Software Engineering

Ravi Sethi

The University of Arizona

July 12, 2017

Preface

"Because so much of what is learned will change over a student's professional career and only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge."

— *The joint IEEE-ACM curriculum guidelines for undergraduate programs in software engineering stress the importance of continued learning.*[1]

For many computer science majors, an introductory course in software engineering will be their only course on the subject.[2] What do they really need to learn in such a course to be productive today and relevant tomorrow? What are the principles and practices that will prepare them to adapt and learn as their careers unfold and software engineering evolves?

Such questions have guided the selection of content for this book. Core concepts that are needed for iterative and agile development are introduced early, so the book can be used for courses with a significant team project. Principles and practices are illustrated by real-world examples and case studies.

Audience: Assume Some Programming Maturity

Software engineering addresses the problems of complexity and scale. The significance of these problems may be lost unless students have some programming maturity. This book assumes that students have had the experience of completing a medium sized programming project.

At the University of Arizona, the introductory software engineering course is at the junior-senior level.

What to Cover? So Much to Choose From

Content selection for an introductory course is a challenge because there is so much to choose from and only so much time in any course. SWEBOK V3, a guide to the software engineering body of knowledge, lists fifteen knowledge areas, together with computing, mathematical, and engineering foundations.[3]

This book favors iterative and agile development over plan-driven processes. Hence, a chapter on working with customers and only a mention of software requirements specifications. The change from plan-driven to iterative and agile processes ripples through the activities of software development. Planning and estimation become adaptive and are distributed across iterations. The distinction between development and testing blurs.

Many of the changes are changes in emphasis, not in the underlying principles. For example, architecture and design principles remain relevant—"The question is not whether or not to design, the question is when to design," as Kent Beck notes in *Extreme Programming Explained*.[4] Customer feedback, goals and metrics, validation and verification are all based on ideas that carry over.

Alignment With a Team Project

The ACM-IEEE computer science curriculum guidelines recommend that

> "the best way to learn to apply software engineering theory and knowledge is in the practical environment of a [team] project."[5]

Software engineering courses with a significant project face the challenge of aligning classroom lectures with the needs of the project. Without alignment, a course appears to consist of two loosely coupled tracks: a principles and practices track and a project track.

The experience at the Rochester Institute of Technology (RIT) is instructive; RIT was the first university in the United States to offer an undergraduate degree program in software engineering. James R. Vallino of RIT observed in 2013,

> "Of all the courses in our software engineering curriculum, our introduction to software engineering course is the one course that we never feel we have done correctly. ... In the course's seventeen year history, we have reworked it seven or eight times."[6]

The ordering of topics in this book is informed by experience with a junior-senior level introductory course at the University of Arizona. The Arizona course has also gone through several iterations.

Use of this Book

The course at Arizona includes a semester-long project. Students form teams of four, pick their own projects, prepare a project proposal addressing a customer need, and then iteratively develop a useful system. They are encouraged to have a real customer, and many do.

The alignment between classroom lectures and the team project is roughly as follows (project iterations need not coincide with chapter boundaries):

Lectures	Project
Chapters 1-2: introduce software engineering and development processes	form teams
Chapters 3-5: iterative and agile processes; customer needs; use cases	write project proposal, with a focus on the customer benefit
Chapters 6-8: estimation; goals and metrics; architecture	Iteration 1: build a minimal viable system; submit design
Chapters 9-10: architectural patterns; software quality	Iteration 2: complete most features; reflect on what went well, what didn't
Chapters 11-??: testing; additional topics	Iteration 3: complete project; submit a comprehensive project report

Even when lectures and project activities are aligned, students may need help in connecting what they are learning in the classroom with what they were doing in their projects. Active-learning activities can help bridge the gap. For example, the Think-Pair-Share technique invites students to briefly think about a question; explore possible answers with their neighbors or team-mates; and then share their deliberations with the class.[7] The entire activity takes only a few minutes. Questions can be posed to invite students to reflect on how the lecture material applies to their their project experience.

Acknowledgements

At Bell Labs, I came to appreciate both the Unix group's iterative approach to refining software tools based on user feedback and the Switching product unit's disciplined plan-driven approach to developing large 99.999% reliable systems, with thousands of software engineers.

At Avaya Labs, David Weiss's Software Technology Research department worked closely with the business units to "improve the state of software in Avaya and know it." I am grateful to David and his department, especially Randy Hackbarth, Audris Mockus, and John Palframan, for their briefings and insights. When I joined the University of Arizona in 2014, David generously

shared the materials from his software engineering course at Iowa State University.

Ravi Sethi
Tucson, Arizona
July 2017

Notes

[1]See Guideline 9 in the Software Engineering 2014 Curriculum Guidelines [4, p. 43].

[2]The opening line borrows from the title of James R. Vallino's position paper [8]

[3]The knowledge areas in SWEBOK 3.0, the software engineering body of knowledge, include requirements, design, construction, testing, maintenance, configuration management, process, models and methods, quality, professional practice, and economics [3]. See also the ACM-IEEE 2013 guidelines for undergraduate computer science curricula, which include software engineering as a knowledge area [1].

[4]In the second edition of *Extreme Programming Explained* [2, ch. 7] , Kent Beck notes that some of the teams misinterpreted the first edition as recommending deferring design until the last moment—they created brittle poorly designed systems. He recommends deferring until the last "responsible" moment. The quote is from the same chapter.

[5]The ACM and IEEE curriculum guidelines for both computer science [1, p. 174] and software engineering [4, p. 45] strongly recommend a significant team project.

[6]For reflections on the sophomore-level introductory software engineering course at the Rochester Institute of Technology, see the 2005 report by Stephanie Ludi, Swaminathan Natarajan, and Thomas Reichlmayr [5]. See also the 2013 position paper by James R. Vallino [8] and the retrospective by Michael J. Lutz, Fernando Naveda, and James R. Vallino [6].

[7]The Think-Pair-Share technique is due to Frank Lyman, Jr. [7].

References

1. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013.* ACM Press and IEEE Computer Society Press (December 2013) `http://dx.doi.org/10.1145/2534860` .

2. Kent Beck, with Cynthia Andres, *Extreme Programming Explained: Embrace Change, 2nd Ed.* Addison-Wesley, Reading, Mass. (2005).

3. Pierre Borque and Richard E. Fairley (eds) *Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0.* IEEE Computer Society (2014) `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1265988` .

4. IEEE Computing Society and ACM. *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* `http://www.acm.org/education/se2014.pdf` .

5. Stephanie Ludi, Swaminathan Natarajan, and Thomas Reichlmayr. An introductory software engineering course that facilitates active learning. *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)* (February 2005) 302-306.

6. Michael J. Lutz, Fernando Naveda, and James R. Vallino. Undergraduate software engineering: Addressing the needs of professional software development. *ACM Queue* 12, 6 (June 2014) 30-39.

7. Frank Lyman, Jr. The responsive classroom discussion: the inclusion of all students. In *Mainstreaming Digest: A Collection of Faculty and Student Papers*, Audrey Springs Anderson (ed.) University of Maryland (1981) 109-113.

8. James Vallino. What should students learn in their first (and often only) software engineering course? *IEEE Conference on Software Engineering Education and Training* (2013) 335-337.

Contents

Chapter 1

What is Software Engineering?

"Today, we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes—build the whole thing, push it off the cliff, let it crash, and start over again."

— *Robert M. Graham on the state of the art around 1968.*[1]

Software remains invisible when it works as expected. It stands out when it fails us in some way. Occasionally, software fails in ways that evoke the following:

> "The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time."[2]

This quote is from a 1968 international NATO workshop on the problems of software development. The organizers chose the title Software Engineering for the workshop to highlight the need for making software development an engineering discipline based on scientific principles and rigorous practices.

Since then, software engineering has progressed from an aspiration to a field in its own right. There is a growing body of knowledge, incorporating hard-won lessons from both successful and unsuccessful projects. The professional societies, ACM and IEEE, have jointly published curricula for undergraduate

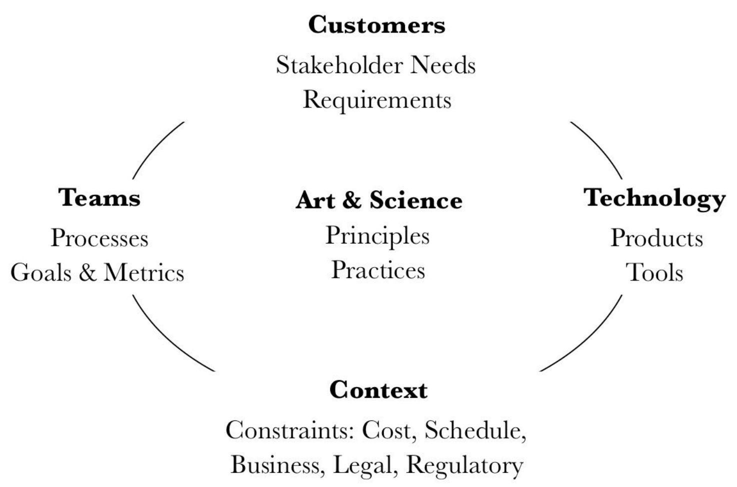


Figure 1.1: A framework for discussing aspects of software engineering.

courses and degree programs. Thousands of books have been published on the subject.

Still, there continue to be differences of opinion about whether software engineering has truly matured into an engineering discipline.[3] What are its underlying principles? What are its best practices? What are the concepts that every software engineer must master?

1.1 Introduction to Software Engineering

Software engineering involves both product and process. Informally, product refers to what is developed and process refers to who does what by when. Products are artifacts. Processes deal with how teams and development are organized.

But, there is more to software engineering than products and processes. Software is commissioned by customers and is developed for their benefit. Software projects live within a context: they are subject to resource constraints as well as business, legal, and regulatory constraints.

The framework in Fig. 1.1 touches on these four key aspects of software engineering. Customers are at the top. To the right is technology, which includes both software products and the software tools that are used to build products. To the left are teams—the framework groups processes under teams, instead of the other way around, to emphasize that software systems are built by people in teams. Context is at the base of the framework.

1.1.1 Customers: Needs and Requirements

At the top of the framework in Fig. 1.1 are customer needs, which drive requirements. *Requirements* define what a software system must do and what it must not do. The term customer is to be interpreted broadly: requirements are based on the needs of stakeholders, where a *stakeholder* is anyone with a stake in the system. The stakeholders in a flight-control system include not only the airline executives who sign the contract for the system, but the pilots who rely on the system to fly the plane, the technicians who must maintain the system, not to mention the passengers who are served by the airline.

Different stakeholders may have different, perhaps conflicting, ideas about what they need, ideas that have to be elicited and reconciled into a prioritized set of requirements.

Knowing the requirements—knowing exactly what to build—can be a challenge. Furthermore, requirements typically change during the life of a software project. For the software for the space shuttle, NASA and its contractors made over 2,000 requirements changes over six years, before the first flight in 1981.[4]

1.1.2 Teams: Software Development Processes

To the left in Fig. 1.1 are teams and how they are organized for developing and delivering a system. Development activities include design, coding, testing, and maintenance. Associated with each activity are goals that motivate the activity and metrics that measure progress toward the goals.

The rules for organizing activities during software development are called *software development processes*, or simply *processes*. Processes can vary from project to project, even within the same organization.

The prevailing wisdom about software development in the 1970s was akin to "measure twice, cut once:" gather detailed requirements and do a careful design before implementing the system. The problem with this linear process—gather requirements, do a design, implement, and deliver—is that requirements changes can result in rework and wasted effort. The 2,000 requirements changes for the space shuttle software resulted in massive cost over-runs: the project cost $200 million instead of the planned $20 million.[4]

Iterative processes are a class of processes for incrementally delivering a system a little at a time, starting with a minimal working system with just enough functionality to get stakeholder feedback. The feedback is then used to make corrections and to decide what to deliver in the next iteration. Increasingly, software is delivered using iterative processes.

1.1.3 Technology: Deliver Products Using Tools

To the right in Fig. 1.1 is technology, which takes two forms: products and tools. Here, *product* refers broadly to an artifact that results from a development activity. For example, architects create designs, developers write code, testers come up with tests. The designs, code, and tests are artifacts, as is the documentation that accompanies an artifact. When talking about software development, it is convenient to use the familiar term *system* for any product delivered by a development team.

A *tool* is a piece of software that is used to develop artifacts. Tools include programming environments, languages and compilers, and testing tools. Unlike artifacts, which are for external delivery to customers, tools are used internally to facilitate development.

1.1.4 Context: All Projects Face Constraints

At the bottom of Fig. 1.1 is the context for a software project. All projects face cost and schedule constraints. Most projects face additional constraints. For example, a company may have enough budget and staff and still face resource constraints if it does not have enough staff with the right skills, or staff with the desired skills at a given geographic location. Projects also face regulatory and legal constraints, such as safety regulations and export controls.

Open source software projects have evolved their own ground rules for organizing the efforts of independent contributors. The ground rules act as constraints on contributions.

1.1.5 Working Definition of Software Engineering

Many have tried, but no simple definition has captured all of the aspects of software engineering. The art and science of software engineering is therefore shown in the middle of Fig. 1.1, linking customers, teams, technology, and context. These aspects are strongly connected; they are part of a whole.

Example 1.1 : Consider security and privacy.

Customers. As a rule, the privacy of personal information is a requirement.

Context. Regulations such as the U.S. Health Insurance Portability and Accountability Act of 1996 (HIPAA) protect individually identifiable health information.

Teams. Development activities include testing to prevent inadvertent security holes in the code. (In 2012, an extra unwanted `goto` introduced a security vulnerability that affected hundreds of millions of Apple iOS and Mac OS X users.[5])

Technology. Systems must be deployed and operated to keep hackers out.

Thus, security and privacy touch on customer needs, team activities, the regulatory context, and delivered technology. □

The following simple definition touches on the various aspects of software engineering in Fig. 1.1. It does not pretend to be as complete or as general as it could be.

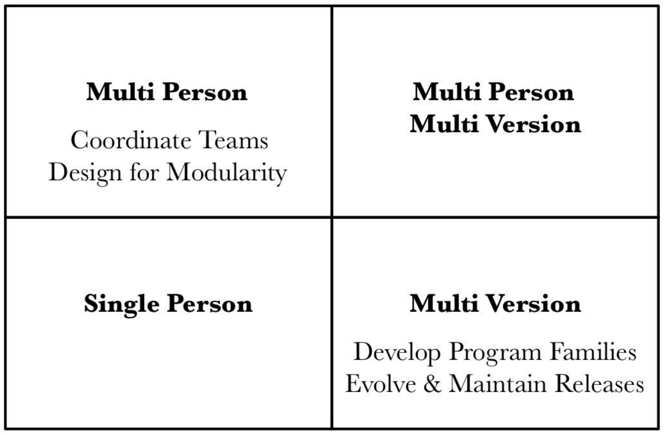


Figure 1.2: Beyond single person programming by a single person for personal or friendly use.

> *Software engineering* is the art and science of developing reliable software systems that address customer needs, subject to resource, business, and societal constraints.[6]

1.1.6 Software Engineering is More than Programming

A standalone program written for personal or friendly use can be an order of magnitude less expensive than a software product that has been carefully designed, cleanly implemented, extensively tested, and properly documented. Programming is an essential part of software engineering, and there is more to software engineering than programming.

The following concise characterization identifies some distinctions between software engineering and programming:

> Software engineering is "multi-person development of multi-version programs."[7]

.

Multi-Person

Multi-person implies the need for coordination and communication between people; see the top-left box in Fig. 1.2. The larger the project, the greater the need for coordination. For example, in 1995, Microsoft enrolled all 18,000 employees in a "companywide emergency" to build a web browser. Several teams worked in parallel in a concerted effort to catch up with Netscape Navigator.[8]

Multi-person products benefit from designs that (a) support the distribution of work to different team members and (b) the subsequent integration of individual contributions into a cohesive system.

Multi-Version

Multi-version development can take at least two forms: a family of versions; and, a sequence of releases. (See the bottom-right box in Fig. 1.2.)

A *product family* is a set of products designed from the start to (a) take advantage of what they have in common (their *commonalities*), and (b) manage what varies between them (their *variablilities*). For example, different versions of a mobile banking app are needed for iOS and Android. The user features of the app would be among the commonalities, while the iOS and Android specific dependencies would be among the variabilities.

Multi-version implies the need to design, code, test, and maintain multiple versions. Such products benefit from the sharing of designs and components across versions.

Thus, multi-person and/or multi-version programming goes beyond programming for personal or friendly use.

1.2 A Software Engineering Success Story

Each application has its own priorities and concerns. Reliability was an overriding concern for the mission to Mars, discussed in this section. The users of a self-publishing system may be willing to live with an occasional crash, as long as the finished product looks good. Payment systems, social media, digital libraries, games, ... all have their own priorities and concerns, be it security, scale, accessibility, or responsiveness. Since applications vary, so do software systems and the software engineering approaches used to produce them

For a mission-critical system, such as the software to control a spacecraft, development activities have to be rigorously organized to ensure the success of the mission. This section outlines the extraordinary software development measures that were taken by the team to ensure a successful mission to Mars.

1.2.1 Mission to Mars

After traveling 350 million miles in 274 days, the Mars rover, Curiosity, made a flawless soft landing on the planet's surface on August 5, 2012. For perspective on the distance to Mars, it took 14 minutes for signals about the landing to reach Earth.

The landing sequence took seven minutes, from entry into the Mars atmosphere to touchdown on the surface. The engineers who designed the sequence referred to it as "seven minutes of terror" because so many things had to go right in perfect synchrony for the landing to be successful.[9]

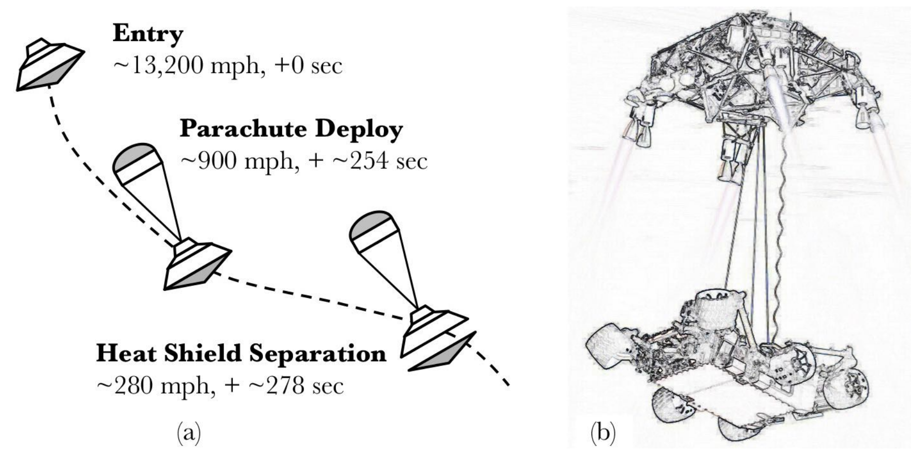


Figure 1.3: (a) Preparing for landing on Mars. (b) The sky-crane lowering the Mars rover. Diagrams adapted from NASA/JPL illustrations.

All functions on the rover and its spacecraft were controlled by software. The landing sequence was choreographed by 500,000 of the 3 million lines of code, mostly in the C programming language. The code was the work of a team of 35 people.

1.2.2 The Crucial Landing Sequence

The Mars atmosphere is 100 times thinner than Earth's, which posed challenges for slowing the spacecraft down from 13,200 miles an hour to 0 upon landing. During entry and descent, the rover was enclosed in a backshell and protected by a heat shield; see Fig. 1.3(a).; Besides the rover, the backshell contained a sky crane that handled the final descent.

With less the a minute to go before landing, the heat shield and backshell had separated. The sky crane took over. It used rockets to slow the descent and gradually lowered the rover for a soft landing; see Fig. 1.3(b). Finally, the sky crane disconnected and flew a safe distance away, to avoid crashing on top of the rover.

1.2.3 Overriding Concern: Reliability

Brief descriptions of the team's development activities appear in Fig. 1.4. The activities are grouped under three headings: design, coding, and verification. *Verification* refers to activities that ensure that a system is built correctly.

Design	• Design for fault tolerance • Document the rationale behind design decisions
Coding	• Practice defensive coding to guard against unforeseen events • Annotate programs with assertions about expected behavior
Verification	• Peer code reviews for identifying design flaws • Static checking of source code for compliance with rules • Rigorous testing (unit and integration) • Model checking of parallel tasks

Figure 1.4: Selected development activities for the software for the Mars mission.

Design

The team designed a software architecture that was clean, with well defined interfaces. The design was fault tolerant, with a redundant backup computer system. Not only was the hardware redundant to recover from hardware faults, the software was redundant as well, to recover from software faults. During the crucial landing sequence, the backup computer ran independently written control software, so that the backup software did not simply run into the same fault, if the main computer system failed.

Verification

The emphasis on reliability drove the team's practices, especially related to verification; see Fig. 1.4 for a summary. Verification was deeply entwined with the other development activities.

- Designs and code were reviewed line-by-line by peers, using a formal process. All 10,000 comments gathered during 145 peer code reviews were individually tracked and addressed.

- All code was automatically checked nightly for consistency and compliance with the team's coding standards. Four different commercial static analysis tools were used, since each tool picked up issues that the others did not.

- The code for the mission was highly parallel, with 120 tasks—parallel programs are notoriously hard to verify. The team used a powerful verification technique called logic model checking.

Overall, every precaution was taken to ensure the success of the mission.

- *Public.* Software engineers shall act consistently with the public interest.
- *Client and Employer.* Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- *Product.* Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
- *Judgment.* Software engineers shall maintain integrity and independence in their professional judgment.
- *Management.* Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
- *Profession.* Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- *Colleagues.* Software engineers shall be fair to and supportive of their colleagues.
- *Self.* Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.5: Short version of the ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice. These aspirations are to be taken with the examples and details in the full version of the code, since "without the details, the aspirations can become high sounding but empty."

The software development practices deployed for the Mars project are almost all standard. Countless other projects have used them. What distinguishes the Mars project is not the novelty of the development practices, but the rigor with which the practices were deployed.

1.3 Ethics: A Cautionary Tale

In 1986, a patient died after radiation treatment by a software-controlled medical device called Therac-25. It was one of six known accidents with the device. Nancy Leveson and Clark Turner examined information from lawsuits, correspondence, and regulatory agencies and concluded that basic software engineering principles had apparently been violated during the development of the software for the medical device.[10]

The Therac-25 accidents are a cautionary tale of the societal impact of software projects. The professional societies ACM and IEEE have published a Code of Ethics; see Fig. 1.5.[11] The Code recommends that software engineers act in the public interest, in the best interests of clients and employers, and maintain integrity and independence in their professional judgment.

1.3.1 Therac-25: Malfunction 54

When the patient came to the East Texas Cancer Center for his ninth treatment on March 21, 1986, more than 500 patients had been treated on the Therac-25 radiation therapy machine over a period of two years. The planned dose was 180 rads. Nobody realized that the patient had actually received between 16,500 and 25,000 rads over a concentrated area, in less than 1 second. He died five months later due to complications from the massive overdose.

On March 21, when the technician pressed the key for treatment, the machine shut down with an error message: "Malfunction 54," which was a "dose input 2" error, according to the only documentation available. The machine's monitor showed that a substantial underdose had been delivered, instead of the actual overdose. The machine was shut down for testing, but no problems were found and Malfunction 54 could not be reproduced. The machine was put back in service within a couple of weeks.

Four days after the machine was put back in service, the Therac-25 shut down again with Malfunction 54 after having delivered an overdose to another patient, who died three weeks later. An autopsy revealed an acute high-dose radiation injury.

After the second malfunction, the physicist at the East Texas Cancer Center took the Therac-25 out of service. Carefully retracing the steps by the technician in both accidents, the physicist and the technician were eventually able to reproduce Malfunction 54 at will. If patient treatment data was entered rapidly enough, the machine malfunctioned and delivered an overdose. With experience, the technician had become faster at data entry, until she became fast enough to encounter the malfunction.

The accidents in Texas were later connected with prior accidents with Therac-25, for a total of six known accidents between 1985 and 1987. After the 1985 accidents, the manufacturer made some improvements and declared the machine fit to be put back into service. The improvements were unrelated to the synchronization problems that led to the malfunctions in Texas in 1986. On January 17, 1987, a different software problem led to an overdose at the Yakima Valley Memorial Hospital. The Yakima problem was due to a coding error in the software.

1.3.2 Lessons Learned

The software for Therac-25 was reused from an earlier machine, called Therac-20. A related software problem existed with Therac-20, but that earlier machine had a hardware interlock to prevent an accidental overdose. Therac-25 did not have a hardware interlock.

The lessons from the Therac-25 accidents include the following:

- *System Failures.* A system can fail due to interaction between its components or between the system and its environment. The failure in Texas was due to an interaction between the technician and the machine. Specifically, the Therac-25 software had concurrent tasks and Malfunction 54 was due to a task synchronization problem.

- *Defensive Design.* Design for fault tolerance, so a failure in one part of the system is contained rather than cascaded. Therac-20 had the same synchronization problem, but it had a hardware interlock to guard against an overdose.
- *Software Engineering Practices.* The practices followed for Therac-25 stand in sharp contrast to those followed for the successful Mars mission; see Fig. 1.4. For Therac-25, the first safety analysis did not include software. Testing was inadequate. Documentation was lacking.

1.4 Conclusion

Software engineering is a rich subject that encompasses technology and teams, customer needs and contextual constraints. It is not just about technology or programming and it is not just about teams or processes. The following definitions of software engineering from Sections 1.1 and 1.1.6 touch on the many aspects of the subject (see also the framework in Fig. 1.1):

- Software engineering is the art and science of developing reliable software systems that address customer needs, subject to resource, business, and societal constraints.
- Software engineering is multi-person development of multi-version programs.

The successes of software engineering are all around us. The mission to Mars described in Section 1.2 is a noteworthy example. Meanwhile, the Therac-25 accidents are a reminder that software engineers need to conduct themselves ethically and professionally; see Section 1.3.

Exercises for Chapter 1

Exercise 1.1 : Based on a thorough investigation of the Therac-25 accidents, the following software engineering practices were violated:[12]

- Specifications and documentation should not be an afterthought
- Establish rigorous software quality assurance practices and standards
- Keep designs simple; avoid dangerous coding practices
- Design audit trails and error detection into the system from the start
- Conduct extensive tests at the module and software level
- System tests are not enough

- Perform regression tests on all software changes
- Carefully design user interfaces, error messages, and documentation

How would each of these practices have helped avoid the Therac-25 accidents? Provide 2-3 bullet items per practice to convey that you understand the practice would be sufficient.

Notes for Chapter 1

[1]Comment by Robert M. Graham during the 1968 NATO workshop [10, p. 17].

[2]Kenneth W. Kolence, during the 1968 NATO conference [10, p. 16].

[3]The description of the 2014 IEEE-ACM software engineering curriculum recognizes “that software engineering, as a discipline, is relatively immature and that common agreement on the definition of an education body of knowledge is evolving” [6, p. 24]. Note also the comments by Parnas [11]: “For each of the traditional engineering disciplines, there is agreement on a core body of knowledge, which comprises the skills and knowledge that all of those licensed to practice in that discipline must have. No such body of knowledge has been identified for software developers. There are a number of proposals but they do not have the broad acceptance that the core body of knowledge for disciplines like Civil Engineering has received.”

[4] Lynn Killingbeck interview reported by Tomayko [12, chapter 4, p. 108].

[5]The security vulnerability was in the Secure Sockets Layer (SSL) code; see Bland [3].

[6]The working definition of software engineering is close to the following comment from the 2013 ACM-IEEE Curriculum Guidelines [2, p. 172]: “Software engineering is the discipline concerned with the application of theory, knowledge, and practice to effectively and efficiently build reliable software systems that satisfy the requirements of customers and users.”

[7]Parnas [11, p. 415] attributes “multi-person development of multi-version programs” to Brian Randell.

[8]MacCormack [9].

[9]Holzmann [4, 5] describes software development for the Mars mission.

[10]The Therac-25 accidents were never officially investigated. Leveson and Turner [8] had to infer information about “the manufacturer’s software development, management, and quality control” practices.

[11]The short version of the ACM/IEEE-CS Code of Ethics [1] is reprinted by permission. ©1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

[12]Leveson and Turner [7, 8] conducted an unofficial investigation of the Therac-25 accidents.

References for Chapter 1

1. ACM/IEEE-CS. *Software Engineering Code of Ethics and Professional Practice* (1999). `https://www.acm.org/about/se-code`.

2. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013.* ACM Press and IEEE Computer Society Press (December 2013) `http://dx.doi.org/10.1145/2534860`.

3. Mike Bland. Finding more than one worm in the apple. *Comm. ACM* 57, 7 (July 2014) 58-64.

4. Gerard J. Holzmann. Landing a spacecraft on Mars. *IEEE Software* (March-April 2013) 17-20.

5. Gerard J. Holzmann. Mars code. *Comm. ACM* 57, 2 (February 2014) 64-73.

6. IEEE Computing Society and ACM. *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* `http://www.acm.org/education/se2014.pdf` .

7. Nancy G. Leveson. Medical Devices: The Therac-25. Appendix A of *Software: System Safety and Computers* by Nancy Leveson. Addison Wesley, Reading, Mass. (1975) `http://sunnyday.mit.edu/papers/therac.pdf` .

8. Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer* 26, 7 (July 1993) 18-41. `http://courses.cs.vt.edu/professionalism/Therac_25/Therac_1.html` .

9. Alan D. MacCormack. Product-development processes that work: How Internet companies build software. *Sloan Management Review* 42, 2 (Winter 2001) 75-84.

10. Peter Naur and Brian Randell (eds). *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, 7th to 11th October 1968. (January 1969). `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF` .

11. David L. Parnas. Software engineering: multi-person development of multi-version programs. In *Dependable and Historic Computing*, C. B. Jones and J. L. Lloyd (eds.), *Lecture Notes in Computer Science* 6875 (2011) 413-427.

12. James E. Tomayko. *Computers in Spaceflight: The NASA Experience*. NASA Contractor Report 182505 (March 1988). `https://archive.org/details/nasa_techdoc_19880069935` .

Chapter 2

Introduction to Processes

"Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them."

— *Doug McIlroy, Elliot Pinson, and Berkley Tague from their 1978 introduction to the Unix program development style.*[1]

A *process* is a systematic method for meeting the goals of a project by

- defining and organizing development activities,
- determining the roles and skills to do the project, and
- setting criteria and deadlines for progressing from one activity to the next.

Informally, a process guides "who will do what by when and why."[2]

The selection of a development process is one of the earliest decisions during the life of a software project. The selection is strongly influenced by the nature of the project. A process that works for one project may not be a good fit for another. The right process can mean the difference between success and failure.

Processes are sometimes called *methods*, as in "agile method" for a specific agile process. Classes of processes are sometimes called *models*, as in "iterative model" for the class of iterative processes.

2.1 Software Development Processes

This section introduces two broad classes of software development processes, called plan-driven and iterative. As we shall see, the trend has been away from plan-driven to various forms of iterative processes, although plan-driven processes continue to be used for mission-critical projects; e.g., for medical devices or for flight-control software.

A *plan-driven process* is characterized by up-front planning, design, and documentation. Then come coding, testing, and other implementation activities. Customers are involved during planning, but there is minimal customer involvement during implementation.

An *iterative process* has a sequence of steps or iterations that produce increasingly functional versions of a system. The idea is to start with an early working version that provides just enough functionality for customers to extrapolate what they will get from the finished system. With each iteration, the software does more of what customers want. Their feedback on previous iterations guides what gets added or changed in the next iteration.

A characterization of the two classes appears in Fig. 2.1. Early developments and plan-driven processes are covered in this chapter. Iterative and agile processes are covered in Chapter 3. Agile processes are are a refinement of iterative processes. Agile processes include Scrum, discussed in Section 3.4, and Extreme Programming (XP), discussed in Section 3.5.

2.1.1 Build versus Grow

Informally, a plan-driven process *builds* a system, whereas an iterative process *grows* a system. The build analogy is with physical buildings. A building goes through phases: an architect prepares a blueprint; a contractor then builds according to the blueprint; an inspector finally certifies that the completed building is fit for occupancy. With each phase, the building takes shape, but it is not usable until the end. With plan-driven software processes, the system comes together at the end, when coding is all done.

The grow analogy is with biological organisms. There is a living organism from the start; its capabilities grow as it evolves. With iterative processes, a minimal usable system grows in functionality as the project progresses. As Fred Brooks observed,

> "Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a graphics system appears on the screen, even if it is only a rectangle. One always has, at every stage in the process, a working system."[3]

2.1.2 Examples of Plan-Driven and Iterative Processes

All software development projects share the same broad goals: identify customer needs; build a system; deliver it or deploy it. The next example describes a simple plan-driven process that addresses these goals sequentially.

Example 2.1: The plan-driven process in Fig. 2.2 has four activities, represented by the boxes in the figure: identify customer needs; plan an app; build

Plan-Driven
Pure Waterfall
Iterative
Agile

PLAN-DRIVEN PROCESSES

- Up front planning, design, and documentation drive development
- Minimal customer feedback during implementation
- Delivery times measured in months; e.g., 18-24 months

ITERATIVE PROCESSES

- Iterations deliver increasingly functional versions of the system
- Customer feedback guides iteration planning
- Quick iterations, measured in weeks; e.g., weekly agile iterations

Figure 2.1: Characteristics of two classes of processes: plan-driven and iterative.

the app; and submit the app to an app store. The three roles—not shown in the figure—are customer, product owner, and developer.

The following is a brief description of the process:

1. *Identify Customer Needs.* The product owner, with the help of the development team, works with selected customers to identify their needs. The discussions with the customers continue until the customers are satisfied that the product owner really understands their needs. The result of the discussions is a set of requirements for an app that would delight the customers.

2. *Plan an App.* Based on the requirements, the development team designs an app and creates a plan for implementing the design. This activity continues until the design specification and the implementation plan are completed.

3. *Build the App.* According to plan, the development team builds the app. The development of the app completes when the product owner and the

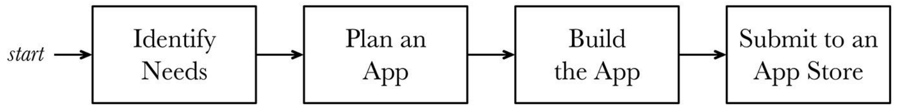


Figure 2.2: A sequential plan-driven process.

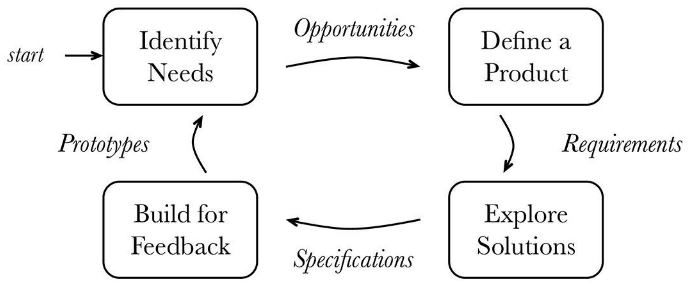


Figure 2.3: An iterative process.

development team are satisfied that the app correctly implements the design specification.

4. *Submit to an App Store.* The development team submits the completed app to an app store. The submission completes when the app store accepts the app.

The above description outlines the activities, identifies who does what, and includes the criteria for progressing from one activity to the next. □

The next example illustrates an iterative process adapted from Scenario-Focused Engineering, which has been used within Microsoft.[4]

Example 2.2 : The process in Fig. 2.3 has four activities, represented by the boxes in the figure. The arrows represent transitions between activities. The label on an arrow from activity A to activity B shows the artifact produced by A and passed to B. For example, the "Identify Needs" activity results in a list of opportunities that is passed to the "Define a Problem" activity.

This example provides an overview of the process; see also Section **??**.

1. *Identify Needs.* The development team works with selected customers to identify their needs. The discussion results in a list of opportunities for addressing customer needs.

2. *Define a Product.* The development team prioritizes the opportunities and defines a product that will address some unmet customer needs. The result of this activity is a set of requirements for a product.

3. *Explore Solutions.* The development team then explores possible solutions that meet the requirements. The exploration ends with the design specification for an implementation.

4. *Build for Feedback.* The team builds a prototype, based on the specification. The purpose of the prototype is to get customer feedback on

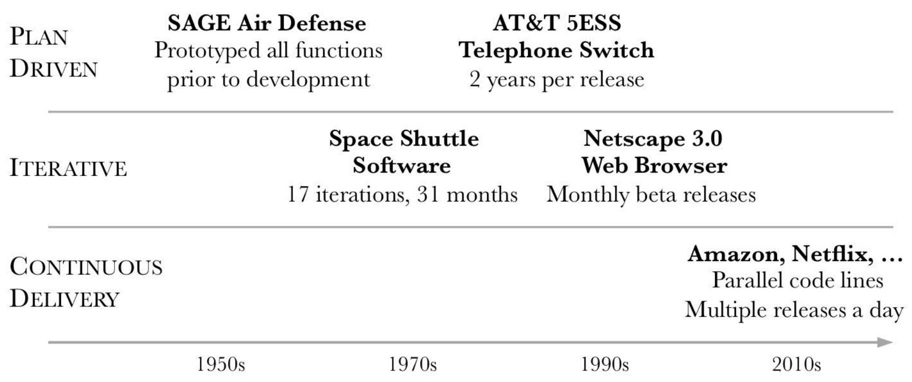


Figure 2.4: Rough timeline with examples of successful software projects using three classes of processes: plan-driven, iterative, and continuous delivery.

the proposed solution. The feedback will be used in the next iteration to refine or even replace the solution approach. The feedback session may also uncover additional needs.

The cycle begins anew at the first activity. The development team works with the selected customers to identify any further needs. The process completes when the prototype has evolved into a product that delights the customers. □

2.1.3 The Process Landscape

For perspective on software development processes, consider the timeline in Fig. 2.4. The successful U.S. SAGE Air Defense system and the flagship AT&T 5ESS Switching System were developed using plan-driven processes. They took years to develop.

The software for the Space Shuttle and for the Netscape Navigator 3.0 web browser were delivered incrementally, using iterative processes. Rapid iterations delivered relatively small increments of software, compared to the complete systems delivered for SAGE and 5ESS. The iterations for the shuttle took roughly 8 weeks and the ones for the browser took 4 weeks.

Delivery times continue to shrink. Technology companies like Amazon, Facebook, Google, and Netflix now practice continuous delivery; they deploy pieces of software multiple times of day, sometimes hundreds of times a day. Each individual piece may have taken days or weeks to develop, but with many teams working in parallel, software from some team or the other is deployed every so many minutes or every so many seconds.

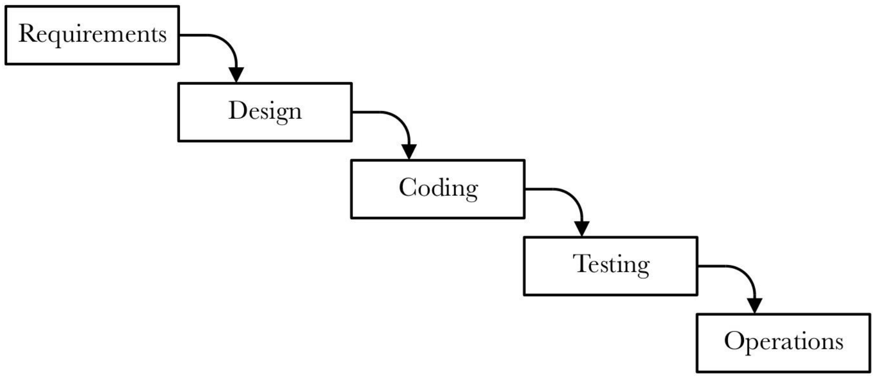


Figure 2.5: A pure waterfall process.

2.2 Early Developments

In the late 1950s, the computer industry faced a software crisis. Software development was a craft that relied on skilled programmers and there was a shortage of skilled programmers. Ad hoc "code and fix" programming methods were not meeting the challenge of increasingly complex software projects.

At the time, efficient use of computing resources was all important. Computers were expensive and bulky and filled large rooms. Computer time was precious: an hour of computer time was 300 times the cost of an hour of a programmer's time.[5] Software was written in assembly language. Code clarity and maintainability were often sacrificed in the name of efficiency.

2.2.1 Waterfall: Seeking Order Amid Chaos

To manage software development, the computer industry adopted processes that were similar to the phase-gate processes used by the chemical industry. In Fig. 2.5, phases are represented by boxes and the flow from one phase to the next is represented by arrows. "Gate" in phase-gate refers to a review at the end of a phase. The gates were to decide whether to continue a project by proceeding to the next gate, or to redirect or even cancel the project.

Waterfall processes are named after diagrams like the one in Fig. 2.5. The flow of control from phase to phase looks a little like water streaming over a series of drops in a waterfall.[6]

A *pure waterfall* process divides software development into sequential phases, such as requirements gathering, design, coding, and testing. It is a defining characteristic of pure waterfall processes that each phase completes before the next one begins.

Waterfall diagrams are neat and tidy and were viewed as an ideal, to the point where the Space Shuttle project felt the need to justify the use of an iterative rather than a waterfall process:[7]

> "From an idealistic viewpoint, software should be developed from a concise set of requirements that are defined, documented, and established before implementation begins. The requirements on the Shuttle program, however, evolved during the software development process."

(See Section 3.2 for the iterative process used for the Space Shuttle software.)

2.2.2 Limitations of Waterfall Processes

Thousands of useful software systems were built using variants of the waterfall model. But, there were also many failures. Many pure waterfall projects failed to live up to their promise or failed entirely.[8]

Waterfall processes assume that we can specify what the system must do before we design it, we can design it before we code it, we can code it before we test it, and that everything will go according to plan.

The problems with pure waterfall processes are twofold:

- *Requirements change.* During the months or years between initial requirements gathering and final testing, customer needs may have changed significantly. Since design and implementation are based on the initial requirements, it is highly likely that the end result will not meet the customer's changed needs.
- *Issues surface late in the project's lifecycle.* Integration and testing come at the end of the development cycle. Design, coding, and performance issues can therefore lurk undetected until the end. In practice, late discovery of major issues has resulted in replanning and rework, leading to significant cost and schedule overruns.

These problems were known all along. The 1970 paper that introduced waterfall diagrams notes that the pure waterfall model is risky and invites failure. To reduce risk, the paper suggests steps like prototyping, customer involvement, and early test planning.

Example 2.3 : In 2013, integration and testing came at the tail end of the rollout of the `healthcare.gov` website, too late to address usability and performance issues.

The Affordable Care Act was the most significant overhaul of the U.S. healthcare system in decades. The software system to implement the law was the result of a two-year project, with components built by several contractors. The system included a website, `healthcare.gov`, for people to enroll for health insurance.

The October 1, 2013 launch of the website did not go well. The chairman of the oversight committee opened a congressional hearing on October 24 with

> "Today the Energy and Commerce Committee continues our ongoing oversight of the healthcare law as we examine the many problems, crashes, glitches, system failures that have defined open enrollment [for health insurance]."

When questioned, one of the contractors admitted that integration testing for the website began two weeks before launch. Another contractor admitted that full end-to-end system testing did not occur until "the couple of days leading up to the launch."

At the time, \$118 million had already been spent on the website alone.[9] □

2.3 Plan-Driven Processes in Practice

Given the known limitations of waterfall processes, how did they become—and remain—the dominant process model for several decades? What made some software projects successful where others failed?

Some successful projects didn't use a waterfall process: instead, they used an iterative process to accommodate changing requirements. But, many did use a waterfall variant. They were successful because they were good at managing the risks associated with software development.

2.3.1 Software Risk Factors

In practice, project managers rely on experience and intuition to deal with the factors that contribute to success or failure. Although risk can be defined formally by

$$\textit{Risk} \text{ of outcome} = (\textit{Probability} \text{ of outcome}) \times (\textit{Impact} \text{ of outcome})$$

few project managers explicitly estimate probabilities and impacts.

The software risk factors in Fig. 2.6 are adapted from published checklists of serious risks. A 1998 study asked project managers in three countries to identify and rank order risk factors. The results were consistent: 7 of the 11 most serious risks were related to customer requirements. Also on the most serious list: shortfalls in staff and skills and lack of top management commitment. The list may be biased by the fact that the project managers in the study tended to give more weight to risks that were outside their control. They treated uncontrollable risks as being more severe.[10]

The case studies in this section illustrate techniques for managing risks related to customer requirements and system implementation.

2.3.2 Disposable Prototype Lowers Risk

In the 1950s, the team for the SAGE air defense system used prototyping to gain a deep understanding of both the problem and of a workable solution. In other words, they gained a deep understanding of both the requirements and of

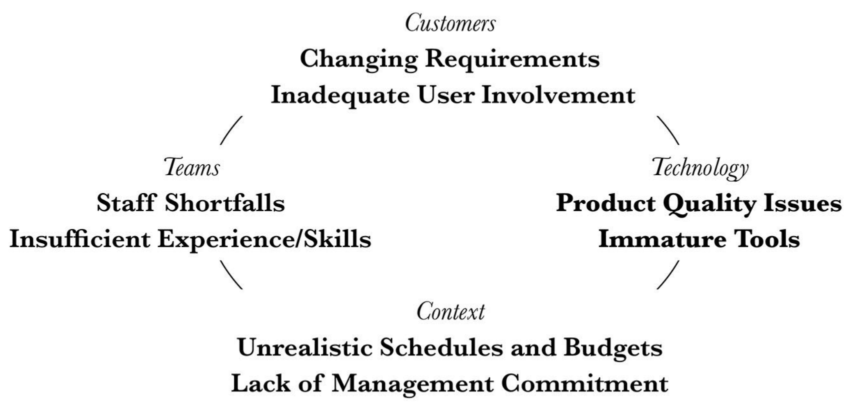

Customers
Changing Requirements
Inadequate User Involvement
Teams
Staff Shortfalls
Insufficient Experience/Skills
Technology
Product Quality Issues
Immature Tools
Context
Unrealistic Schedules and Budgets
Lack of Management Commitment

Figure 2.6: Examples of risks faced by software projects.

a possible implementation. Then, they successfully built SAGE, as described in the following example.

Example 2.4 : SAGE (Semi-Automated Ground Environment) was an ambitious distributed system that grew to 24 radar-data collection centers and 3 combat centers spread across the United States.

Development of the 100,000 instruction system began with an initial prototyping phase to explore the tradeoffs between performance, cost, and risk. Herbert Benington writes,[11]

> "The experimental prototype ... performed all the bare-bones functions of air defense. Twenty people understood in detail the performance of those 35,000 instructions; they knew what each module would do, they understood the interfaces, and they understood the performance requirements."

Having done a realistic prototype, the team was well prepared to build the final system. □

The SAGE team also did careful testing to ensure product quality. Their development process is discussed next.

2.3.3 V-Processes: Levels of Specification and Testing

A *V-process* consists of a sequence of specification phases followed by coding and a sequence of testing phases. The specification and testing phases are paired, like opening and closing parentheses, surrounding the coding phase. The purpose of each testing phase is to verify that the code implements the corresponding specification.

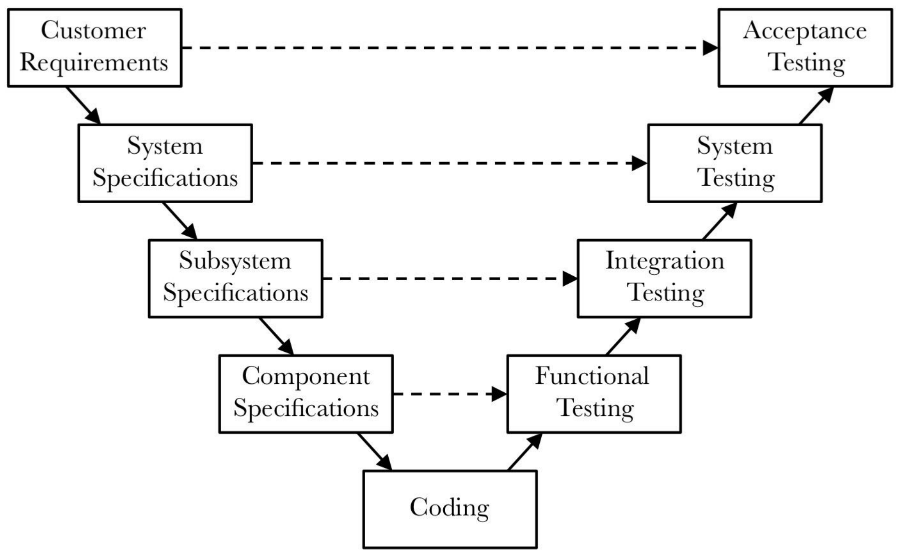


Figure 2.7: Each specification phase has a corresponding verification phase.

Diagrams for V-processes resemble the letter V. As in Fig. 2.7, the specification phases are drawn going down and to the right. Coding is at the bottom. The testing phases are drawn going up and to the right.

Example 2.5 : The process in Fig. 2.7 is inspired by the development process for the SAGE air defense system.[12]

The first four phases develop increasingly detailed specifications. The customer-requirements phase specifies the usage and operation of the system from a user perspective. The system-specifications phase outlines the behavior of the system as a black box. The subsystem-specifications phase organizes the components of the system. The components-pecifications phase describes what each component must do.

The dashed arrows in Fig. 2.7 link a specification phase with its corresponding testing phase. □

If we think of testing as having two parts, test planning and test execution, then test planning can be done early. (Modern test-driven development is based on the idea of starting with tests and then writing the code so it passes the tests.) Test planning can be done in the down part of the V, before coding begins. Test execution must follow coding, so test execution is in the up part of the V. Including relevant tests with a specification strengthens the specification.

V-processes are essentially waterfall processes. Note that the solid arrows in Fig. 2.7 trace the sequential flow from customer requirements to acceptance

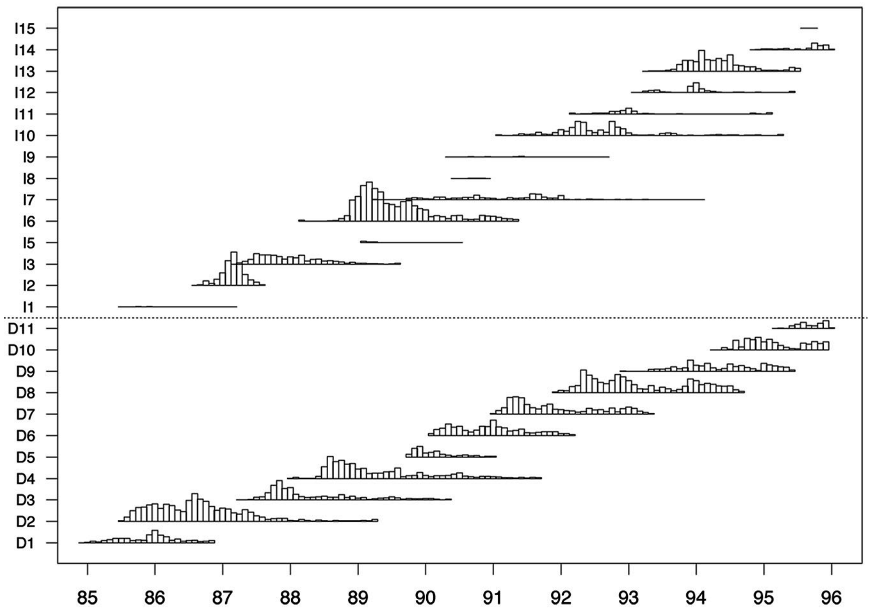


Figure 2.8: Data for releases of the 5ESS switching system, 1985-1996.

testing. There may be several testing phases in a V process.

2.3.4 Multiple Plan-Driven Releases

Successful software products have a long life. They evolve: each release includes enhancements that are based on feedback and experience. Experience with releases 1 through n can be very helpful in managing the risk of developing release $n + 1$.

Example 2.6 : During the period, 1985-1996, the highly reliable 5ESS Switching System was a flagship AT&T product. It aimed for and achieved 99.999% reliability. Using carefully planned releases, many new features were added, and the underlying technology was refreshed several times. New features were not added lightly. Product managers stayed in close contact with customers and maintained a backlog of features and enhancements.

Historical data for multiple releases is summarized in Fig. 2.8.[13] Each row represents a release. Releases D1-D11 were for the U. S. market; releases I1-I15 were for international markets. Each release took roughly two years. Development of releases overlapped. The histograms in each row correspond to the level of effort on that release at each phase of the project.

Each release used a plan-driven process. The requirements for each individual release were fixed; changes were permitted if they fit the plan. Any changes that could not be accommodated during a release were deferred to the next release. A release might have 3,000 pages of functional requirements, 9,000 pages of high-level design, 40,000 pages of detailed design. The detailed design, coding, and testing phases overlapped significantly.

5ESS was a large project. It had roughly 10 million lines of code, divided into 50 subsystems. At any time, there were roughly 3,000 developers working on the multiple phases of multiple releases.

It takes careful planning to manage such a large effort, □

2.4 Cost of Change Curve

What is the cost of changing requirements? How hard is it to introduce a change during a software project? As we shall see when we discuss iterative processes, such questions have implications for how much to invest in design and architecture at the start of a project.

2.4.1 A Proxy for the Cost of a Change

Since data for the cost of making a change is not readily available, consider the data that is available for the cost of fixing a defect. Making a change and fixing a defect are closely related, since both require an understanding of the code for a system.

The curves in Fig. 2.9 show changes to the cost of fixing a severe defect as a plan-driven project progresses. The cost is lowest during the requirements and design phases. The solid curve is based on a chart published in 1976 by Barry W. Boehm. The underlying data was from three companies: GTE, IBM, and TRW. Boehm added data from smaller software projects in a 2006 version of the chart.[14]

For large projects, the relative cost of fixing a severe defect rises by a factor of 100 between the initial requirements and design phases and the phase where the system has been delivered and put into operation. The cost jumps by a factor of 10 if the fix is during coding and jumps by another factor of 10 if the fix is during operation. The dashed curve for smaller projects is much flatter: the cost of a fix during operation is 7 times the cost a fix during the requirements phase.

For non-severe defects, the ratio may be 2:1, instead of the 100:1 ratio for severe defects.

Other companies reported similar ratios during a workshop in 2002 (the data is for plan-driven processes):[15]

- A 117:1 increase in effort was observed at IBM Rochester. The increase was 13:1 between coding and testing, and then a further 9:1 between testing and operation.
- A 137:1 ratio was observed at Toshiba, for a software factory of 2,600 workers, comparing the time needed to fix a severe defect before and after shipment.

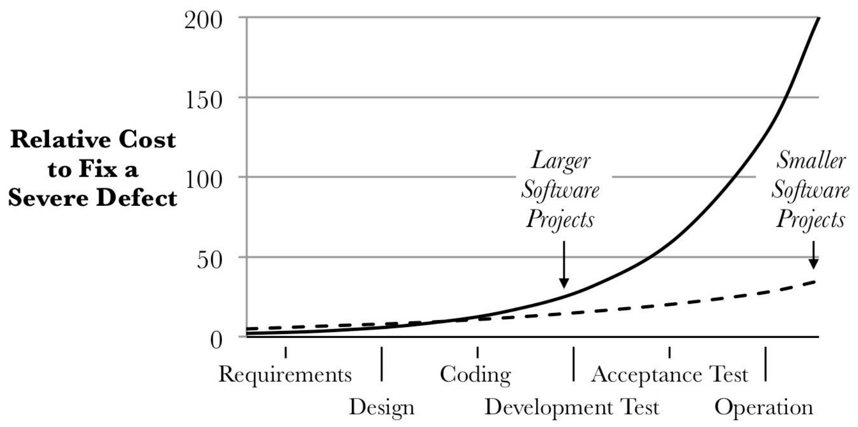


Figure 2.9: For severe defects, the later the fix, the greater the cost. (The original diagram had a log scale for the relative cost).

- Workshop participants from other organizations reported similar experiences.

2.4.2 Implications

The following table summarizes the above discussion comparing the cost of late fixes (during operation, after delivery) to the cost of early fixes (during initial requirements and design):

Project Size	Defect Severity	Ratio
Large	Severe	$\sim 100 : 1$
Small	Severe	$\sim 7 : 1$
Large	Non-Severe	$\sim 2 : 1$

(The symbol "$\sim$" stands for "roughly.")

A high ratio motivates up-front work to catch defects early, thereby avoiding the high cost of fixing the defects during operation. High ratios have been cited as a motivation for plan-driven processes: careful up-front planning, if feasible, can reduce the need for costly late changes. Planning can also allow for anticipated changes. Their cost will likely be comparable to the cost of non-severe fixes.

Agile processes embrace change, on the assumption that the cost of change curve is relatively flat.

2.5 Conclusion

A software development process is a systematic method for meeting the goals of a project by defining and organizing development activities; determining the roles or skills needed to do the project; and setting criteria and deadlines for progressing from one activity to the next.

Plan-driven processes divide software development into phases; e.g., requirements gathering, design, coding, and testing. Phases may have gates or reviews to decide whether to go on to the next phase.

Waterfall processes are a subclass of plan-driven processes, where the phases are sequential; that is, one phase completes before the next phase begins. In the 1970s, waterfall was the dominant process model for software development. In its pure form, it is now held up as an example of what not to do. The risk with pure waterfall processes is twofold. First, while the project is underway, customer requirements will likely have changed. The completed system may therefore fail to meet changed customer expectations. Second, testing occurs at the end of the development cycle, so design and performance issues may not surface until the very end.

V-processes are a variant of waterfall processes, in which test planning and test execution are separated. Test planning is done early, along with the specification and design phases. Test execution has to wait until the system is available for testing; that is, after coding. Each specification phase has a corresponding testing phase.

In practice, waterfall and V-processes may be used along with other methods to manage the risks associated with software development. For example, the team that built the successful SAGE air-defense system began by thoroughly investigating the functions of air defense by building a disposable prototype. Only then did they begin development, using a V-process.

Exercises for Chapter 2

Exercise 2.1 : The process in Fig. 2.10 is a variant of Infosys's development process circa 1996.[16] Compare the process in Fig. 2.10 with each of the following. In each case, discuss the similarities (if any) and differences (if any).

a) Waterfall processes.
b) V-processes.
c) Iterative processes.

Notes for Chapter 2

[1]The opening quote is from McIlroy, Pinson and Tague's foreword to a special issue of the *Bell System Technical Journal* on Unix [12].

[2]"Who will do what by when" is a simpler version of Boehm's [4, p. 76] "Why, What,When, Who, Where, How, How Much."

[3]The grow analogy for iterative processes is due to Fred Brooks [6].

[4]De Bonte and Fletcher [7] developed a Scenario-Focused Engineering workshop that was delivered to over 22,000 Microsoft engineers over a six-year period starting in 2008.

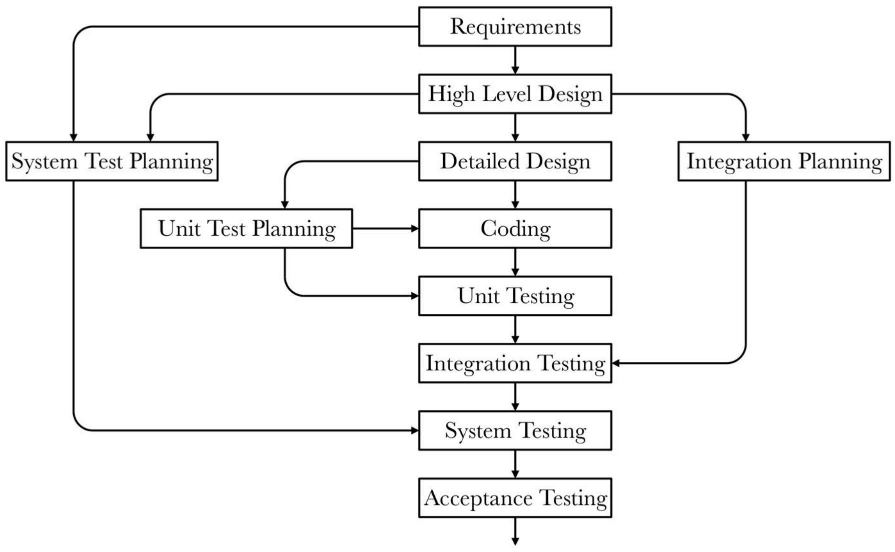


Figure 2.10: A variant of the Infosys development process circa 1996.

[5]Boehm [5] writes, "On my first day on the job, my supervisor showed me the GD ERA 1103 computer, which filled a large room. He said, 'Now listen. We are paying $600 an hour for this computer and $2 an hour for you, and I want you to act accordingly.' "

[6]The waterfall diagram in Fig. 2.5 is adapted from an influential 1970 paper by Winston W. Royce [13]. The first published account of a waterfall process, however, is a 1956 paper by Benington [1] on the development process for the U.S. SAGE air defense system; see Fig. 2.7.

[7]Madden and Rone [11]

[8]U.S. government contracts promoted the use of waterfall processes through standards such as Military Standard MIL-STD-1521B dated June 4, 1985 [16]. A much quoted study of 1995 Department of Defense (DoD) software spending concluded that 75% of the $35.7B worth of software was either never used or never delivered [9].

[9]The committee chairman's opening remarks are from [17, p. 2]. For the extent of system and end-to-end testing, see [17, p. 57]. The cost of the website is from [18, p. 19].

[10]Checklists of top software risks appear in Boehm [3] and Keil at al. [10], among others. The 1998 study by Keil et al. convened independent panels of project managers in Finland, Hong Kong, and the United States.

[11]Herbert Benington's 1956 paper [1] was reprinted in 1983 with a fresh Foreword by the author, in which he noted, "I do not mention it in the [1956] paper, but we undertook the programming only after we had assembled an experimental prototype."

[12]The V-shaped process diagram for SAGE in Fig. 2.7 is a redrawing of the original diagram [1]. Paraphrasing Benington [1], a programmer must prove that the program satisfies the specifications, not that the program will perform as coded. Furthermore, "test specifications ... can be prepared in parallel with coding."

[13]Fig. 2.8 is from Siy and Perry [15]. Used courtesy of Harvey P. Siy.

[14]The chart on the cost of fixing a defect is due to Boehm [5, 2].

[15]Shul et al. [14].

[16]Jalote [8, p. 37] describes the Infosys development process. Infosys is known for its highly mature processes; specifically, at the time, Infosys was a CMM level 5 company.

References for Chapter 2

1. Herbert D. Benington. Production of large computer programs. *Proceedings, Symposium on Advanced Programming Methods for Digital Computers*, Office of Naval Research Symposium (June 1956). Reprinted with a Foreword by the author in *Annals of the History of Computing* 5, 4 (October 1983) 350-361.

2. Barry W. Boehm. Software engineering. *IEEE Transactions on Computers* C-25, 12 (December 1976) 1226-1241.

3. Barry W. Boehm. Software risk management: principles and practices. *IEEE Software* (January 1991) 32-41.

4. Barry W. Boehm. Anchoring the software process. *IEEE Software* (July 1996) 73-82.

5. Barry W. Boehm. A view of 20th and 21st century software engineering. *Proceedings International Conference on Software Engineering (ICSE ?06)* (2006) 12-29.

6. Frederick P. Brooks, Jr. No silver bullet—essence and accident in software engineering. *Proceedings of the IFIP Tenth World Computing Conference*. Elsevier, Amsterdam (1986) 1069-1076. Reprinted in *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* by Fred Brooks, Addison-Wesley (1995) 177-203.

7. Austina De Bonte and Drew Fletcher. *Scenario-Focused Engineering.* Microsoft Press, Redmond, Wash. (2013).

8. Pankaj Jalote. *Software Project Management in Practice.* Addison-Wesley, Boston, Mass. (2002).

9. Stanley J. Jarzombek. *Proceedings, 5th Annual Joint Aerospace Weapons Systems Support, Sensors, and Simulation Symposium (JAWS S3).* U.S. Government Printing Office (1999). There are many references to this elusive report. See for example, Theron R. Leishman and David A. Cook, Requirements risks can drown software projects, *Crosstalk, The Journal of Defense Software Engineering* (April 2002).

10. Mark Keil, Paul E. Cule, Kalle Lyytinen, and Roy C. Schmidt. A framework for identifying software project risks. *Comm. ACM* 41, 11 (November 1998) 76-83.

11. William A. Madden and Kyle Y. Rone. Design, development, integration: Space Shuttle primary flight software system. *Comm. ACM* 27, 9 (1984) 914-925.

12. M. D. McIlroy, E. N. Pinson, and B. A. Tague. Foreword: Unix time-sharing system. *Bell System Technical Journal* 57, 6 (July-August 1978) 1899-1904.

13. Winston W. Royce. Managing the development of large software systems. *Proceedings IEEE WESCON* (August 1970).

14. Forrest Shul, Vic Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What have we learned about fighting defects. *Proceedings Eighth IEEE Symposium on Software Metrics (METRICS '02)* (2002) 249-258.

15. Harvey P. Siy and Dewayne Perry. Challenges in evolving a large scale software product. *International Workshop on Principles of Software Evolution* (April 1998).

16. U.S. Department of Defense. *Military Standard: Technical Reviews and Audits for Systems, Equipments, and Computer Software* MIL-STD-1521B (June 4, 1985). `http://www.dtic.mil/dtic/tr/fulltext/u2/a285777.pdf`

17. U.S. House of Representatives. PPACA implementation failures: answers from HHS? Hearing before the Committee on Energy and Commerce, 113th Congress, *Serial No.* 113-87 (October 24, 2013).

18. U.S. House of Representatives. PPACA implementation failures: didn't know or didn't disclose? Hearing before the Committee on Energy and Commerce, 113th Congress, *Serial No.* 113-90 (October 30, 2013).

Chapter 3

Iterative and Agile Processes

"The advice to XP teams is not to minimize design investment over the short run, but to keep the design investment in proportion to the needs of the system so far. The question is not whether or not to design, the question is when to design."

— *Kent Beck, who introduced Extreme Programming (XP).*[1]

From Section 2.1, an *iterative process* has a sequence of steps or iterations that produce increasingly functional versions of a system. Customer feedback at the end of each iteration helps to keep the project on track. The feedback allows changes in customer needs and expectations to be accommodated during later iterations.

This chapter begins with two case studies that illustrate the benefits of iterative development. The rest of the chapter concentrates on agile processes. The term "agile" applies to several classes or processes—classes of processes are also called *process models*. Adoption surveys show that Scrum is by far the most widely used agile model. A hybrid combination of Scrum and Extreme Programming (XP) is a distant second.[2] Section 3.4 covers Scrum and Section 3.5 covers XP. Section 3.6 covers user stories, which are closely associated with XP, but are used together with other agile models as well.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	*over*	processes and tools
Working software	*over*	comprehensive documentation
Customer collaboration	*over*	contract negotiation
Responding to change	*over*	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Figure 3.1: The Agile Manifesto.

3.1 Introduction

The roots of iterative and agile processes go back to the "plan-do-study-act" quality improvement cycles proposed in the 1930s by Walter Shewhart at Bell Labs. They were applied to software projects in the mid 1950s—they came through NASA to IBM, a federal contractor.[3]

Since iterative processes deliver functionality in increments, they are sometimes called *iterative and incremental*.

Iterative and incremental delivery "within weeks" is a hallmark of the Unix style of program development. In 1978, McIlroy et al. described the Unix programming style as follows:[4]

- "Make each program do one thing well. To do a new job, build fresh rather than complicate old programs by adding new 'features.'"
- "Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them."
- Software utilities "were continually improved by much trial, error, discussion, and redesign."

3.1.1 The Agile Manifesto

In February 2001, a group of self-described "independent thinkers about software development" met to explore "an alternative to document driven, heavyweight processes." The group included proponents of a range of process models including Extreme Programming (XP) and Scrum. They found common ground in the Agile Manifesto in Fig. 3.1.[5]

We use the term *agile method* or *agile process* to refer to any software development process that complies with the values in the Agile Manifesto. Based on the Manifesto and the principles that accompanied it, an agile process emphasizes

- satisfying customers through collaboration,
- delivering working software frequently (in weeks, not months),
- accommodating changes during development, and
- valuing simplicity and technical excellence.

The distinction between the agile and iterative process models is more about values and culture than it is about specific techniques and practices. Both models deliver functionality incrementally, although agile processes tend to have much shorter iterations. Both involve customer feedback at the end of each iteration. The agile model is therefore an evolutionary successor of the iterative model.

Proponents of agile methods speak of "the agile movement" and emphasize respect and collaboration in the work environment.[6] Teams self organize, developers take responsibility, and communication is ideally face-to-face and informal.

Software development is a people business. Culture, values, and a supportive work environment can make a big difference in the productivity and retention of team members.

3.2 Iterative Processes

Customer feedback at the end of each iteration allows iterative (and agile) processes to

- handle uncertain or dynamically changing requirements;
- improve design and quality as early users uncover issues; and
- enable parallel development of a software subsystem and the components that depend on it.

The first case study in this section describes the use of quick iterations to handle changing requirements. Customer feedback helped improve design and quality along the way. The second case study describes the use of iterations for the parallel development the software,the hardware, and the training modules of a system.

3.2.1 Uncertain and Dynamic Requirements

Uncertainty arises because stakeholders may not know what they need, or because the development team may not have fully grasped what stakeholders want. Dynamic changes are often the result of new information, such as a product announcement by a competitor.

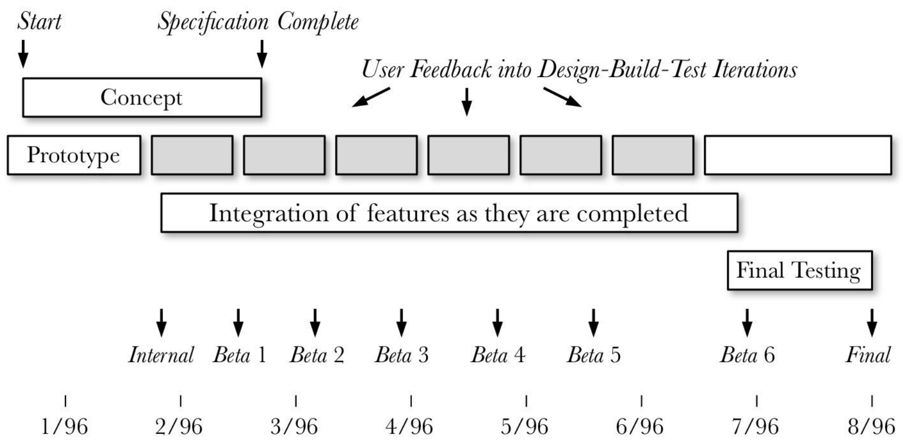


Figure 3.2: A stylized version of the iterative process for the Netscape Navigator 3.0 web browser project.

Uncertainty refers to a known unknown: we know there is a requirement, but we have yet to converge on exactly what it is. Uncertainty can be anticipated when proposing a solution or design.

Dynamic change refers to an unknown unknown: we don't know anything about the change or even whether there will be a change. Dynamic changes are unexpected and can lead to a redesign.

Example 3.1 : In the early days of the World-Wide Web, Netscape Navigator was the dominant web browser, with 70% market share. Microsoft appeared to have missed the Internet "Tidal Wave" until it unveiled its Internet strategy on December 7, 1995. It compared itself to a sleeping giant that had been awakened, and launched an all-out effort to build a competing browser, Internet Explorer 3.0.[7]

In the race with Microsoft, time to market was paramount for the Navigator 3.0 project. The development process emphasized quick iterations; see Fig. 3.2, where the gray boxes represent iterations. There were six beta releases between the start of the project in January 1996 and the final release of the product in August.

The project had both uncertain and dynamically changing requirements.

- *Uncertain*. The initial requirements were based on extensive interactions with customers; however, there was uncertainty about whether customers would like the designs and usability of the features.[8]
- *Dynamic*. The team carefully monitored beta releases of Microsoft's Explorer 3.0, ready to change requirements dynamically to keep Navigator 3.0 competitive with Explorer 3.0.[9]

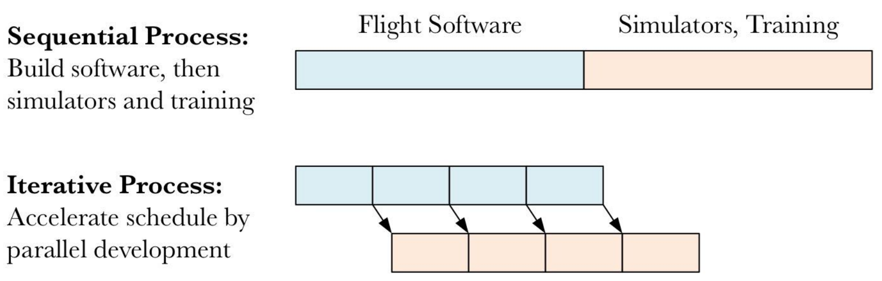

Sequential Process:
Build software, then simulators and training
Flight Software
Simulators, Training
Iterative Process:
Accelerate schedule by parallel development

Figure 3.3: Schedule acceleration through parallel development.

The Navigator 3.0 project began with a prototype that was quickly released internally within the company as Beta 0. The prototype was followed by quick design-build-test iterations, each iteration leading to an external beta release, available for public download. The beta releases of working software elicited valuable user feedback that guided the content of the next beta version.

Navigator 3.0 and Explorer 3.0 both hit the market in August 1996. □

3.2.2 Parallel Development of Subsystems

The NASA Space Shuttle project used an iterative process to accelerate the development of the orbiter hardware, the flight software, and simulators and training modules. Since the software tracked the hardware, and the training tracked the software, a sequential process would have imposed the following ordering (see Fig. 3.3):

1. Develop the shuttle orbiter vehicle.
2. Develop the software for guidance, navigation, and flight control of the orbiter.
3. Develop the simulators and training that would use the software.

The team at IBM, the software contractor, used an iterative process:[10]

> "This approach was based on incremental releases. ... The first drop for each release represented a basic set of operational capabilities and provided a structure for adding other capabilities on later drops."

Early drops of working software allowed development of the simulators to start before all the software was completed (again, see Fig. 3.3). Early releases allowed more extensive verification of the software through testing and use on simulators. (The flight software and the orbiter hardware were also developed in parallel.)

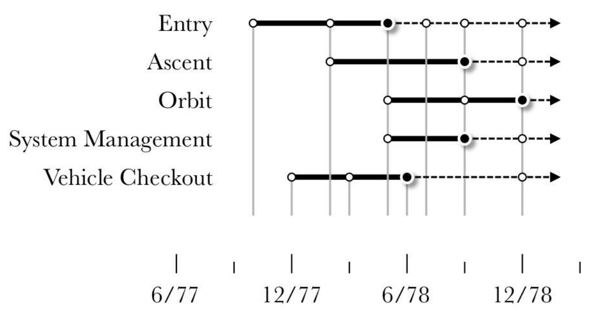


Figure 3.4: The first 9 of 17 releases of the flight software for the space shuttle.. Rows represent subsystems and vertical lines represent releases. Circles represent code drops.

Example 3.2 : The first 9 of 17 software releases for the shuttle are illustrated in Fig. 3.4. The rows are for the subsystems, Entry, Ascent, Orbit, System Management, and Vehicle Checkout. Ascent was for launch from Earth. Entry was for re-entry into the Earth's atmosphere and landing. The black bars represent the period between the first code drop and the completion of a subsystem.

The first 8 releases included code for one or more subsystems. Circles along a bar denote inclusion of the subsystem in a given release. The filled circles denote completion of the functionality for that subsystem. Code drops after the completion point were to incorporate requirements changes.

The ninth and subsequent releases were for the flight software system as a whole. In other words, each row of Fig. 3.4 represents an iterative subprocess for a subsystem. From the ninth release onward, the subsystems were integrated and an iterative process used for the overall software system.

Parallel development meant that the software requirements changed as the orbiter changed. Another reason for changes was that, early in development, it became clear that the software system would exceed the memory and processing capacity of the on-board computer. The requirements could not be met as stated at the time. So, some functions were deleted and the performance requirements relaxed, resulting in changes to the software architecture and the detailed design of the system.

NASA and its contractors made almost 2,000 software requirements changes between 1975 and the first flight of the shuttle on April 12, 1981. □

3.3 Enabling Practices for Iterative Processes

Iterative processes reduce the risk of changing requirements, but they do not eliminate it. This section begins with a troubled project. It then describes best practices from a study of successful iterative projects.

3.3.1 A Troubled Iterative Project

Netscape successfully used an iterative process for Navigator 3.0 (see Example 3.1), but the very next project, Communicator 4.0, using the same process, produced a poor quality product.

Example 3.3 : The Communicator 4.0 project faced multiple risks.

Customers. With 4.0, Netscape shifted its strategy from focusing on individual consumers to selling to enterprises—enterprises include businesses, non-profits, and government organizations. The senior engineering executive later described the shift as "a complete right turn to become an airtight software company." Enterprises have much higher expectations for product quality.[11]

The team took on new features, including email and groupware. Well into the 4.0 project, the requirements for the mailer changed dynamically: the company changed the competitive benchmark. As the engineering executive put it, "Now that's an entire shift!"

The groupware features were unproven and were not embraced by customers. Three quarters of the way through the project, a major new feature was added.

Communicator 4.0 "was built on the old code base, which was beginning to run out of steam."

Technology. The project had technology issues related to both the code for the system and with the tools to build the system. Communicator 4.0 was built on the existing code base from Navigator 3.0. The existing code base needed to be re-architected to accept the new features, but the schedule did not permit a redesign.

With respect to tools, the team chose to use Java, so the same Java code would run on Windows, Mac OS, and Unix. Java was relatively new at the time and did not provide the desired product performance.

Team. The team faced skills shortages. With multiple platforms to support (Windows, Mac, Unix), the team did not have enough testers.

When Communicator 4.0 was released in June 1997, it faced quality problems. The iterative development process, with early customer feedback, addressed some of the risk related to requirements, but there were other risks that were not addressed, related to the existing code base, the use of Java, and insufficient testing. □

3.3.2 Success Factors for Iterative Processes

Why do some projects succeed, while others fail? What are the best practices for iterative processes?

Some answers to these questions can be found in a study of 29 completed projects from 17 companies. The conclusions from the study carry over to agile processes, although the study itself was about the encompassing class of iterative processes. Based on interviews, surveys, and expert panels between 1996 and 1998, the study measured (1) *product quality*, defined as a combination of performance, functionality, and reliability; and (2) *team productivity*.[12]

The rest of this section discusses the four practices that were found to be associated with successful projects.

Earlier Customer Feedback

Feedback on less complete versions allows projects to prioritize features that customers find useful and to benefit from suggestions about new or missing features. Projects that had as little as 30%-40% of their final functionality at first beta had better quality than products that had 70% or more of their final functionality.

Of course, earlier first betas need to have enough of the core functionality that customers take them seriously.

Rapid Trouble Reports

Projects with overnight or daily trouble reports had higher product quality than projects where the trouble reports took 30 or more hours. Half of the projects in the sample used daily builds to incorporate new code and integrate all of the components into a working version.

Agile methods go further: they combine coding with automated unit testing and continuous integration—see Section 3.5. Automated tests verify that new code does not break any existing functionality. Continuous integration builds a working version several times a day.

Flexible System Architecture

Projects with major investments in system architecture had higher quality.

Architectures that are modular and loosely coupled are more likely to be flexible than architectures that are optimized for a specific purpose, such as performance. Iterative processes must balance flexibility and performance.

See also Section 3.7 for a discussion of up-front flexible design versus just-in-time incremental design.

Developer Experience on Diverse Projects

The traditional measure of experience is years of service. Service experience by itself did not lead to either higher product quality or higher team productivity. However, diversity of experience did correlate with higher productivity. Here diversity of experience means experience on diverse completed projects, where the system was not simply a derivative of a previous system.

A broad team, where the developers have seen diverse projects through to completion, is better positioned to anticipate and respond to changes due to requirements and technology.

3.4 The Scrum Framework

Scrum is a class of agile software development processes that dates back to the early 1990s.[13] Its originators, Jeff Sutherland and Ken Schwaber, later became signatories of the Agile Manifesto. Scrum is a class of processes rather than a specific process, since scrum teams are free to choose how they organize themselves and do their work.

Scrum is characterized by rules for three elements: events, roles, and artifacts. The relationships between these elements are illustrated in Fig. 3.5. This section explores the following elements of the Scrum framework:

- *Scrum Team.* A *scrum team* consists of a product owner, a development team, and a scrum master.
- *Sprints.* Iterations called *sprints* are at the heart of Scrum.
- *Product Backlog.* The product owner maintains a *product backlog*, consisting of a prioritized list of items to be implemented.
- *Sprint Backlog.* For each sprint, the team picks items from the product backlog. The picked items form the *sprint backlog* to be implemented during the sprint.
- *Increments.* As with any iterative process, the deliverable from a sprint is an *increment* of relevant functionality. The increments grow a working system that delivers more and more of the functionality that stakeholders care about.

3.4.1 Scrum Roles

Scrum defines three roles: product owner, developer, and scrum master.

Product Owner

The product owner is responsible for the content of the product. This owner must be a person; ownership is not to be spread across team members. In all team meetings, the product owner serves as the voice of the customer, and sets priorities for what the team implements.

Development Team

The development team has sole responsibility for implementation. They come up with estimates for the time and effort needed to implement the items in the product backlog.

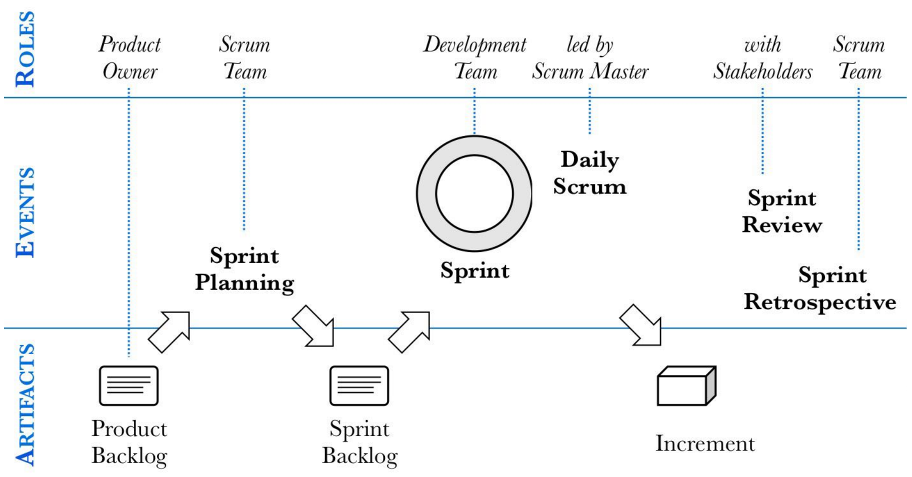


Figure 3.5: Elements of Scrum.

Scrum Master

The scrum master acts as coach, facilitator, and moderator. The scrum master is responsible for arranging all meetings, keeping them focused, and time boxed. The scrum master does not tell people how to do their jobs. Instead, for each activity, he or she highlights its purpose and its rules of order.

The scrum master also takes responsibility for removing any external impediments for the team.

3.4.2 Scrum Events

Scrum defines five kinds of time-boxed events: sprints, sprint planning, daily scrums, sprint reviews, and sprint retrospectives.

Sprints

Sprints last one month or less. The purpose of a sprint is to add an increment of functionality to a working system. By working system is meant that the system is in a state where it could potentially be released to stakeholders.

During a sprint, there are no changes to the functionality to be delivered by the sprint (the sprint goal). Developers may clarify or renegotiate the sprint goal, but the intent is to have short sprints with fixed goals.

Sprint Planning

Sprints begin with a planning meeting that lasts 8 hours or less for a one-month sprint. The purpose of sprint planning is to select items to be implemented during the sprint. The items, selected from the product backlog, constitute the sprint backlog.

The selection of items for the sprint backlog is guided by two things: (1) the product owner's description of customer priorities and (2) the development team's forecasts of what they can implement during the sprint. They are free to choose the implementation; however, they must be able to articulate how they intend to implement the sprint backlog.

Daily Scrum

The name Scrum comes from the sport of rugby, where a scrum is called to regroup and restart play.

During a sprint, a *daily scrum* is a short 15-minute meeting to keep the project on track. The purpose of a daily scrum is to regroup and restart the sprint. Daily scrums are led by the scrum master who ensures that the developers stay focused on addressing the following three questions:

- What did I do yesterday toward the sprint's goal?
- What will I do today?
- Are there any impediments to progress?

These questions keep the whole development team informed about the current status and work that remains to be done in the sprint.

The 15-minute time limit works for small teams. The process can be scaled to larger teams by grouping the the teams into smaller subteams responsible for subsystems. The daily scrum for the larger team is then a *scrum of scrums*, with representatives from the subteams. The representatives are typically the scrum masters of the subteams.

Sprint Review

A *sprint review* is an at most 4-hour meeting at the end of a one-month sprint. The purpose of the review is to close the current sprint and prepare for the next. The review includes stakeholders and the whole scrum team.

Closing the current sprint includes a discussion of what was accomplished, what went well, and what did not go well during the sprint. The development team describes the implemented increment and possibly gives a demo of the new functionality.

Preparing for the next sprint is like starting a new project, building on the current working software. The product owner updates the product backlog based on the current understanding of customer needs, schedule, budget, and progress by the development team. The group revisits the priorities for the project and explores what to do next.

A sprint review sets the stage for the planning meeting for the next sprint.

Sprint Retrospective

A *sprint retrospective* is an at most 3-hour meeting for a one-month sprint. In the spirit of continuous improvement, the purpose of the retrospective is for the scrum team to reflect on the current sprint and identify improvements that can be put in place for the next sprint. The improvements may relate to the system under development, the tools used by the team, the workings of the team within the rules of Scrum, or the interactions between the team members.

3.4.3 Scrum Artifacts

Scrum deals with three kinds of artifacts: the product backlog, the sprint backlog, and the increment. The artifacts are visible to all team members to ensure that everyone has access to all available information.

Product Backlog

The product backlog is a prioritized list of items that evolves as the project proceeds. At any time, the product backlog reflects the current understanding of the scope and quality attributes of the product, based on stakeholder needs, market conditions, and the functionality that has already been implemented.

Sprint Backlog

The sprint backlog is created during sprint planning. It includes items from the product backlog that the team can implement during the current sprint. The items are selected to maximize the value provided by the current increment. The sprint backlog remains fixed for the duration of the sprint. Any changes due to changing requirements are incorporated in the next sprint.

Increment

The increment is the functionality that will be added during the current sprint. If we think of the product as a sequence of increasingly functional releases, the increment is the difference between the previous and the current release.

3.5 XP: No Longer Extreme

Extreme Programming (XP) may have seemed extreme when Kent Beck introduced it around 1996, but its basic premise has long since become mainstream:

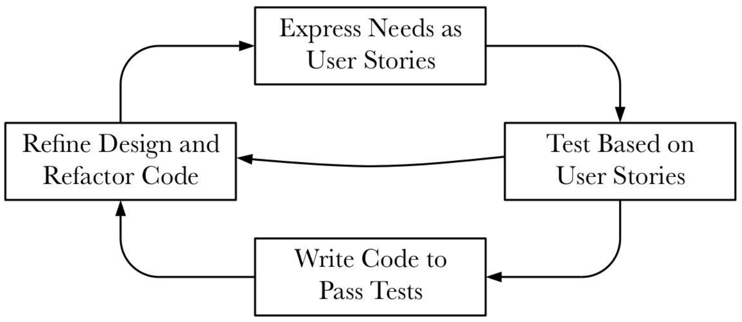

Express Needs as User Stories
Test Based on User Stories
Write Code to Pass Tests
Refine Design and Refactor Code

Figure 3.6: Software development using XP.

> "an always-deployable system to which features, chosen by the customer, are added and automatically tested on a fixed heartbeat."

Beck positioned it as being "about social change" and a "philosophy of software development."[14]

This section provides brief overviews of the key practices of XP. The outer loop in Fig. 3.6 illustrates how the practices fit together during an iteration. Starting at the top, an iteration proceeds as follows: review customer needs and identify what to build; define tests before coding; write just enough code and integrate it into a working system; and then clean up the design and code so that it is ready for the next iteration.

This section is organized around the four values in the Agile Manifesto: customer collaboration; responding to change; working software; and individuals and interactions.

3.5.1 Customer Collaboration: User Stories

At the start of a project and after each iteration, the development team engages users and other stakeholders to review their needs and wants. This engagement results in a set of stories that embody the project's goals and requirements. Such stories have three aspects:

- *User Stories.* A user story is a brief description of a feature or a piece of functionality that a stakeholder wants in a system.
- *Acceptance Tests.* Each user story is accompanied by one or more acceptance tests to validate the implementation of the story.
- *Estimates.* A rough estimate of the development effort for a user story is essential for cost-benefit tradeoffs across stories.

A software system of any size may have dozens or perhaps hundreds of user stories. Stories are written in simple language and are expected to be short

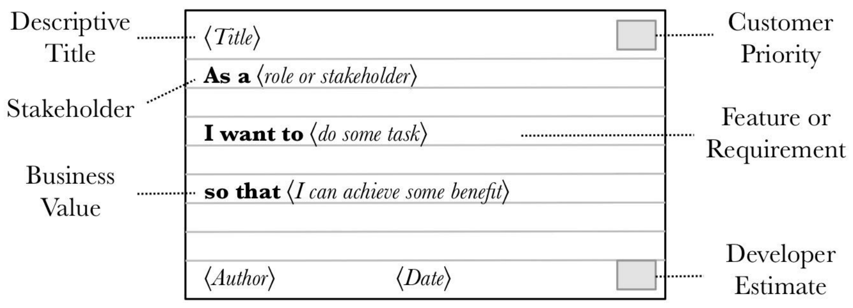


Figure 3.7: A rendering of a user story template created by Connextra in 2001.

enough to fit on an index card. Their purpose is to stimulate and facilitate conversations about the features they represent. A story is meant to identify a customer need, not to capture the details of the need. These properties of user stories are highlighted by the acronym *3C*, for Card, Conversation, and Confirmation: fit on a card; spark a conversation; confirm understanding through an executable acceptance test.[15]

A template for writing user stories appears in Fig. 3.7.[16] The three main elements of the template are (1) the role or stakeholder who wants the story to work; (2) the feature or functionality to be implemented; and (3) the business value of the feature to the stakeholder.

XP was motivated by a payroll system for Chrysler, so here is a payroll example:

As a	payroll manager
I want to	print a simple paycheck
so that	the company can pay an employee

Stories get clarified and sharpened during conversations between stakeholders and developers.

See Section 3.6 for further discussion of user stories.

3.5.2 Responding to Change: Iteration Planning

The collaboration between stakeholders and developers extends beyond reviewing needs and revising the backlog of user stories to be implemented. The collaboration includes prioritizing the backlog and planning what gets implemented in the next iteration.

Agile iterations are *time boxed*, which means that the time interval of an iteration is fixed. A time box constrains what the development team can accomplish during an iteration. Any functionality that does fit into a time-boxed iteration is either dropped or added to the backlog of functionality to be addressed during a future iteration.

Prioritization of stories for an iteration is done by the stakeholder or the customer representative. Prioritization is based on a rough cost-benefit analysis, where the cost is the developers' estimate of the implementation effort needed for a story and the benefit is the value of the story to the stakeholder.

A team's *velocity* is the sum of the estimates of the stories that a team can implement during an iteration. Velocity determines how many of the highest priority stories are selected for the current iteration. Velocity can also be used to revise estimates as a project proceeds.

Planning of iterations and projects is discussed in Section 3.6.

3.5.3 Working Software: The Role of Testing

With agile processes, the ideal is to have simple working software all the time, not just at the end of an iteration. Three forms of testing play an essential role in maintaining a state of clean working software: automated tests; continuous integration; and test-driven development.

Automated Regression Testing

. With any change to a system, there is a risk that the change will break something that was working. The risk of breakage can be significantly reduced by ensuring that the system continues to pass all tests that used to work. This process of running all tests is called *regression testing.*

The burden of regression testing can be significantly reduced by automating it. Automated regression testing provides a safety net while developers are making changes. The more complete the regression tests, the greater the safety net provided by automated tests.

Continuous Integration

. The goal of working software applies to the overall system, not just to the components. Continuous integration means that overall system integration is done several times a day. Integration every couple of hours is more frequent than the daily builds that were cited as an enabling practice for iterative processes in Section 3.2.

System integration includes a complete set of system tests. If a change causes system tests to fail, then the change is rolled back and reassessed. The complete set of tests must pass before proceeding.

Test-Driven Development

With test-driven development, each new feature, each new piece of functionality, begins with a test of what the feature should do. The test should fail, since the code has not yet been written. Furthermore, the test should fail for the expected reason. If, however, the test passes, then either the feature already exists, or the test is inadequate.

In addition to acceptance tests, developers may write tests that are relevant to the proposed implementation.

Once the tests are written, the idea is to write just enough code so the software passes all tests. Regression testing acts as a safety net during test-driven development, since it runs all tests to verify that the new code did not break some existing feature.

3.5.4 Working Software: Refactoring

Refactoring consists of a sequence of correctness-preserving changes to clean up the code. Correctness-preserving means that the external behavior of the system stays the same. Each change is typically small. After each change, all tests are run to verify the external behavior of the system.

Without refactoring, the design and code of the system can drift with the accumulation of incremental changes. The drift can result in a system that is hard to change, which undermines the goal of working software.

3.5.5 Individuals and Interactions

As Kent Beck himself noted, "The individual practices in XP are not by any means new." What distinguishes XP is its social aspect, its emphasis on community and values and trust and teamwork. In Beck's words, XP includes a "philosophy of software development based on the values of communication, feedback, simplicity, courage and respect."[17]

The philosophy of XP is beyond the scope of this section. For now, we concentrate on one of the social aspects of XP that has been perhaps the most controversial: pair programming.

Pair programming is the practice of two developers working together on one task, typically at the same machine. The claimed benefit of pair programming is that it produces better software more effectively. Two people can share ideas, discuss design alternatives, review each others' work, and keep each other on task. As a side benefit, two people know the code, which helps spread knowledge within the team. So, if one person leaves or is not available, there is likely someone else on the team who knows the code.

One of the concerns with pair programming is that having two people working on the same task could potentially double the cost of development. An early study with undergraduate students concluded that pair programming added 15% to the cost, not 100%. Further, the 15% added cost was balanced by 15% fewer defects in the code produced by a pair.[18]

A decade later the controversy remained:

> "We are no longer in the first flush of pair programming, yet the gulf between enthusiasts and critics seems as wide as ever."[19]

3.6 User Stories

The output of the framing or problem-definition step is a set of requirements for a solution. With agile processes, requirements are embodied in user stories and acceptance tests. Other forms of requirements include scenarios (Section 4.4) and use cases (Chapter 5).

A user story is a brief descriptions of a feature or a piece of functionality. The following template for writing user stories was introduced in Section 3.5:

As a	⟨*role or stakeholder*⟩
I want to	⟨*do some task*⟩
so that	⟨*I can achieve a benefit*⟩

This template identifies not only a want or a need for a feature, it also captures the benefit or business value of the feature for some stakeholder.

3.6.1 SMART User Stories

Stories benefit both users and developers. They need to be written collaboratively so users can relate to them and developers can act on them. The first attempt to write a user story may not meet both these objectives.

Example 3.4 : Here is a first draft of a user story for a payroll example:

As a	payroll manager
I want to	print a simple paycheck
so that	the company can pay an employee

In this story, what does "simple paycheck" mean? Paychecks can be very complicated, between various forms of compensation and various deductions for taxes, savings, medical insurance, ...

The following refinement of the payroll story is specific about compensation and taxes:

As a	payroll manager
I want to	print a paycheck that accounts for monthly wages and federal taxes
so that	the company can pay an employee and withhold federal taxes

While this story can be refined further, it may specific enough to facilitate meaningful conversations between customers and developers. □

Good user stories are *SMART*: Specific, Measurable, Achievable, Relevant, and Time-bound.

- A story is *specific* if a developer can implement it.
- It is *measurable* if acceptance tests can be defined for it.

- It is *achievable* if the development team knows how to implement it.
- It is *relevant* if the feature in the story contributes to the goal in the story, and the goal provides business value for the stakeholder.
- It is *time bound* if it can be implemented in one iteration.

If one of these criteria is not met, then the story needs to be refined until it is SMART. For more on SMART criteria, see Section 7.1.2.

3.6.2 Clarifying the Benefit

The template for user stories contains both a feature (the want) and the benefit that drives the want.

Clear benefits help to keep a project on track. If the feature in a user story does not contribute to the stated benefit, then either the feature is not relevant, or there may be an unstated goal that needs to be clarified. Features that are not relevant can be dropped and unstated goals can be made explicit. A mismatch between features and benefits may be a sign that there are missing stories.

Clear benefits can also help avoid *gold plating*, which refers to continuing to work on a project beyond the point of meeting all the stakeholder needs.

When clarifying benefits, ask "Why?" Why does a stakeholder want a feature? "Why" questions tend to elicit cause and relevance, as we shall see in Chapter 7. Respectfully continuing to ask "Why?" questions can surface an underlying need that has higher priority for the user than the starting point. The "Why" questions may be posed explicitly or they may be something to keep in mind as you observe or interact with a stakeholder.

3.6.3 Acceptance Tests

User stories are accompanied by acceptance tests for verifying that a story has been implemented correctly. Acceptance tests are part of the conversation about stories between customers and developers.

The following template has been found useful for writing acceptance tests:[20]

Given ⟨*a precondition*⟩
when ⟨*an event occurs*⟩
then ⟨*ensure some outcome*⟩

The payroll user story might be accompanied by a test of the form:

Given an employee is on the payroll and is single
when the paycheck is printed
then use the federal tax tables for singles to compute the tax

Acceptance tests are written using language that is meaningful to stakeholders. At the same time, the wording needs to be specific enough for a developer to implement the test. Acceptance tests focus on what the outcome is, not

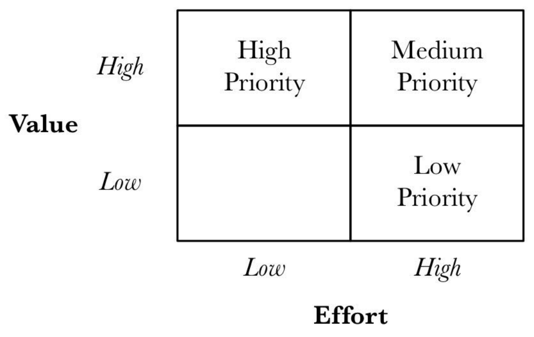


Figure 3.8: Prioritizing nice-to-have user stories.

on how the outcome is implemented. For example, when the payroll story is implemented, it is up to the developer to decide how to access the tax tables and compute the tax. It is best not to build the tax tables into the story or into an acceptance scenario, since tax tables change from year to year.

3.6.4 Prioritizing User Stories

With an iterative process, customers (or their representatives) participate in iteration planning. Customers decide on the value and priority of a story, balancing benefit and cost. Developers provide feedback in the form of the estimated cost or effort needed to implement a story. The highest priority stories are selected for implementation during the next iteration.

Fortunately, rough estimates suffice for assigning high, medium, or low priority to stories. Some stories are "must haves;" for example, security-related or performance related items may be required, even if stakeholders do not specifically ask for them.

The remaining items can be prioritized, as in Fig. 3.8: high priority for high-value low-effort items; medium priority for high-value high-effort items; low priority for low-value high-effort items; leaving low-value low-priority items to be dropped or retained at the discretion of the customer.

3.6.5 Rough Estimates: Agile Story Points

Quick estimates can be made by assigning *points* to a user story, where a point is a unit of work. Points are relative: a 1 point story is simpler than a 2 point story, and so on. Point-based estimation techniques rely on the judgment of developers. Based on their experience and intuition, the developers agree on how to assign points to a story. Point systems are specific to a team; a different team might assign points differently.

Point Values 1, 2, 3

The simplest point system is a system with point values 1, 2, and 3, corresponding to easy, medium, and hard. For example, the developers might assign points as follows:

- 1 *point* for a story that the team knows how to do and could do quickly (where the team defines quickly).
- 2 *points* for a story that the team knows how to do, but the implementation of the story would take some work.
- 3 *points* for a story that the team would need to figure out how to implement.

Hard or 3-point stories are candidates for splitting into simpler stories.

Fibonacci Story Points

With experience, as the team gets better at assigning points to stories, they can go beyond an easy-medium-hard or three-value scale. A Fibonacci scale uses the point values $1, 2, 3, 5, 8, \ldots$ The reason for a Fibonacci, rather than a linear $1, 2, 3, 4, 5, \ldots$ scale is that points are rough estimates and it is easier to assign points if there are some gaps in the scale.

Velocity

With any point scale, a team's *velocity* is the number of points of work it can complete in an iteration.

Example 3.5 : In Fig. 3.9, the team's estimated velocity was 17, but it only completed 12 points worth of stories in an iteration. For the next iteration, the team can adjust its estimated velocity based on its actual velocity from recent iterations. One possible approach is to use the average velocity for the past few, say 3, iterations. □

3.6.6 Limitations of User Stories

Big Picture?

As the number of stories increases, it is easy to lose sight of the "big picture" of a system, which can be helpful during system design. User stories are detailed; they are at the level of individual features. Use cases, considered in Chapter 5, do provide context. As we shall see, user stories and use cases can be used together.

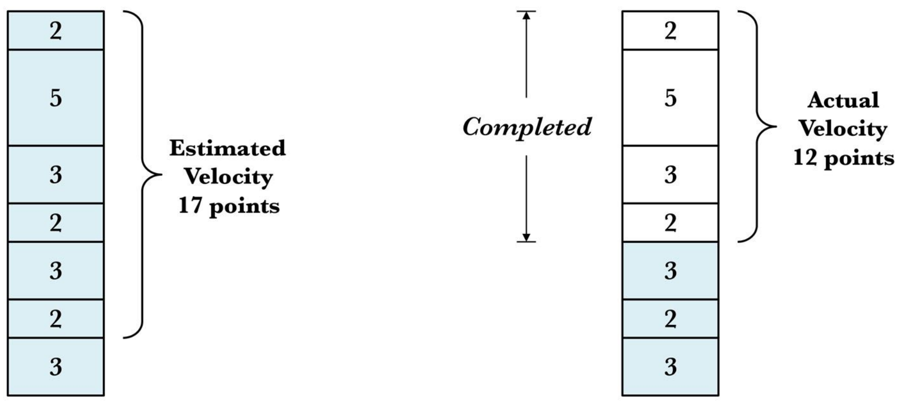


Figure 3.9: Estimated and actual velocity for an iteration. The team planned 17 points worth of work, but completed only 12 points worth.

Completeness?

Completeness is another issue with user stories: there is no guarantee that a collection of user stories describes all aspects of a system.

Customers may have left out something they that take for granted, something that "everybody knows" in their environment. Furthermore, non-functional requirements, such as performance and privacy, may not have been discussed in the conversation between customers and developers.

Deeper Needs?

Conversations, interviews, and surveys are well suited to identifying expressed needs. The "**I want to**" phrasing in the user story template is also suited to expressed needs.

A skilled development team may uncover additional needs through observations and empathy; however, observations and empathy are not an explicit part of the user story template. Scenarios, discussed in Sec 4.4, do include a primary customer's feelings and emotional state. Without emotions and intuition, latent needs are likely to remain latent.

3.7 When to Design?

The promise of iterative and agile processes is that they can accommodate requirements changes while a software project is underway. For the promise to be realized, the design and architecture must evolve as the system grows. The following are two approaches to evolving or maintaining a design:

- *Flexible Design.* Start with a design that is flexible enough to handle later changes without major rework.
- *Incremental Design.* Start with a minimal plausible design, add to it as needed, and refactor at the end of each iteration to incrementally clean up the design.

Both approaches involve some up-front design. The difference between them is in their point of view. Flexible design invests in anticipation of a later need. Incremental design defers investment until there is a need. The balance between up-front flexible design and just-in-time incremental design depends on the goals and risks of the project.

Flexible Design

Flexible design was identified as a success factor in a study of 29 iterative projects; see Section 3.3. Major initial investments in software design and architecture correlated with higher quality. All of the projects used iterative processes with short iterations to cope with uncertain and dynamically changing requirements. Their short iterations were comparable to those of agile processes.

Flexibility comes at a cost. The challenge with flexible design is that of over-investment; some of the investment may turn out to have been unnecessary.

Incremental Design

The premise of the incremental design approach is that the overall cost can be reduced by doing just-in-time design. Agile software teams have a slogan, *yagni*, which comes from "you aren't going to need it."[21] Yagni has come to mean: don't anticipate; don't implement any functionality until you need it.

However, yagni does not mean no design up front. As Kent Beck notes, "The question is not whether or not to design, the question is when to design." He recommends deferring until the last "responsible" moment.[22]

Automated regression testing and refactoring are essential to incremental design. Regression testing ensures that the current design is safe, that the features that used to work continue to work. Refactoring keeps the design clean, which lowers the cost of making a change.

The argument in favor of incremental design is that for clean code, the cost-of-change curve is relatively flat.[22] The cost of change curve in Section 2.4 plots the cost of fixing a severe defect against the phase in which the defect is discovered—the later the fix, the greater the cost. With disciplined use of refactoring, the claim is that the curve does not rise as steeply, which means that there is less of a penalty for changes as the project proceeds.

The challenge with incremental design is that of running into a dead end that might have been avoided with some preplanning.

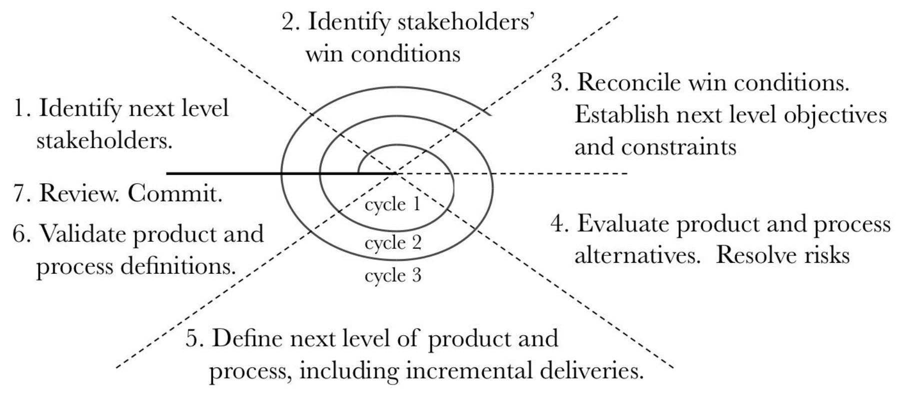

2. Identify stakeholders' win conditions
3. Reconcile win conditions. Establish next level objectives and constraints
1. Identify next level stakeholders.
7. Review. Commit.
6. Validate product and process definitions.
cycle 1
cycle 2
cycle 3
4. Evaluate product and process alternatives. Resolve risks
5. Define next level of product and process, including incremental deliveries.

Figure 3.10: Spiral Risk-Reduction Framework.

3.8 Spiral Risk-Reduction Framework

The *spiral framework* is an iterative approach to risk reduction during software development.[23] With each iteration, risk shrinks and the system grows. Iterations are called *cycles* to distinguish them from iterations in iterative processes.

The framework takes its name from diagrams like Fig. 3.10, where cycles are depicted by the loops of a spiral. The first cycle is at the center. The widening cycles reflect the increasing levels of investment and commitment to the project. The greater the risk, the more careful the investment. The level of effort and investment in a cycle is guided by the risk reduction planned during the cycle. For example, a first-of-its-kind project might have a prototyping cycle to investigate whether a proposed approach will work well. Better to invest a little up front in prototyping than to launch a full implementation, only to have it fail.

Each cycle corresponds to a pass through the numbered actions in the diagram. (Although the actions are numbered, they are expected to be concurrent, where possible.) The pass begins with identifying the next level of stakeholders and proceeds clockwise all the way around to the commitment to proceed with the next cycle.

The upper half of the diagram relates to the customer problem, the bottom half to the evolving solution:

- *Problem Defintion.* Identify the stakeholders and their objectives. Reconcile any conflicts and respect the constraints.
- *Solution Progress.* Resolve risks through benchmarking, modeling, and/or prototyping. Build the next increment for the next level of risk-reduction. Review with stakeholders and get a commitment to proceed.

The framework accommodates any methods or processes for carrying out the actions listed in Fig. 3.10. In fact, the reason for calling it a "framework" rather than a process model is that the framework has been adapted for use with various process models. For example, risk resolution in Action 4 can be through analysis, prototyping, or by building an increment to the system. In Action 5, the process for the next level of system definition and development can be different from the process used during the previous cycle.

For a simple project, some of the actions may be combined and the number of cycles reduced. A small project may have just one cycle. For a complex project, a more formal approach with more cycles might be helpful.

Example 3.6 : This example illustrates the flexibility of the spiral risk-reduction framework. It deals with a very large contract for a system to control remotely-piloted vehicles (drones). The contract had several risk-reduction cycles.

The challenge was to improve productivity by a factor of 8, from each drone piloted remotely by 2 people to 4 drones controlled by one person.[24] In other words, improve eightfold from a 1:2 ratio to a 4:1 ratio of drones to pilots.

The project started with four competing teams in the first risk-reduction cycle. Each team was awarded \$5M (M is for million). With an additional \$5M for evaluation, the initial investment for the four teams was \$25M. The review at the end of the first cycle concluded that a 4:1 ratio was not realistic, but that some improvement was possible.

Three competing teams remained in the second risk-reduction cycle. Each was awarded \$20M to build a scaled-down system. With an additional \$15M for evaluation, the incremental investment for the three teams was \$75M. The review at the end of the second cycle concluded that a ratio of 1:1 was possible.

For the final cycle, one team was selected to build a viable system that achieved a ratio of 1:1, resulting in a twofold productivity improvement.

Each of the competing teams could choose its own process for a given cycle.
□

3.9 Conclusion

Iterative processes and agile processes share the following properties:

- *Deep Customer Involvement.* Get the insight, guidance, and commitment of stakeholders early and often.

- *Time-Boxed Iterations.* Deliver working software frequently to get feedback and test hypotheses. Iteration intervals are now measured in weeks or days rather than months.

- *Working Software.* During development, maintain the integrity of the system. Test early and often. After each change, run all tests to have confidence that the change did not break something.

Since projects vary, software development processes vary from project to project. Consider for example the varying needs of systems for scientific computing, online banking, video streaming, self-driving cars, ... Each such project benefits from a process tailored to its needs.

If the actual process differs from the intended process, then document the decisions behind the actual process. The documentation will help the maintainers of the system.

Exercises for Chapter 3

Exercise 3.1 : For each of the following statements, answer whether it is True or False.

a) Very few projects have been completed using waterfall processes.

b) With a waterfall process, testing comes very late in the process.

c) Plan-driven processes call for careful up-front planning, so there are fewer errors.

d) A disadvantage of plan-driven processes is that they are inefficient and unsuccessful around constantly changing requirements.

e) With an iterative process, each delivery builds on the last.

f) Agile processes began with the Agile Manifesto.

g) With agile processes, project timelines can be hard to predict.

h) Agile processes are great for when you are not sure of the target of a project.

i) Pair programming doubles the cost of a project.

j) Sprint planning is essentially the same as iteration planning.

Exercise 3.2 : A usage survey identified the following as the top five Agile techniques (percentages refer to the organizations that practiced the technique):[25]

a) Daily Standup	83%
b) Prioritized Backlogs	82%
c) Short Iterations	79%
d) Retrospectives	74%
e) Iteration Planning	69%

For each technique, describe

- the purpose or role of the technique.
- how the technique is practiced.

Exercise 3.3 : Describe how you would do iterative development of *Moo*, a century-old game. When the game begins, the program thinks of a secret number made up of 4 different random digits, say `4271`. The program then invites the player to make guesses about the secret number. With each guess, the program provides feedback about the goodness of the guess. Two of the digits in the guess `1234` appear in the secret `4271`: the digit `2` is in the secret and is in the right position; the digits `1` and `4` are in the secret, but either one is in the wrong position.

Call a right digit in the right position a *bull* and a right digit in the wrong position a *cow*. With secret `4271`, the game might proceed as follows:

```
Moo: Decipher the secret number
guess: 1234
1 bull, 2 cows
guess: 5678
1 bull, 0 cows
...
```

Use more than one iteration. Be explicit about the deliverables at the end of each iteration.

Exercise 3.4 : Your client is a nation-wide insurance company that prides itself on its customer responsiveness. Each customer has a designated insurance agent who is familiar with the customer's needs and preferences.

The client has engaged you to create an application that routes customer phone calls and messages to their designated agent. But, there may be times when the designated agent is not in the office or is busy with someone else If a customer needs to speak to someone—say, to report an accident—then, as a backup, the automated application offers to connect them with another agent at the local branch (preferred) or at the regional support center, which is staffed 24 hours a day, 7 days a week. The regional support center is equipped to take customer calls, since along with any call, the application provides the answering agent with the customer's name and insurance coverage information. At any choice point, customers can choose to leave a message or request a call-back.

a) Putting yourself in the client's shoes, write 5 user stories based on the above description—let these be the 5 user stories that you believe are the highest priority, where priority is based on your perceived business value for the client. Make your stories SMART, where SMART stands for Specific, Measurable, Achievable, Relevant, and Time-bound.

b) For each user story, write Acceptance scenarios, using the Acceptance Criteria Template. Note that you may need more than one acceptance test to adequately cover a user story.

Notes for Chapter 3

[1]In the second edition of *Extreme Programming Explained* [2, chapter 7] , Kent Beck notes that some of the teams misinterpreted the first edition as recommending deferring design until the last moment—they created brittle poorly designed systems. He recommends deferring until the last "responsible" moment. The quote is from the same chapter.

[2]The Annual State of Agile Report contains usage data about agile practices. [19].

[3]Larman and Basili [13] trace the history of iterative processes.

[4]McIlroy, Pinson and Tague [16] describe the Unix program development process in a foreword to a special issue of the *Bell System Technical Journal* on Unix.

[5]Agile Manifesto: ©2001 by Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. The Agile Manifesto may be freely copied in any form, but only in its entirety through this notice. `http://agilemanifesto.org/` .

[6]The Agile Manifesto is accompanied by historical notes that touch on the "deeper theme" of trust and respect and organizational models that value people. [10].

[7]The Tidal Wave reference is from the title of an internal Microsoft memo by Gates [9]. The sleeping giant allusion is from Bill Gates's opening comments at Microsoft's December 1995 analysts conference [7, p. 109]

[8]Bill Turpin: "The original way we came up with the product ideas was that Marc Andreeson was sort of our product marketing guy. He went out and met with lots of customers. He would meet with analysts. He would see what other new companies were doing." [7, p. 251]

[9]Iansiti and MacCormack [11] note that companies in a wide range of industries "from computer workstations to banking" had adopted iterative product development to deal with uncertain and dynamic requirements.

[10]The discussion of the Space Shuttle software in Section 3.2 is based on the account by Madden and Rone [15].

[11]For the risks faced by the Netscape Communicator 4.0 project, see the remarks by Rick Schell, senior engineering executive [7, p. 187].

[12]MacCormack [14].

[13]"Ken Schwaber and Jeff Sutherland first co-presented Scrum at the OOPSLA conference in 1995. This presentation essentially documented the learning that Ken and Jeff gained over the previous few years applying Scrum." [18, 17].

[14]Beck and Andres [2] open with, "Extreme Programming (XP) is about social change." The basic premise of XP is from [2, ch. 17].

[15]The 3C acronym for Card, Conversation, Confirmation is due to Ron Jeffries [12].

[16]The user story template is attributed to Connextra, a London startup; see
`http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html`

[17]Beck [1] briefly describes the roots of XP and provides references. The quote about XP including a philosophy is from Beck and Andres [2, ch.1].

[18]Cockburn and Williams [6]

[19]Wray [20] reviews work on pair programming in a position paper prompted by his own experience with pair programming.

[20]Dan North [10] writes, "[Starting with user stories, Chris] Matts and I set about discovering what every agile tester already knows: A story's behaviour is simply its acceptance criteria .... So we created a template to capture a story's acceptance criteria. ... We started describing the acceptance criteria in terms of scenarios, which took the 'Given-when-then' form."

[21]Martin Fowler [8] attributes the acronym *yagni* to a conversation between Kent Beck and Chet Hendrickson, in which Hendrickson proposed a series of features to each of which Beck replied, "you aren't going to need it."

[22]In the second edition of *Extreme Programming Explained* [2, chapter 7].

[23]Barry Boehm introduced the spiral framework in the the 1980s [3]. The treatment in Section 3.8 follows [4]. The framework is not a process model: ,Boehm [4] desribes it as a

"risk-driven process model generator."

[24]Example 3.6 is based on Boehm [5], who compares Total (up-front) and Incremental (spiral) commitment of funds. The Incremental approach in Example 3.6 achieved twofold improvement in 42 months for an overall investment of $1B (B for billion). The winning bidder of a Total commitment project promised eightfold improvement in 40 months for $1B, but delivered only twofold improvement in 80 months for $3B.

[25]2015 State of Agile Report [19].

References for Chapter 3

1. Kent Beck. Embracing change with Extreme Programming. *IEEE Computer* (October 1999) 70-77
2. Kent Beck, with Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Ed.* Addison-Wesley, Reading, Mass. (2005).
3. Barry W. Boehm. A spiral model of software development and enhancement. *Computer* 21, 5 (May 1988) 61-72,
4. Barry W. Boehm. *Spiral Development: Experience, Principles, and Refinements.* Software Engineering Institute Report CMU/SEI-2000-SR-008 (July 2000).
5. Barry W. Boehm. The Incremental Commitment Spiral Model (ICSM): principles and practices for succesful software systems. *ACM Webinar* (December 17, 2013).
6. Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. `http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF`.
7. Michael A. Cusumano and David B. Yoffie. *Competing on Internet Time.* The Free Press, New York (1998).
8. Martin Fowler. Yagni. (May 26, 2015) `https://martinfowler.com/bliki/Yagni.html`.
9. William H. Gates, III. The Internet Tidal Wave. Internal Microsoft memo. (May 26, 1995). `http://www.justice.gov/atr/cases/exhibits/20.pdf`.
10. John Highsmith, for the Agile Alliance. History: The Agile Manifesto. `http://agilemanifesto.org/history.html`.
11. Marco Iansiti and Alan D. MacCormack. Developing products on Internet time. *Harvard Business Review* (September-October 1997).
12. Ron Jeffries. Essential XP: Card, Conversation, Confirmation. (August 30, 2001) `http://ronjeffries.com/xprog/articles/expcardconversationconfirmation/`.
13. Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer* 36, 6 (June 2003) 47-56.
14. Alan D. MacCormack. Product-development processes that work: How Internet companies build software. *Sloan Management Review* 42, 2 (Winter 2001) 75-84.
15. William A. Madden and Kyle Y. Rone. Design, development, integration: Space Shuttle primary flight software system. *Comm. ACM* 27, 9 (1984) 914-925.
16. M. D. McIlroy, E. N. Pinson, and B. A. Tague. Foreword: Unix time-sharing system. *Bell System Technical Journal* 57, 6 (July-August 1978) 1899-1904.
17. Ken Schwaber. Scrum development process. *OOPSLA '95 Proceedings* (1995) 117-134. In *Business Object Design and Implementation*, Jeff Sutherland et al. (eds) Springer Verlag (1997).
18. Jeff Sutherland and Ken Schwaber. *The Scrum Guide.* `http://www.scrumguides.org/`.

19. VersionOne. 10th Annual State of Agile Report. (2015) `http://stateofagile.versionone.com/`.

20. Stuart Wray. How pair programming really works. *IEEE Software* (January-February 2010) 51-55. See also: Responses to “How pair programming really works.” *IEEE Software* (March-April 2010) 8-9.

Chapter 4

Working with Customers

"The hardest single part of building a software system is deciding precisely what to build. ... the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. For the truth is clients do not know what they want."

— *Fred Brooks, in a 1986 essay on the inherent versus the incidental impediments to progress in software production.*[1]

Faced with a string of product failures in the late 1990s, Scott Cook, the founder of Intuit

> "reembraced a [Proctor & Gamble] fundamental—that new products should be based on actual customer behaviors, not on what customers *said* they wanted to do. ... Cook resolved that for future new product development, Intuit should rely on customer actions, not words."[2]

Identifying what customers need and want is one of the inherent challenges in software development. This chapter contains techniques for interacting with customers to identify and prioritize needs. The iterative process in Section 4.1 provides a context for discussing customer interactions. Customers are people, and people have levels of needs, as we shall see in Section 4.2. Some needs are conscious, easy to talk about; some are subconscious; some needs they may not even be aware of. The discussion of customer satisfiers and dissatisfiers in Section 4.3 is helpful for clarifying what customers find attractive and what they take for granted. Like user stories (Section 3.6), scenarios (Section 4.4) are a mechanism for recording customer requirements. What distinguishes scenarios is that they include not only customer needs and frustrations, but also a customer's emotional state.

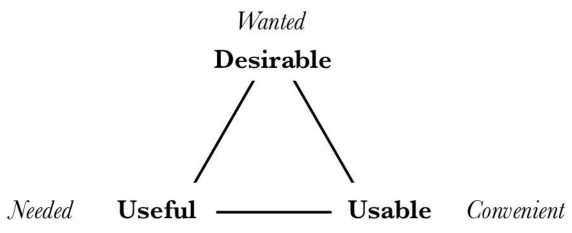


Figure 4.1: Great products address needs, wants, and ease of use.

Great Products: Useful, Usable, Desirable

Great products and systems have all of the following properties (see Fig. 4.1):[3]

- *Useful.* They serve a need and will be used.
- *Usable.* They are easy to use and can either be used immediately or can be readily learned.
- *Desirable.* Customers want them!

These properties are independent of each other. A system can be useful and desirable, but not usable; e.g., publishing software that produces great-looking books, but crashes often. Another example is a system with a complex interface that is hard to use.

A system can be usable and desirable, but not useful; e.g., fashionable apps that are dowloaded, but rarely used.

A system can be useful and usable, but not desirable; e.g., an application that gets treated as a commodity or is not purchased at all. Section 4.3 explores "must-have" features that are taken for granted if they are implemented, but cause dissatisfaction if they do not live up to expectations.

Terminology: Requirements and Specifications

For clarity, let the term *requirement* denote a constraint that must be met, and let *specification* denote a complete, precise, verifiable description of a solution or of some aspect of a solution.

We use the term requirement by itself as an abbreviation for a "customer requirement" that must be met to solve a customer problem or to achieve a customer objective. We use the term specification by itself as an abbreviation for "solution or product specification."

Traditionally, requirements reflect the development team's understanding of customer needs and wants. Requirements define what a system must do and what it must not do. The term is also used to denote constraints in other contexts; e.g., functional requirements, performance requirements, interface requirements, ...

Functional requirements relate to the job to be done. If the system can be described in terms of inputs and outputs, its functional requirements define the mapping from inputs to outputs. Similarly, if the system can be described in terms of stimuli and responses, its functional requirements define the stimulus-response behavior of the system.

Non-functional requirements relate to how well the job is done. Properties of the system such as performance, scale, and reliability are covered by non-functional requirements. Non-functional requirements are also known as *quality requirements*.

In practice, the usage of the term requirements blurs into the use of specifications. For example, IEEE Standard 830-1984 is a guide for writing a "software requirements specification," or "SRS." For completeness, a brief overview of the standard appears in Section 4.5.

4.1 A Context for Customer Interactions

4.1.1 Iterative Customer Feedback

Software companies recognize the need to listen to, observe, and empathize with customers. Intuit's three-step product design process is called *Design for Delight*:[4]

1. Develop deep empathy with customers through understanding what really matters to them and what frustrates them.
2. Offer a broad range of options before settling on a choice.
3. Use working prototypes and rapid iterations to get customer feedback.

The iterative process in Fig. 4.2 is similar. It is adapted from Scenario-Focused Engineering, which has been used within Microsoft.[5] The boxes in the diagram represent activities and the arrows represent outputs from the activities. (An overview of this process appears in Section **??**.)

An iteration begins at the top left: work with customers to identify their needs and potential opportunities for addressing unmet needs.

Moving right, the next step is to *frame* or define a problem by sifting through the opportunities. For the distinction between identifying needs and framing the problem, consider the following example:

Customer Need:	Listen to music
Problem Definition 1:	Offer songs for purchase and download
Problem Definition 2:	Offer a free streaming service with ads
Problem Definition 3:	Offer paid subscriptions without ads

In general, there are multiple ways of addressing a customer need.

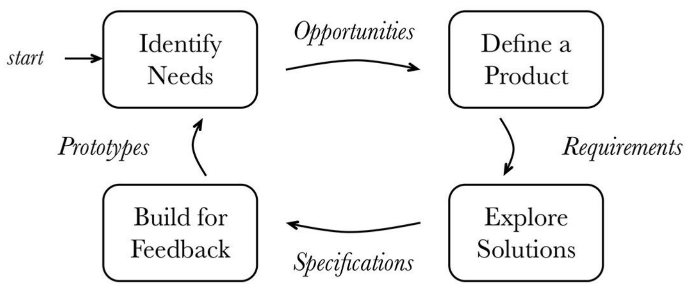


Figure 4.2: Iterative process for identifying and addressing customer needs. This figure repeats Fig. 2.3.

The lower half of the diagram represents activities internal to the development team: brainstorm possible solutions and then build prototypes to get customer feedback. As the iterations continue, the prototypes are refined into a deliverable product.

The quadrants in Fig. 4.3 elaborate on the steps in Fig. 4.2. The upper half of the figure focuses on the problem to be solved and the bottom half focuses on the solution. During needs identification and problem definition (the steps in the upper half), it is important to have implementation-free discussions, using language that customers can relate to.

4.1.2 Example: Personal Audio Player

The iterative process in Fig. 4.2 and 4.3 can be used for any product, not just for software. The following example illustrates the development process for the Sony Walkman personal audio player; see Fig. 4.4.

Example 4.1 : Introduced in 1979, the Sony Walkman was the first personal audio player. MP3 players and iPods followed in its footsteps.

Identify Customer Needs. Masaru Ibuka, co-founder of Sony, liked to listen to operas on long trans-Pacific flights. Sony had a high quality audio tape cassette device, but it was large and expensive. Ibuka wanted a smaller lighter device for his personal use.[6]

Define the Problem. The problem was to build a lightweight reliable personal player with high quality stereo audio. The requirement for high quality stereo audio follows from the need to listen to operas. Reliability was added as a requirement by the product team.

Explore Possible Solutions. At the time, audio-cassette devices were mono and had both record and playback functions; audio output was through a

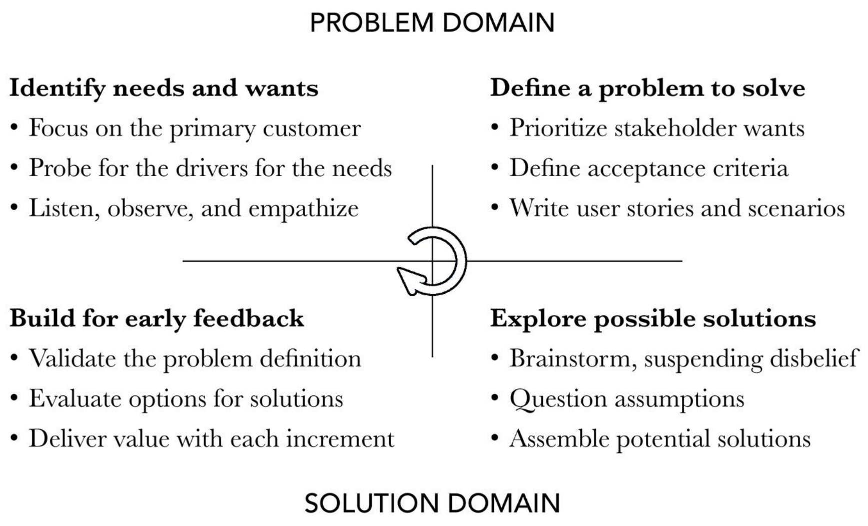

PROBLEM DOMAIN
Identify needs and wants
• Focus on the primary customer
• Probe for the drivers for the needs
• Listen, observe, and empathize
Define a problem to solve
• Prioritize stakeholder wants
• Define acceptance criteria
• Write user stories and scenarios
Build for early feedback
• Validate the problem definition
• Evaluate options for solutions
• Deliver value with each increment
Explore possible solutions
• Brainstorm, suspending disbelief
• Question assumptions
• Assemble potential solutions
SOLUTION DOMAIN

Figure 4.3: Iterative process for identifying and addressing customer needs.

speaker. Given the customer need for music on flights, the team focused on a stereo playback-only device. Dropping the ability to record saved weight, as did dropping the speaker and supporting only a headset. (Dropping the ability to record was controversial. In fact, when the Walkman was introduced, the press lampooned the inability to record. Retailers did not want to carry a playback-only device.)

Build in Increments. The initial prototype was created quickly by adapting an expensive product called Pressman, which was marketed only to the press. The final product was a sub-$200 device that went on to sell over 300 million units. □

4.1.3 Delight the Primary Customer

While the needs of all stakeholders must be considered, it helps to concentrate on a primary customer set, for two reasons.

First, a solution that delights the primary customers has the potential for benefiting secondary customers as well. For example, when telephone linesman jobs opened up to women in the late 1960s, the tools and procedures were redesigned with women as the primary "customers." The jobs were strenuous: linesmen used to climb telephone poles, carrying heavy tools. The redesign was an opportunity to rethink the tools and procedures. With the redesign, everybody benefited, women and men. Productivity went up.

Second, a primary customer or user provides a focus for the members of

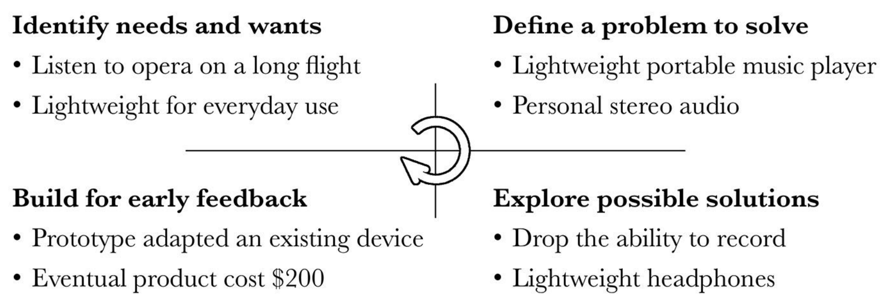

Identify needs and wants
• Listen to opera on a long flight
• Lightweight for everyday use
Define a problem to solve
• Lightweight portable music player
• Personal stereo audio
Build for early feedback
• Prototype adapted an existing device
• Eventual product cost $200
Explore possible solutions
• Drop the ability to record
• Lightweight headphones

Figure 4.4: Development of the Sony Walkman.

the development team. The developers can visualize the user, hear their voice, and empathize with them. Otherwise, each developer may potentially have a different mental image of the users of the system.

4.2 Levels of Needs

4.2.1 A Model of Customer Needs

Words and actions correspond to the top two levels in the model of customer needs in Fig. 4.5:[7]

- *Expressed Needs* relate to what people say and think. They are what people want us to hear. Listening accesses expressed needs.
- *Observable Needs* are needs that are displayed or can be inferred from actions and behaviors. They relate to what people do and use. Direct observations and usage data access observable needs.
- *Tacit Needs* are conscious needs that stakeholders cannot readily articulate. They relate to what people feel and know. Empathy and trading places—walking in someone else's shoes—are needed to access tacit needs.
- *Latent Needs* are needs that are not conscious or are not recognized. Empathy and intuition can help in making hypotheses about customers' latent needs.

4.2.2 Accessing Expressed and Observable Needs

Interviews and surveys are techniques for accessing expressed needs. They are classified in the left column in Fig. 4.6. Interviews are qualitative; that is, interview findings cannot be readily measured. Surveys, on the other hand, are quantitative; responses can be measured.

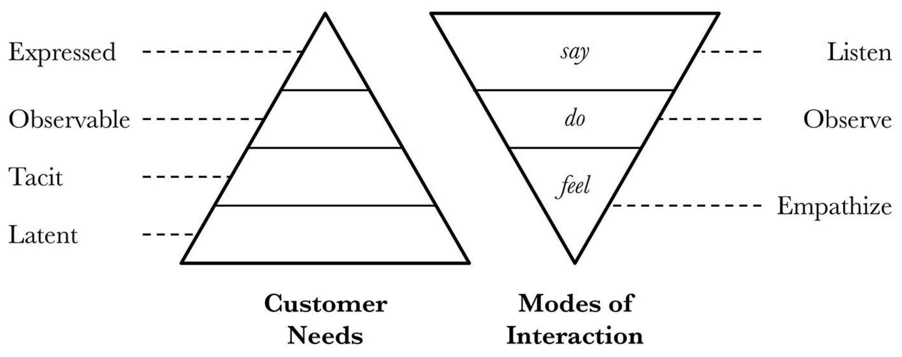


Figure 4.5: Levels of customer needs and corresponding modes of interaction for accessing needs.

Observation of customer behavior and analysis of usage logs are techniques for accessing observable needs; see the right column in Fig. 4.6. Observations are qualitative; usage logs are quantitative.

The next example illustrates the use of these techniques.[8]

Example 4.2 : The intended users for Intuit's QuickBooks app for the iPad were one-person business owners who spent most of their time out of the office, serving their customers. The business owners wanted to manage all aspects of their business while mobile. For example, they wanted to provide professional estimates and invoices on the spot, at a customer site, rather than waiting until they got back to the office. Without the ability to manage their business while mobile, work was piling up and they were spending evenings and weekends catching up.

Using interviews and observations to understand user needs and goals, the Intuit team came up with an initial list of requirements, which they evolved based on user feedback. The requirements were captured as user stories.

Starting with an initial working prototype, the team used an agile process to implement the stories. The app was instrumented to provide usage data. During iteration planning, prioritization of stories was based on both observations and analysis of usage data.

Careful observations guided the design of the user experience. As an example, users held their tablets differently while they were mobile and while they were at home. On the go, they favored portrait mode, while at home they favored landscape mode. Why? The task mix on the go was different from the mix at home. At home, they dealt with more complex transactions and landscape mode allowed them to display more data on a line. □

Abstracting from Example 4.2, the needs that are identified in the early stages of a project are likely to be expressed needs. As the development team

		Customer Needs	
		Expressed	*Observable*
Collected Data	*Qualitative*	Interviews	Observations
	Quantitative	Surveys	Usage Logs

Figure 4.6: Classification of techniques for identifying expressed and observable needs.

establishes rapport with stakeholders and as the stakeholders provide feedback on prototypes, additional needs are likely to surface. In the example, it was only after the app had been prototyped that the team gained the insight that customers held their iPads differently in the field and at home.

4.2.3 Listening with Understanding

Listening, really listening, is an essential skill when working with customers. In a classic paper on the barriers to communication, Carl Rogers wrote,

> "there is one main obstacle to communication: people's tendency to *evaluate*. Fortunately, I've also discovered that if people can learn to *listen* with understanding, they can ... greatly improve their communication with others."[9]

Listening with understanding is easy to state; it takes practice to do well. It involves not only understanding the content of what the other person is saying, but sensing how they are feeling about the subject of the communication.

Active listening involves repeating back to the person what they have said in a way that leaves them feeling that they have been heard. For example, when a customer asks for a feature, instead of jumping in with a question or losing the thread of the conversation, begin with something like

> "What I heard you say is ..."

These need not be the exact words—it would make for an awkward conversation if too many sentences began with, "What I heard you say is, ..." The objective is to establish rapport before going on to a question or changing the direction of the conversation.

Words and phrases carry nuances that are important to people. When taking notes, capture the person's words verbatim. Resist the urge to paraphrase or reword. Marketers use the term *Voice of the Customer* for a customer need expressed "in the customer's own words, of the benefit to be fulfilled by the product or service."[10]

4.3 Customer Satisfiers and Dissatisfiers

What makes one product or feature desirable and another taken for granted until it malfunctions? An analysis of satisfiers and dissatisfiers is helpful for addressing such questions. Such analysis can unearth latent needs.

4.3.1 Background: Job Satisfiers and Dissatisfiers

In the late 1950s, Frederick Herzberg and his colleagues discovered that the factors that lead to job satisfaction are different from the factors that lead to job dissatisfaction.[11]

Job satisfaction is tied to the work, to what people do: "job content, achievement on a task, recognition for task achievement, the nature of the task, responsibility for a task and professional advancement."

Job dissatisfaction is tied to "an entirely different set of factors." It is tied to the situation in which the work is done: supervision, interpersonal relationships, working conditions, salary. Improving working conditions alone reduces dissatisfaction, but it does not increase job satisfaction because the nature of the work does not change. Similarly, improving the nature of the work alone does not reduce job dissatisfaction, because working conditions and salary do not change.

4.3.2 Kano Analysis

Noriaki Kano and his colleagues carried the distinction between job satisfiers and dissatisfiers over to customer satisfiers and dissatisfiers.[12] *Kano analysis* assesses the significance of product features by considering the effect on customer satisfaction of (a) building a feature and (b) not building the feature. Kano analysis has been applied to user stories and work items in software development.[13]

Paired Questions

Kano et al. used questionnaires about product features that had paired positive and negative questions of the form

- If the product *has* this feature ...
- If the product *does not have* this feature ...

For example, consider the following positive and negative forms of a question:

- If this product *can* be recycled ...

		Satisfied	*Neutral*	*Dissatisfied*
Feature is Built	*Satisfied*		Attractor	Key
	Neutral	Reverse	Indifferent	Must-Have
	Dissatisfied	Reverse	Reverse	

Feature is Not Built

Figure 4.7: Classification of features.

- If this product *cannot* be recycled ...

With each form, positive, and negative, they offered five options:

a) I'd like it
b) I'd expect it
c) I'm neutral
d) I can accept it
e) I'd dislike it

4.3.3 Classification of Features

The results from paired positive and negative questions can be classified using the nine-box grid in Fig. 4.7. For simplicity, the classification is based on the following three, instead of five, options:

a) I'd be satisfied
b) I'm neutral
c) I'd be dissatisfied

The rows in the nine-box grid correspond to the cases where a feature is built. The columns are for the cases where the feature is not built.

Key Features

The top-right box in Fig. 4.7 is for cases where customers are satisfied if the feature is built and would be dissatisfied if it is not built. These are key features to build; the more the better, subject to the project's schedule and budget constraints. For example, with time-boxed iterations, a constraint would be the number of features that the team can build in an iteration.

Reverse Features

The shaded box to the bottom left is for the case where customers would be dissatisfied if the feature is built and satisfied if it is not built. The two adjacent shaded boxes are also marked Reverse. Reverse features are detractors; the fewer that are built the better.

Example 4.3 : Features that customers do not want are examples of reverse features. Clippy, an "intelligent" Microsoft Office assistant would pop up when least expected to cheerily ask something like, "I see that you're writing a letter. Would you like help?"

Clippy was so unpopular that an anti-Clippy web site got about 22 million hits in a few months after the launch of the web site.[14] □

Attractor Features

The box in the middle of the top row is for cases where customers would be satisfied if the feature is built and neutral if it is not. This box represents features that can differentiate a product from its competition. Features that address latent needs are likely to show up as attractors.

Example 4.4 : Around 2000, mobile phones with web and email access were a novelty. Kano analysis in Japan revealed that young people, such as students, were very enthusiastic about the inclusion of the new features, but were neutral about their exclusion. Web and email access were therefore attractors.

Indeed, phones with web and email access rapidly gained market share. □

Must-Have Features

The box in the middle of the right column is for cases where customers would be neutral if the feature is built and dissatisfied if it is not built. Features customers take for granted fit in this box. Performance, reliability, and quality are typical attributes that customers take for granted. If a feature does not meet the performance threshold—that is, if performance is not built in—then customers would be dissatisfied. But if performance is built in, then customers may not even notice it.

Indifferent Features

The middle box in Fig. 4.7 is for the case where customers would be neither satisfied nor dissatisfied if the feature were built. Features that customers consider unimportant would also fit in this box.

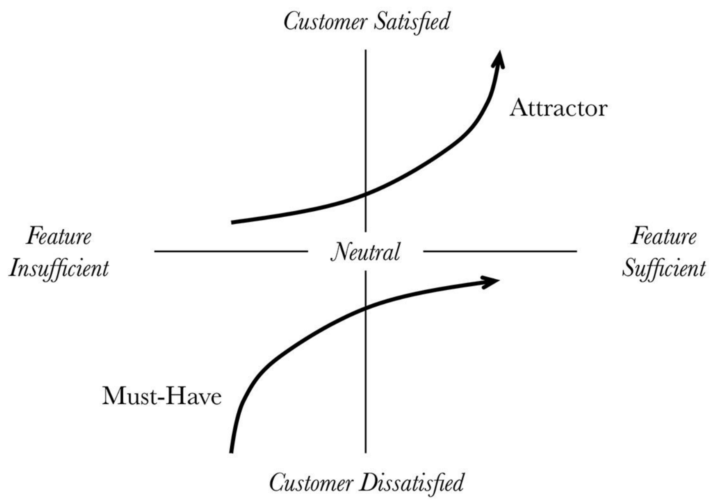


Figure 4.8: Conceptual diagram illustrating the increase in customer satisfaction for Attractor and Must-Have features.

4.3.4 Degrees of Sufficiency

The classification of features in Fig. 4.7 is based on a binary choice: either the feature is built or it is not built. Response time and capacity are examples of system attributes that are not binary. They potentially improve along a sliding scale.

The conceptual diagram in Fig. 4.8 illustrates features that have degrees or levels of sufficiency. The horizontal axis represents degrees of sufficiency, increasing from *Insufficient* on the left to *Sufficient* on the right. The vertical axis represents degrees of satisfaction, from *Dissatisfied* at the bottom to *Satisfied* at the top.

The top curve is for an attractor. With an attractor, customers are satisfied if the feature is built, but are neutral if it is not built. In Fig. 4.8 the *Attractor* curve rises from roughly *Neutral* to *Satisfied* as the degree of the feature increases from insufficient to sufficient.

The bottom curve is for a must-have feature. With a must-have feature, customers are dissatisfied if the feature is not built, but are neutral if it is built. The *Must* curve rises from *Dissatisfied* to roughly *Neutral* as the degree of the feature increases from insufficient to sufficient.

The curves for key, indifferent, and reverse features are not shown in Fig. 4.8. The curve for a key feature would increase diagonally up and to the right; the curve for an indifferent feature would stay at *Neutral*; and the curve for a reverse feature would decrease diagonally down and to the right.

4.3.5 Life Cycles of Attractiveness

The attractiveness of features can vary over time. Consider again web and email access features for mobile phones. When these features were first introduced in Japan in the late 1990s, young people found them to be attractive, but middle-aged people were indifferent—they had not experienced the need for the features. In the early days of the World Wide Web, people outside the technical community may have heard the term, but few felt the need for it. Today, web and email access; have become musts for any new smartphone.

For features, a possible progression is as follows:

1. *Indifferent.* When an entirely new feature is introduced, people may be indifferent because they do not understand its significance.

2. *Attractor.* As awareness of the new feature grows, it becomes an attractor, a delight to have, but people who do not have it are neutral.

3. *Key.* As usage of the feature grows, people come to rely upon it and are dissatisfied if they do not have it.

4. *Must-Have.* Eventually, as adoption of the feature grows, it comes to be taken for granted if it is present and a dissatisfier if it is not present.

The above is not the only progression. A fashionable feature may rise from indifferent to key and fade back to indifferent as it goes out of fashion.

4.4 Scenarios: End-to-End Experience

Latent customer needs can be uncovered when members of a project team empathically immerse themselves in the customer's end-to-end experience.

A *scenario* is a personalized narrative of a primary customer's end-to-end experience.[15] It brings to life the customer, their situation, their delights, frustrations, and emotional state in the situation. It provides a context for insights into the customer's needs and wants.

The scenario identifies

- the primary customer,
- what the customer wants to experience or accomplish,
- the proposed benefit to the customer,
- how the proposed benefit contributes to their overall experience, and
- the customer's emotional response in the overall experience.

Example 4.5 : Konica was a pioneering Japanese camera manufacturer. In the 1970s, it sought to differentiate its cameras in a crowded competitive market.[16]

Konica engineers conducted extensive interviews to identify what customers wanted in a camera. The results from the interviews were disappointing, since customers wanted more of the same. Their requests were for minor changes to the existing designs.

The engineers then tried a different approach. At the time, cameras captured images on photographic film, which was sent to a photo lab for processing and printing. The engineers visited photo labs to learn about issues encountered during processing.

They discovered that the top two reasons for poor photo quality were underexposure and blurry out-of-focus images. Underexposure was due to pictures taken in low light without a flash. Flash units were separate and bulky and people would either forget them or leave them at home. Blurry images were due to difficult manual focus controls.

These insights prompted the engineers to design cameras with a built-in flash unit and an auto-focus mechanism. For further ease of use, the cameras were auto loading, to allow the film to be quickly and simply changed. The point-and-shoot Konica C35 AF, introduced in 1977, was the first mass produced auto-focus camera.

The insights that led to the point-and-shoot camera were uncovered by a change in viewpoint, from product features to the end-to-end experience. The interviews that identified only minor changes asked about camera features. The visits to the photo lab studied the experience of taking pictures. □

4.4.1 Writing Scenarios

A scenario is a narrative or a story, not a checklist of items. As a helpful suggestion, consider the following elements when writing a scenario:

- *Title.* A descriptive title.
- *Introduction.* An introduction to the primary customer; their motivation; what they might be thinking, doing, and feeling; and their emotional state—that is, whether they are mad, glad, or sad.
- *Situation.* A brief description of the real-world situation and the need or opportunity. The description puts the need in the context of an overall experience.
- *Outcome.* The outcome for the customer, including success metrics and what it would take for the solution to delight the customer.

Together, the introduction and situation in a scenario correspond to the current state. The outcome corresponds to the desired state. What a scenario must not include is any premature commitment to a potential implementation, to how to get from the current to the desired state.

The narrative in the next example has two paragraphs, one paragraph for the introduction and situation and another paragraph for the desired outcome.

Example 4.6 : The fictionalized scenario in this example is based on the case of the Sony Walkman in Example 4.1.

> *Title.* Masaru Ibuka enjoys opera on a long trans-Pacific flight.
>
> *Introduction.* On long flights across the Pacific, Masaru Ibuka wished that he could listen to opera, to while away the time. He knew that Sony made high quality stereo cassette tape recorders, but they were too bulky to take on a flight. He wanted something convenient and lightweight that he could carry with him.
>
> *Outcome.* Ibuka is delighted with the new device that the engineers have created. The stereo audio quality is excellent; it's almost as if he were at a live performance. The device is light enough and small enough that he can carry it with him, along with the baggage for his trip. Furthermore, the device can play without power for many hours, long enough to last the whole flight.

□

4.4.2 A Checklist for Scenarios

The acronym SPICIER provides a checklist for writing a good scenario:

S:	tells the beginning and end of a *story*
P:	includes *personal* details
I:	is *implementation-free*
C:	it's the *customer's* story, not a product story
I:	reveals deep insight about customer needs
E:	includes *emotions* and *environment*
R:	is based on *research*

Example 4.7: Applying the SPICIER checklist to the Masaru Ibuka scenario in Example 4.6, we get

S: *Story.* The narrative begins with Masaru Ibuka's desire to carry his music with him on long flights. It ends with him being delighted with a small lightweight device that has excellent stereo audio quality and can power itself for many hours.

P: *Personal.* The personal details include his love of opera.

I: *Implementation-Free.* The scenario focuses on the need to listen to opera and attributes such as bulk and playing time. It does not say anything about how the device works or how to implement it. Although it notes the existence of cassette recorders with the desired audio quality, the scenario does not say that the device has to be a tape recorder. The Walkman product line later included CD players and MP3 players, among others.

C: *Customer's Story.* The scenario is about the experience of listening to opera, not about a product feature.

I: *Insight.* Looking back, there were two key insights that emerged from "listen to opera" and "on a flight." At the time, cassette recorders had both record and playback functions, and playback was through a speaker. The first insight was that the record function could be dropped; playback was enough. The second insight was that the need was for personal audio, since a flight is a public space. The engineers designed for a private headset rather than a speaker, which could disturb a seat-mate.

E: *Emotions and Environment.* The scenario is specific about the environment (a flight). It hints at emotions, but is not explicit about them. "While away the time" hints at boredom on a long flight. "Too bulky to take on a flight" hints at unhappiness with bulky cassette recorders.

R: *Research.* The scenario is based on the primary customer's needs, it was not based on further research. In serving the primary customers, the engineers created a whole new product category: personal audio players.

□

4.5 SRS: Software Requirements Specifications

With plan-driven processes, the next step after gathering customer requirements is to write specifications. A specification is a complete, precise, verifiable description of a system or of an aspect of a system. Developers use specifications to build a system, testers use them to write system tests, and project managers use them to estimate the cost and schedule for a project.

IEEE Standard 830-1984 is a guide for writing software specifications. The standard refers to them as *software requirements specifications* (*SRS*). The outline of an SRS in Fig. 4.9 is adapted from the standard. According to the standard a good SRS is[17]

- *Unambiguous.* Natural language descriptions can be ambiguous, so care is needed to ensure that the descriptions in the SRS permit only one interpretation.
- *Complete.* For an SRS to be complete, it must cover all requirements and define the system behavior for both valid and invalid data and events.
- *Verifiable.* An SRS is verifiable if all requirements can be tested.
- *Consistent.* An SRS is consistent if no two specifications are inconsistent and occurrences of the same behavior are described using the same terminology across specifications.
- *Modifiable.* An SRS is modifiable if the specifications in the SRS can be easily modified without violating the above properties, such as consistency and completeness.
- *Traceable.* An SRS is traceable if every specification can be traced or connected with a customer requirement (*backward traceability*) and has

1. Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms, and abbreviations
- 1.4 References
- 1.5 Overview

2. Overall description
- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 General Constraints
- 2.5 Assumptions and dependencies

3. Specific requirements
- 3.1 Functional Requirements
  - 3.1.1 Functional Requirement 1
    - 3.1.1.1 Introduction
    - 3.1.1.2 Inputs/Stimulus
    - 3.1.1.3 Processing
    - 3.1.1.4 Outputs/Responses
  - 3.1.2 Functional Requirement 2
  - ...
  - 3.1.*n* Functional Requirement *n*
- 3.2 External Interface Requirements
  - 3.2.1 User interfaces
  - 3.2.2 Hardware interfaces
  - 3.2.3 Software interfaces
  - 3.2.4 Communication interfaces
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Attributes
- 3.6 Other Requirements

Figure 4.9: An outline of a software specification, from IEEE Standard 830-1984.

a reference to any documents that depend on a specification in the SRS (*forward traceability*).

- *Usable for Maintenance.* The operations and maintenance staff are typically different from the development team. The SRS must meet the needs of the operations and maintenance staff.

4.6 Conclusion

The distinction between what people say and what they do has long been recognized in product development, not just software development. What people say relates to the top or surface level of a four-level model of customer needs in Section 4.2.

The four levels are: expressed, observable, tacit, and latent. Expressed needs relate to what people say and think and want us to hear. Observable needs can be inferred from what people do and use. Tacit needs relate to what people feel and know; empathy with the other person is needed to access them. Latent needs are needs that people are not aware of.

Customer needs and wants are the starting point for the iterative process in Section 4.1. The steps in the process are: identify needs; define the problem; explore possible solutions; and grow the system by building an increment. After each incremental improvement to the system, customer feedback is helpful, not only for validating that the project is on track, but for surfacing additional needs. From Example 4.2, observations and analysis of usage data uncovered further needs, once customers got their hands on prototypes of the QuickBooks app for the iPad.

Tacit and latent needs require deep insights into the customer's end-to-end experience. Scenarios, Section 4.4, are a personalized narrative about a customer, their situation, and their emotional state in the situation. In Example 4.5, Konica engineers were initially disappointed with the results from customer interviews about camera features: customers suggested minor changes to existing features. The engineers identified the problems of poor lighting and blurry images only after they immersed themselves in their customers' picture-taking experience.

Kano analysis, Section 4.3, assesses the significance of product features. The assessment is based on the distinction between customer satisfiers and dissatisfiers. Satisfiers are different from the dissatisfiers. Features that customer find attractive relate to satisfaction. Features that customers consider must-haves lead to dissatisfaction if they are missing or malfunction.

With plan-driven processes, requirements are typically written as a list of features. IEEE Standard 830-1984 is a guide to writing SRS (Software Requirements Specifications) documents.

Exercises for Chapter 4

Exercise 4.1 : Come up with your own examples of products that are

a) Useful, but neither usable nor desirable
b) Usable, but neither useful nor desirable
c) Desirable, but neither useful nor usable
d) Useful and usable, but not desirable
e) Useful and desirable, but not usable
f) Usable and desirable, but not useful
g) Useful, usable, and desirable

Your examples need not relate to software.

Exercise 4.2 : Use the 9-box grid in Fig. 4.10 to classify features during Kano Analysis. Give a one-line explanation for why a given class of features belongs in one of the boxes in the grid. (Note that the ordering of rows and columns in Fig. 4.10 is different from the ordering in Fig. 4.7.)

Exercise 4.3 : Kano analysis classifies features using a nine-box grid (Fig. 4.7).

a) How would you classify features using the four-box grid in Fig. 4.11, instead of the nine-box grid in Fig. 4.7. Explain your answer.

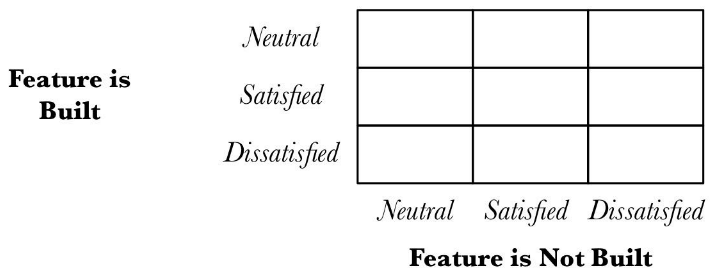


Figure 4.10: Classification of features during Kano analysis. The rows and columns have been scrambled, relative to Fig. 4.7.

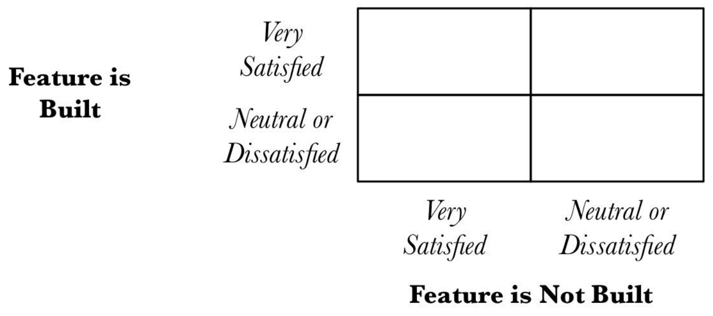


Figure 4.11: Classification of features during Kano Analysis.

b) For each box in the four-box grid in Fig. **??**, give an example of a specific feature that belongs in that box. Why does it belong in the box?

Exercise 4.4 : The conceptual diagram in Fig. 4.8 illustrates the increase in customer satisfaction for Attractor and Must-Have features as the sufficiency of a feature increases. Draw the corresponding curves for Key, Reverse, and Indifferent features.

Exercise 4.5 : Based on Example 4.5 about Konica cameras,

a) Write a SPICIER scenario.
b) Show how your scenario meets the criteria represented by the SPICIER checklist.

Notes for Chapter 4

[1]Fred Brooks [1].

[2]Before starting Intuit, Scott Cook worked at Proctor & Gamble (P&G), where he "learned an encyclopedia's worth of business. ... P&G's obsession with customers resonated with Cook." [16, p. 6]. The quote about basing new products on customer behaviors, not words, is from [16, p. 221].

[3]Sanders [14].

[4]For Intuit's Design for Delight, see for example Ruberto [13].

[5]The iterative process in Fig. 4.2 is adapted from the Fast Feedback cycle in De Bonte and Fletcher [2]. See also the overview of the iterative process in Section **??**.

[6]Hormby [5]

[7]Sanders [14, 15] uses an inverted pyramid for levels of customer needs. She writes about a "shift in attitude from designing **for** users to one of designing **with** users."[15]

[8]Rabinowitz [11].

[9]Carl Rogers [12] cites "a natural urge to judge, evaluate, and approve (or disapprove) another person's statement" as a major barrier to communication.

[10]Griffin and Hauser [3] refer to Voice of the Customer as an industry practice.

[11]Herzberg [4, ch. 6] et al. interviewed 200 engineers and accountants to test the hypothesis that people have two sets of needs: "the need as an animal to avoid pain" and "the need as a human to grow psychologically." They asked about times when the interviewees felt especially good about their jobs and probed for the reasons behind the good feelings. In separate interviews, they asked about negative feelings about the job. The results from the study were that satisfiers were entirely different from dissatisfiers. Herzberg et al. used the term "motivators" for job satisfiers and the term "hygiene" for job dissatisfiers. The corresponding terms in this chapter are *attractors* and *must-haves*.

[12]Kano [7] is the source for the treatment of Kano analysis in Section 4.3. The oft-cited paper by Kano et al. [8] is in Japanese.

[13]Kano analysis is included in De Bonte and Fletcher's [2] description of Scenario Focused Engineering at Microsoft.

[14]Clippy was included in Microsoft Office for Windows versions 97 through 2003 [18]. A USA Today article dated February 6, 2002 noted, "The anti-Clippy site has gotten 22 million hits since launching April 11." [17].

[15]For an in-depth treatment of scenarios, see the book on Scenario-Focused Engineering by De Bonte and Fletcher [2].

[16]Example 4.5 on Konica cameras is based on the description by Kano [7].

[17]IEEE Standard 830-1984 [6, p. 11]

References for Chapter 4

1. Frederick P. Brooks, Jr. No silver bullet" essence and accident in software engineering. *Information Processing '86* (1986) Elsevier, Amsterdam, 1069-1076. Reprinted in *IEEE Computer* (April 1987) 10-19.

2. Austina De Bonte and Drew Fletcher. *Scenario-Focused Engineering.* Microsoft Press, Redmond, Wash. (2013).

3. Abbie Griffin and John R. Hauser. The Voice of the Customer. *Marketing Science* 12, 1 (Winter 1993) 1-27.

4. Frederick Herzberg. *Work and the Nature of Man.* Cleveland World Publishing Co. (1966).

5. Tom Hormby. The story behind the Walkman. *Low End Mac* (August 13, 2013). `http://lowendmac.com/2013/the-story-behind-the-sony-walkman/`.

6. *IEEE Guide to Software Requirements Specifications.* IEEE Standard 830-1984 (February 10, 1984).

7. Noriaki Kano. Life cycle and creation of attractive quality. `http://huc.edu/ckimages/files/KanoLifeCycleandAQCandfigures.pdf` .

According to Löfgren and Wittel [9] a paper with this title was presented at the *4th International QMOD Quality Management and Organizational Development Conference* at Linköping University (2001).

8. Noriaki Kanoi, Nobuhiko Seraku, Fumio Takahashi, and Shin-ichi Tsuji. Attractive quality and must-be quality (in Japanese). *Journal of the Japanese Society for Quality Control* 14, 2 (1984) 147-156.

9. Martin Löfgren and Lars Wittel. Two decades of using Kano's theory of attractive quality: a literature review. *The Quality Management Journal* 15, 1 (2008) 59-75.

10. Dan North. Behavior modification. *Better Software Magazine* (March 2006). See `https://dannorth.net/introducing-bdd/`.

11. Dorelle Rabinowitz. `http://www.aiga.org/inhouse-initiative/intuit-quickbooks-ipad-case-study-app-design/`.

12. Carl R. Rogers and F. J. Roethlisberger. Barriers and gateways to communication. *Harvard Business Review* (July-August 1952). Reprinted, *Harvard Business Review* (November-December 1991). `https://hbr.org/1991/11/barriers-and-gateways-to-communication`

13. John Ruberto. Design for Delight applied to software process improvement. *Pacific Northwest Software Quality Conference* (October 2011). `http://www.pnsqc.org/design-for-delight-applied-to-software-process-improvement/`.

14. Elizabeth Sanders. Converging perspectives: product development research for the 1990s. *Design Management Journal* 3, 4 (Fall 1992) 49-54.

15. Elizabeth Sanders. From user-centered to participatory design approaches. In *Design and the Social Sciences*, J. Frascara (ed.) Taylor & Francis Books (2002).

16. Suzanne Taylor and Kathy Schroeder. *Inside Intuit*. Harvard Business School Press, Boston, Mass. (2003).

17. USA Today. Microsoft banks on anti-Clippy sentiment. (February 6, 2002). `http://usatoday30.usatoday.com/tech/news/2001-05-03-clippy-campaign.htm`.

18. Wikipedia. Office Assistant. `https://en.wikipedia.org/wiki/Office_Assistant`.

Chapter 5

Use Cases

"The use cases capture the goals of the system. To understand a use case we tell stories. ... Use cases provide a way to identify and capture all the different but related stories in a simple but comprehensive way."
— *Ivar Jacobson, Ian Spence, and Brian Kerr. Use cases were introduced by Ivar Jacobson in 1986.*[1]

A *use case* models interactions between users and the system, interactions that result in an observable benefit for some user. Each use case is built around a sunny-day sequence of actions—"sunny day" means that all goes well and the system delivers the desired benefit. The sunny day sequences are called basic flows. Basic flows and the other elements of use cases are defined in Section 5.1.

A well written collection of use cases provides a readable overview of the expected behaviors of the system. Readability makes use cases suitable for discussions with customers and users. The initial discussions with customers can focus on basic flows and their associated benefits. Special cases and exceptions need not cloud the initial discussions.

Basic flows also provide a starting point for iterative development. Planning for the first iteration can begin with the basic flow from the most important use case. Special cases can be handled in later iterations.[2]

Use case diagrams, Section 5.2, depict collections of related use cases. They show who or what interacts with the system. In diagrams, a use case is represented by its associated goal or benefit.

System behaviors correspond to functional requirements. Collections of use cases give the set of functional requirements for a system. Use cases have to be supplemented with descriptions of non-functional requirements, such as performance and availability requirements.

Actors:	Every role, human and automated, in the use case
Basic Flow:	Sequence of actions—text, readable by stakeholders
Alternative Flows:	Variations on the basic flow
Extension Points:	Insertion points in a flow, for additional behavior

Figure 5.1: Key elements of use cases.

5.1 Elements of Use Cases

Use cases were motivated by the problem of modeling telephone calls.[3] While the internal workings of a telephone system can be enormously complex, a caller's intentions can be simply stated: connect with a callee. The system's response to a caller is outlined in the following example.

Example 5.1 : A telephone caller wants to connect to a callee. The following sequence of actions is one way of achieving this user goal:

1. The caller enters a number to be called.
2. The system collects the number.
3. The system maps the number to a destination.
4. The system routes the call through the telephony infrastructure.
5. The system rings the callee's phone.
6. The callee answers.
7. The system starts logging the duration of the call.
8. The caller disconnects.
9. The system records the duration of the call.
10. The system releases the resources for the call.

The above sequence is one of the simplest way of achieving the goal of placing a telephone call. There are many possible variations of the sequence. □

The complexity of most systems is due to the many options, special cases, exceptions, and error conditions that the systems must be prepared to handle. For the telephone call in Example 5.1, what if the callee does not answer? What if the destination phone number is no longer in service? What if the callee has turned on a do-not-disturb feature? There are hundreds of possibilities.

The rest of this section introduces the key elements of use cases; see Fig. 5.4.

5.1.1 Actors Represent Roles

An *actor* represents a role played by a person or a thing. The same person may play multiple roles. For example, consider an employee of a company who manages a team. This person has two roles: employee of the company and manager of a team. Each of these two roles would be represented by a different actor. Same person, two roles.

The purpose of a use case is to provide a benefit for one or more actors. The benefit is called the *goal* and the initiator of the use case is called the *primary actor* of the use case. The *start* and *end* of a use case refer to the start and end of a sequence of actions that is called the basic flow.

5.1.2 Flows Represent Sequences of Actions

The term *flow*, by itself, represents a sequence of actions from the start to the end of a use case. Flows represent interactions between actors and the system; each action is either initiated by an actor or by the system.

Example 5.2 : The sequence of actions for placing a phone call in Example 5.1 constitutes a flow. A variant of the flow is obtained if, in Action 8, the callee disconnects, instead of the caller.

The flow involves two actors: the caller and and the callee. The caller is the primary actor. □

The *basic flow* of a use case is a successful sequence of actions that achieves the goal of the use case. A well written basic flow represents the simplest way of achieving the goal. Special cases and exceptions are handled separately by variants of the basic flow.

For the moment, an alternative flow is simply a variant of the basic flow. (The connection between basic and alternative flows is explored below, in Section 5.1.3.)

Alternative flows represent behaviors that are required of a system, but are not central to an intuitive understanding of the system. For example, many systems begin by authenticating a user. In a use case, the basic flow would handle successfully authentication. An alternative flow would handle the case where authentication fails. The system is required to gracefully handle authentication failures, but their handling does not tell us much about the purpose of the system.

Example 5.3 : Cash withdrawal from an automated teller machine (ATM) is a classic example of a use case.[4] For simplicity, the actions in the flow are high level

Basic Flow: Withdraw Cash

1. The cardholder inserts a card.
2. The system reads the card and prompts for a passcode.
3. The cardholder enters a passcode.
4. The system authenticates the cardholder.
5. The system displays options for bank services.
6. The cardholder selects Withdraw Standard Amount.
7. The system checks the account for availability of funds.
8. The system dispenses cash and updates the account balance.
9. The system returns the card.

The basic and alternative flows for cash withdrawals correspond to paths through the graph in Fig. 5.2. The basic flow goes sequentially through the numbered actions, from the start to the end of the use case. The two alternative flows in the figure are for handling special situations: authentication fails or there are insufficient funds in the account. Many other alternative flows are possible; for example, the card may get stuck; the ATM may run out of cash; and so on.

The following alternative flow corresponds to the path for failed authentication:

Alternative Flow 1: Authentication Failed

1. The cardholder inserts a card.
2. The system reads the card and prompts for a passcode.
3. The cardholder enters a passcode.
4. The system authenticates the cardholder.

$5'$. The system prints an Authentication-Failed message.

9. The system returns the card.

The connections between basic and alternative flows are through extension points, which are explained below. □

5.1.3 Extension Points and Alternative Flows

Extension points are named points between actions, just before the first action, or just after the last action of a flow. The name extension point is motivated by their use to attach additional behavior that extends the system.

Specific alternative flows attach between two named extension points, thereby creating alternative paths through the use case. *Bounded alternative flows* attach anywhere between two named extension points.

The next example illustrates specific alternative flows.

Example 5.4 : The following basic flow was formed by adding four extension points to the basic flow of Example 5.3. The extension points are between numbered actions and are in boldface.

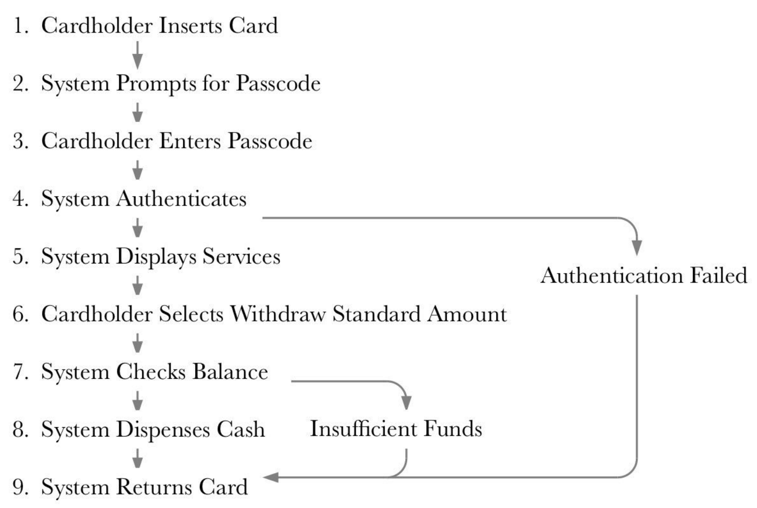

1. Cardholder Inserts Card
2. System Prompts for Passcode
3. Cardholder Enters Passcode
4. System Authenticates
5. System Displays Services
6. Cardholder Selects Withdraw Standard Amount
7. System Checks Balance
8. System Dispenses Cash
9. System Returns Card
Authentication Failed
Insufficient Funds

Figure 5.2: Basic and alternative flows for cash withdrawals from an ATM.

Basic Flow: Withdraw Cash, Version 2

1. The cardholder inserts a card.
   { **Read Card** }
2. The system reads the card and prompts for a passcode.
3. The cardholder enters a passcode.
4. The system authenticates the cardholder.
   { **Bank Services** }
5. The system displays options for bank services.
6. The cardholder selects Withdraw Standard Amount.
7. The system checks the account for availability of funds.
   { **Dispense Cash** }
8. The system dispenses cash and updates the account balance.
   { **Return Card** }
9. The system returns the card.

The falternative flow for failed authentication attaches between {**Bank Services**} and {**Return Card**}:

Alternative Flow 1: Authentication Failed

At {**Bank Services**} if authentication has failed
. Display "Authentication failed."
Resume the basic flow at {**Return Card**}

The alternative flow for insufficient funds attaches between {**Dispense Cash**} and {**Return Card**}:

Alternative Flow 2: Insufficient Funds
At {**Dispense Cash**} if account has insufficient funds
. Display "Insufficient funds in account."
Resume the basic flow at {**Return Card**}

The flows in this example correspond to paths through the flow graph in Fig. 5.2.
□

The next example illustrates bounded alternative flows, which attach anywhere between two named extension points.

Example 5.5: What if the network connection between the ATM and the server is lost? The loss of a network connection is an external event that can happen at any time; it is not tied to any specific point in the basic flow.

For simplicity, we assume that the actions in the basic flow are atomic. In particular, we assume that the action "Dispense cash and update account balance" is handled properly. A real system would use transaction processing techniques to ensure that the account balance reflects the cash dispensed.

With the basic flow and extension points from Example 5.3, the following bounded alternative flow returns the card if the network connection is lost before cash is dispensed:

Bounded Alternative Flow: Loss of Network Connection
At any point between {**Read Card**} and {**Dispense Cash**}, if the network connection is lost
. Display "Sorry, out of service."
Resume the basic flow at {**Return Card**}

□

5.2 Use-Case Diagrams

A *use case digram* summarizes the actors, the use cases, and the interactions between actors and use cases. As in Fig. 5.3, an actors is represented by a stick figure. A use case is represented by an ellipses labeled with the goal of the use case. Interactions are represented by arrows. The direction of an arrow indicates the initiator of an interaction. Note that an arrow represents a dialogue that potentially involves an exchange of messages. If there is no arrowhead, then either party can initiate the interaction.

5.2.1 A Diagram Provides a Big Picture Summary

Diagrams are intended for an overall understanding of a system with multiple use cases. Collectively, the goals of the use cases describe the purpose of the

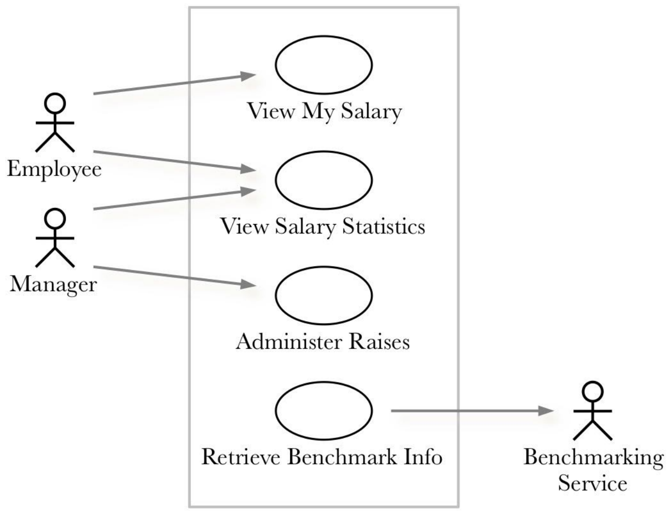


Figure 5.3: Use case diagram for a salary system.

system. During requirements discussions with stakeholders, diagrams are helpful for confirming that all the key stakeholders have been included and that no major goals have been missed.

Example 5.6: The diagram in Fig. 5.3 shows the actors and use cases for a salary system. The two primary actors are employee and manager.

Employees can view their own salaries by initiating the View My Salary use case. They can also initiate View Salary Statistics for perspective on how their salaries compare with similar roles both within their company and within their industry. Information about salaries outside the company is provided by a benchmarking service, which is shown as a secondary actor (on the right).

Managers can view salary statistics and can administer raises for the people they manage. The system can initiate the Retrieve Benchmark Info use case to get industry salary statistics from the Benchmarking Service actor. □

5.2.2 Use Case Diagrams in Practice

Use case diagrams are supported by UML (Unified Modeling Language), a standard language for visualizing and specifying software systems.[5]

Based on empirical studies of UML usage, it appears that use-case diagrams are not used by themselves. However, the concepts of use cases are widely used, and diagrams may be used in combination with textual descriptions. The following quotes are from one of the studies:

> "It's hard to design without something that you could describe as a use case."

"Many described use cases informally, for example, as: 'Structure plus pithy bits of text to describe a functional requirement. Used to communicate with stakeholders.'"[6]

5.3 Writing Use Cases

In discussions with customers and users, use cases are a tool for identifying their needs, goals, and requirements. With developers and testers, use cases guide decisions about what to implement. The right level of detail in a use case depends on the audience and the situation.

A fully described use case needs to be SMART (specific, measurable, achievable, relevant, time-bound); see Section 3.6.1 for the application of SMART criteria to user stories.

5.3.1 General Guidelines

The first guideline for writing a use case is: evolve it iteratively. Invest just enough time and provide just enough detail for the current audience and situation. An outline may be enough to get started. Add details as needed.

Jacobson and his colleagues lay down "six basic principles [that are] at the heart of any successful application of use cases."[7] The first three principles are guidelines for writing use cases:

1. *Keep it simple by telling stories.* Use simple textual descriptions for flows, in language that is meaningful to stakeholders. Accompany the descriptions with acceptance tests that ensure the relevance of the use case to the goal. Acceptance tests are also helpful for resolving ambiguities.

2. *Understand the big picture.* Use an overall view of the system's behavior to align with stakeholder goals and to guide design and implementation decisions and tradeoffs. Use-case diagrams accompanied by crisp snippets of text are helpful for an overall understanding.

3. *Focus on value.* Structure each use case to emphasize and quantify the value to the primary actor. Order the basic and alternative flows so that the simplest way of achieving the goal becomes the basic flow. Use value and simplicity to prioritize the alternative flows.

5.3.2 Conversational Form

A flow is typically written as a single sequence of actions, where actor actions and system responses are interleaved; as in the cash-withdrawal examples in Section 5.1. In *conversational form*, there is a column for actor actions and another for system responses:[8]

Example 5.7 : The following is a conversational form for the cash-withdrawal example in Section 5.1:

	Actor Action	**System Response**
1.	Insert card	Prompt for passcode
2.	Enter passcode	Authenticate cardholder
3.		Display Bank Services
4.	Select Withdraw Standard Amount	Check availability of funds
5.		Dispense cash
6.		Return Card

□

Conversational form is often used to highlight the most significant usability-related behaviors. Sequential form is more common; in sequential form, actor actions and system responses are interleaved. It is easier to use the sequential form when there is more than one actor. Furthermore, it is easier to add extension points to the sequential form of a use case.

5.3.3 User Intentions versus System Interactions

For perspective on the right level of detail in a use case, consider the distinctions between the three words intention, interaction, and interface, as in user intentions, system interactions, and user interfaces.

Intention is technology-free and implementation-free. Consider again a bank customer wanting to withdraw cash from an ATM. The customer's intention is to get cash. Most customers withdraw the same amount most of the time. As Larry Constantine puts it,

> "The bank customer wants to be able to say 'It's me. The usual. Thanks!' and be off."[9]

"It's me" is about the customer's identity and "The usual" is about withdrawing the usual amount of cash from the usual account; say, from the checking account.

An *essential use case* is a use case that is technology-free and implementation-free: it describes the user's intentions and the system's response. This definition of essential use cases does not specify either conversational or sequential form. However, essential use cases are closely associated with conversational form.[9]

Example 5.8 : The following is an essential use case for cash withdrawal in conversational form:

	Customer Intention	**System Response**
1.	Identify self	Authenticate customer
2.		Offer Bank Services
3.	Withdraw usual amount	Dispense cash

Exceptions and special cases need not be come up during a discussion about user intentions, so essential use cases need not have any alternative flows. □

Interactions are implementation-free. Although regular use cases are meant to be implementation-free, they may have some technology choices built into them.

Compare the regular use case in Example 5.7 with the essential use case in Example 5.8. The regular use case begins with, "Insert card" and ends with "Return card." The implicit assumption behind these actions is that identity will be authenticated using a card and a passcode. The essential use case does not make this assumption. What if identity were authenticated by fingerprint?

The regular use case also includes "Check availability of funds." This check is required, so it belongs in the interaction described by the use case. However, the check is internal to the system, so it need not be spelled out at the level of user intention.

The distinction between interaction and intention is therefore twofold. First, an interaction may make some technology assumptions, which an intention does not. Second, an interaction may include some internal system actions, which an intention does not.

Interfaces. The description of a user interface includes such design choices as the layout of buttons, text fields, and graphics, not to mention color schemes and fonts. Some of these design and implementation choices may creep into a use case, but by and large, use cases are implementation-free.

To summarize, user intention, system interaction, and user interface reflect three increasing levels of detail. User intention focuses on what the user wants, system interaction focuses on the system's behavior in response to a user's actions, and user interface focuses on design and implementation.

The resulting guideline for writing a use case is as follows:

- Start by capturing user intentions, perhaps as an essential use case.
- Once intentions solidify, write a use case that describes system interactions.
- Include user interface dependencies in a use case only as necessary for the audience and the situation.

5.3.4 A Template for Use Cases

There is no standard format for use cases. In its simplest form, a use case description may consist only of a goal and a basic flow. Even with more complex use cases, alternative flows may be represented only by their name or goal. Any details that are missing can be filled in later, as needed.

Templates for use case do exist, however; a representative template appears in Fig. 5.4. The first element in the template is a name for the use case. The name is preferably a short active phrase such "Withdraw Cash" or "Place Order." Next, if needed, is a brief description of the goal of the use case. If the name is descriptive enough, there may be no need to include a goal that says essentially the same thing.

Name:	Active phrase for what will be achieved for the actor(s)
Goal:	Brief description of the purpose of the use case
Actors:	Every role, human and automated, in the use case
Basic Flow:	Sequence of actions—text, readable by stakeholders
Alternative Flows:	Variations on the basic flow
Extension Points:	Insertion points in the flow for additional behavior
Preconditions:	State of system for the use case to operate correctly
Postconditions:	State of the system after the use case completes
Relationships:	Possible communication with other use cases

Figure 5.4: A representative template for writing use cases.

Use cases for real systems can have multiple actors, so it is worth listing the actors, human or automated.

The basic flow is the heart of a use case. It is required. A basic flow begins with an actor action. The first action typically triggers the use case; that is, it initiates the use case.

The basic flow is followed by alternative flows. Long use cases are hard to read, so alternative flows are often identified only by their names or goals. In the early stages of a project, the only available information about an alternative flow may be its goal. Since requirements can change, alternative flows need not be fleshed out until they are needed.

Alternative flows must be about optional, exceptional, or truly alternative behavior. If the behavior is required, it belongs in the basic flows. Furthermore, if the behavior is not conditional, it is not alternative behavior and does not belong in the use case. It may belong in some other use case.

Alternative flows, both specific and bounded, attach to a basic flow at named extension points. In Section 5.1, the alternative flows for failed authentication and for insufficient funds attached at named extension points in the basic flow for the Withdraw Cash use case.

Preconditions are assertions that must be true for the use case to be initiated. For example, a Cancel Order use case may have a precondition that an order exists or is in progress. If there is no order, there is nothing to cancel. Similarly, postconditions are assertions that must be true when the use case ends. In simple examples, preconditions and postconditions are often omitted.

Relationships between use cases are discussed in the next section.

5.4 Relationships Between Use Cases

5.4.1 Subflows

Even for simple systems, the readability of flows can be enhanced by defining subflows: a *subflow* is a self-contained subsequence with a well defined purpose. The logic and alternatives remain tied to the basic flow if subflows are linear, where all the actions are performed or none of them are.

A subflow is *private* to a use case if it is invoked only within that use case.

5.4.2 Inclusion

Subsequences, again with a well-defined purpose, that are common to multiple use cases can be handled by *inclusion* of one use case within another. The inclusion appears as an action within a basic flow; that is, the basic flow invokes the included use case at some point and then resumes at that point.

In the ATM example in Section 5.1, authentication was treated as an action. In practice, authentication involves subactions such as verifying a passcode and logging the event. Authentication will likely be required for other use cases as well; e.g., for transferring money between accounts. The following sequence includes a use case, *Authenticate Cardholder*

1. Start the use case when the actor inserts a card
2. Read the card
3. Include the use case *Authenticate Cardholder*
4. Upon authentication, display options for bank services

5.4.3 Extensions

A use case that is complete by itself can be augmented through a form of inheritance. Suppose that a use case U has an extension point p within its basic flow. A use case V can specify that it is to be performed at the point p in U. The use case V is then called an *extension* of U.

The purpose of an extension is to support an additional goal for an actor.

The distinction between inclusion and extension is that a basic flow is unaware of an extension, but it knows about the inclusion and explicitly calls the included use case by name. For an extension, the basic flow provides extension points, however, it does not know the use cases that take advantage of those extension points. Conversely, an inclusion is unaware that it is being included in some basic flow, but an extension knows about the basic flow and explicitly specifies the extension point where it is to be invoked.

For an example of an extension, suppose that the basic flow for a use case *Place Order* includes an extension point {**Display Products**}:

Get product category entered by customer
{ **Display Products** }
Display products and prices

An extension use case, *Social Place Order*, can specify the extension point {**Display Products**}, where it is invoked to add recommendations based on the product category and friend data from Facebook.

As another example, consider a surveillance system with a basic flow to monitor an area for intruders. Another use case notifies authorities. Surveillance and notification can be combined by making notification an extension of the surveillance system. At an extension point {**Intruder Detected**} in the surveillance flow, the notification system can be invoked to notify the authorities that an intruder has been detected.

Most systems can be described by collections of self-contained use cases without inclusions and extensions.[10]

5.5 Conclusion

Summary of Use Cases

A well written collection of use cases and their goals provides an overview of the requirements for the system. Each use case is built around a basic flow, which is a single successful sequence of actions. The alternative flows in a use case handle exceptions and special cases.

The full use case does not need to be fleshed out up front. The list of use cases and their goals are helpful in early discussions with stakeholders to ensure that no major goal is overlooked. Basic flows provide the next level of detail. Alternative flows can be developed as needed during the project. They serve as backup in case stakeholders have detailed questions.

Use Cases and User Stories

Use cases and user stories are complementary; they provide different perspectives on requirements and can be used together. A user story corresponds to an individual feature, whereas a use case is closer to a collection of related user stories. The tradeoff is between between use cases and user stories is between context and weight: use cases provide context for system behaviors and user stories are lighter weight, which means that they require less effort.[11]

Use Cases and Iterative Development

Jacobson and his colleagues provide three guidelines for iterative software development based on use cases:[12]

- *Build the system in slices.* Instead of implementing an entire use case all at once, consider slicing it, where a "slice" is a subset of the flows in the use case, along with their test cases. Alternatively, a slice corresponds to a set of paths through a flow graph, such as the one in Fig. 5.2.
- *Deliver the system in increments.* Use an iterative process, based on delivering slices of use cases. Begin with the slice or slices that provide the most value.
- *Adapt to meet the team's needs.* Fit the development process to the project. A small cohesive team in a close collaboration with stakeholders, might document just the bare essentials of use cases, relying on informal communication to address any questions along the way. A large team would likely require documented use cases with key details filled in.

Exercises for Chapter 5

Exercise 5.1 : Write a full use case for the insurance company scenario from Exercise 3.4.

Exercise 5.2 : Write a use case for the software to control a self-service gasoline pump, including handling payments, choice of grade of gas, and a receipt. In addition, when the screen is not being used otherwise, the system must permit targeted advertising on the screen, where the targeting is based on the customer's purchase history with the gas vendor.

Your use case must include the following.

a) A basic flow

b) Extension points

c) At least one specific alternative flow

d) At least one bounded alternative flow

e) Inclusion

In each case, explain how the use case illustrates the relevant concept. For alternative flows, include the full flow, not just the name of the flow.

Exercise 5.3 : Prior to meeting with the customer, all you have is the following brief description of a proposed system:

> The system will allow users to compare prices on health insurance plans in their area; to begin enrollment in a chosen plan; and to simultaneously find out if they qualify for government healthcare subsidies. Visitors will sign up and create their own specific user account first, listing some personal information, before receiving detailed information about the plans that are available in their area.[13]

Write a use case based on this description.

Exercise 5.4: HomeAway allows a user to rent vacation properties across the world. It has multiple web sites that support user interaction in different languages. Write a use case for a renter to select and reserve a vacation property for specific dates in a given city.

Exercise 5.5: Write a use case for an airline flight-reservations system. For cities in the United States, the airline either has nonstop flights or flights with one stop through its hubs in Chicago and Dallas. Another team is responsible for the pricing system, which determines the price of a round-trip ticket, based on the source, destination, departure time, and frequent flier status of the passenger. Your reservations system is responsible for offering flight options (there may be several options on a given day), seat selection, method of payment (choice of credit card or frequent flier miles).

Your use case must include the following.

a) A basic flow
b) Extension points
c) A specific alternative flow
d) A bounded alternative flow
e) Inclusion

In each of the above cases, briefly explain the concept and show the part of the use case that illustrates the concept. For alternative flows, include the full flow, not just the name of the flow.

Exercise 5.6: Write a use case for the software to send a text message between two mobile phones, as described below.

> Each phone has its own Home server, determined by the phone?s number. The Home server keeps track of the phone?s location, billing, and communication history. Assume that the source and destination phones have different Home servers. The destination Home server holds messages until they can be delivered. Also assume that the network does not fail; that is, the phones stay connected to the network.

Your use case must include the following.

a) A basic flow
b) Extension points
c) At least one specific alternative flow
d) At least one bounded alternative flow

In each case, explain how the use case illustrates the relevant concept. For alternative flows, include the full flow, not just the name of the flow.

Notes for Chapter 5

[1]Jacobson, Spence, and Kerr [8] provide guidelines for software development based on use cases.

[2]Jacobson, Spence, and Kerr [8] recommend that the system be built incrementally, slice by slice, where a slice corresponds to the work items for an iteration.

[3]Use cases were first presented at OOPSLA '87 [6]. Jacobson [7] provides a retrospective.

[4]The ATM use case in Example 5.3 is based on a fully worked out use case in Bittner and Spence [1], which Ivar Jacobson, the inventor of use cases, called "*THE* book on use cases" [7].

[5]Grady Booch, Ivar Jacobson, and James Rumbaugh [2] created UML in the mid 1990s. It was adopted as a standard by Object Management Group (OMG) in 1997 and by the International Standards Organization (ISO) in 2005.

[6]Petre [9, p. 728] conducted "interviews with 50 professional software engineers in 50 companies and found 5 patterns of UML use," ranging from no use of UML (70%) to selective use (22%) and wholehearted use (0%). Selective users mentioned informal use of use cases. Only one of the 50 developers found use case diagrams to be useful.

[7]Jacobson, Spence, and Kerr [8]

[8]Rebecca Wirfs-Brock [10] proposed conversational form to "clearly demarcate actor actions and system responses.

[9]Larry Constantine and Lucy Lockwood [4] propose that user-interface design be based on a model of user intentions; specifically, on essential use cases.

[10]In a chapter entitled "Here There Be Dragon," Bittner and Spence [1] note that the behavior of most systems can be specified without inclusions and extensions. They also note, "If there is one thing that sets teams down the wrong path, it is the misuse of the use-case relationships."

[11]For a discussion of use cases and user stories, see Cockburn [3] and the accompanying comments.

[12]Jacobson, Spence, and Kerr [8]

[13]The description in Exercise 5.3 is adapted from the "Background and functionality" section of the Wikipedia entry for HealthCare.gov.
`https://en.wikipedia.org/wiki/HealthCare.gov` . Text used under the Creative Commons CC-BY-SA 3.0 license.

References for Chapter 5

1. Kurt Bittner and Ian Spence. *Use Case Modeling* Addison-Wesley Professional (2003).

2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, 2nd Ed.* Addison-Wesley Professional (2005).

3. Alistair Cockburn. Why I still use use cases. (January 9, 2008).
`http://alistair.cockburn.us/Why+I+still+use+use+cases` .

4. Larry L. Constantine. Essential modeling: use cases for modeling user interfaces. *ACM Interactions* 2, 2 (April 1995) 34-46.

5. John Erickson. A decade or more of UML: an overview of UML semantic and structural issues and UML field use. *Journal of Database Management* **19**, 3 (2008) i-vii.

6. Ivar Jacobson. Object oriented development in an industrial environment. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (October 1987) 183-191.

7. Ivar Jacobson. Use cases: yesterday, today, and tomorrow. *Software and Systems Modeling* (2004) 210-220.

8. Ivar Jacobson, Ian Spence, and Brian Kerr. Use-Case 2.0: The hub of software development. *ACM Queue* 14, 1 (January-February 2016) 94-123.

9. Marian Petre. UML in practice. *International Conference on Software Engineering (ICSE '13)*. (2013) 722-731.

10. Rebecca Wirfs-Brock. Designing scenarios: making the case for a use case framework. *The Smalltalk Report* 3, 3 (November-December 1993). `http://wirfs-brock.com/PDFs/Designing Scenarios.pdf`.

Chapter 6

Estimation

"Overwhelming evidence documents a tendency toward cost and effort overruns in software projects. On average, this overrun seems to be around 30 percent. Furthermore, comparing the estimation accuracy of the 1980s with that reported in more recent surveys suggests that the estimation accuracy hasn't changed much since then."

— *Magne Jørgensen, assessing the state of the art of estimation in 2014. Estimation is a key activity during project planning.*[1]

During project planning, size and effort estimates guide decisions about schedules, budgets, work assignments, team sizes, required skills, and other resources.

A *project* is a set of activities with a start, a finish, and deliverables, subject to schedule and cost constraints. The deliverables from a project can be anything: a product, like a web browser; an event, like landing a rover on Mars; a service delivered from the cloud; an assessment; ...[2]

Project management consists of planning, organizing, tracking, and controlling a project, from initial concept through final delivery. *Project planning* includes the selection of a software development process; work assignment; size, schedule, and cost estimation; risk assessment; and quality planning.[3]

6.1 Introduction

The selection of a software-development process is an early decision during project planning. The development process guides customer interactions, team coordination, work distribution, progress monitoring, and course corrections.

There are two approaches to project planning: predictive and adaptive.

6.1.1 Predictive Planning

Predictive planning is geared to plan-driven processes. Such planning is up-front, before design, coding, and testing. In the early stages of a project, there is great uncertainty about the customer problem and about potential solutions. Predictive planning therefore goes to great lengths to create the best possible plans based on the information that is available at the time.

Unfortunately, requirements change. Design and performance issues crop up. The implemented solution can therefore diverge not only from the planned functionality, but from the planned schedule and budget.

6.1.2 Adaptive Planning

Adaptive planning is geared to iterative processes, especially agile processes. Instead of planning the whole project up front, amid great uncertainty, adaptive planning is spread evenly across the project. For each iteration, there is just enough planning to cover that iteration. Adaptive plans need not be perfect, since iterations are short and plans can be adapted at the end of an iteration.

In other words, an agile approach can be applied to planning itself, by doing it an iteration at a time.

Adaptive planning achieves two goals at the same time: it improves planning accuracy and reduces planning effort. Plans improve because they adapt as the project progresses. The plan for the current iteration can benefit from information from past iterations. The overall planning effort is reduced as follows. It is easier to plan for the next 2-4 weeks than it is to plan for the next 12-18 months. With short planning horizons, the sum of the incremental planning efforts adds up to less than the effort for careful up-front predictive planning.

6.1.3 The Role of Estimation

Planning involves estimation of the size and effort required for a project. Estimation begins with questions like the following: Is the project small, medium, or large? Does it require special skills? Is there a hard deadline? Has something like this been done before?

During adaptive planning, estimation is spread across iterations. The selection of work items for an iteration is based on a combination of (1) value to the customer and (2) estimated software development effort. Fortunately, rough estimates suffice for assigning high, medium, or low priority to work items.

Despite all the attention it has received, estimation remains more art than science. The state of the art of estimation for software development can be summarized as follows:[4]

- Historical data about similar past projects is a good predictor for current projects.

- Simple models tailored to the local work environment can be at least as if not more accurate than advanced statistical models.
- Estimation accuracy can be improved by combining independent estimates from a group of experts.
- Most estimation techniques ultimately rely to a lesser or greater extent on expert judgment.
- There is no one best estimation model or method.

People are better at estimating relative magnitude than they are at estimating absolute magnitude. Given work items A and B, it is easier to estimate whether A requires more or less effort than B, or whether A is simpler or more complex than B. It is harder to estimate the number of staff-days or the code size needed to implement either A or B.

6.2 Planning Constraints

The Mythical Man Month, Fred Brooks's influential series of essays on software project management, begins with

> "In many ways, managing a large computer programming project is like managing any other large undertaking—in more ways than most programmers believe. But in many other ways it is different—in more ways than most professional managers expect."[5]

The similarities between software projects and other large undertakings have to do with people and organizations. With respect to the framework for software engineering in Fig. 1.1,

$$\begin{tikzcd} & \textbf{Customers} \arrow[dl, dash] \arrow[dr, dash] & \\ \textbf{Teams} \arrow[dr, dash] & & \textbf{Technology} \arrow[dl, dash] \\ & \textbf{Context} & \end{tikzcd}$$

the similarities have to do with team management, customer interactions, and the organizational context. The differences relate to the nature of the technology: software is complex; software appears readily changeable (on the surface).

This section begins with the Iron Triangle, which applies to any project, and goes on to the Adaptive Iron Triangle, which applies to time-boxed iterative software projects.

6.2.1 The Iron Triangle

The *Project Management Triangle* (also known as the *Iron Triangle*) illustrates the traditional constraints faced by any project: scope, time, and cost;[6] see Fig. 6.1. Scope refers to two things: (1) the functionality to be delivered or the customer requirements to be met by the project; and (2) quality attributes

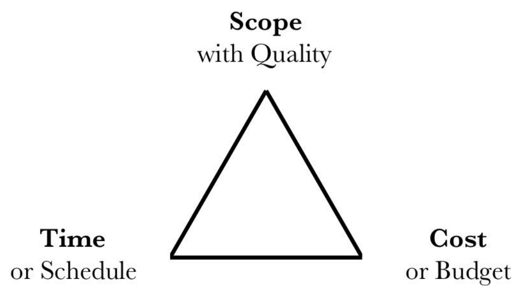


Figure 6.1: The Project Management Triangle, with constraints at the vertices. A variant of this diagram attaches the constraints to the edges of the triangle, instead of the vertices.

such as correctness (few defects), security, performance, and availability. The two vertices at the base of the triangle represent time or schedule constraints and cost or budget constraints.

The edges of the triangle represent the connections between the constraints. For example, if scope increases, then time and or cost are bound to be affected. Similarly, if time or cost are reduced, then scope is bound to be affected. Schedule overruns have typically been accompanied by cost overruns or scope reductions.

In practice, there is rarely enough time or budget to deliver the full scope with quality. The challenge of meeting the triple constraints simultaneously has led to the quip, "Time. Cost. Scope. Pick any two!" An alternative version is, "Fast. Cheap. Good. Pick and two!"

Example 6.1 : The challenges represented by the Iron Triangle are illustrated by the experience of a company that will remain nameless.

In a rush to get new products to market, the executives of Company X pressed its development teams for a 15% reduction in project schedules, compared to similar past projects. The budget remained the same. The teams responded by spending fewer days, on average, on every activity: design, coding, verification.

Once the new products were released, it became evident that scope and quality had suffered. Early customers complained about missing features and product defects. The company reacted to the trouble reports by issuing upgrades to improve the products that had already been delivered. Eventually, the problems with the products did get fixed, but the company's reputation had been tarnished.

In a bid to repair its reputation, the company prioritized quality over schedule for its next set of projects. □

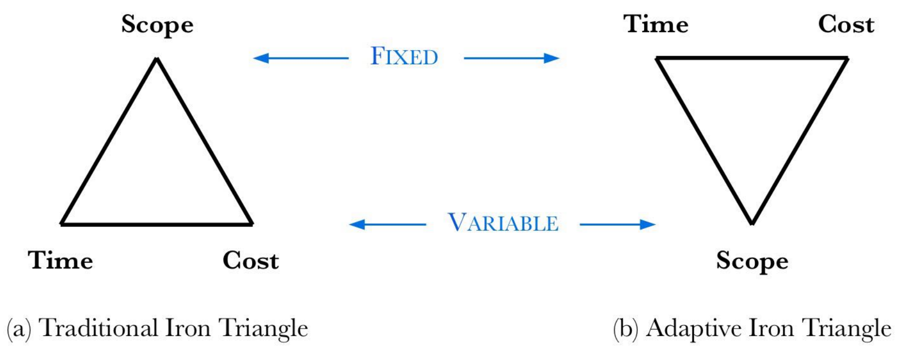


Figure 6.2: Project Management Triangles for two development processes.

6.2.2 The Adaptive Iron Triangle

The variants of the Iron Triangle in Fig. 6.2 illustrate two approaches. The triangle in Fig. 6.2(a) is for a traditional (say, waterfall) process that prioritizes scope. It fixes the scope and allows the time and cost to vary. Such a project is run until the full scope can be delivered, even if there are schedule and cost overruns.

The inverted triangle in Fig. 6.2(b) is for a time-boxed iterative process. With time-boxed iterations, time and cost are fixed; scope varies. Assuming that quality is a given, "scope varies" means that functionality varies. Lower priority features that cannot be completed in an iteration are dropped from that iteration. The completed scope grows with each iteration.

An inverted triangle that fixes time and cost, as in Fig. 6.2(b), is called an *Adaptive Iron Triangle*.

Many variants of the Iron Triangle have been proposed to illustrate different project constraints. The double triangle in Fig. 6.3 adds a second triangle with risk, resources, and quality at its vertices.[7] Risk reduction will be discussed in Section **??**. Resources refers to the skills, facilities, and equipment needs of a project. Sometimes "resources" is used as a euphemism for headcount.

In Fig. 6.3, quality has been separated out from scope. When scope and quality are separated, scope refers to functionality and quality refers to defects. Depending on the project, attributes such as security and performance may be included under either scope or quality.

6.3 Anchoring and Cognitive Bias

Before considering estimation techniques that rely on expert judgment, we pause to reflect on the nature of human judgment itself.

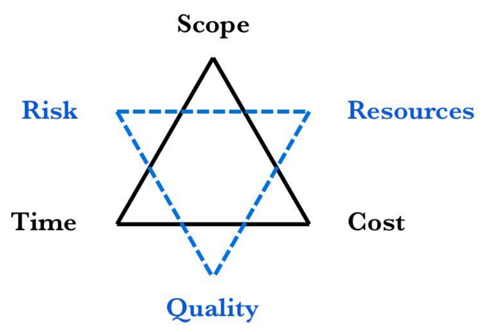


Figure 6.3: A variant of the Project Management Triangle.

Amos Tversky and Daniel Kahneman introduced the term *cognitive bias* for the human tendency to make systematic errors in judgment under uncertainty. They defined *anchoring* as follows:

> "In many situations, people make estimates by starting from an initial value that is adjusted to yield the final answer. ... different starting points yield different estimates, which are biased toward the initial values. We call this phenomenon anchoring."[8]

Anchoring has implications for both predictive and adaptive planning. Simply stated, avoid anchoring during estimation. In order to get unbiased estimates, do not convey customers or management expectations about effort, schedule, or budget. As the next example indicates, including an anchor point in customer requirements or developer instructions can lead to estimates that are very different from the estimates that would be produced without the anchor.

Example 6.2 : Participants in a case study were asked to estimate the time it would take to deliver a software application.[9] Each participant was given a 10-page requirements document and a 3-page "project setting" document. The project setting document had two kinds of information: (1) a brief description of the client organization, including quotes from interviews; and (2) background about the development team that would implement the application, including the skills, experience, and culture of the developers.

The 23 participants were divided into three groups. The only difference between the instructions to the three groups was a quote on the second page of the project setting document; see Fig. 6.4. The quote was supposedly from a middle manager.

Group 1, the control group, got a quote with no mention of the time it might take to develop the application. Group 2 got a quote with a low anchor, 2 months. Group 3 got a high anchor, 20 months.

- **Group 1 was given a quote with no mention of time**
  - "I admit I have no experience estimating."
- **Group 2 got a quote that mentioned 2 months.**
  - "I admit I have no experience with software projects, but I guess this will take about 2 months to finish."
- **Group 3 got a quote that mentioned 20 months.**
  - "I admit I have no experience with software projects, but I guess this will take about 20 months to finish."

Figure 6.4: Case study of the effect of anchoring on software effort estimation.

The results confirmed the phenomenon of anchoring. The mean development-time estimates from the three groups were 8.3 months for the control group, 6.8 months for the 2-month group, and 17.4 months for the 20-month group. The mean estimates from the experienced participants in the three groups were 9 months, 7.8 months, and 17.8 months, respectively. The differences between the low-anchor and high-anchor groups are clear: 7-8 months versus 17-18 months. Further studies would be needed to explain the small differences between the control group and the low-anchor group. □

6.4 Estimation Uncertainty

In the early stages, while a project is still being defined, any estimates can at best be informed guesses, sometimes called *guesstimates*. Using historical data from similar projects, guesstimates can be narrowed by setting upper and lower bounds for size and effort.

During estimation, the range between a pair of upper and lower bounds is a measure of uncertainty. Uncertainty takes three main forms:

- *customer-related*, linked to emerging or changing requirements;
- *solution-related*, linked to the unfinished design and implementation; and
- *team-related*, linked to the skills and experience of the development team.

6.4.1 Cone of Uncertainty for Predictive Planning

Estimation uncertainty decreases as a project progresses and more information becomes available. That is, the range between upper and lower bounds narrows. This intuition motivates the dashed lines in Fig. 6.5. The range between the upper and lower dashed lines represents the inherent or unavoidable uncertainty at a given phase of a waterfall process. In the figure, the range is 0.5x-2.0x at

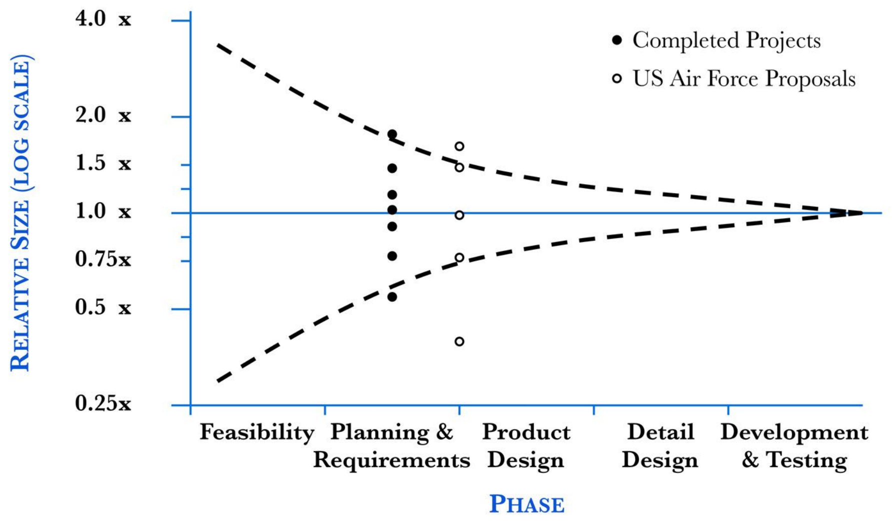


Figure 6.5: The Cone of Uncertainty illustrates the intuition that uncertainty decreases as a plan-driven project progresses.

the end of the Feasibility phase. The range narrows to 0.75x-1.5x at the end of the Planning & Requirements phase. In other words, the best we can do at the end of the Planning & Requirements phase is a size estimate that is between 75% and 150% of the size of the eventual completed system.

The diagram in Fig. 6.5 has been dubbed the *Cone of Uncertainty*, after the shape enclosed by the dashed lines. The Cone of Uncertainty was introduced by Barry Boehm as a conceptual diagram. Later, Boehm and his colleagues provided some supporting data that is represented by the dots in Fig. 6.5.[10]

The filled dots reflect estimates from completed projects; the estimates were based on "partially defined specifications." The spread of the filled dots is from 0.5 to 2.0; that is, these estimates based on partially defined specifications were accurate to within a factor of 2.

The unfilled dots reflect estimates from proposals submitted to the US Air Force Electronic Systems Division. The proposals were based on "a fairly thorough specification."

6.4.2 Levels of Uncertainty During Adaptive Planning

The shorter the planning horizon, the lower the estimation uncertainty. The ovals in Fig. 6.6 represent levels of uncertainty at various stages during adaptive planning. The smaller the oval, the lower the uncertainty.[11]

A project involves one or more releases. Each release is the result of a sequence of iterations. During each iteration, there is day-to-day planning;

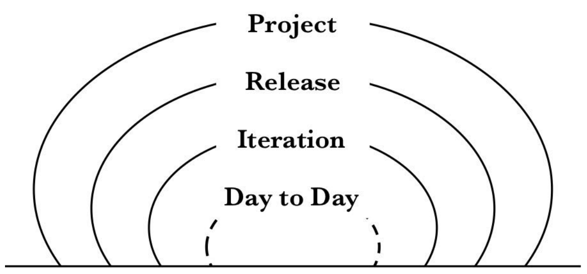


Figure 6.6: Levels of uncertainty during adaptive planning.

e.g., see the discussion of daily scrums in Section 3.4. The planning horizon for a project may be several months, the horizon for a release may be a few months, and the horizon for an iteration may be 1-4 weeks. The shorter the planning horizon, the lower the uncertainty.

At the project level, there are likely to be broad goals, such as "build a child-friendly app for a hospital" or "calculate the solar-power potential of a rooftop," accompanied by high-level constraints on the schedule and budget. The child-friendly app may be needed by a certain date. The project to display solar-power maps may be abandoned if similar previous projects cost significantly more than the projected budget.

Planning for agile projects is typically focused on release and iteration planning. Estimation planning for agile iterations is discussed in Section **??**. The oval for day-to-day planning is dashed, since day-to-day planning consists of small adjustments within he encompassing plans. Iteration planning on the other hand creates the plan for the next iteration adaptively, while the project is underway.

When an inner plan changes, the enclosing plans are adapted to reflect the change.

6.4.3 Three-Point Estimation

Instead of a single value or a range, *three-point estimation* involves three values: b, corresponding to the best case; m corresponding to the most likely case; and w, corresponding to the worst case. These values are then combined by taking a weighted average, using the formula

$$\textit{estimate} \ = \ (b + 4m + w)/6$$

This formula is based on a statistical model developed for time estimates in conjunction with an operations research technique called PERT. The assumptions behind the formula are as follows:[12]

- Estimates b, m, and w for a work item are independent of the other work items in the project. For example, if module M uses module N, then the estimate for M must be independent of N and any other module.

- Estimates b, m, and w are independent of schedule, budget, resource, or other project constraints. Furthermore, the estimates are assumed to be independent of anchoring or cognitive bias.
- Technically, the best case b is assumed to be the 95th percentile case and the worst case w is assumed to be the 5th percentile case.

In practice, the weighted-average formula is roughly right, even though the assumptions do not fully hold. Some bias is unavoidable. Estimates are not exact, so there are no guarantees that the best and worst case estimates correspond to the 95th and 5th percentile cases.

6.5 Collective Judgment

Under the right conditions, the consensus estimate from a group can be more accurate than individual expert judgment. The notion of group wisdom or collective judgment dates back to Aristotle:

> "the many, of whom each individual is an ordinary person, when they meet together may very likely be better than the few [experts] ... for some understand one part, and some another, and among them they understand the whole."[13]

Example 6.3 : Consider the experience of Best Buy in estimating the number of gift cards they would sell around Christmas. The estimates from the internal experts proved to be 95% accurate. The consensus estimates from 100 randomly chosen employees proved to be 99.9% accurate. Both groups began with the actual sales number from the prior year.[14] □

Group estimation techniques differ in how they address two issues:

- *Avoiding cognitive bias.* Groups are subject to anchoring and groupthink, where some members are swayed and adapt to the group rather than giving their own opinion. Structured estimation techniques begin with each member independently supplying an initial estimate.
- *Converging on a consensus.* How are the various estimates by the group members combined into a consensus estimate? For example, the "consensus" may be a weighted average, the median, or a true consensus that emerges from structured group interaction.

6.5.1 The Original Delphi Method for Group Consensus

The Delphi method is a technique for reaching group consensus, while avoiding cognitive bias, as much as possible. Its designers were concerned that simply bringing a group together for a roundtable discussion

	Round 1	2	3	4
Expert 1	1,000	525	332	349
2	200	256	300	292
3	300	250	250	276
4	150	184	200	206
5	125	158	166	167

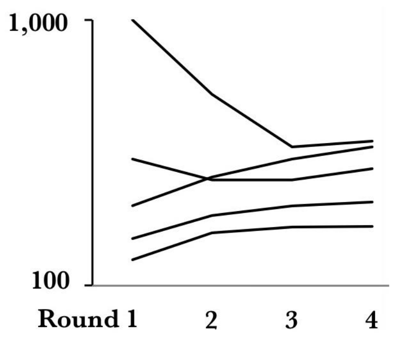


Figure 6.7: An application of the Delphi method with 5 experts. The same data appears in both tabular and graphical form.

> "induces the hasty formulation of preconceived notions, an inclination to close one's mind to novel ideas, a tendency to defend a stand once taken or, alternatively and sometimes alternately, a predisposition to be swayed by persuasively stated opinions of others."[15]

Rounds of Estimates and Feedback

With the original Delphi method, the group members were kept apart and anonymous. Instead of direct contact with each other, they were provided with feedback about where their estimate stood, relative to the others.

Consensus was achieved by having several rounds of estimates and feedback.

Example 6.4: The data in Fig. 6.7 is adapted from a forecasting exercise in the 1950s. The same data appears in tabular form on the left and in graphical form on the right. The data is for four rounds of forecasts submitted by a group of five experts.

The first round forecasts range from a low of 125 to a high of 1,000. The median forecast for the first round is 200. In the second round, the range of forecasts narrows from a low of 158 to a high of 525. The ranges for the third and fourth rounds are close: 166-332 and 167-349, respectively.

Note that the group is converging on a range, not on a single forecast. With forecasts, it is not unusual for experts to have differences of opinion. □

Examples of Feedback Between Rounds

in different forecasting exercises, the designers experimented with different forms of feedback.[16] For example, the feedback might consist of (1) the median of the group estimates; (2) a weighted average of the group estimates; (3) a range of estimates, after dropping outliers (for outliers to be dropped, the number of estimates has to be large enough).

In the next example, the feedback consists of the median and a range.

Example 6.5 : In one version of the Delphi method, the feedback consisted of the median and the range of the middle 50% of the group estimates. In other words, 25% of the estimates were below the range and 25% were above the range.

For the next round, a group member who stayed with an estimate that was outside the middle 50% was asked to provide the reasons for submitting an estimate that was much higher or much lower than the estimates from the rest of the group. □

The Delphi method was also used for planning exercises. The next example considers the allocation of a fixed budget, based on cost-benefit estimates for a list of items. Iteration planning can also be thought of as the allocation of a time budget, based on effort-benefit estimates for user stories or work items.

Example 6.6 : In an application of the Delphi method in the mid-1960s, a group of educators was given a fixed budget and rough cost estimates for a list of potential educational innovations. Working individually, each educator did an intuitive cost-benefit appraisal of each item on the list.

The moderators then synthesized the individual opinions into a summary and provided the summary as feedback to the group members. Delphi rounds were then used to reach a group consensus. □

In practice, a majority of applications of the Delphi method resulted in group consensus. Even in cases where the rounds were stopped before the group's opinions had converged, the method was helpful in clarifying the issues and highlighting the sources of disagreement, leading to better decisions.[17]

6.5.2 The Wideband Delphi Method

For software projects, group discussion can lead to valuable insights that can be useful during design, coding, and testing. Any concerns can be recorded and addressed, pitfalls avoided. A shortcoming of the original Delphi method is that it isolates group members so it prevents group discussion. The isolation softens anchoring and cognitive bias, but, perhaps, the benefits of group discussion outweigh the risks of bias.

The Wideband Delphi method, outlined in Fig. 6.8, combines anonymous individual estimates with group discussion between rounds. It has been used successfully for both predictive planning[18] and for adaptive planning.

Planning Poker is a Variant of Wideband Delphi

A variant of the Wideband Delphi method, called *Planning Poker* has been applied to estimation for agile projects. In Planning Poker, participants are given cards marked with Fibonacci story points $1, 2, 3, 5, 8, ...$ Each developer

```
repeat
    Participants anonymously submit individual estimates
    if the estimates have converged enough
        done
    else
        The moderator convenes a group meeting to discuss outliers
```

Figure 6.8: The Wideband Delphi Method.

independently and privately picks a card, representing their estimate for the story under discussion. All developers then reveal their cards simultaneously. In the unlikely event that the individual estimates are close to each other, a consensus has been reached. More likely, there will be some high cards and some low cards, with the others being in the middle. The developers with the high and low cards may have good reasons for assigning high or low story points. Or, it may be that they are missing something.

Following some group discussion, the developers then do a second round of making independent estimates, which they reveal simultaneously. If a consensus has not been reached, then the group engages in further discussion and rounds of making independent estimates.

A moderator captures key comments from the group discussion of a story, so that the comments can be addressed during implementation and testing.[19]

6.6 Empirical Models Based on Historical Data

Estimation models rely on historical data to estimate future development effort. Effort estimates are then used to address questions about how long a project will take and how much it will cost.

The "cost" of a work item can be represented by either the estimated program size or the estimated development effort for the item. Size and effort are related, but they are not the same. As size increases, effort increases, but by how much?

The challenge is that, for programs of the same size, development effort can vary widely, depending on factors such as team productivity, problem domain, and system architecture. Team productivity can vary by an order of magnitude.[20] Critical applications require more effort than casual ones. Effort increases gradually with a loosely coupled architecture; it rises sharply with tight coupling.

Project managers can compensate for some of these factors, further complicating the relationship between size and effort. Experienced project managers can address productivity variations when they form teams; say, by pairing a novice developer with someone more skilled. Estimation can be improved by

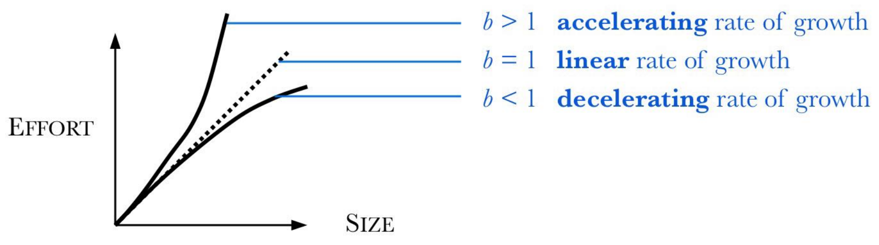

$b > 1$ **accelerating** rate of growth
$b = 1$ **linear** rate of growth
$b < 1$ **decelerating** rate of growth
EFFORT
SIZE

Figure 6.9: How does effort grow with size?

relying on past data from the same problem domain; for example, smartphone apps can be compared with smartphone apps, embedded systems with embedded systems, and so on. Design guidelines and reviews can lead to cleaner architectures, where effort scales gracefully with size.

The relationship between size and effort is therefore context dependent. Within a given context, however, historical data can be used to make helpful predictions.

For large projects or with longer planning horizons, it is better to work with size. Iteration planning, with its short 1-4 week planning horizons, is often based on effort estimates. The discussion in this section is in terms of effort—the same estimation techniques work for both size and effort.

6.6.1 Estimating Effort from Size

Consider the problem of estimating development effort E from program size S. In other words, determine a function f such that

$$E = f(S)$$

If f is a linear function, then estimated effort E is proportional to size S: if size doubles, then effort doubles.

Effort does indeed increase with size, but not necessarily linearly. Does it grow faster than size, as in the upper curve in Fig. 6.9? Or does it grow slower than size, as in the lower curve? The dashed line corresponds to effort growing linearly with size.

The three curves in Fig. 6.9 were obtained by picking suitable values for the constants a and b in the equation

$$E = aS^b \tag{6.1}$$

The curves in Fig. 6.9 correspond to the following cases:

- *Case* $b = 1$ (dashed line). The function in Equation (6.1) is then linear.

- *Case* $b < 1$ (lower curve). This case corresponds to there being economies of scale; for example, if the team becomes more productive as the project proceeds, or if code from a smaller project can be reused for a larger project.
- *Case* $b > 1$ (upper curve). The more usual case is when $b > 1$ and larger projects become increasingly harder, either because of the increased need for team communication or because of increased interaction between modules as size increases. In other words, the rate of growth of effort accelerates with size.

6.6.2 The Cocomo Family of Estimation Models

Equation (6.1), expressing effort as a function of size is from a model called Cocomo-81. The name Cocomo comes from Constructive Cost Model. The basic Cocomo model, introduced in 1981, is called Cocomo-81 to distinguish it from later models in the Cocomo suite.[21]

For a given project, the constants a and b in Equation (6.1) are estimated from historical data about similar projects. IBM data from waterfall projects in the 1970s fits the following (E is effort in staff-months and S is in thousands of lines of code):[22]

$$E = 5.2\, S^{0.91} \tag{6.2}$$

Meanwhile, TRW data from waterfall projects fits the following (the three equations are for three classes of systems):[23]

$$\begin{array}{ll} E = 2.4\, S^{1.05} & \text{Basic Systems} \\ E = 3.0\, S^{1.12} & \text{Intermediate} \\ E = 3.6\, S^{1.20} & \text{Embedded Systems} \end{array} \tag{6.3}$$

The constants in (6.3) can be adjusted to account for factors such as task complexity and team productivity. For example, for a complex task, the estimated effort might be increased by 25% (the actual percentage depends on historical data about similar projects by the same team). Such adjustments can be handled by picking suitable values for the constants a and b in the general equation (6.1).

6.6.3 Cocomo II: A Major Redesign

Cocomo-81 was applied successfully to waterfall projects in the 1980s, but it lost its predictive value as software development processes changed. As processes changed, the data relating effort and size changed and the earlier statistical models no longer fit.

A major redesign in the late 1990s resulted in Cocomo II. The redesign accounted for various project parameters, such as the desired reliability and the use of tools and platforms. With Cocomo II and its many variants, the relationship between effort E and size S is given by the general form

$$E = aS^b + c \tag{6.4}$$

Constants a, b, and c are based on past data about similar projects. Factors like team productivity, problem complexity, desired reliability, and tool usage are built into the choice of constants a, b, and c.

New estimation models continue to be explored. With each advance in software engineering, the existing models lose their predictive power.[24] Existing models are designed to fit historical data, and the purpose of advances in software development is to improve upon (disrupt) the historical relationship between development effort, program size, and required functionality.

6.7 Conclusion

Project planning takes two forms: predictive for plan-driven processes and adaptive for iterative agile processes.

Predictive planning is up-front, prior to design. Uncertainty is greatest in the early stages of a project, so key decisions about team formation, work assignments, schedules, and budgets are made in the face of inherent uncertainty. The decisions may be revisited and refined as the project proceeds, but the overall direction of the project is set early.

Adaptive planning is ongoing: plans are improved by adjusting them during iteration planning. The plan for the current iteration benefits from learnings from prior iterations. Planning accuracy increases across iterations; uncertainty decreases. Rough estimates suffice for prioritizing work items during iteration planning.

During both predictive and adaptive planning, cost-benefit tradeoffs are based on estimates of development effort and program size. Expert judgment and statistical models are the two main approaches to estimation. Both have strengths and limitations:[25]

- Expert judgment can be more effective and accurate, but it is time consuming, so it cannot be used widely.
- Statistical models may be less effective, but they can run automatically to complement expert judgment.

Estimates are subject to cognitive bias, the human tendency to make systematic errors of judgment under uncertainty. Anchoring is the tendency to make estimates by starting from an initial value that is adjusted to yield the final answer. The initial value is referred to as an anchor. Different anchors lead to different estimates.

The Wideband Delphi method develops consensus estimates by tapping the collective judgment of a group of experts. In an attempt to avoid cognitive bias, the experts are asked to create their own independent initial estimates. The group then meets to discuss the individual estimates. After group discussion and feedback, the experts independently create another round of estimates. The process continues through rounds of estimation and feedback until the group converges on a consensus estimate.

When all is said and done, programmers are optimists. The tendency to underestimate has prompted quips like the following:

> "The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time."[26]

Exercises for Chapter 6

Exercise 6.1: Are the following statements True or False?

a) The expected overall planning effort is less with predictive planning than with adaptive planning.
b) The Iron Triangle illustrates connections between time, cost, and scope.
c) The Adaptive Iron Triangle fixes time and scope and lets costs vary.
d) Anchoring is the human tendency to stick to a position, once taken..
e) The shorter the planning horizon, the lower the uncertainty.
f) Anchoring reduces uncertainty during planning.
g) Planning Poker involves successive rounds of individual estimation and group discussion.
h) Wideband Delphi involves successive rounds of individual estimation and group discussion.

Exercise 6.2: In order to clarify the relative merits of formal models and expert judgment for development effort estimation, Magne Jørgensen and Barry Boehm engaged in a friendly debate [13].

Summarize their arguments, pro and con, for

a) expert judgment.
b) formal models.

Based on their arguments, when and under what conditions would you recommend estimation methods that reoly on

c) expert judgment.
d) formal models.

Notes for Chapter 6

[1]Jørgensen [12] reviews "what we do and don't know about software development effort estimation." See also the friendly debate between Jørgensen and Boehm [13] about expert judgment versus formal models.

[2]The Project Management Institute publishes PMBOK, a guide to the project management body of knowledge [17].

[3]The IEEE Software Engineering Body of Knowledge [8] defines project planning to consist of process planning[deliverable planning; effort, schedule, and cost estimation; resource allocation; risk management, quality planning; plan management. Plan management consists of managing changes to the plan itself.

[4]Jørgensen [12].

[5]The essays in the *Mythical Man Month* [7] are based on Fred Brooks's experience as the project manager for IBM's Operating System/360, a very large and complex software project.

[6]Martin Barnes is credited with creating the Iron Triangle for a 1969 course [22]. Trilemmas have been discussed in religion and philosophy for centuries.

[7]A Wikipedia article [23] credits the double Iron Triangle to PMBOK 4.0 [17].

[8]Tversky and Kahneman [20]. Kahneman won the 2002 Nobel Prize in Economic Sciences for his insights into "human judgment and decision making under uncertainty."

[9]Example 6.2 is based on a case study by Aranda and Easterbrook [1].

[10]Diagrams similar to the Cone of Uncertainty were in use in the 1950s for cost estimation for chemical manufacturing [16]. Boehm [4, p. 311] introduced the cone as a conceptual diagram for software estimation. The supporting data in Fig. 6.5 is from a 1995 paper by Boehm et al. [5].

[11]Cohn [9, ch. 3] discusses the multiple levels of agile planning. Figure 6.6 is adapted from the "planning onion" in [9].

[12]See Moder, Phillips, and Davis [15, ch. 9] for the statistical underpinnings of the weighted-average formula for three-point estimation. PERT (Program Evaluation and Review Technique) is a project management tool that was developed for the U.S. Navy in the 1950s. PERT involves three-point time estimation.

[13]The quote about group wisdom is from Aristotle [2, Book 3, Part 11].

[14]Smith and Sidky [19, ch. 14]

[15]Dalkey and Helmer [10] describe the Delphi method, which was designed at RAND corporation in the 1950s.

[16]Examples 6.5 and 6.6 are based on Helmer's [11] retrospective of the Delphi method.

[17]Helmer [11].

[18]Barry Boehm [4, p. 335] is credited with the Wideband Delphi method.

[19]See Cohn [9, ch 6] for Planning Poker.

[20]Results from numerous studies, going back to the 1960s, support the observation that there are order of magnitude differences in individual and team productivity. From early studies by Sackman, Erikson, and Grant [18], "one poor performer can consume as much time or cost as 5, 10, or 20 good ones." McConnell [14] outlines the challenges of defining, much less measuring, software productivity.

[21]Boehm and Valerdi's [6] review of the Cocomo family of formal models includes some historical perspective on models.

[22]Walston and Felix [21].

[23]Boehm [4].

[24]Boehm and Valerdi [6] note that "although Cocomo II does a good job for the 2005 development styles projected in 1995, it doesn't cover several newer development styles well. This led us to develop additional Cocomo II-related models."

[25]Jørgensen and Boehm [13] debate which is better: formal models or expert judgment.

[26]Attributed to Tom Cargill by Bentley [3].

References for Chapter 6

1. Jorge Aranda and Steve Easterbrook. Anchoring and adjustment in software estimation. *Proceedings 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2005) 346-355.
2. Aristotle. *Politics*. See `http://classics.mit.edu/Aristotle/politics.html` for the Benjamin Jowett translation.
3. Jon Bentley. Programming Pearls: bumper-sticker computer science. *Comm. ACM* 28, 9 (September 1985) 896-901.
4. Barry W. Boehm. *Software Engineering Economics* (1981).
5. Barry W. Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future life cycle processes: COCOMO 2.0, *Annals of Software Engineering* 1, 1 (1995) 57-94.
6. Barry W. Boehm and Ricardo Valerdi. Achievements and challenges in Cocomo-based software resource estimation. *IEEE Software* (September-October 2008) 74-83.
7. Frederick P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. (1975) Addison-Wesley, Reading, Mass. See also the *Anniversary Edition* (1995) with four added chapters.
8. Pierre Bourque and Richard E. Farley (eds). *SWEBOK Version 3.0: Guide to the Software Engineering Body of Knowledge*. IEEE (2014).
9. Mike Cohn. *Agile Estimating and Planning*. Prentice Hall (2006).
10. Norman Dalkey and Olaf Helmer. An experimental application of the Delphi method to the use of experts. *Management Science* 9, 3 (April 1963) 458-467.
11. Olaf Helmer. *Analysis of the Future: The Delphi Method*. RAND Corporation, Report P-3558 (March 1967).
12. Magne Jørgensen. What we do and don't know about software development effort estimation. *IEEE Software* (March-April 2014) 13-16.
13. Magne Jørgensen and Barry Boehm. Software development effort estimation: formal models or expert judgment? *IEEE Software* (March-April 2009) 14-19.
14. Steve McConnell. Measuring software productivity. ACM Learning Webinar (January 11, 2016). `http://resources.construx.com/wp-content/uploads/2016/02/Measuring-Software-Development-Productivity.pdf` .
15. Joseph J. Moder, Cecil R. Phillips, and Edward D. Davis. Project Management with CPM, PERT, and Precedence Programming, 3rd ed. Van Nostrand Reinhold, New York (1983).
16. W. T. Nichols. Capital cost estimating. *Industrial and Engineering Chemistry* 43, 10 (1951) 2295-2298.
17. Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOKGuide)*. Project Management Institute (2008)
18. H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM* 11, 1 (January 1968) 3-11.
19. Greg Smith and Ahmed Sidky. *Becoming Agile: ... in an imperfect world*, Manning Publications (2009).
20. Amos Tversky and Daniel Kahneman. Judgement under uncertainty: heuristics and biases. *Science* 185 (September 27, 1974) 1124-1131.
21. C. E. Walston and C. P. Felix. A method of programming measurement and estimation. *IBM Systems Journal* 16, 1 (March 1977) 54-73.

22. Patrick Weaver. The origins of modern project management. Originally presented at the *Fourth Annual PMI College of Scheduling Conference* (2007).
`http://www.mosaicprojects.com.au/PDF_Papers/P050_Origins_of_Modern_PM.pdf`

23. Wikipedia. Project Management Triangle.
`https://en.wikipedia.org/wiki/Project_management_triangle`

Chapter 7

Goals and Metrics

"I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be."
— *William Thomson, Lord Kelvin.*[1]

Goals and metrics are essential for making informed decisions during software development. Questions abound. The following are just some of the top questions collected during one study:[2]

- "How do users typically use my application?
- "What parts of a software product are most used and/or loved by customers?
- "How effective are the quality gates we run at checkin?"
- "What is the impact of a code change or requirements change to the project and tests?"

Behind each such question is a goal. Why is the questions being asked? What would be the benefit if the question were answered? In other words, what is the goal behind the question?

This chapter deals with goals: their nature; how to clarify them; how to act on them; and how to measure progress toward achieving them.

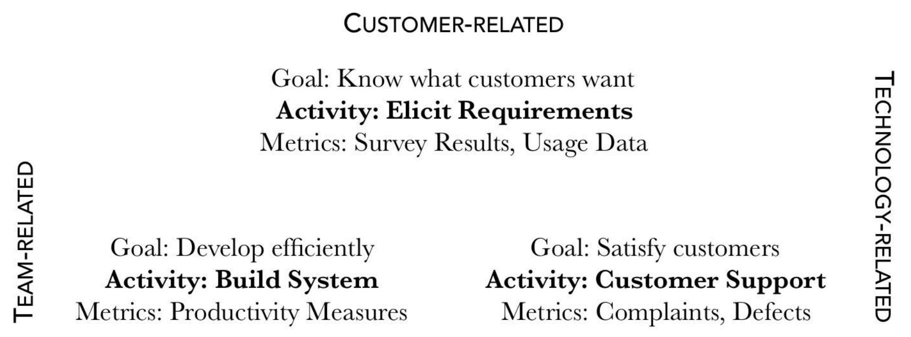

CUSTOMER-RELATED
Goal: Know what customers want
Activity: Elicit Requirements
Metrics: Survey Results, Usage Data
TEAM-RELATED
Goal: Develop efficiently
Activity: Build System
Metrics: Productivity Measures
Goal: Satisfy customers
Activity: Customer Support
Metrics: Complaints, Defects
TECHNOLOGY-RELATED

Figure 7.1: High level activities and their associated goals and metrics.

7.1 Introduction to Goals

With any activity, we can associate *goals* that provide context and *metrics* that define success for the activity. Sample goals and metrics associated with software development activities appear in Fig. 7.1.

In the user-stories template from Fig. 3.7,

> **As a** ⟨*stakeholder*⟩ **I want to** ⟨*do some task*⟩
> **so that** ⟨*I can achieve some benefit*⟩

the *benefit* corresponds to a goal and the *task* corresponds to an activity to achieve the goal.

Goals can be used to make decisions and keep a project on track. When a user request for convenient authentication conflicts with an IT staff requirement for tight security, goals can be used to resolve and prioritize these conflicting requirements. During development, goals can help prioritize between functionality, schedule, and budget.

This chapter deals with a goal-directed approach to identifying actions and metrics for a project. The approach is as follows:

- Set initial goals that represent business value.
- Use questions to refine the goals until they can be turned into actions.
- Identify actions and metrics based on the refined goals.

7.1.1 Soft and Hard Goals

Initial goals may be expressed as aspirations, such as "Increase customer satisfaction" or "Reduce costs." A goal without success or satisfaction criteria is called a *soft goal*. A goal with a success criterion is *hard goal*.

The "so that" part of the following user story is a soft goal:

As a manager **I want to** have timely sales information
so that I can manage inventories

The soft goal, "manage inventories," can be refined:

reorder items with low inventory

Further refinement yields a hard goal that can be tested:

reorder items that will be out of stock in less than a week

This goal appears in the following revised version of the user story

As a manager **I want to** have daily sales and stock information
so that I can reorder items that will be out of stock in less than a week

This user story is specific about the kind of data (sales and inventory), about when the data is needed (daily), and about the benefit (keeping items in stock). From sales data about the rate at which items are selling and current inventory levels, the manager can project when an item will go out of stock. The story assumes that reordered items can be delivered within a week. It can be rewritten to allow for different delivery intervals for different items.

7.1.2 SMART Criteria

The acronym *SMART* stands for Specific, Measurable, Achievable, Relevant, and Time-bound. SMART criteria can be applied to goals, questions, and actions; they were applied to user stories in Section 3.6.1.

SMART goals are hard goals. From SMART goals, we can identify actions that result in progress toward the goal and metrics to assess whether the goal is satisfied.

Specific. A specific goal identifies what needs to be accomplished: an achieve or cease goal identifies the desired state; a maintain or avoid goal identifies the invariant property to be maintained or avoided; an optimize goal identifies what needs to be optimized.

Measurable. A measurable goal is accompanied by a criteria for its attainment. For example, consider the goal of reducing costs by 15%. Costs can be measured. The 15% is the criterion for determining if the cost reduction goal has been attained.

Achievable. A goal that cannot be attained is not very helpful. Achievability is often relative to other constraints, such as time, skills, and resources, since the goal must be achievable within the onstraints.

Relevant. A goal is relevant if attaining the goal will produce business value.

Time-Bound. The goal is achievable within the time bound.

SMART criteria originated in a business setting; they are attributed to Peter Drucker's management by objectives. Drucker wrote about setting relevant goals that reflect "the objective needs of the business rather than what the individual manager wants," and of the need to measure results against the goal. His advice on measurements is relevant to software development:

> "Those measurements need not be rigidly quantitative: nor need they be exact. But they have to be clear, simple and rational. They have to be relevant and direct attention and efforts where they should go."[3]

7.1.3 A Temporal Classification of Goals

Frequently occurring classes of goals include the following:[4]

- *Achieve/Cease* goals eventually get to a desired state; e. g., land a rover on Mars.
- *Maintain/Avoid* goals keep a property invariant; e.g., the shopping site is up 24 hours a day, 7 days a week.
- *Optimize* goals involve a comparison between the current state and the desired state; e.g., reduce costs by 15%.

7.2 Working with Questions to Clarify Goals

High level goals, such as "Increase customer satisfaction," are simple to state and easy to link to the business value of a project. Such soft high level goals can then be refined into hard SMART goals. Even more specific goals, such as "Reduce costs by 15%" may need to be refined further before they can be acted on. Actions to reduce costs may not be obvious until we dig into how costs can be reduced.

This section deals with the following simple kinds of questions that are helpful for identifying and refining goals:[5]

- *Why.* Questions of the form "Why are we doing x?" explore the context for activity x. They are useful for establishing the relevance of x to a goal, or for eliciting the goal behind activity x. They are also useful for eliciting higher goals from lower subgoals.
- *Why Not.* Questions of the form "Why not do x?" explore constraints or conditions on doing x. The phrasing of the question can be changed; e.g., a related form of the question is "What stops us from doing x?"
- *How.* Questions of the form "How can we accomplish x?" explore potential subgoals. "How else ... ," questions bring out alternatives.
- *How Much.* Questions of the form "How will we know whether x" explore criteria for goal x. "How many ...?" or "How much ...?" questions bring out possible metrics.

7.2.1 Why Questions for Cause and Relevance

Repeated *Why* questions can be applied to test the relevance of a goal. In the following example, the starting goal is "Add staff:"

> *Why do we want to add staff?*
>
> To improve the time it takes to respond to complaints.
>
> *Why do we want to improve response time?*
>
> Because it is very effective for improving customer satisfaction.

The questions brought out the goals of improving response time and customer satisfaction. (Improving time to first response is perhaps more important for customer satisfaction than the time it takes to actually fix the problem.)

Five Why's

Five Why's is a technique for identifying the root causes of a problem:[6]

1. Given a problem, respectfully ask "Why?" Why did the problem occur?
2. Identify the possible causes of the problem.
3. For each possible cause, keep asking *Why* questions to uncover successive causes. Often, five levels of *Why* questions are enough to uncover a root cause.
4. Backtrack to explore another line of reasoning to uncover other root causes—there may be more than one.

Example 7.1 : On January 15, 1990, the AT&T Long Distance Network crashed. It took nine hours to track down the causes and restore service.

No single technique can do justice to such a complex problem, so the following is an oversimplified account. Nevertheless, it will serve to illustrate how repeated *Why* questions can quickly dig deep into a problem.

> All circuits are busy.
>
> *Why?* The switches are not accepting calls.
>
> *Why?* Their control software is busy doing a reset.
>
> *Why?* The recovery code keeps reinitializing the switch.
>
> *Why?* A signal about an incoming call triggers the recovery code.
>
> *Why?* The software has a defect and can't handle the incoming call.

The defect was introduced by a software upgrade in mid-December 1989. The software behaved normally until a hardware malfunction on January 15, 1990 uncovered the defect.[7] □

7.2.2 How Questions for Refining Goals

How and *How-else* questions elicit potential subgoals that contribute to the satisfaction of a higher goal. They bring out potential solution approaches.

Consider the following questions, based on the high level goal, "Reduce costs:"

> *How can we reduce costs?*
> Reduce travel expenses.
>
> *How else can we reduce costs?*
> Reduce staff.
>
> *How else can we reduce costs?*
> Reduce service and repair costs after the system is delivered.

The questions elicited three alternative approaches to reducing costs.

"How will we know?" explores criteria for accomplishing a goal. For example,

> *How will we know whether we have designed and built what customers want?*
>
> When the system passes the acceptance tests.

"How many?" and "How much?" elicit metrics:

> *How many of the acceptance tests must the system pass?*
>
> It must pass 100% of them.

7.2.3 No Substitute for Insight

The *Why* and *How* questions in this section are helpful for getting started. They are not a substitute for insightful inquiry.

Where do insights come from? While insights can come from anybody, the people who set the goals and who act upon them are in the best position to supply the insights. They have the expertise. They know their context, what's possible and what's not possible.

Example 7.2 : Consider the starting goal: Track changes during software development. *Why* questions bring out higher or unstated goals; e.g.,

> *Why track changes?*
> To identify risky files.
>
> *Why identify risky files?*
> To improve product quality.

The connection between product quality and risky files is not obvious, so we can follow up with a *How* question:

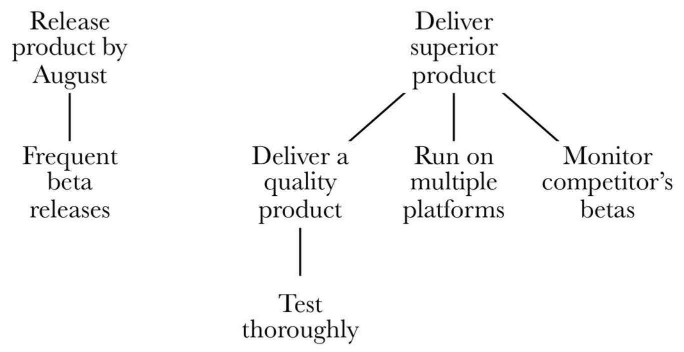

Release product by August
Frequent beta releases
Deliver superior product
Deliver a quality product
Run on multiple platforms
Monitor competitor's betas
Test thoroughly

Figure 7.2: A goal hierarchy.

> *How can identifying risky files help improve product quality?*
> Risky files tend to have many more defects.

The underlying insight—risky files tend to have many more defects—is based on knowledge of and experience with software development.[8]

The following question probes the connection between changes and risky files:

> *How can tracking changes help with identifying risky files?*
>
> Files that have been changed by many developers are riskier than files that have been changed by a single developer.

In general, there is no set order in which to ask *Why* and *How* questions: the order depends on the flow of the conversation. The above conversation would have taken a very different turn, depending on the answers to the questions. For example, the answer to the first question may easily have been different:

> *Why track changes?*
> To do software development more efficiently.

It is left as an exercise to the reader to develop this line of inquiry into efficient software development. □

7.3 Working with Goals: Goal Elaboration

As the number of goals increases, it is helpful to organize them into a hierarchy that links goals and subgoals. The hierarchy in Fig. 7.2 is a tree, but, in general, a subgoal may be shared by multiple higher goals. For example, "Improve product quality" can be a subgoal of both "Improve customer satisfaction" and "Reduce the cost of support after delivery."

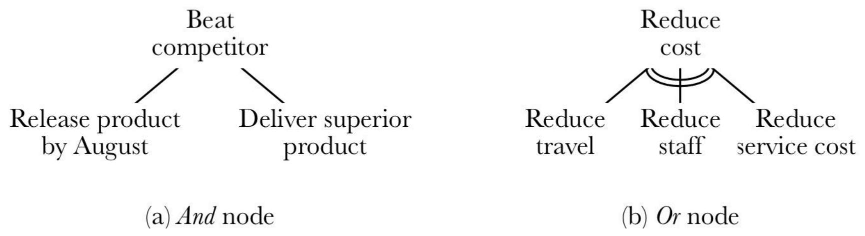


Figure 7.3: Examples of *and* and *or* nodes in a goal hierarchy.

7.3.1 Goal Hierarchies

A *goal hierarchy* has nodes representing goals and edges representing relationships between goals. The nodes in the hierarchy are of two kinds:

- An *and* node represents a goal G with subgoals $G_1, G_2, ..., G_k$, where every one of $G_1, G_2, ..., G_k$ must be satisfied for G to be satisfied. For example, the goal in Fig. 7.3(a) of beating a competitor is satisfied only if both of the following are satisfied: release the product by August; and deliver a superior product.

- An *or* node represents a goal G with subgoals $G_1, G_2, ..., G_k$, where G can be satisfied by satisfying any one of $G_1, G_2, ..., G_k$. For example, the goal in Fig. 7.3(b) of reducing costs can be satisfied by doing any one or any combination of the following: reduce travel costs; reduce staff; or reduce the cost of service after delivery.

Or nodes will be identified by a double arc connecting the edges between the goal node and the nodes for the subgoals.

Example 7.3 : The goal hierarchy in Fig. 7.2 has two initial goals. In general, a project can have multiple initial goals.

In Fig. 7.2, there is only one goal with more than one subgoal: "Deliver superior product." This goal has an *and* node, so its three subgoals must all be satisfied for this goal to be satisfied.

This example is motivated by the browser wars between Netscape and Microsoft in the 1990s; see Section 3.2. Netscape's strategy was to deliver a browser that ran on multiple operating systems; Microsoft's browser ran only on Windows, at the time. Netscape also closely monitored the beta releases from Microsoft to ensure that their browser would be competitive. □

7.3.2 Contributing and Conflicting Goals

A goal such as "Deliver a quality product" contributes strongly to the higher goal, "Deliver a superior product." Monitoring a competitor's beta releases does

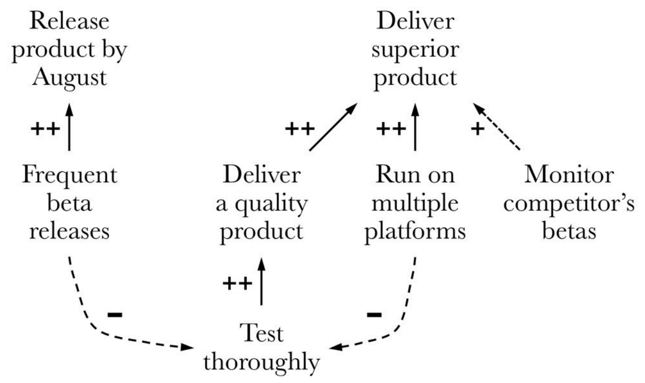

Release product by August
Deliver superior product
++
++
++
+
Frequent beta releases
Deliver a quality product
Run on multiple platforms
Monitor competitor's betas
++
–
–
Test thoroughly

Figure 7.4: Examples of contributing and conflicting goals.

contribute to delivering a superior product, but not as strongly. Meanwhile, there can be conflicting goals: it is not unusual to see "Improve quality" with the potentially conflicting goal "Reduce staff."

Goal G_1 *conflicts* with goal G_2 if satisfaction of G_1 hinders the satisfaction of G_2. For example, "Add testers" conflicts with "Reduce staff." If we add testers, we are adding, not reducing staff. Conversely, if we reduce staff, we are potentially hindering the addition of testers—potentially, since the reductions could be elsewhere.

Goal G_1 *contributes* to goal G_2 if satisfaction of G_1 aids the satisfaction of G_2. For example, "Add testers" contributes to "Improve product quality."

The conflicts and contributes relationships will be represented by directed graphs with goals for nodes. An edge is marked ++ for a strong contribution, + for a weak contribution, −− for a strong conflict, and − for a weak conflict. For clarity, edges representing weak contributions or conflicts will be dashed.

Example 7.4: The graph in Fig. 7.4 shows the contributes and conflicts relations for the goals in Example 7.3.

Since subgoals contribute to higher goals, there are edges from subgoals to higher goals. Two of the edges are for strong contributions. The third edge is for a weak contribution from "Monitor competitor's betas."

There are two conflicts edges, both to "Test throughly." At Netscape, the number of testers was fixed, so more platforms (operating systems) to test meant more work for the same testers. Furthermore, every beta release had to be fully tested, so more frequent beta also meant additional work for the same testers. Hence the conflicts edges from "Frequent beta releases" and "Run on multiple platforms" to "Test thoroughly." □

7.3.3 Business-, Software-, and Project-Level Goals

In practice, it can be helpful to group goals as follows:[9]

- *Business* goals are tied to organizational objectives, such as "Reduce costs" and "Increase revenues." They are independent of specific strategies for achieving the goal.
- *Software* goals are tied to the system to be developed. Software goals can be obtained from business goals by asking *How* questions. For example, "Increase revenues" might lead to a software goal: "Deliver a cloud service to do x," where x relates to the organization's business.
- *Project* goals are tied to the implementation of the system. They can be refined from software goals by asking *How* questions. For example, project goals might be tied to project deliverables or team training.

For each one of these categories—business, software, project—there can be multiple levels. The same elaboration and refinement approaches apply at all levels.

7.3.4 When to Stop Goal Elaboration

Goal refinement can stop when the subgoals become SMART, since actions can be based on SMART goals. For example, consider the following refinement of "Run on multiple platforms:"

> *How many platforms does the browser need to run on?*
> Windows, Mac OS, and Unix.

No further refinement is needed, since the development team knows which operating systems the browser must work on.

7.4 Working with Metrics

A *metric* quantifies progress towards a goal. If the goal is to pass all acceptance tests, then the percentage of successful tests is a metric for quantifying progress toward the goal.

A *criterion* is a specific value of a metric that serves as a reference for determining whether a goal has been achieved. In the above example of acceptance tests, the criterion is 100%. For the goal to be reached, 100% of the tests must pass.

Bowing to popular usage, we make little distinction between the terms "metric" and "measure." Careful readers will note that *measure* corresponds to a class and a specific *metric* corresponds to an instance, as in the following:

measure:	length
metrics:	miles, kilometers

7.4.1 Data Collection

Metrics require data. A metric such as the number of changes by a developer by week requires data about changes made by developers.

Ideally, metrics would come first and, from the metrics, we would know what data to collect. In practice, the data that is available constrains the metrics that can be used. The reasons are twofold. First, for metrics that require historical data, we have no choice but to rely on the available historical data. Second, measurement is often an added activity that gets low priority, compared to design, coding, and testing. Most projects therefore rely on the data collected unobtrusively by the tools that they use for software development.

The main sources of data include the following:

- *Version control systems* that track changes to code and documents.
- *Personnel directories* that track information about team members, such as work location and dates of service.
- *Trouble ticket systems* that track customer service requests.
- *Customer information systems* that track information about systems installed at customer sites.

Systems like the above include a wealth of data, although the raw data may not be in a form that is readily usable.

Example 7.5: In a large organization, with software developers in multiple countries, consider the problem of finding relevant expertise.

The change history of a piece of code identifies the people who touched the code, including the original developers, the people who fixed customer found defects, and the people who made enhancements. By looking at the pattern of changes made by a person, we can infer their experience with the code. Anyone who made significant changes to a given file presumably knows that file. Anyone who made changes in multiple places must have broad knowledge of that set of files.

The expertise browser illustrated in Fig. 7.5 brings together change information from version control system, organizational information from personnel directories, and modification requests for trouble ticket systems.[10] The panes for Developers, Organizations, and Folders are linked, so a selection in one pane triggers changes to the contents in the other panes. Personnel information is needed, since the original developers of a file may have moved on to other projects or may have left the company entirely. □

A key challenge with data collection is that even if the data is available, the desired data may be spread over multiple systems from different vendors. Accuracy is another issue: even if the data was initially correct when entered, it may not be up to date, and hence inaccurate.

In summary, data collection is a major issue.

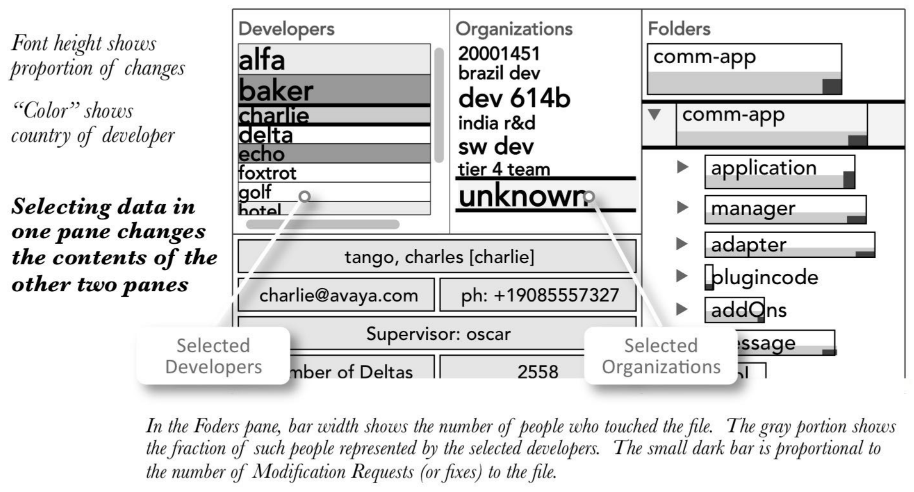

Font height shows proportion of changes
"Color" shows country of developer
Selecting data in one pane changes the contents of the other two panes
Developers
alfa
baker
charlie
delta
echo
foxtrot
golf
hotel
Organizations
20001451
brazil dev
dev 614b
india r&d
sw dev
tier 4 team
unknown
Folders
comm-app
comm-app
application
manager
adapter
plugincode
addOns
essage
tango, charles [charlie]
charlie@avaya.com
ph: +19085557327
Supervisor: oscar
mber of Deltas
2558
Selected Developers
Selected Organizations
In the Foders pane, bar width shows the number of people who touched the file. The gray portion shows the fraction of such people represented by the selected developers. The small dark bar is proportional to the number of Modification Requests (or fixes) to the file.

Figure 7.5: A rendering of an expertise browser due to Audris Mockus.

7.4.2 Case Study: Customer Satisfaction

Sometimes, the relevant metrics are evident from a goal, as in the example of acceptance tests at the beginning of this section.

In general, finding the right subgoal and then the right metric can be a challenge. For example, the authors of an IBM study noted that "you are likely to get two conflicting answers" if you ask which of the following has a greater impact on customer satisfaction:[11]

- the number of problems on a product, or
- service call response time.

The IBM study was based on three years of actual data from service centers that handled customer calls. It examined 15 different metrics, for an operating systems product. The metrics related to customer found defects and to customer service requests; see Fig. 7.6.

The study examined the correlation between the 15 metrics and customer satisfaction surveys. It found the greatest correlation between the following two factors and satisfaction surveys"

1. Number of defects found in previous "fixes" to code or documentation

2. Total number of customer service requests that were closed

The next two factors were much less significant than the first two:

3. Total number of fixed customer-found defects in code or documentation

4. Number of days to resolution for requests handled by Level 2

- Total number of fixed customer-found defects in code or documentation
- Number that were dubbed ?genuine? and reported for the first time
- Number that were rejected by the Level 3 support personnel
- Number of pointers to components touched by a first-time genuine defect
- Number of defects found in previous ?fixes?
- Number of defects in code or documentation that were received
- Backlog of defects in code or documentation
- Total number of customer service requests that were closed
- Number that were for preventive service
- Number that were for installation planning
- Number that were for code or documentation; e.g., 2nd+ defect reports
- Number that were not related to IBM code or documentation
- Number of customer service requests handled by Level 2 support
- Number of days to resolution for requests handled by Level 2
- Number of users, a measure of the size of the installed base

Figure 7.6: Which of these customer-service metrics has the most effect on customer satisfaction? All 15 were managed and tracked.

7.5 Putting It All Together

The extended example in this section is for support services for systems delivered and installed at a customer site. The example starts with two top level goals:

- *Customer Quality.* Improve customer perceptions of the quality of the installed system.
- *Service Experience.* Improve the service experience when customers report troubles with the installed system.

The goal hierarchy in Fig. 7.7 provides a roadmap for the extended example.

7.5.1 Improve Product Quality

The first approach to improving customer quality is to improve the quality of the delivered product. For simplicity, we consider just one way of improving product quality: improve system testing so that there will be fewer defects in the systems that are delivered to customers. A metric associated with improved system testing is the total number of system tests.

- Improve quality as perceived by customers.

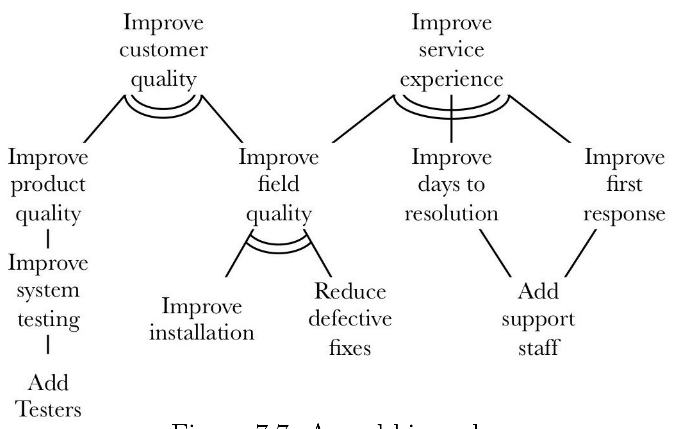


Figure 7.7: A goal hierarchy.

- *How?* Improve the quality of the delivered product.
    - *How?* Improve system testing.
      Metric: Number of system tests.
        - *How?* Add testers to do more product testing.
          Metric: Total number of testers.

Improved system testing has a further subgoal: add testers.

7.5.2 Improve Field Quality

Improving customer quality has an alternative subgoal: improve quality during operations in the field. For clarity, the top level goal of improving customer quality is repeated in the following:

- Improve quality as perceived by customers.
    - *How else?* Improve quality during operations in the field.
        - *How?* Improve installation.
          Metric: Percentage of systems that fail in the first month.
        - *How else?* Reduce the number of defective "fixes."
          Metric: Percentage of defective fixes.

Installation issues tend to show up early, so the percentage of failures in the first month should go down if installations go well. From the IBM study in Section 7.4, reducing the number of defective "fixes" is strongly correlated with customer satisfaction.

7.5.3 Improve Days to Resolution

The second top-level goal is to improve the service experience. The first alternative for this goal—improve field quality—has already been considered. The second alternative is to reduce the number of days it takes to resolve a customer problem. Here, the approach is to add service support staff for faster resolution of problems. The metrics are the median number of days to resolution and the number of support staff added.

- Improve the service experience for customers.
  - *How?* Improve quality during operations in the field.
    Considered above.
  - *How else?* Improve days to resolution of a customer problem.
    - *How?* Add service support staff.
      Metric: Total number of service support staff.
      Metric: Median days to resolution.

7.5.4 Improve Time to First Response

Finally, the third alternative for improving the service experience:

- Improve the service experience for customers.
  - *How else?* Improve time to first response to customer service request
    Add service staff
    See above.
    Metric: Median time to first response.

Exercises for Chapter 7

Exercise 7.1 : The goals for the San Francisco Bay Area Rapid Transit System (BART) included the following:[12]

- Serve more passengers
- Minimize costs
- Improve safety

Apply the Goals and Metrics approach to create an action plan.

a) Show the questions that you use for goal elaboration. Explain the rationale for each question.

b) Draw the goal hierarchy.

c) Identify the supporting and conflicting goals

d) Show the metrics. For each metric, explain why it is a good metric.

Exercise 7.2 : Consider the soft goals

- Take products to market quickly
- Create products efficiently

Apply the Goals-Activities-Metrics approach to create an action plan.

a) Show the questions you used to elaborate the goals. Be sure to consider alternatives during goal elaboration.

b) Show the resulting goal hierarchy. Distinguish between and-nodes and or-nodes.

c) Does any goal contribute to a goal in another subtree? Are there any conflicting goals?

d) Associate metrics with each goal in the hierarchy.

e) Create an action plan based on your hierarchy.

f) Assess the effectiveness of the action plan derived using this approach. In your opinion, will the action plan achieve the initial soft goals? Explain your answer.

Exercise 7.3 : From a State of Agile Report, two of the top reasons for adopting agile software development processes are as follows:

- Accelerate product delivery, 62%
- Improve productivity, 55%

Starting with these goals, apply the Goals and Metrics approach to create subgoals and an action plan. Include sub-subgoals (that is, second-level subgoals) if at all possible.

a) Show the questions that you use for goal elaboration.

b) Draw the goal hierarchy.

c) Show the metrics. For each metric, explain why it is a good metric.

Exercise 7.4 : The following are among the top-rated questions from a 2014 study of questions related to software development.

1. How effective are the code reviews prior to code check-in?

2. What parts of a software product are most used and/or loved by customers?

3. How can we improve collaboration and sharing between teams?

4. What is the impact of tools on productivity?

For each question

a) Identify higher-level goals related to the questions.

b) Refine the higher-level goal to come up with actions. Show your work.

c) Associate metrics with each goal.

Notes for Chapter 7

[1]Lord Kelvin began a lecture on May 3, 1883 with, “In physical science a first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. ! often say ...” [15, p.79-80].

[2]These questions ranked number 1, 2, 3, and 6 of 145 in Begel and Zimmermann’s [3] survey of “questions that software engineers would like data scientists to investigate about software.”

[3]Peter Drucker’s advice on goals and measurements is from [7, ch. 8], an excerpt from his 1954 book, *The Principles of Management*. The acronym SMART is attributed to George T. Doran [6].

[4]The temporal classification of goals is from Dardenne, Lamsweerde, and Fickas [5].

[5]“Lamsweerde, Darimont, and Massonet [10] write, “*why* questions allow higher-level goals to be acquired from goals that contribute positively to them. ... *how* questions allow lower level goals to be acquired as sub-goals that contribute positively to the goal considered.”

[6]The Five Whys technique was developed at Toyota to identify root causes. [16]. Repeated *Why* questions are used in many contexts to elicit higher goals, values, and root causes. Financial planner Carl Richards: “The question I like to start with is ‘Why is money important to you?’ [14].”

[7]A hardware malfunction—a link connected to the New York switch failed—set off a chain of events that crashed the entire AT&T network in 1990. [13].

[8]For risky files and their role in quality improvement, see [8, 11].

[9]Basili et al. [1] describe an approach that they call GQM$^+$Strategies, which extends the Goals-Questions-Metrics approach of Basili and Weiss [2]. GQM$^+$Strategies starts with high-level goals that are refined into what they call “measurement” or “GQM” goals. The approach of [2] is then applied to the measurement goals.

[10]Mockus and Herbsleb [12] describe an expertise browser.

[11]Buckley and Chillarege [4]

[12]Axel van Lamsweerde [9] uses BART as a case study: “This case study is appealing for a number of reasons: it is a real system; it is a complex real-time safety-critical system; the initial document was provided by an independent source involved in the development.”

References for Chapter 7

1. Victor Basili, Mikael Lindvall, Myrna Regardie, Carolyn Seaman, Jens Heidrich, Jürgen Münch, Dieter Rombach, and Adam Trendowicz. Linking software development and business strategy through measurement. *IEEE Computer* 43, 4 (April 2010) 57-65.

2. Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* SE-10, 6 (November 1984) 728-738.

3. Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. *36th International Conference on Software Engineering (ICSE)* (2014) 12-23.

4. Michael Buckley and Ram Chillarege. Discovering relationships between service and customer satisfaction. *Proceedings IEEE Intl. Conf. Software Maintenance (ICSM '95)* (1995) 192-201.

5. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming* 20 (1993) 3-50.

6. George T. Doran. There's a S.M.A.R.T. way to write management's goals and objectives, *Management Review* 70, 11 (1981) 35-36.

7. Peter F. Drucker. *The Essential Drucker*. Harper Business, New York (2001).

8. Randy Hackbarth, Audris Mockus, John D. Palframan, and Ravi Sethi. Improving software quality as customers perceive it. *IEEE Software* (July/August 2016) 40-45.

9. Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. *International Conference on Software Engineering (ICSE)*. (2000) 5-19.

10. Axel van Lamsweerde, Robert Darimont, and Philippe Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. *Second IEEE International Symposium on Reliability Engineering.* (1995) 194-203

11. Audris Mockus, Randy Hackbarth, and John D. Palframan. Risky files: an approach to focus quality improvement effort. *European Conference on Software Engineering and ACM SIGSOFT Symposium on Foundations of Software Engineering (ECSE/FSE ?13).* ACM Press, New York, NY (2013) 691-694.

12. Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. *International Conference on Software Engineering (ICSE).* (2002) 503-512.

13. Peter G. Neumann. Cause of AT&T network failure. *Risks Digest* 9, 62 (January 26, 1990). `http://catless.ncl.ac.uk/Risks/9/62#subj2`.

14. Carl Richards (interviewed by Charles Rotblut). Creating and following a financial plan. *AAII Journal* 37, 8 (August 2015) 29-32.

15. William Thomson (Lord Kelvin). *Popular Lectures and Addresses, Volume I, 2nd Edition* MacMillan (1891).

16. Wikipedia. 5 Whys. `https://en.wikipedia.org/wiki/5_Whys`.

Chapter 8

Software Architecture

"In practice, the terms 'architecture,' 'design,' and 'implementation' appear to connote varying degrees of abstraction in the continuum between complete details ('implementation'), few details ('design'), and the highest form of abstraction ('architecture')."

— *Rick Kazman and Amnon Eden, in a news publication from the Software Engineering Institute.*[1]

Informally, a software architecture is a description of the parts of a software system, together with how the parts fit to form the system. There are different descriptions for different purposes, hence the reference to "*an* architecture," not "*the* architecture," of a system. A formal definition of architecture appears in Section 8.2.

8.1 Role and Benefits of Software Architecture

Software architecture touches all aspects of a project. Examples of the role and benefits of software architecture appear in Fig. 8.1. Each of the categories in the figure—customers, teams, technology, context—is considered below.

8.1.1 Architecture and Customers

In the early stages of a project, a proposed architecture can help address questions such as the following: Will this solution approach meet customer needs and goals? Will it satisfy all stakeholders?

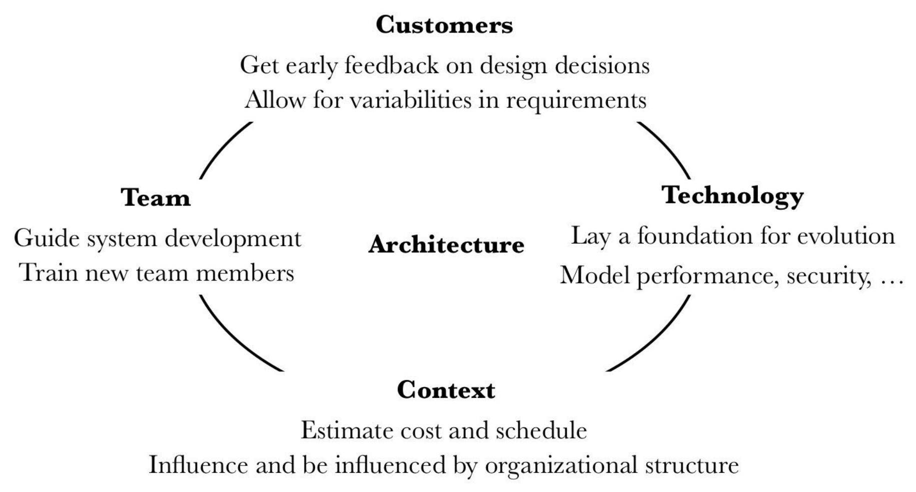

Customers
Get early feedback on design decisions
Allow for variabilities in requirements
Team
Guide system development
Train new team members
Architecture
Technology
Lay a foundation for evolution
Model performance, security, …
Context
Estimate cost and schedule
Influence and be influenced by organizational structure

Figure 8.1: Examples of the role and benefits of software architecture.

A proposed architecture can be used to get early feedback on design decisions. It can also be used to explore the range of possibilities for requirements. Requirements that might change are referred to as variabilities. Design decisions related to variabilities can be isolated, so the effect of a requirements change is confined to one part of a system, with minimal ripple-through effect on the rest of the system.

8.1.2 Architecture and Teams

Project managers can use an architecture to identify the skills and resources needed for a project. They can then assemble a team with the right skills and assign work to team members. When new members join the team, the architectural descripton can be used to train them on the goals and design decisions for the project.

Example 8.1 : Microsoft used a modular architecture to catch up with Netscape during their browser wars of the mid 1990s. As noted in Section 3.2, Microsoft appeared to have missed the Internet disruption until it launched a company-wide effort to build its own web browser. A team member observed,

> "If someone asked what the most successful aspect of [Internet Explorer 3.0] was, I would say it was the job we did in 'componentizing' the product."[2]

A modular architecture meant that the components were relatively independent of each other, so they could be developed in parallel by different subteams. Work assignment for parallel development allowed the browser to be delivered sooner, compared to sequential development of the components. □

8.1.3 Architecture and Technology

Software architecture can be used to

- *Guide system development.* An architecture describes the parts of the system and how the parts come together. It can therefore be used to guide assembly and integration of the code for the parts. During iterative development, architecture can be used for release and iteration planning.
- *Model system properties.* Properties such as scale, performance, availability, and security can be modeled and studied in terms of the architecture of the system; see also Example 8.2, below.
- *Provide a foundation for system evolution.* As noted in Section 3.2, uncertainty during requirements gathering relates to changes that can be anticipated, whereas dynamic changes are unexpected. The architecture of a system can be designed to handle system evolution based on anticipated changes.

Example 8.2 : In 2012, Apple banned all apps from Qihoo as being potential security risks. The decision was based on the high level architecture of the apps.

Apps on Apple iOS devices have two main parts: (a) the application-specific code for the app; and (b) the features provided by the underlying iOS system through iOS Application Programming Interfaces (APIs).

Some of the iOS APIs are for Apple's own use and are referred to as *private APIs.* Misuse of private APIs can pose security risks, so third-party apps are not permitted to use them. The Qihoo ban was because the company distributed apps that used private APIs.[3]

Thus, Apple cited architecture (specifically, the use of private APIs) to reach a conclusion about a system property (specifically, potential security risk). □

8.1.4 Architecture and Context

Project managers use an architecture to estimate the cost and schedule for a project. Estimation is done by decomposing a problem into simpler subproblems. The subproblems may themselves by decomposed further, until they becomes simple enough to permit work items to be defined and estimates to be made. Estimates for the subproblems are then combined to create estimates for the overall problem.

Non-technical considerations exert an influence on architecture. Organization structure influences system structure—this observed phenomenon is known as Conway's Law..

Conway's Law

Sociological or organizational context can exert a powerful influence on software architecture. In 1968, Melvin Conway observed

> "Any organization that designs a system [defined broadly] will inevitably produce a design whose structure is a copy of the organization's communication structure."[4]

The premise for this sociological observation is that two software modules A and B cannot interface correctly with each other unless the designer of A communicates with the designer of B. Thus, the interface structure of the system necessarily reflects the social structure of the organization that produces it.

8.2 What is Software Architecture?

There is no shortage of definitions of the term "software architecture." The Software Engineering Institute (SEI) has collected over two hundred definitions from visitors to its web site.[5]

Why is software architecture so hard to define?

The reason is that the appropriate description of a software architecture varies with the intended audience and purpose. A descriptions that is geared to a specific application is called a "view" of the architecture. A view that is suitable for discussions with customers and users may not be suitable for modeling system properties like performance. The analogy with building architecture is that electricians and plumbers focus on different elements. An electrician focuses on electrical outlets and circuits, while a plumber focuses on taps and drains.

With software, there is an additional distinction, between static and dynamic description of the system. Descriptions of the static program elements in the source text are potentially quite different from descriptions of the dynamic object or process elements that are created at run time.

8.2.1 Structure

Common to the many definitions of software architecture is the notion of structure: given a complex task, break it down into smaller more manageable elements. Formally, a *structure* consists of a set of elements and a binary relation on the elements.

Structures are often represented by box-and-line diagrams (see Fig. 8.2), in which boxes represent elements and directed lines (arrows) represent a binary relation on the elements. The direction of the arrows is significant.

Example 8.3: The two structures for a compiler in Fig. 8.2 (a) and (b) look very similar. Both structures are on the same five elements, represented by the

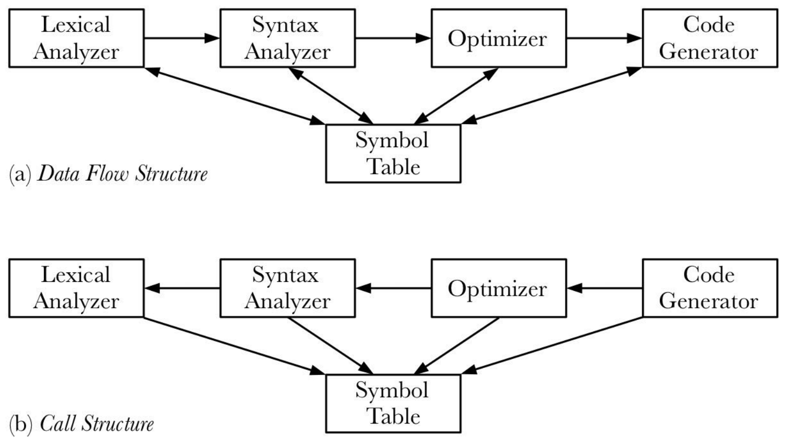


Figure 8.2: Two structures for a compiler. While the diagrams in (a) and (b) look very similar, the directions of the arrows are different in the two diagrams, since the arrows represent different binary relations on the same five elements.

five boxes in (a) and (b). The differences between the two diagrams lie in the directions of the arrows.

The arrows in Fig. 8.2(a) represent the flow of data between the five elements. The *Data-Flow* structure includes the following relation:

From	To
Lexical Analyzer	Syntax Analyzer
Syntax Analyzer	Optimizer
Optimizer	Code Generator
Lexical Analyzer	Symbol Table
Syntax Analyzer	Symbol Table
Optimizer	Symbol Table
Code Generator	Symbol Table
Symbol Table	Lexical Analyzer
Symbol Table	Syntax Analyzer
Symbol Table	Optimizer
Synbol Table	Code Generator

The arrows in Fig. 8.2(b) represent "who calls who" or "who uses who." Note that while data flows in and out of the symbol table, the calling relation is one sided: the other elements call the symbol table; the symbol table itself does not call the others. Hence there are arrows from the other elements to the symbol table in Fig. 8.2(b), but not in the other direction.

The *Call* structure includes the following relation:

Code Generator	*calls*	Optimizer
Optimizer	*calls*	Syntax Analyzer
Syntax Analyzer	*calls*	Lexical Analyzer
Lexical Analyzer	*calls*	Symbol Table
Syntax Analyzer	*calls*	Symbol Table
Optimizer	*calls*	Symbol Table
Code Generator	*calls*	Symbol Table

□

8.2.2 Definition of Software Architecture

The potential for multiple structures leads to the following definition:[6]

> "The *software architecture* of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

Architecture deals with the externally visible software elements and their relationships; see Fig. 8.3. The *design* of a system covers both the internal and external structure of the system. Architecture is therefore a subset of design.[7]

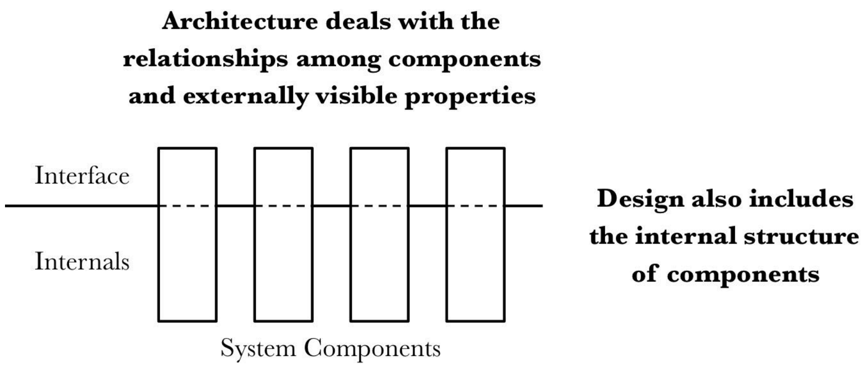

Architecture deals with the relationships among components and externally visible properties
Interface
Internals
System Components
Design also includes the internal structure of components

Figure 8.3: Architecture is a subset of design.

8.2.3 Views and Structures

An *architectural view* is a representation of an architectural structure. Thus, structures and views are related: a structure is a binary relation, which is a mathematical abstraction; a view is a concrete form of the abstraction.

A view focuses on some aspect of a given architecture. Typically, a view focuses on a problem and outlines how the architecture solves that problem. Other aspects of the system that are not relevant to the solution are suppressed. No one view (or structure) captures everything about an architecture.

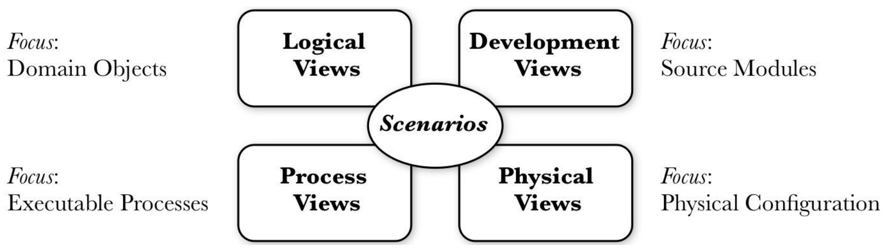


Figure 8.4: The 4+1 grouping of architectural views.

The concept of views has been incorporated into international standards for architectural descriptions.[8]

The *4+1* grouping of views is as follows (see Fig. 8.4):[9]

- *Logical Views* focus on structures that support requirements and end-user functionality. With an object-oriented approach, logical views would depict the objects and classes that are relevant to the application domain.
- *Development Views* focus on modules in the static source code. Modules are discussed in Section 8.4.
- *Process Views* focus on dynamic or run-time processes. Process views are relevant for distributed or concurrent systems. The concerns addressed by process views include synchronization and fault tolerance.
- *Physical Views* focus on the configuration and physical distribution of the system. For example, the allocation of processes to servers would be addressed by a physical view.

The "+1" in "4+1" refers to selected *scenarios* or use cases, which span the four views: logical, development, process, and physical.

8.3 Lessons from Architecture for Buildings

The centuries old tradition of building architecture has valuable lessons to offer for software architecture:

- *Principles.* What makes for a good architecture? The classical principles of utility, strength, and beauty carry over to software.
- *Views.* Since buildings are tangible, they are convenient for illustrating the role of different views of an architecture.
- *Patterns.* A pattern provides the core of a solution to a recurring problem. Patterns are discussed in Chapter 9.

The analogy between buildings and software is not perfect. Buildings are static; software has both static (source code) and dynamic (run-time) structure. Buildings are tangible: they can be seen and touched and visualized. Software is intangible. Buildings do not readily change, although buildings do get remodeled from time to time. Software is seemingly easy to change, and refactor.

Despite the differences, there are enough similarities that software architects have drawn inspiration from building architecture.

8.3.1 Principles from Classical Architecture

In the first century BC, the Roman architect Vitruvius laid out three principles that are still applied to buildings.[10] These principles can be interpreted as follows:

- *Utility.* Does the building conveniently serve its intended purpose?

  Software Equivalent: Does the system meet its requirements?

- *Strength.* Will the building stand? Are the foundations solid and have the materials been wisely selected?

  Software Equivalent: Is the system robust? Will it scale and perform? Is the technology appropriate?

- *Beauty.* Is the appearance of the building pleasing and in good taste? Are the elements of the building in due proportion?

  Software Equivalent: Is the implementation of the system elegant? Is it easy to understand and modify?

Example 8.4: Inaugurated in 537 AD, the Hagia Sophia in Istanbul was the largest cathedral in the world for a thousand years; see Fig. 8.5.[11] Converted to a mosque in 1453, elements of its architecture were replicated in many other mosques in the Ottoman empire. The Hagia Sophia became a museum in 1931.[12]

By any standards, the Hagia Sophia is an exceptional building.

Utility. It fulfills Emperor Justinian I's wish for a majestic church, grander and more imposing than all its predecessors. For centuries, it reigned as the greatest cathedral ever built. As a mosque, it showed the way for many others to come.

Strength. The Hagia Sophia stands tall, almost 1500 years after it was built. The main dome was re-architected, but that was in 562 AD.

Beauty. The beauty of the Hagia Sophia is intrinsic to its architecture and proportions. The main dome soars 182 feet from the floor. The vast open well-lit interior was richly decorated with mosaics and marble pillars. As a museum, it attracts millions of visitors every year. □

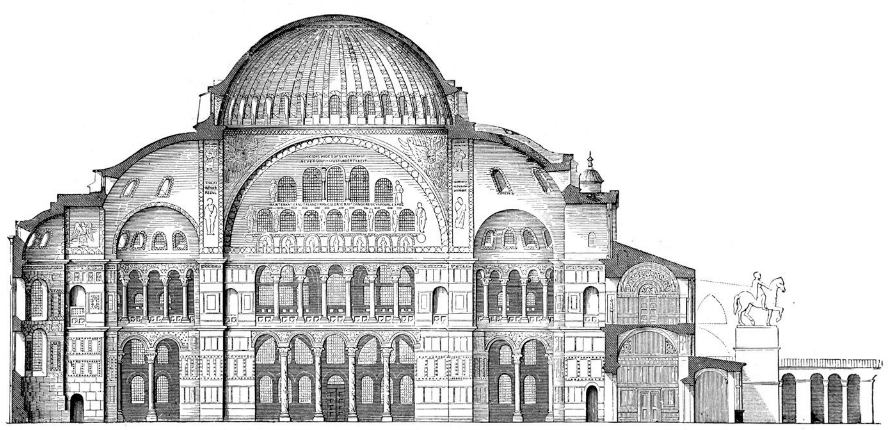


Figure 8.5: Cross section of the Hagia Sophia, built 1500 years ago..

8.3.2 Architectural Views

Utility, strength, and beauty are different perspectives on the same architecture. A functional (utility) view of the Hagia Sophia includes the galleries and interior spaces and the functions that they serve. An aesthetic (beauty) view includes the external and internal appearance. The next example takes a structural (strength) view of how the walls support the load of the domes.

Example 8.5 : The cross-section of the Hagia Sophia in Fig. 8.5 is a view of its proportions and spaces. A key aspect of the architecture is the relationship between the round main dome and the square base provided by the supporting walls.

The load from the round dome is distributed to the square base by an innovative part called a *pendentive*; see Fig. 8.6.[13] The original dome collapsed during the earthquake of 558. The rebuilt dome is 30 feet higher to better distribute its weight to the supporting walls. □

8.4 Information Hiding and Modules

Information hiding is the principle of isolating design decisions to make software easier to change and understand. The design decisions are grouped and assigned to program units called modules. Modules are implemented using related collections of classes, methods, values, types, and other program elements.

The term *design decisions* refers to the decisions that guide an implementation of some task, including the choice of data representation, algorithms, and program organization.

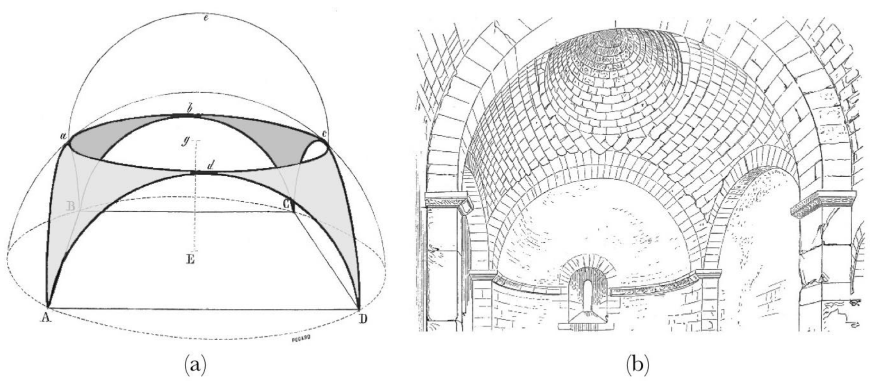


Figure 8.6: (a) The geometry of a pendentive (shaded). (b) The use of a pendentive.

A *module* contains a related collection of program elements that embody design decisions. A subset of the set of program elements is declared to be the *interface* of the module; the remaining program elements are said to be *private* to the module. The interface defines the functionality of the module: what the module does, and the interactions between this module and the other modules. The functionality is also called the *behavior* of the module. The private elements are referred to as the *implementation* of the module. The design decisions embodied in the implementation of a module are said to be *hidden* in the module.

Hidden design decisions are referred to as a module's *secrets*.

Example 8.6 : A store wants counts of the items that it sells, so it can manage its inventory and in-store displays.

How are the counts represented?
Where are they stored?
How is the top-selling item identified?

These questions relate to design decisions.

Information about how the counts are implemented can be hidden in a module M that is responsible for maintaining the counts. In other words, the specific representation of counts in M can be hidden from a module U that uses the counts to manage the store's inventory and displays. The using module U does not need to know which database system M uses to store the counts. Nor does U need to know whether the counts are stored locally in the store or remotely at a data center.

The using module U interacts with the implementing module M by asking questions such as the following:

What is the count for item i?
What is the item with the highest count?

The hidden implementation of the counts is the secret of module M.

Later, if the store wants real-time counts, as opposed to, say, day-old data, then the hidden implementation can be changed to meet the additional requirement, without perturbing the rest of the modules. □

Information hiding is more about conceiving and thinking of programs than it is about specific programming languages or constructs. Such constructs vary from language to language, and might be called modules or packages. Information hiding is also applicable to classes, although the examples in this chapter deal with collections of related classes.

8.4.1 Coupling and Cohesion

Information hiding results in systems with loose coupling and high cohesion, where *coupling* between modules is the degree to which they are inter-related, and *cohesion* within a module is the degree to which the elements of a module belong together.

Coupling. Two modules are *loosely coupled* if they interact only through their interfaces; they are *tightly coupled* if the implementation of one module depends on the implementation of the other. The following list progresses from looser (better) coupling to tighter (worse) coupling:

- *Message Coupling.* Modules pass messages through their interfaces.
- *Subclass Coupling.* A subclass inherits methods and data from a superclass.
- *Global Coupling.* Two modules share the same global data.
- *Content Coupling.* One module relies on the implementation of another.

Depending on the language, there may be other possible forms of coupling. For example, in a language with pointers, module A can pass module B a pointer that allows B to change private data in A.

Cohesion. A module has *high cohesion* if the module has one secret and all its elements relate to that secret; it has *low cohesion* if its elements are unrelated. The Unix philosophy of having each tool do one thing well leads to tools with high cohesion.[14] The following forms of cohesion reflect different approaches to grouping program elements into modules. The list progresses from higher (better) cohesion to lower worse cohesion:

- *Functional Cohesion.* Group elements based on a single well defined decision or functionality.

1. Understanding the rationale behind a piece of code	66%
2. Having to switch tasks often because of ... teammates or manager	62%
3. Being aware of changes to code elsewhere that impact my code	61%
4. Finding all the places code has been duplicated	59%
5. Understanding code that someone else wrote	56%
6. Understanding the impact of changes I make on code elsewhere	55%
7. Understanding the history of a piece of code	51%
8. Understanding who "owns" a piece of code	50%

Figure 8.7: Percentage of Microsoft developers agreeing that the statement represented a "serious problem for me." The results are from a 2005 survey.

- *Sequential Cohesion.* Group based on processing steps. The phases of a compiler—lexical analysis, syntax analysis, optimization, code generation—represent processing steps. A module with sequential cohesion can potentially be split into submodules, where the output of one submodule becomes the input to the next.
- *Informational Cohesion.* Group based on the data that is being manipulated. Such a module can potentially be split by grouping based on the purpose of the manipulations.
- *Temporal Cohesion.* Group based on the order in which events occur; e.g., grouping initializations or grouping housekeeping events that occur at the same time. Redesign modules based on object-oriented design principles.
- *Coincidental Cohesion.* The elements of a module have little to do with each other. Redesign.

8.4.2 Guidelines for Designing Modules

Although the concepts of information hiding, coupling/cohesion, and object-oriented design date back to the 1970s, software developers still face problems like the following:

> "Being aware of changes to code elsewhere that impact my code"
> "Understanding the impact of changes I make on code elsewhere"

These quotes are from a 2005 Microsoft survey of software architects, developers, and testers.[15] Of the top eight problems that emerged from the survey, seven relate to the modular structure of systems; see Fig. 8.7.

The following guidelines represent best practices for module design:[16]

- Begin with a list of significant design decisions or decisions that are likely to change. Then put each such design decision into a separate module so that it can be changed independently of the other decisions. Different design decisions belong in different modules.

- Keep each module simple enough to be understood fully.
- Minimize the number of widely used modules.
- Hide implementations, so that the implementation of one module can be changed without affecting the behavior and implementation of the other modules.
- Plan for change, so that likely changes do not affect module interfaces; less likely changes do not affect the interfaces of widely used modules; and only unlikely changes affect the modular structure of the system.
- Allow any combination of old and new implementations of modules to be tested, provided the interfaces stay the same.

8.5 Module Descriptions

Beyond a dozen or so modules, it becomes increasingly difficult to find relevant modules. This difficulty can be addressed by describing the rationale for the design of the modules in addition to detailed specifications of module interfaces.

The principle of information hiding was explored by David Parnas in the early 1970s.[17] As a demonstration of its applicability, Parnas and his colleagues built a modular system that duplicated the functionality of the flight software for a military aircraft, the A-7E.[18] The modular system had to meet all of the requirements of the existing flight software, including all of the real-time constraints. In addition, it had to be structured so that the overall behavior of the system could be inferred from the behavior of the modules, without looking at their implementations.

The modular system had hundreds of modules, many of them small ones. Finding relevant modules was a challenge. This challenge was addressed by

- organizing the modules into a hierarchy, and
- providing a guide to the modules, written in plain English.

Module hierarchies and module guides help with many of the problems identified in the Microsoft survey (see Fig. 8.7), including the following:

> "Understanding the rationale behind a piece of code"
> "Understanding code that someone else wrote"

Finding Relevant Modules. Someone new to the system can use the guide to navigate through the module hierarchy and find the modules that are relevant to a proposed change. Someone familiar with the system can use the guide to convince themselves that any change is propagated to all the modules that are affected by the change.

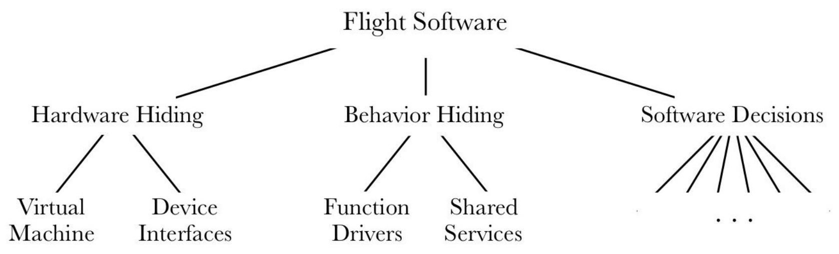


Figure 8.8: A partial module hierarchy for the A7-E flight software.

Validating the Design. The preparation of a hierarchy and a guide can be a useful exercise, for systems large and small. In the process of preparing them, a developer must think about and describe the design of each of the modules in the system, so the exercise serves as a validation of the design.

8.5.1 Module Hierarchy

A *module hierarchy* is formed by grouping related modules into a tree-structured hierarchy, where a parent module is composed from its child modules. By design, the secret of a child module is a subsecret of the secret of its parent module. One branch of the hierarchy can therefore be studied with minimal knowledge about modules that belong to unrelated branches.

Example 8.7 : For the A-7E flight software, the module hierarchy was grouped under three top-level modules:

- *Hardware-Hiding Module*, which implemented a virtual machine that was used by the rest of the software.

- *Behavior-Hiding Module*, which hid decisions related to the user requirements. The purpose of this module was to isolate the impact of external requirements changes.

- *Software-Decision Modules*, which hid implementation decisions, say, for algorithms and data structures.

A partial module hierarchy for the A7-E flight software appears in Fig. 8.8. The partial hierarchy shows only the top-level modules and their sub-modules. Furthermore, the names of the six sub-modules of the Software-Decision Module are not shown. □

- **Module Name**
  - **Textual Description**
    - The responsibility of the module
    - Overview and context for the service and the secret of the module
  - **Service Provided**
    - Service provided to the other modules through the module interface
  - **Secret**
    - Primary design decision that is hidden by the module
    - Any secondary design decisions that are needed for the implementation
  - **Error and Exception Handling**
    - List of possible errors and exceptions

Figure 8.9: Template for a Module Guide.

8.5.2 Module Guide

A *module guide* is a description of a module hierarchy in plain English. A plain English guide is intended to supplement, not replace, a specification of module interfaces. The purpose of a guide is threefold:

- provide an overview of the system;
- bring out the context and assumptions behind the design approach; and
- describe the responsibilities and behavior of the modules.

The template in Fig. 8.9 touches on the main points about a module that need to be covered by a guide. The template begins with the name and a plain English textual description of the module. The description includes the responsibility of the module and an overview of the design decisions that are hidden by the module. Any notes to the reader can also be included as part of the description.

The subsection on the module's service helps the reader find relevant modules. From the service, readers can decide whether the module is relevant to the aspect of the system that is of interest to them. If the module seems relevant, they can consult a module interface specification for more information.

The guide includes the module's secret, but not the implementation of the secret. Recall that the purpose of the guide is to allow the overall behavior of the system to be inferred, without looking at the implementation. In some cases, it is helpful to include secondary secrets that are uncovered during the implementation of the primary secret.

8.6 Software Product Lines

The term *family* originally meant a set of programs where it was worth

- first studying the common properties of the set and
- then determining the special properties of the individual family members.[19]

The term program family or *software product line* now refers to a set of programs that are specifically designed and implemented as a family. Each program or system in a family is a complete product in its own right.

The common properties of the family or product line are called *commonalities* and the special properties of the individual family members are called *variabilities*.

The annual Software Product Lines Conference has a Hall of Fame that lists about 20 organizations that have been honored for commercially successful product lines. The application areas for their product lines include automotive software, avionics, financial services, firmware, medical systems, property rentals, telecommunications, television sets, and training,[20]

Product lines arise because products come in different shapes, sizes, performance levels. and price points, all available at the same time. Without a family approach, each version would need to be developed and maintained separately. A family approach can lead to an order of magnitude reduction in the cost of fielding the members of the family.

Example 8.8 : HomeAway, a startup in the web-based vacation home rental market, grew quickly through acquisitions. Each acquired company retained the look and feel of its website.[21]

HomeAway's first implementation approach was to lump the systems for the various web sites together, with conditionals to guide the flow of control. This umbrella approach proved unwieldy and unworkable due to the various websites having different content management, layouts, databases, and data formats.

The second approach was to merge the various systems onto a common platform, while still retaining the distinct look and feel of the different websites. This approach had its limits:

> "A thorough code inspection eventually revealed that over time 29 separate mechanisms had been introduced for managing variation among the different sites."
>
> "Testing ... impoverished though it was, discovered 30 new defects every week—week after week—with no guarantee that fixing one defect didn't introduce new ones."

The company then turned to a software-product line apprach. Within weeks, the product-line approach paid for itself. The software footprint went down, quality went up, deployment times went down. Modularity meant that changes to one site no longer affected all the other sites. □

8.6.1 Software Architecture and Product Lines

Software architecture plays a key role in product-line engineering. To the guidelines for defining modules in Section 8.4 we can add:

- Address commonalities before variabilities, when designing modules.
- Hide each implementation decision about variabilities in a separate module. Related decisions can be grouped in a module hierarchy.

One of HomeAway's goals for a product line approach (see Example 8.8) was to make the cost of implementing a variation proportional to that variation, as opposed to the previous approaches, where the cost was proportional to the number of variations.

Support for the significance of architecture in product-line-engineering comes from SEI's experience with helping companies implement product lines:

> "The lack of either an architecture focus or architecture talent can kill an otherwise promising product line effort."[22]

8.6.2 Economics of Product-Line Engineering

Product-line engineering requires an initial investment. Here are some areas for investment: identify commonalities and variabilities; build a business case that encompasses multiple products; design a modular architecture that hides variabilities; create test plans that span products; and train developers and managers. One of the keys to the success of product-line engineering at Bell Labs was a small dedicated group that worked with development groups on their projects.

Management support is therefore essential. Product-line engineering projects that have lacked management support or initial investment have failed to deliver the promised improvements in productivity, quality, cost, and time to market.

The schematic in Fig. 8.10 illustrates the economic tradeoffs.[23] With the traditional approach, each family member is of built separately, so costs rise in proportion to the number and complexity of the family members. For simplicity, the schematic shows costs rising linearly with the number of family members.

With product-line engineering, there is an initial investment, which adds to the cost of the first product. The payoff begins as more products are delivered, since the incremental cost of adding a product is lower. Based on the Bell Labs experience, the crossover point is between 2 and 3 family members. The greater the number of family members, the greater the savings, past the crossover point.

8.7 Summary

Software architecture touches all aspects of a project. For customers, an architecture is helpful for confirming the project's direction. For development teams,

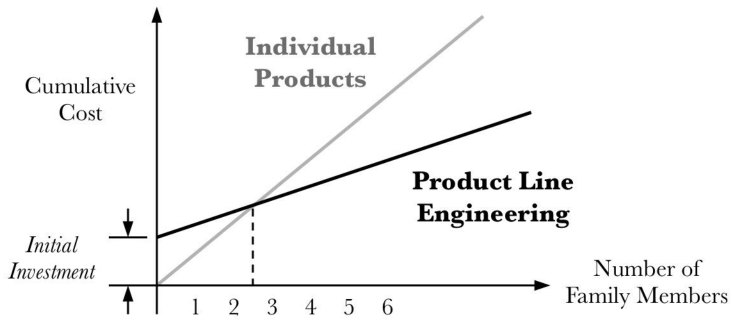


Figure 8.10: Schematic of the economics of software product lines.

it is helpful for assembling skills and resources and for assigning work to developers. For system architects, it is helpful for modeling system properties such as performance, security, and availability. For project managers, it is helpful for estimating budgets and schedules.

These many uses of architecture lead to multiple views of an architecture, where a *view* focuses on some aspect of the system or on the solution to some specific problem. The 4+1 grouping of views further illustrates the many roles for software architecture:

- *Logical Views* focus on end-user functionality and the problem domain.
- *Development Views* focus on program structure and the source text.
- *Process Views* focus on run-time objects and processes.
- *Physical Views* focus on system configurations and deployment.

These views are accompanied by *scenarios* or use cases, which span the four views.

Technically, a view is a structure, where a *structure* consists of a set of components and a binary relation on the components. Just as there are multiple views, there are multiple structure; e.g., the modular or component structure of the system and the calling or the "who calls who" structure for components.

No one view or structure defines an architecture. Hence the following definition (see Section 8.2.:

> "The *software architecture* of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

The main principle for creating an architecture is *information hiding*: isolate design decisions into program units called modules, so individual decisions can be changed without affecting the rest of the system. A *module* is a collection of related program elements, like classes, that implement design decisions.

The following guidelines for designing modules are from Sections 8.4 and 8.6:

- Isolate each significant design decisions into a separate module.
- Keep each module simple enough to be understood fully.
- Minimize the number of widely used modules.
- Hide implementations, so that they can be changed independently.
- Plan for change, so that likely changes are easy and only unlikely changes result in a redesign.
- Allow any combination of old and new implementations of modules to be tested, provided the interfaces stay the same.
- For a product family, design for commonalities before variabilities.
- Hide implementation decisions about variabilities in separate modules.

A *product family* or *product line* is a set of related programs that is specifically designed to be implemented together. The common properties of the family are called *commonalities* and the special properties of individual family members are called *variabilities.* Versions of a program can be treated as a family, as can successive working versions that are implemented during an iterative development process. Thus, the family approach is helpful for planning iterations of the same system.

Beyond a dozen or so modules, it is helpful to create a *module hierarchy*, a tree-structured grouping of related modules that share a secret; the secret of a submodule is a subsecret of its parents' secret. The *secret* of a module refers to the design decisions that are hidden in the module. In effect, the secret is the implementation of the behavior of the module.

Someone tasked with maintaining or changing a system would benefit greatly from a module guide. A *module guide* is a plain English description of the module hierarchy that provides an overview of the modules and includes the rationale for the design decisions that are relevant to a module.

A module guide is highly recommended for even a small system. For a large system with tens or hundreds of modules, it is invaluable.

Exercises for Chapter 8

Exercise 8.1 : Modules are a key concept in software architecture and design.

- What is Information Hiding? Define it, explain it, and give an example.
- What is a Module Guide? List its main elements and their purpose or roles.

Exercise 8.2 : Are the following statements True or False?

a) The Information Hiding Principle refers to hiding the design decisions in a module.

b) A Module Guide describes the implementation of each module.

c) A module with a secret cannot be changed.

d) A Module Interface Specification specifies the services provided and the services needed by modules.

e) With the XP focus on the simplest thing that could possibly work and on refactoring to clean up the design as new code is added, there is no need for architecture.

f) A system that obeys the Information Hiding principle is secure.

g) If module A uses module B, then A and B must have an ancestor-descendant relation in the module hierarchy.

h) If module A uses module B, then B must be present and satisfy its specification for A to satisfy its specification.

i) A Development View of an architecture specifies the internal structure of components.

j) Conway's "law" implies that the architecture of a system reflects the social structure of the producing organization.

Exercise 8.3: A video-streaming service wants to track the most-frequently requested movies. They have asked your company to bid on a system that will accept a stream of movie orders and incrementally maintain the top 10 most popular movies so far. Orders contain other information besides the item name; e.g., the movie's price, its category (e.g., Historical, Comedy, Action).

Exercise 8.4: Coupling is the degree to which modules are inter-related. Forms of coupling include:

a) *Message*: pass messages through their interfaces.

b) *Subclass*: inherit methods and data from a superclass.

c) *Global*: two or more modules share the same global data.

d) *Content*: one module relies on the implementation of another.

For each of the above cases, suppose modules A and B have that kind of coupling. How would you refactor A and B into modules $M_1, M_2, \cdots$ that comply with Information Hiding and provide the same services as A and B. That is, for each public function $A.f()$ or $B.f()$ in the interfaces of A and B, there is an equivalent function $M_i.f()$, for some refactored module M_i.

a) Using Information Hiding, give a high-level design. Include each module's secret.

b) Change your design to track both the top 10 movies and the top 10 categories; e.g., to settle whether Comedy movies are more popular than Action moves.

Exercise 8.5 : For the system in Exercise 8.3, you decide to treat the system as a product family because you recognize that the same approach can be applied to track top selling items for a retailer or the top most emailed items for a news company.

a) What do product family members have in common?

b) What are the variabilities; that is, how do product family members differ?

Exercise 8.6 : KWIC is an acronym for Key Word in Context. A KWIC index is formed by sorting and aligning all the "significant" words in a title. For simplicity, assume that capitalized words are the only significant words. As an example, the title `Wikipedia the Free Encyclopedia` has three significant words, `Wikipedia`, `Free`, and `Encyclopedia`. For the two titles

```
KWIC is an  Acronym for Keyword in Context
Wikipedia the Free Encyclopedia
```

the KWIC index is as follows:

```
                 KWIC is an   Acronym for Keyword in Conte
is an Acronym for Keyword in   Context
          Wikipedia the Free   Encyclopedia
               Wikipedia the   Free Encyclopedia
      KWIC is an Acronym for   Keyword in Context
                               KWIC is an Acronym for Keywo
                               Wikipedia the Free Encyclope
```

Design an architecture for KWIC indexes that hides the representation of titles in a module.

a) Give brief descriptions of the modules in your architecture

b) For each module, list the messages to the module and the corresponding responses from the module.

c) Give a module hierarchy

d) Describe the secret of each module

Notes for Chapter 8

[1]Kazman and Eden [8] propose the following distinction between architecture and design: architecture is non-local and design is local, where non-local means that the specification applies "to all parts of the system (as opposed to being limited to some part thereof)."

[2]The quote about the architecture of Internet Explorer 3.0 is from MacCormack [14, p. 77].

[3]Zheng et al. [26].

[4]This version of Conway's law [5] is from his web site `http://www.melconway.com/Home/Conways_Law.html` .

[5]See `http://www.sei.cmu.edu/architecture/start/glossary/community.cfm` for definitions of software architecture contributed by visitors to the site.

[6]Rather than add to the proliferation of definitions of software architecture, we follow Bass, Clements, and Kazman [2]. The authors were all at the Software Engineering Institute when they published the first two editions of their book.

[7]The notion of architecture as a subset of design follows Klein and Weiss [9].

[8]Views have been incorporated into the standards IEEE 1471 and ISO/IEC/IEEE 42010.

[9]The 4+1 grouping of architectural views is due to Kruchten [11].

[10]From Vitruvius's *de Architectura*, Book 1, Chapter 3, Verse 2: "All these should possess strength, utility, and beauty. Strength arises from carrying down the foundations to a good solid bottom, and from making a proper choice of materials without parsimony. Utility arises from a judicious distribution of the parts, so that their purposes be duly answered, and that each have its proper situation. Beauty is produced by the pleasing appearance and good taste of the whole, and by the dimensions of all the parts being duly proportioned to each other." `http://penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/1*.html`

[11]The cross section of the Hagia Sophia in Fig. 8.5 is from Lübke and Semrau [13]; see `https://commons.wikimedia.org/wiki/File:Hagia-Sophia-Laengsschnitt.jpg` .

[12]`https://en.wikipedia.org/wiki/Hagia_Sophia`

[13]The diagrams of the pendentives in Fig. 8.6 are adapted from Viollet-le-Duc [23]; see `https://commons.wikimedia.org/wiki/Category:Pendentives` .

[14]McIlroy and Pinson [15] note "a number of maxims [that] explain and promote" Unix style.

[15]LaToza, Venolia, and DeLine [12].

[16]The first guideline for module design is from Parnas [17]. The remaining guidelines are based on Britton and Parnas [4, p. 1-2].

[17]Parnas [17].

[18]Parnas, Clements, Weiss [19].

[19]The definition of program family is due to Parnas [18]. David Parnas credits the idea of program family to Dijkstra [6], who referred in passing to "a program as part of a family or 'in many (potential) versions.'"

[20]See `http://splc.net/fame.html` for the Software Product Line Conferences Hall of Fame.

[21]The HomeAway example is based on Kreuger, Churchett, and Buhrdorf [10].

[22]Northrop [16].

[23]See Weiss and Lai [25] for the economics of product-line engineering at Bell Labs. David Weiss led a group in Bell Labs Research that worked closely with a small dedicated group in the business unit to support product-line engineering across the parent company, Lucent Technologies.

References for Chapter 8

1. Mark A. Ardis and Janel A. Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal* 3, 3 (July-September 1998) 10-20.

2. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice* (3rd ed.). Addison-Wesley (2013).

3. Kent Beck. Embracing change with Extreme Programming. *IEEE Computer* (October 1999) 70-77.

4. Kathryn H. Britton and David L. Parnas. *A-7E Software Module Guide.* Naval Research Laboratory Memorandum 4702 (December 1981).

5. Melvin E. Conway. How do committees invent? *Datamation* (April 1968) 28-31.

6. Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*, J. N. Buxton and B. Randell (eds.) NATO Science Committee (April 1970) 84-88. Report on the NATO Software Engineering Conference in Rome (October 1969).

7. Birgit Geppert and Frank Roessler. *Multi-Conferencing Capability* United States Patent 8,204,195 (June 19, 2012).
8. Rick Kazman and Amnon Eden. Defining the terms architecture, design, and implementation, *news@sei* 6, 1 (First Quarter 2003).
9. John Klein and David Weiss, What is architecture? (2009) [21, p. 3-24].
10. Charles W. Kreuger, Dale Churchett, and Ross Buhrdorf. HomeAway's transition to software product line practice: engineering and business results in 60 days. *12th International Software Product Line Conference (SPLC '08).* IEEE (September 2008) 297-306.
11. Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software* (November 1995) 42-50.
12. Thomas LaToza, Gina Venolia, and Rob DeLine. Maintaining mental models: a study of developer work habits. *28th International Conference on Software Engineering (ICSE '06).* ACM, New York (2006) 492-501.
13. Wilhelm Lübke and Max Semrau. *Grundriß der Kunstgeschichte.* 14. Auflage. Paul Neff Verlag, Esslingen (1908).
14. Alan D. MacCormack. Product-development processes that work: How Internet companies build software. *Sloan Management Review* 42, 2 (Winter 2001) 75-84.
15. M. D. McIlroy, E. N. Pinson, and B. A. Tague. Foreword: Unix time-sharing system. *Bell System Technical Journal* 57, 6 (July-August 1978) 1899-1904.
16. Linda M. Northrop. SEI's software product line tenets. *IEEE Software* (July-August 2002) 32-40.
17. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (December 1972) 1053-1058.
18. David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (March 1976) 1-9.
19. David L. Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Trans. Software Engineering* SE-11, 3 (March 1985) 259-266.
20. Mary Shaw. Patterns for software architectures. *First Annual Conference on Pattern Languages of Programming.* (1994) 453-462.
21. Diomidis Spinellis and Georgios Gousios (eds.). *Beautiful Architecture.* O?Reilly Media, Sebastopol, Calif. (2009).
22. W. P. Stevens, G. J. Meyers, and L. L. Constantine. Structured design. *IBM Systems J.* **13**, 2 (June 1974) 115-138.
23. Eugène Viollet-le-Duc. *Dictionary of French Architecture from 11th to 16th Century.* (1856).
24. Marcus Vitruvius Polilo. *de Architectura.* (circa 15 BC). See `http://penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/1*.html` for the Joseph Gwilt English translation (1826), maintained by Bill Thayer.
25. David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process* Addison-Wesley, Reading Mass. (1999).
26. Min Zheng, Hui Xue, Yolong Zhang, Tao Wei, and John C. S. Lui. Enpublic apps: security threats using iOS enterprise and developer certificates. *10th ACM Symposium on Information, Computer and Communications Security (Asia CCS '15).* ACM, New York (April 2015) 463-474.

Chapter 9

Architectural Patterns

"Because patterns are (by definition) found in practice, one does not invent them; one discovers them. ... there will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions, and as long as those conditions change, new patterns will emerge."

— *Len Bass, Paul Clements, and Rick Kazman.*[1]

Rather than start each design from scratch, architects tend to adapt earlier successful designs for similar problems. Certain problems, such as the design of a graphical user interface or the design of a compiler, have been addressed over and over again in practice. The design of another graphical user interface is likely to have much in common with previous designs for graphical interfaces. Similarly, the design of another compiler is likely to have much in common with a previous design for a compiler.

An *architectural pattern* consists of a set of design decisions that address a recurring problem. These design decisions outline the components of a software solution, the behavior of the components, and the connections or relationships between the components. Since architecture is a subset of design, architectural patterns are a subset of design patterns.

There were early attempts to classify patterns in terms of the nature of their components and connectors. However, given the diversity of problems that occur in practice, any catalog of patterns is likely to be incomplete.

This chapter introduces some frequently occurring architectural patterns.

9.1 Alexander's Patterns

The above definition of architectural patterns builds on the work of the architect and urban planner, Christopher Alexander, who introduced the term *pattern* for a problem that occurs over and over again, together with a "the core of the solution to that problem."[2]

This notion of patterns has inspired object-oriented design patterns, software architecture patterns, and the *Pattern Languages of Programming* conferences. Extreme Programming was influenced by Alexander's work, especially the belief that the occupiers of a building should design it.[3]

Alexander's patterns range from large scale to small scale, from patterns for communities, to buildings, to rooms. The description of each pattern has three main parts: context, problem, solution. Let us consider them in the order problem, solution, context.

Problem. The problem addressed by a pattern is some fundamental aspect of a design, such as

- the design of the main entrance to a building, or
- the design of roofs for a cluster of buildings, or
- the design of the ceiling height of a room.

The Hagia Sophia consists of a cluster of buildings, with a high central building surrounded by lower wings with smaller rooms; see Fig. 8.5.

Solution. The "core" of a solution consists of guidance for designing a specific structure. For example, the pattern for ceiling heights includes the following guidance: "make ceilings high in rooms which are public or meant for large gatherings (10 to 12 feet), lower in rooms for smaller gatherings (7 to 9 feet), and very low in rooms or alcoves for one or two people (6 to 7 feet)."[4]

Context. Each pattern fits within a context, both larger and smaller.[5] For the larger context, consider that a roof completes a room, a room fits within a building, a building is part of a community. For the smaller context, consider the ceiling heights in a room or the positioning of windows in a room.

9.2 Software Layering

Layered architectures have been used for applications ranging from virtual machines to network protocols to enterprise applications to apps on mobile phones. Both Android and iOS have layered architectures. Apps running on a mobile phone have a top layer with application logic, middle layers providing media and platform services, and a bottom layer for the operating system.[6]

Informally, the modules in a layered architecture can be grouped into layers, where the layers are one on top of the other, as in a cake. They are depicted by

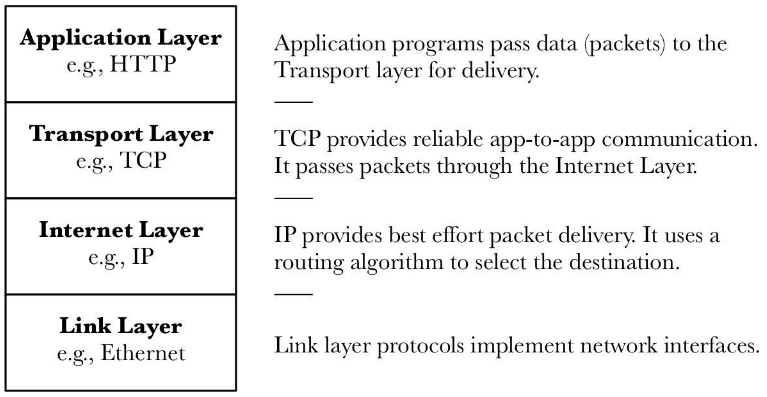

Application Layer
e.g., HTTP
Application programs pass data (packets) to the Transport layer for delivery.
Transport Layer
e.g., TCP
TCP provides reliable app-to-app communication. It passes packets through the Internet Layer.
Internet Layer
e.g., IP
IP provides best effort packet delivery. It uses a routing algorithm to select the destination.
Link Layer
e.g., Ethernet
Link layer protocols implement network interfaces.

Figure 9.1: Layers of the Internet Protocol Suite.

vertical diagrams, such as Fig. 9.1. Each layer is cohesive; it has a responsibility. Together, the modules in the layer carry out that responsibility.

The modules in a given layer use services provided by modules in the layer(s) below. The term "use" has a precise meaning: a module A *uses* module B if B must be present and satisfy its specification for A to satisfy its specification. In other words, A relies on B and B must work for A to work.

If A uses B and B uses A, then A and B belong in the same layer.

Example 9.1 : Internet Protocol (IP) and Transport Control Protocol (TCP) are the main protocols of the Internet. TCP is layered over IP. More precisely, TCP uses IP.[7] Their responsibilities differ:

- TCP supports reliable delivery, but packets might be delayed. TCP is *connection oriented*: it delivers packets between a source and a destination.
- IP supports timely delivery, but packets might be dropped. IP is *connectionless*: it forwards packets toward a destination. Successive packets may take different routes to get to the destination.

TCP and IP belong to the middle two layers of the Internet Protocol Suite; see Fig. 9.1. The layers describe software that runs in endpoints connected to the Internet. Starting at the top, applications such as web browsing and email in the Application layer use the Transport layer to deliver packets to another endpoint. The Application layer is not concerned with how the packets get to the other endpoint.

The TCP protocol in the Transport layer deals with end-to-end issues such as ensuring that packets are delivered reliably and in order; if a packet is missing, TCP requests retransmission of the packet through the Internet layer. TCP is not concerned with the topology or the connectivity of switches in the network.

The IP protocol in the Internet layer routes packets through the network and provides best effort service. Best effort means that there is no guarantee that packets will get to the other end. The network consists of switches connected by links that use various wired and wireless technologies. The Link layer isolates IP from the specifics of the network technology that is used to carry packets over a given link.

Originally, the roles of TCP and IP were performed by a single protocol. IP was separated from TCP, however, because a single protocol could not ensure both reliable and timely delivery of packets.[8]

Applications like email require reliable delivery; delay is acceptable, but packet loss (due to dropped packets) is not. Voice conferencing, on the other hand, requires timely delivery of speech packets; some packet loss can be smoothed over and tolerated, but delay is not acceptable.

Strict layering, where each layer uses only the layer just below it, has served the Internet well. For example, HTTP, an Application layer protocol for web browsing, was added in 1991, without touching the other layers. HTTP was added long after TCP and IP were designed in the 1970s. □

9.2.1 The Layered Pattern

Context. The objects that are relevant to an application are often quite different from the objects in an underlying implementation. A web browser is concerned with the retrieval and presentation of content. The browser uses a network, but does not know or care if the links in the network are wired or wireless. A payroll system is concerned with salaries and tax laws. The payroll system may use a database, but it need not know how the data is represented or stored.

Problem. Design a system where application concerns are separated from implementation concerns.

Core of a Solution. Partition the system into *layers*, where each layer consists of a cohesive set of modules. The layers are ordered and are depicted vertically, one on top of the other. The modules in a layer use only the modules in the layer below, unless explicitly stated otherwise. Design the layers, so that each layer has a specific responsibility.

For example, consider an application that runs on a virtual machine that runs on an underlying operating system. The application uses the services of the virtual machine, and the virtual machine uses the services of the underlying operating system.

9.2.2 Layered Pattern: Assessment

Strict layering, where each layer uses only the layer below, has proven enormously successful in settings such as the Internet, virtual machines, and mobile apps. Layers can be replaced and new layers added, without touching the other layers, as long as the interfaces are unchanged. With the Internet, the web-browsing protocol HTTP was added in 1991 by layering it over TCP. HTTP was added without touching the worldwide infrastructure of the Internet.

Strict layering favors clarity for the user over ease for the designer. The designer of an enhancement to a layered architecture must partition the enhancement to fit into the layers. The simple case, of course, is when the enhancement fits entirely within a layer.

Performance Tradeoff. Strict layering can result in a performance penalty. Consider Internet packets from a source to a destination. At the source, going down the protocol stack, each layer adds identifying and routing information in the form of a header. At the destination, going up the protocol stack, the headers are stripped to recover the payload between the source and the destination. With the Internet protocols the flexibility provided by layering is worth the overhead of transmitting and processing the headers.

The compromise between strictness of layering and performance depends on the setting. As we shall see in Section 9.5, the Portable C compiler compromised strictness to gain performance.

9.3 Dataflow: Pipe and Filter

A *dataflow network* consists of independent processes connected by unbounded queues. The concept was introduced by Melvin Conway, who called the processes *coroutines*. The motivating application for coroutines was the design of a compiler.[9] Google Dataflow provides flexible constructs for dealing with streams of data between processes. For example, advertisers can use it to set up dataflow networks to calculate "the time and length of each video viewing, who viewed it, and with which ad or content it was paired."[10]

A *pipeline* is a dataflow network in which the processes are connected in a line. Pipelines are named after the Unix *pipe* operator "|": if p and q are processes, then $p \,|\, q$ connects the output of p with the input of q. For the pipeline $p \,|\, q$ to work, the tools p and q must be written so that q can read what p writes.

Example 9.2 : The pipeline in Fig. 9.2 has four processes, represented by boxes. The arrows represent queues; the first arrow represents an input queue and the last arrow represents an output queue.

The processes in the pipeline transform the lines in a document into a sorted list of words, one per line. Suppose that a document has two lines:

```
Omit needless words!
Omit needless words!
```

The first process converts the document into a list of words, each word on a separate line. It does so by translating all non-alphabetic characters into

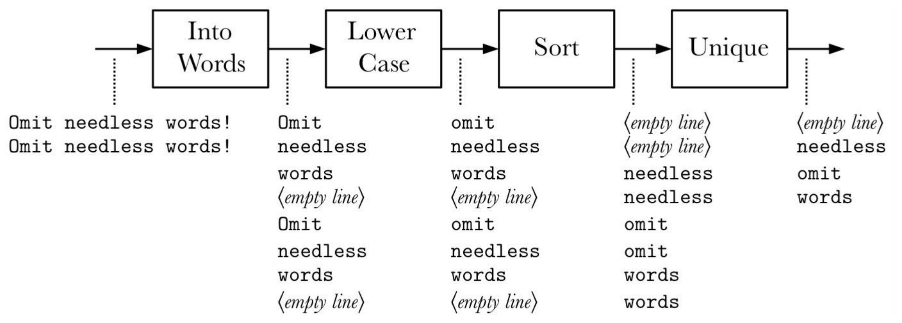


Figure 9.2: Pipeline for making a list of words in a document.

newline characters. Thus, the translation of each blank starts a new line, and so does the translation of each exclamation point, "`!`". The resulting 8-lines are shown between the first box and the second.

The second process translates uppercase letters into lowercase. The two instances of `Omit` are therefore each translated into `omit`. The third process in the pipeline sorts the lines. The last process removes adjacent duplicate lines.

The output of the last process in Fig. 9.2 is a sorted list of words, preceded by a blank line. We could another process to remove the blank line. □

9.3.1 The Pipe-and-Filter Pattern

Context. A large class of applications, from data mining to text processing, can be thought of in terms of sequences of operations on streams of data.

Problem. Assemble software applications from independent components that can be mixed and matched and used as building blocks.[11]

Core of a Solution. Connect the components so they form a pipeline, where the output of one component becomes the input to the next component in the pipeline. The components are referred to as *filters* since they transform or filter the data as it progresses through the pipeline.

This pipe and filter architecture pattern carries over to other configurations where the output of one component goes to several components or where a component has several inputs. Note that this pattern can be used for any data objects, not just text files.

A pipe and filter architecture where components can have multiple outputs or multiple inputs can be implemented on Windows using named pipes: "*Named pipes* are used to transfer data between processes that are not related processes and between processes on different computers."[12] Windows pipes can also be used to set up two-way communication between processes.

9.3.2 Unix Pipelines

The Unix pipe construct "|" leads to linear pipelines, where one tool follows the next in sequence. This linearity is for readability and for ease of use. The underlying implementation can support duplication of streams, so the output of one tool can become the input for more than one tool.

Example 9.3: For completeness, here is the Unix implementation of the pipeline in Fig. 9.2 for converting a document into a sorted list of words in the document:

```
tr -C a-zA-Z '\n' | tr A-Z a-z | sort | uniq
```

The Into Words box in Fig. 9.2 is implemented by using the `tr` (short for "translate") command, with appropriate parameters:

```
tr -C a-zA-Z '\n'
```

The flag `-C` represents "complement'," so the command translates any character that is not a lowercase or an uppercase letter, represented by `a-zA-Z`, into a newline character, represented by `'\n'`:

The translation from uppercase to lowercase letters (the second box in Fig. 9.2) is done by

```
tr A-Z a-z
```

To complete the explanation of Fig. 9.2, the `sort` command sorts its input and `uniq` removes adjacent duplicate lines from its input. □

9.3.3 A Dynamic Variant of Pipelines

There is more to dataflow networks than static linear pipelines, as the following example illustrates. The filters in the example are created dynamically and inserted into the pipeline at run time. [13]

Example 9.4: The Greek philosopher Eratosthenes is credited with the *sieve method* for computing prime numbers. The idea is that n is a prime if it is not a multiple of any prime smaller than n. Thus, 3 is a prime because it is not a multiple of 2. And, 5 is a prime because it is not a multiple of 2 and it is not a multiple of 3.

The dataflow network for generating prime numbers has three kinds of processes:

- Process *count* enumerates the integers, $2, 3, ...$
- Processes called *filter*(p), for $p = 2, 3, 5, ...$, where *filter*(p) removes multiples of p.
- Process *spawn*, which reconfigures itself by creating *filter*(p) whenever an integer p reaches it.

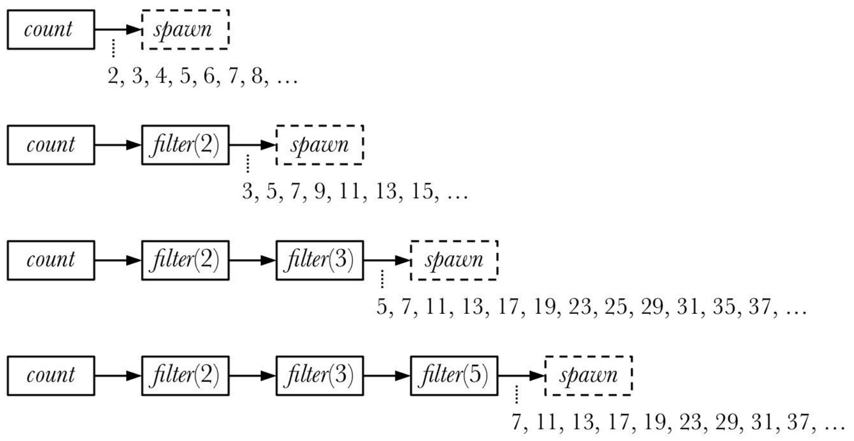


Figure 9.3: A reconfigurable dataflow network for generating prime numbers.

The dataflow network starts out with two processes, *count* and *spawn*; see the pipeline at the top of Fig. 9.3. When 2 reaches *spawn*, it creates *filter*(2) and inserts it to its left, as in the the second pipeline from the top. Since 3 is not a multiple of 2, it passes through *filter*(2) and reaches *spawn*, which creates and inserts *filter*(3), as shown in the third pipeline from the top of the figure.

The next integer from *count* is 4. It is a multiple of 2, so it is removed by *filter*(2).

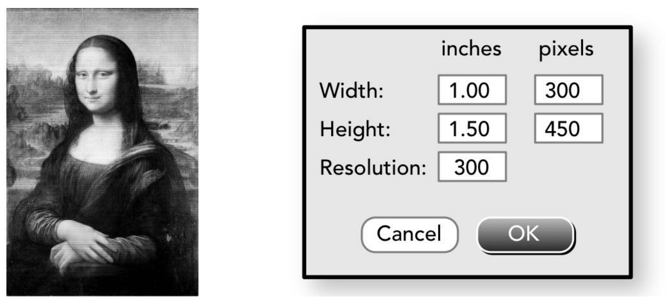
The next integer from *count*, 5, is neither a multiple of 2 nor is it a multiple of 3, so it reaches *spawn*, which inserts *filter*(5) into the pipeline, as shown in the last snapshot of the pipeline in the figure.

Since any integer p reaching *spawn* is not a multiple of any integer smaller than p, it must be a prime. □

9.4 User Interfaces: Model-View-Controller

Variants of the model-view-controller pattern have been used for interactive user interfaces ever since the pattern was introduced in 1979 for Smalltalk-80.[14] The terms model, view, and controller refer to logical components of an interface; a very simple interface may have a single module that performs all these logical functions.

This section begins with the distinction between a model and its views. The roles of models and views have remained essentially the same across the many variants of the architectural pattern. A *model* is a collection of related decisions and objects that implement an application. A *view component* manages a portion of the display. Let *screen view* refer to what a view component displays on a screen. When the context is clear, both view components and screen views



	inches	pixels
Width:	1.00	300
Height:	1.50	450
Resolution:	300	

Figure 9.4: Two views of a photo object.

will be referred to simply as *views.*

A *controller* links a model and one or more views. The role of a controller has evolved from the original Smalltalk-80 version, to the point where some of the variants have a different name: model-view-presenter. Complex user interfaces may require decisions and computations that are view specific, decisions that do not fit the role of the model. A *presenter* handles view-specific decisions and computations. This section has a sequence of examples that illustrate the roles of controllers and presenters.

9.4.1 Models and Views

The popular saying, "The map is not the territory," distinguishes between a collection of related objects (the territory) and a view of the objects (the map). There can be many maps for the same territory; for example, consider a satellite map, a street map, or a topological map. Even for the same map, say a street map, the level of information depends on the scale and size of the map. Additional details may appear on the screen as we zoom in from state to city to neighborhood.

Example 9.5 : For the distinction between a model and its views, consider the two views of a photo in Fig. 9.4. This example also illustrates that a view can have subviews.

The image view on the left displays the photo as it might appear on a screen. The dialog view on the right shows numeric values for the photo's height and width. The height and width are in both inches and in pixels; the values in inches are accompanied by the resolution in pixels per inch. For this example, assume that the proportions of the photo are fixed. That is, the ratio of height to width is fixed: if the height changes, the width changes accordingly, and vice versa.

Changes to the size of the photo through user interaction with one view trigger corresponding changes in other views. When the size in the image view

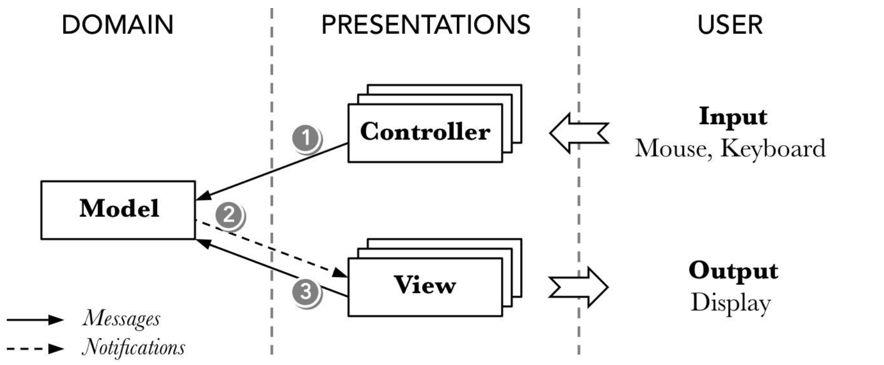


Figure 9.5: Handling input in the original Smalltalk-80 Model-View-Controller.

is changed by dragging a corner of the image, the heights and widths in the dialog view change accordingly. Similarly, if the numeric values in the dialog view are edited, then there are corresponding changes in the image view.

The dialog view is a composite: it is made up of subviews that display the dimensions of the photo in inches and in pixels. The view and its subviews are composed from four kinds of elements: text labels like "Height:" and "pixels"; text fields for values that can be edited by the user; dialog buttons for "Cancel" and "OK"; and a bitmap for the background. □

The distinction between models and views is one of the fundamental contributions of Smalltalk-80.

Example 9.6 : Based loosely on the original Smalltalk-80 model-view-controller, this example illustrates architectural decisions. The architecture may not be appropriate for practical use today.

The architecture in Fig. 9.5 separates the model from its views. The dashed lines separate the figure into three columns. The left column is for the real-world application domain, which is implemented by the model. The right column is for user interaction through a display for output to the user and through a mouse and keyboard for input from the user. The middle column, labeled Presentations, is for the components that handle user interaction.

The two main roles of the presentations are the handling of input and output. In this example, input and output are handled by controllers and views, respectively. There is a view-controller pair for an overall screen view and for each view element. For the two views of a photo object in Fig. 9.4, there is a view-controller pair for the image view on the left, and another view-controller pair for the dialog view on the right. There is also a view-controller pair for each of the elements in the dialog view. Each text field has its own view-controller pair; so does each button.

Views get their data from the model. In the dialog view in Fig. 9.4, suppose that the user increases the height from 450 to 600 pixels. This change in one field in the dialog view triggers changes in both the image view and the other fields in the dialog view itself.

In the image view, the representation of the photo must grow to reflect the increase in height. From Example 9.5, photo proportions are fixed, so the increase in height triggers a corresponding increase in width.

In the dialog view, the displayed dimensions in inches and pixels must all change. The width in pixels must change from 300 to 400. The height and width in inches must change from 1.00 and 1.50 inches to 1.33 and 2.00 inches, respectively.

How is a change in one field communicated to the other fields? The view-controller pairs are unaware of each other. How do they communicate?

The numbered arrows Fig. 9.5 illustrate how a view gets its data.

1. The change from 450 to 600 is interpreted by the controller for the height in pixels. The controller sends a message to the model with the value 600 pixels.

2. The model updates its state to change the height of the photo to 600 pixels and adjusts the width accordingly to 400 pixels. The model then sends out notifications of the changes, which are observed by the views.

3. The observing view components respond to the notifications by retrieving state information from the Model and updating their part of the display.

View-specific decisions are handled by Views. The views in this example are simple enough that decisions related to the views can be handled by the views themselves. Specifically, conversions from pixels to inches or vice versa can be done by the views. □

9.4.2 The Model-View-Controller Pattern

subsubContext User interaction with computers is predominantly through graphical and touch interfaces.

Problem. Given an application, design a user interface that supports user interaction through one or more views.

Core of a Solution. Partition the user interface into logical components called model, view, controller, and presenter. As we shall see, for complex views, a *presenter* handles view-specific decisions and computations.

Model. Recall that a model implements an application by a collection of related objects. The values of these objects represent the state of the model. For example, the model would hold the actual photo displayed in Fig. 9.4 and keep track of the photo's dimensions and resolution.

The other components are layered on top of the model. Thus, the other components use the model, but the model is unaware of them. In other words, the implementation of the model is independent of the implementations of the other components. Layering allows multiple views to use the same model.

Changes to the state are communicated to the other components through a list of observers maintained by the model. When the state changes, the observers are notified, and can retrieve state information from the model, as needed. The model is unaware of the identity of its observers.

View. Recall that a view manages a portion of the display. The information displayed by a view comes from the model. When a view is created, it puts itself on the model's list of observers to be notified of changes.

Controller. A *controller* links a model with one or more views. In the original Smalltalk-80 implementation, the controller was responsible for interpreting user input through a mouse and a keyboard. Based on the input, the controller sent updates to the model, as appropriate. For example, changes through either the image view or the dialog view in Fig. 9.4 would have gone through the controller to the model, so the model could update the dimensions of the photo.

As operating systems have taken over more and more of the role of interpreting user gestures, the role of the controller has diminished, to the point where the controller is combined with the view. The view then handles both input and output.

Presenter. For complex views, an optional *presenter* handles view-specific decisions and computations. Instead of observing a model directly, a view may interact indirectly, through a presenter.

A complex view may be more than a passive display of values from the model; it may involve some decisions and computations. For example, in a network map, should an overloaded link be highlighted by changing its color or by flashing it, or both. The decision that the link is overloaded belongs to the application; the decision about how to highlight the link belongs to the presentation of the link.

The rest of this section considers the evolution of the model-view-controller.

We get a *model-view-controller* when the presenter is not needed, or when its role is simple enough to be merged into the view or the controller. We get a *model-view-presenter* if the controller is not needed, or when its role is simple enough to be merged into the view or the presenter.

9.4.3 An Intermediate Step: Presentation Model

A presentation is *passive* if it simply displays information that it is provided, without additional decisions or computations. With passive presentations, the model-view-controller separation in Fig. 9.5 is clean. All values are computed by the model, and the computations are presentation independent.

But, what about complex presentations, which do involve logic that is specific to the presentation? As noted earlier, the decision about how to display an

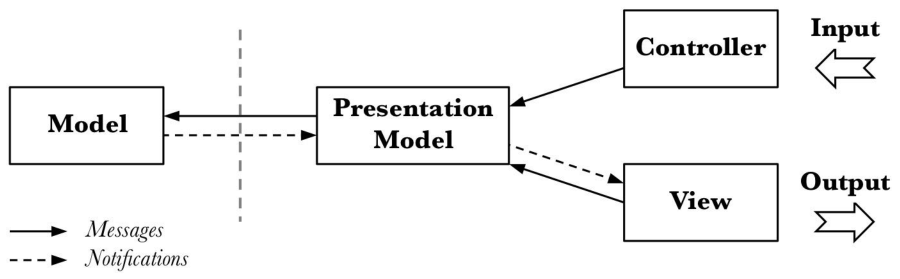


Figure 9.6: Isolate presentation decisions in the Presentation Model.

overloaded link is related to the presentation. Whether the link is overloaded is an application or domain decision. How the overloaded link is highlighted (change its color? animate it?) is a presentation decision. As another example of a complex presentation, consider a 3D rendering of a human brain. When a user interacts with the rendering by rotating it in 3D, the underlying data about the brain does not change. All that changes is the perspective on that data. The underlying data belongs with the application; the rendering belongs with the presentation.

Presentation logic does not fit cleanly into the original model-view-controller separation in Fig. 9.5. Where does it belong?

An early variant of Smalltalk-80 put presentation logic into an additional component shown as Presentation Model in Fig. 9.6.[15] The Presentation Model acts as a mediator between the Model and the View-Controller pair. Theoretically, the pair interacts with the Presentation Model is it did with the Model. The View continues to be responsible for the display of information and the Controller continues to be responsible for interpreting user gestures. In practice, the Presentation Model was often more tightly coupled with the View.

9.4.4 Model-View-Presenter

In the *model-view-presenter* architecture of Fig. 9.7, the roles of the view and the controller are combined. In the original Smalltalk-80 implementation, the operating system provided little support, so the controller had to do all the work of interpreting user gestures. When the operating system does more of the interpreting, the controller has less to do. It therefore makes sense to combine the view and controller, and let the view handle both input and output.[16]

Complex presentations are handled by putting presentation logic in the component called the Presenter in Fig. 9.7. The Presenter is therefore a variant of the Presentation Model in Fig. 9.6. The Presenter and the View work together, with the view doin

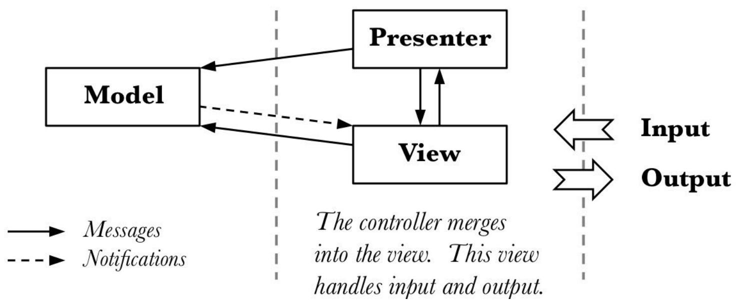

Presenter
Model
View
Input
Output
Messages
Notifications
The controller merges into the view. This view handles input and output.

Figure 9.7: The view in a model-view-presenter architecture, handles both input and output. For clarity, the

Testing User Interfaces

The problem with testing user interfaces is that it is hard to compare a display with the an expected snapshot of the display.

A passive view makes it easier to do automated testing of user interfaces. A passive view that simply displays values without need for decisions or computations. These values can be intercepted and compared during automated unit tests. The values are generated by the presenter. Messages from the presenter to the view can be intercepted for unit testing.

9.5 Case Study: Unix Portability

For a practical example of architectural principles and design tradeoffs, consider the Unix portability project. The project illustrates layering, information hiding, dataflow pipelines, and product families. It also illustrates design tradeoffs, since the designers were willing to trade some architectural purity for performance.

Unix is a portable operating system that has spawned numerous derivatives, including Linux. This family of operating systems runs on a range of devices, from smartphones to servers in data centers. Apple's iOS is a descendant of Unix; Android uses the Linux kernel.

The Unix operating system forms a machine-independent layer that insulates software applications from the underlying hardware. For example, see Fig. 9.8. The application can be ported or moved, essentially unchanged, to any other machine running Unix.

Unix was created around 1970 for the PDP-11, a minicomputer. In 1977, the Unix system kernel and much of its software were ported from the PDP-11 to a very different machine, the Interdata 8/32. Prior to the 1977 port of Unix, operating systems were closely tied to the machine: different machine, different operating system.

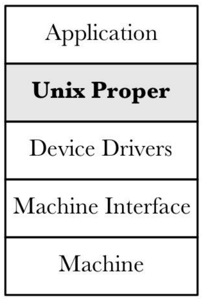


Figure 9.8: The Unix layer insulates an application from the underlying machine.

Steve Johnson and Dennis Ritchie's account of the Unix portability project describes both architectural principles and efficiency tradeoffs.[17] The project had three goals

- Write a portable C Compiler "that could be changed without grave difficulty to generate code for a variety of machines."
- Refine and extend the C language itself for portability.
- Port Unix by rewriting it in portable C, "detecting and isolating machine dependencies."

9.5.1 Portable C Compiler

The Portable C Compiler began as a product line with two members: the compilers for the PDP-11 and the Interdata 8/32. The family members shared the module hierarchy in Fig. 9.9. The compilers had a front end that hid language dependencies and a back end that hid machine dependencies. The modules at the leaves of the hierarchy formed a pipeline: lexical analysis, syntax analysis, expression tree matching, register allocation, and code generation.

The front end translated C programs into an intermediate representation consisting mostly of expression trees and stylized code for procedure entry and exit. The intermediate representation was independent of C. Thus, the secret of the front end was the source language, C.

The back end translated the intermediate representation into machine code. Thus, the secret of the back end was the machine.

Ideally, with strict information hiding, all machine dependencies would be hidden in the back end. The front end would then be 100% machine independent. In fact, the compiler traded some portability for efficiency, so 4,000 (87%) of the 4,600 lines in the front end were machine independent. Recall that the goal was a compiler that could be ported from one machine to another "without grave difficulty," as opposed to ported unchanged.

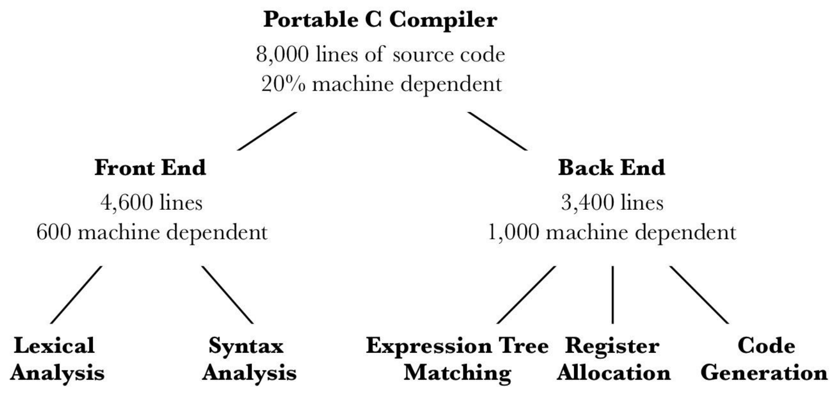


Figure 9.9: Module Hierarchy for a member of the Portable C Compiler family. The data about the proportion of machine-specific and machine independent lines of code is for the Interdata version.

The portable C compiler therefore contained rather than hid the underlying machine. Containment carried over to the implementation of the back end. The back end generated code for the underlying machine, so it was inherently machine dependent. Surprisingly, only 1,000 (29%) of the lines in the back end were machine dependent. Even in the machine dependent routines, only a third to half of the lines varied across machines.

Within months, the Portable C Compiler was running on a multitude of machines. In the machine independent portions, a bug could be fixed in all versions "almost mechanically."

As further validation of the design, a family of Fortran 77 compilers for the PDP-11 and Interdata 8/32 were soon created by reusing the back ends.

9.5.2 Porting Unix

Unix has a layered architecture, as illustrated in Fig. 9.10. The lowest layer, just above the machine, is the hardware/software interface, written in machine-specific assembly language. Most of the bugs appeared in the assembly language routines. The next layer consists of device drivers for handling interrupts, input/output, and errors.

The operating system proper is shown straddling the two machines, since this layer is essentially machine independent; on the Interdata, only 350 out of the 7,000 lines differed from the PDP-11 version. Finally, the top layer in Fig. 9.10 is for utilities such as assemblers, compilers, loaders, and debuggers, which are not part of the Unix kernel. (The hardware/software interface, the device drivers, and the operating system proper are part of the kernel.)

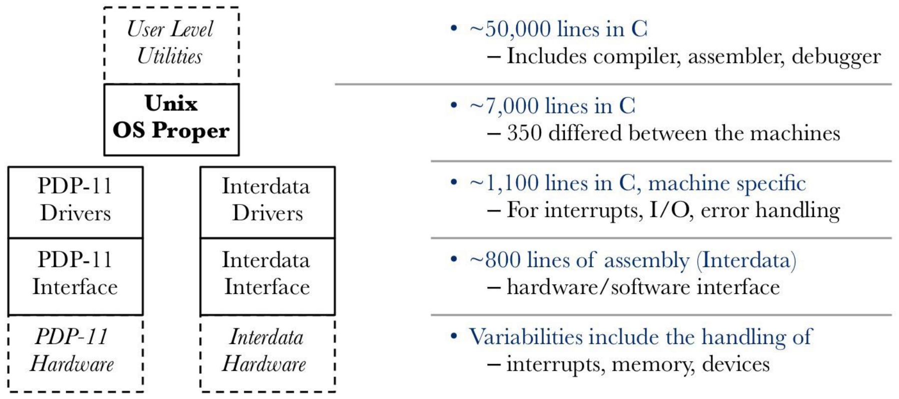

User Level Utilities
Unix OS Proper
PDP-11 Drivers
PDP-11 Interface
PDP-11 Hardware
Interdata Drivers
Interdata Interface
Interdata Hardware
• ~50,000 lines in C
– Includes compiler, assembler, debugger
• ~7,000 lines in C
– 350 differed between the machines
• ~1,100 lines in C, machine specific
– For interrupts, I/O, error handling
• ~800 lines of assembly (Interdata)
– hardware/software interface
• Variabilities include the handling of
– interrupts, memory, devices

Figure 9.10: Unix as a program family, running on two machines, the PDP-11 and the Interdata 8/32.

Overall, a high degree of portability was achieved, since the operating system proper (not including the hadware/software interface and the device drivers) was 95% the same on the two machines.

With user-level utilities, we need to distinguish between three times;

- *Compiler-Creation Time.* The time when the compiler itself is translated.
- *Compile Time.* The time when the compiler translates a source program in C into target code for a machine.
- *Run Time.* The time when the target code is run on the target machine.

The compiler is inherently machine-aware, since it has to generate target code at compile time.

Once the Unix kernel was ported to the Interdata, user-level utilities were ported (at compiler-creation time). The utilities included debuggers and assemblers, which had to be modified to work with Interdata code. Despite being machine-aware, user-level utilities were 75%-80% the same on the PDP-11 and the Interdata 8/32.

9.6 Summary

An *architectural pattern* consists of a set of design decisions that outline a solution to a recurring problem. Patterns are distilled from practical experience; they represent best practices for software architecture. This chapter considers some frequently occurring patterns.

9.6.1 The Layered Pattern

Context. The modules that are relevant to an application are often quite different from the objects in an underlying implementation. The application and implementation may themselves be addressing concerns that can be separated.

Problem. Design a system where application concerns are separated from implementation concerns.

Core of a Solution. Partition the system into *layers*, where each layer consists of a cohesive set of modules. The layers are ordered and are depicted vertically, one on top of the other. The modules in a layer use only the modules in the layer below, unless explicitly stated otherwise. Design the layers, so that each layer has a specific responsibility.

The term "use" has a precise meaning: a module A *uses* module B if B must be present and satisfy its specification for A to satisfy its specification. In other words, A relies on B and B must work for A to work.

9.6.2 The Pipe-and-Filter Pattern

Context. A large class of applications from data mining to text processing can be thought of in terms of sequences of operations on streams of data.

Problem. Assemble software applications from independent components that can be mixed and matched and used as building blocks.

Core of a Solution. Connect the components in a pipeline, where the output of one component becomes the input to the next component in the pipeline. The components are referred to as *filters* since they transform or filter the data as it progresses through the pipeline.

9.6.3 The Model-View-Controller Pattern

Context. User interaction with computers is predominantly through graphical and touch interfaces.

Problem. Given an application, design a user interface that supports user interaction through one or more views.

Core of a Solution. Partition the user interface into the following logical components:

- The *model* hides decisions related to the application domain. The model maintains a list of observers to be notified of changes to its state. It is unaware of the identity of its observers. Its implementation is independent of the other components.
- A *view* manages a portion of the display. Typically, views put themselves on the model's list of observers. They change the display when the model changes.
- A *controller* links a model with one or more of its views.
- For complex views, an optional *presenter* handles view-specific decisions and computations. Instead of observing a model directly, a view may interact indirectly, through a presenter.

We get a *model-view-controller* when the presenter is not needed, or its role is simple enough to be merged into the view or the controller. We get a *model-view-presenter* if the controller is not needed, or its role is simple enough to be merged into the view or the presenter.

Exercises for Chapter 9

Exercise 9.1 : Relate the following excerpt about beds and tables from Plato's *The Republic* (circa 380 B.C.)[18] to patterns:

> " there are beds and tables in the world—plenty of them, are there not?
>
> "Yes.
>
> "But there are only two ideas or forms of them—one the idea of a bed, the other of a table.
>
> "True.
>
> "And the maker of either of them makes a bed or he makes a table for our use, in accordance with the idea ...."

Notes for Chapter 9

[1]Bass, Clements, and Kazman concluded that "there will never be a complete list of patterns" [5, ch. 13]. Earlier, the authors had explored the classification of patterns in terms of temporal and static features [15].

[2]Alexander et al. [2] describe a sequence of 253 patterns. For the definition of patterns, see [2, p. x]. See pattern 116, Cascade of Roofs, for roofs for a building cluster and pattern 190, Ceiling Height Variety, for ceiling heights.

[3]Gamma et al. note that "The Alexandrian point of view has helped us focus on design trade-offs—the different 'forces that help shape a design." [12, p. 356]. Shaw [20] writes, "[Alexander's patterns] helped shape my views on software architecture." Beck [3] acknowledges the influence of Alexander's writings on Extreme Programming, For the history of the *Patterns of Programming Languages conferences*, see
`http://www.c2.com/cgi/wiki?HistoryOfPatterns` .

[4]Alexander et al. [2, p. 881].

[5]Alexander et al. [2, p. xiii]: “In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns that are embedded in it.”

[6]For the architecture of apps on mobile phones, see the overviews of Android [3] and iOS [4].

[7]The communication layers of Internet Protocol Suite are described in RFC 1122 [7].

[8]The separation of TCP and IP was motivated by applications such as conferencing and debugging. The cross-Internet debugger, XNET, needed access to all available information when dealing with stress or failure in the network; it mattered if packets were dropped or out of order. Dave Clark [9] traces the evolution of the design philosophy of TCP and IP. The original design of TCP was due to Robert Kahn and Vinton Cerf [8]. Dave Clark “joined the project in the mid 1970s and took over architectural responsibility for TCP/IP in 1981.” [9].

[9]Conway [10] partitioned programs so that the output of one module becomes the input to another and “the entire program can be laid out so that ... all information items flowing between modules have a component of motion to the right.”

Note that a dataflow network defines a relation on the set of modules, where a pair (p, q) is in the relation if the output of module p becomes the input of module q.

[10]Akidau et al. [1] describe the processing model of Google Cloud Dataflow. See `https://cloud.google.com/dataflow/examples/wordcount-example` for a word-count pipeline, similar to the pipeline in Example 9.2. Accessed November 7, 2015.

[11]Doug McIlroy [16] envisioned a catalog of standard components that could be coupled “like garden hose—screw in another segment when it becomes necessary to massage data in another way.”

[12]For named pipes on Windows, see
`https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574%28v=vs.85%29.aspx`
Accessed July 13, 2015.

[13]Doug McIlroy “cooked up” the prime-number sieve in Example 9.3.3 for a 1968 talk on coroutines [17].

[14]Trygve Reenskaug’a original May 1979 proposal for the Smalltalk user interface is entitled Thing-Model-View-Editor [19]. *Thing* referred to the application, “something that is of interest to the user,” *model* to the objects that represent the thing, *view* to a “pictorial representation” of the model, and *editor* to “an interface between the user and one or more views.” “After long discussions,” the editor was renamed *controller*.

[15]Fowler notes that the VisualWorks variant of Smalltalk put presentation logic into a component called Application Model. Following Fowler, the component is called Presentation Model in Fig. 9.6.

[16]Potel and Bower and McGlashan.

[17]Johnson and Ritchie [14] describe the Unix portability project in 1977. At the time, “Transportation of an operating system and its software between non-trivially different machines [was] rare, but not unprecedented.”

[18]Plato’s theory of Forms or theory of Ideas appears in many of the dialogues, including *The Republic*. The excerpt about beds and tables is from Book X.

References for Chapter 9

1. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernáncez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015) 1792-1803.

2. Christopher Alexander, Sara Ishikawa, Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York (1977).

3. Android Open Source Project. Android interfaces and architecture. `https://source.android.com/devices/`.

4. Apple Inc. iOS Technology Overview (September 17, 2014) `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iOSTechOverview.pdf`.

5. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice* (3rd ed.). Addison-Wesley (2013).

6. Andy Bower and Blair McGlashan. Twisting the triad. *European Smalltalk User Group (ESUG)* (2000).

7. R. Braden (ed). *Requirements for Internet Hosts: Communication Layers* Internet Engineering Task Force RFC-1122 (October 1989).

8. Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications* COM-22, 5 (May 1974) 637-648.

9. David D. Clark. The design philosophy of the DARPA Internet Protocols. *Computer Communications Review* 18,4 (August 1988) 106-114.

10. Melvin E. Conway. Design of a separable transition-diagram compiler. *Comm. ACM* 6, 7 (July 1963) 396-408.

11. Martin Fowler. GUI architectures (July 18 2006). `http://martinfowler.com/eaaDev/uiArchs.html`.

12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Mass. (1995).

13. Derek Greer. Interactive application and architecture patterns. (August 25 2007). `http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/`.

14. Stephen C. Johnson and Dennis M. Ritchie. Portability of C programs and the UNIX system. *Bell System Technical Journal* 57, 6 (July-August 1978) 2021-2048.

15. Rick Kazman, Paul Clements, Len Bass, and Gregory Abowd. Classifying architectural elements as a foundation for mechanism matching. *Computer Software and Applications Conference (COMPSAC '97)* (August 1997) 14-17.

16. M. Douglas McIlroy. Typescript (October 11, 1964). `http://doc.cat-v.org/unix/pipes/`.

17. M. Douglas McIlroy. Coroutine prime number sieve. (May 6, 2015). `http://www.cs.dartmouth.edu/~doug/sieve/`.

18. Mike Potel. MVP: Model-View-Presenter (1996). `http://www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

19. Trygve Reenskaug. The original MVC reports (February 12, 2007). `http://folk.uio.no/trygver/2007/MVC_Originals.pdf`.

Chapter 10

Software Quality: Reviews

"When we analyze our conception of quality, we find that the term is used in several different ways."

— *Walter A. Shewhart's 1931 classic book on manufacturing quality distinguishes between quality as it relates to customer wants, to manufacturing processes, and to inherent product properties. The software equivalents of these three forms of quality are functional, process, and product quality, respectively.*[1]

There are severe limitations on the number of chunks of information that people can "receive, process, and remember." We can keep track of about seven chunks, be they bits, words, colors, tones, or tastes. Beyond about seven, confusion and errors set in.[2]

No wonder software has defects! Software is complex. Even with a short seven-line program, it is easy to make a mistake. Large software systems can have thousands of unresolved defects, despite the best efforts of their developers.

Section 10.1 sets the stage for the discussion of software quality in this book. This chapter deals with static properties of systems—a *static* property can be analyzed without executing the program. By contrast, a *dynamic* property is a run-time property of the behavior of a program.

10.1 Overview of Software Quality

Terms like quality are inherently difficult to define since different stakeholders view quality differently. Users may focus on whether a product does what they want, whether it works reliably, and whether it is worth the price. Developers

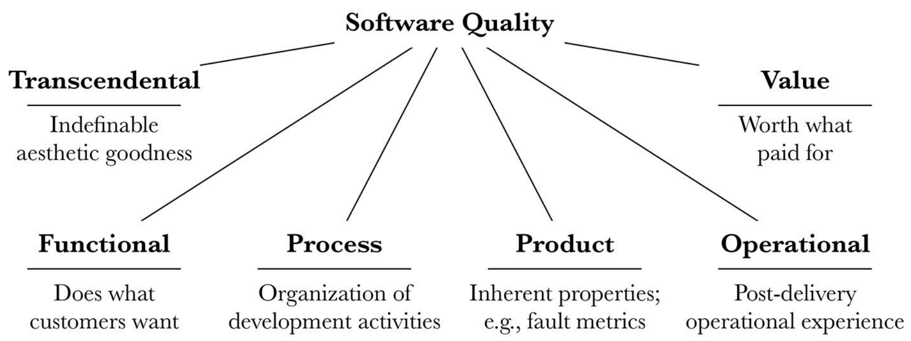

Software Quality
Transcendental
Indefinable aesthetic goodness
Value
Worth what paid for
Functional
Does what customers want
Process
Organization of development activities
Product
Inherent properties; e.g., fault metrics
Operational
Post-delivery operational experience

Figure 10.1: A model for software quality.

may focus on the product's code: whether it meets its specification, whether it is free of defects, and whether it is clean or sloppy. Thus, while users focus on what the code code does for them, developers focus on the quality of the code itself.

It is therefore more appropriate to talk about "qualities" (plural) than it is to talk about "quality" (singular).

10.1.1 Views of Software Quality

For our purposes, the term *software quality* is a general term for any of the six forms of quality illustrated in Fig. 10.1.[3]

Functional Quality

Functional quality is the degree to which a software system does what the user wants. Specifically, it is the degree to which the system meets user requirements for functionality. For example, functional quality might be measured in terms of how well the system supports the use cases or scenarios for the system.

Process Quality

Process quality has two possible interpretations:

- *Effectiveness.* With the effectiveness interpretation, process quality refers to the effectiveness of a process in organizing software development activities and teams.
- *Compliance.* With the compliance interpretation, process quality is the degree to which a process conforms to or follows the documented process model. This notion of process quality is motivated by manufacturing, where manufacturing processes are controlled to ensure consistency of output:

We shall take the effectiveness interpretation: process quality refers to process effectiveness. For example, iterative processes tend to be more effective than waterfall processes at delivering the right product on time and within budget. Note that process quality is relatively independent of the product being developed. Consider the process decision that all code must be reviewed before it becomes part of the code base. This decision is independent of the product.

Product and Operational Quality

Product quality refers to inherent properties of a product that cannot be altered without altering the product itself.[4] For example, the number of known defects in a system is a measure of product quality.

Operational quality refers to a customer's operational experience after the product is delivered. If a product does not work correctly or if it fails during operation, the customer's operational experience will suffer and the product will be said to have poor operational quality.

Example 10.1 : This example illustrates the distinction between product and operational quality. It mentions testing and code coverage, which will be discussed in Chapter 11. Hopefully, the following description is self-explanatory.

A team at Nortel spent months improving product quality:[5]

> "We spent almost 6 months in test between manual and automated testing. Then we went to the customer base, with [a code coverage] tool turned on ... what we found was that we only tested 1/10 of 1% of what they use. And of course it was a bad release in the customers' eyes."

In other words, the team spent 6 months improving product quality as measured by the number of defects in the source code. However, the defects they fixed through testing were largely in portions of the code that were not reached during operation. Enough defects remained in the portions of the code that customers did use that the operational quality of the product was poor.

For the next release the team focused on improving operational quality:

> "We changed the automation tools and manual testing to test what the customer used, about 1% of the code ... and the next release was a fantastic release in the customers' eyes.
>
> "Through this tool we also learned that the customer base rarely (aka almost never) used new features. It was all about everything that they currently use still working in the new release."

The conclusion is that customers care about operational quality, not product quality. (Developers typically focus on product quality.) The defects that matter to customers are the defects that they encounter when they use a product. □

Transcendental and Operational Quality

The remaining two forms of quality are included for completeness. *Transcendental quality* refers to the indefinable "I'll know it when I see it" goodness of a product. It refers to perceptions and aesthetics and cannot be measured. Finally, the *value* notion of quality refers to a customer's willingness to pay for a system.

10.1.2 Defects, Faults, and Failures

When discussing software quality, software engineers use a variety of terms, including anomaly, bug, defect, failure, fault, and error. Two of these terms—fault and failure—are widely used and have generally accepted meanings.

Distinction Between Faults and Failures

A *fault* is a flaw in a system. The flaw could be in the source code, the design, the documentation, or some other artifact related to the system. The term fault also applies to any deviations from programming style guidelines. Faults are a static property of a system. The number of faults is a measure of product quality.

A *failure* occurs when the behavior of an implementation is incorrect; that is, the behavior does not match the specification. Failures are a dynamic property of a system; they occur when the system is run. The number of failures is a measure of operational quality.

The following example explores the distinction between faults and failures.[6]

Example 10.2 : There had been 300 successful trial runs of the airborne guidance system for NASA's Atlas rockets. The system relied on signals from the ground to keep an Atlas vehicle on course during flight. Then, an Atlas Agena rocket was launched, carrying Mariner I, an unmanned probe to the planet Venus.

Shortly after launch, signal contact with the ground was lost. The developers had planned for such an eventuality. The airborne guidance system was supposed to behave as follows:

```
if not in contact with the ground then
    ignore course correction
```

But, due to a programming error, the **not** was missing. The guidance system blindly steered the rocket off course: hard left, nose down. At 293 seconds after launch, the rocket and its payload were destroyed by the safety officer.

The missing **not** was a fault in the source code. This fault lay undetected through 300 successful trial runs. During these trials, the fault did not trigger a failure.

The failure occurred when contact with the ground was lost and control reached the fault in the code. □

Severity of Defects

When precision is needed, we shall on the terms fault and failure. For convenience, the terms defect and error will be used as follows:

- A *defect* is a fault or an omission from a software artifact.
- An *error* is a fault, a failure, or an omission. The reason for this broad interpretation is that the term *error* has multiple meanings, including: (1) a fault in a software artifact; (2) the difference between actual and expected behavior (e.g., 10% sampling error); (3) a human action that results in a failure; (4) an undesirable system state, potentially leading to a failure.

All defects are not equal: some are critical; some are harmless. There is general agreement on the following four levels of *severity of defects*:

1. *Critical.* Total stoppage. Customers cannot get any work done.
2. *High.* Major error. Some required functionality is unavailable and there is no workaround.
3. *Medium.* Minor error. There is a problem, but there is a workaround, so the problem is an inconvenience rather than a roadblock.
4. *Low.* Cosmetic error. All functionality is available.

The lower the number, the more severe the defect. Companies tend not to ship a product with known critical or high severity defects.

10.2 Validation and Verification

Errors have to be found before they can be fixed. Error detection takes two forms:[7]

Validation:	"Am I building the right product?"
Verification:	"Am I building the product right?"

Validation refers to checking that a software artifact meets customer needs and requirements. With an iterative process, the development team relies on customer feedback to validate that it is building the right product. Customer acceptance testing is also a form of validation.

Verification refers to checking that an implementation is correct. Checking for correctness implies that there is a specification of what the system is supposed to do.

In terms of overall effort, verification is a much bigger problem than validation.[8]

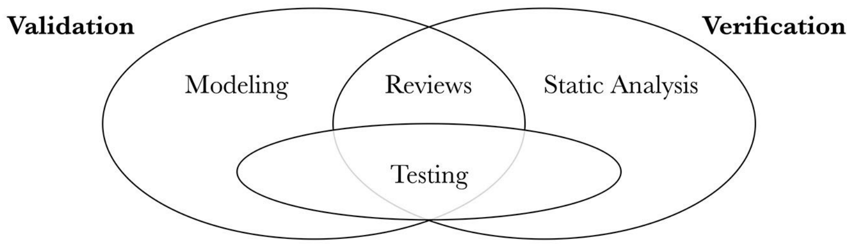


Figure 10.2: Selected techniques for verification and validation.

10.2.1 Techniques for Validation and Verification

Validation and verification techniques overlap, as illustrated in Fig. 10.2. The techniques are defined here and discussed in later sections.

- A *review* is a process or meeting for examining a software artifact for "comment or approval," according to IEEE Standard 1028-2008.[9] An *inspection* is a formal review by a carefully selected group of independent experts, with the purpose of finding "anomalies, including errors and deviations from standards and specifications."
- *Static analysis* examines a program without running it. The purpose of static analysis is to find potential defects. Note that the term defect applies to both flaws and to deviations from guidelines about programming practices.
- *Testing* is the process of running a program or component in a controlled environment to check whether the program behaves as expected. Often, testing produces output that is compared with the expected output.
- Modeling is beyond the scope of this book. It consists of building a theoretical or software model of a system or some aspect of a system to predict some properties or behavior of the system.

The following example illustrates verification techniques.

Example 10.3 : As discussed in Example 10.2, Mariner 1 was lost because of a programming error, a missing **not**. During a congressional hearing a few days after the loss, the committee members had difficulty understanding why such a simple error had not been uncovered before the launch. As one of the committee members put it (remarks edited):[10]

> Mr. FULTON. ... a loss up to $18 to $20 million, plus the time, plus the loss of prestige ... Doesn't any outside inspector check that the computers are correctly programmed?

The Mariner 1 loss touches on the following techniques:

- *Reviews.* During the congressional hearing, Mr. Fulton asked, "Doesn't any outside inspector check ...?" Formal reviews (inspections) are a cost-effective technique for error detection.
- *Static Analysis.* A missing **not** in a conditional can result in unreachable code. Unreachable code can be detected by static analysis. Such code can be a symptom of an underlying programming error.
- *Testing.* 300 trials failed to uncover the programming error. Testing cannot prove the absence of errors. □

The focus of architecture and design reviews in Section 10.3 is on validation: will the architecture meet customer needs? The focus of code reviews in Section 10.5 is on verification: does the code have errors? The description of software inspections (formal reviews) in Section 10.4 is relevant for both validation and verification; specifically, for both architecture and code reviews. Reviews are useful for finding flaws in any software-related artifacts, including designs, code, and documentation.

Static analysis, which is essentially an automated review, is covered in Section 10.6. Testing, in Chapter 11, spans validation and verification. As we shall see, there are several forms of testing,

10.3 Architecture Reviews

The primary purpose of an architecture review is to

a) clarify the goals of the proposed architecture and
b) confirm that a solution based on the architecture will meet those goals.

The goals of an architecture follow from customer needs, so architecture reviews are a validation technique; they improve functional quality.

Far too often, either the architecture does not adequately address the goals or the goals are not completely or clearly defined.[11] In addition a review may uncover other issues related to the project, such as the lack of management support or the inadequacy of the team's tools and domain knowledge.

10.3.1 Guiding Principles for Architecture Reviews

This section explores the guiding principles for reviews in Fig. 10.3.[12] A discussion of how to conduct a review will be deferred until Section 10.4, since similar review processes are used for both architecture and code reviews (Section 10.5).

The full benefits of reviews are realized when they follow a formal process. The guiding principles are also helpful for informal peer reviews, where developers go over each other's work.

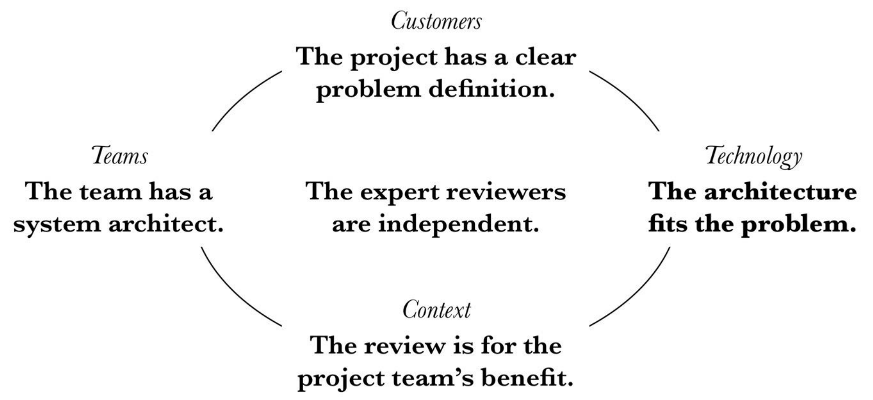

Customers
The project has a clear problem definition.
Teams
The team has a system architect.
The expert reviewers are independent.
Technology
The architecture fits the problem.
Context
The review is for the project team's benefit.

Figure 10.3: Guiding principles for architecture reviews.

The Review is For the Project Team's Benefit

The purpose of a review is to provide a project team with objective feedback. It is then up to the project team and its management to decide what to do with the feedback. The decision may be to continue the project with minor changes, to change the project's direction, or even to cancel the project.

Reviews have been found to be cost effective for both projects that are doing well and for projects that need help. They are not meant to be an audit on behalf of the project's management. They are not for finding fault or assigning blame.

Since the project team will be the one to act on reviewer feedback, members of the development team have to participate in the review. The team's participation also helps to build trust between the team and the reviewers, which increases the likelihood that the team will act on the feedback from the review.

The Expert Reviewers are Independent

For the review to be objective, the reviewers need to be independent of the project and its immediate management. For the review to be credible, the reviewers need to be respected subject-matter experts.

The independent experts may be either from outside the company or from other parts of the company. A side benefit of drawing reviewers from other parts of the company is that (a) they spread best practices to other projects and (b) the company builds up a stable of experienced reviewers.

The Project has a Clear Problem Definition

As discussed in Section 4.1.1, there may be multiple ways of addressing a customer need. For example, consider the following need and options for a problem definition:

Customer Need:	Listen to music
Option 1:	Offer songs for purchase and download
Option 2:	Offer a free streaming service with ads

During an architecture review, the independent experts respectfully provide feedback on the clarity and completeness of the problem definition.

Issues can arise with the problem definition if there are multiple stakeholders with conflicting needs or goals. For example, one stakeholder may be fanatical about keeping costs low, while another is equally passionate about maximizing performance. As another example, conflicts can arise when balancing convenience and security.

The Architecture Fits the Problem

The reviewers confirm that the architecture will provide a reasonable solution to the problem. Developers often focus on what the system is supposed to do; that is, they focus on the functional requirements for the system. They tend to pay less attention to non-functional goals, such as performance, security, and reliability. Reviewers therefore pay particular attention to non-functional requirements. Early reviews help because non-functional requirements such as performance and security need to be planned in from the start. Otherwise, they can lead to redesign and rework later in the project.

For example, with iterative and agile processes, early iterations focus on a minimal viable system, in order to get early customer feedback on the basic functionality. Special cases, alternative flows, and error handling get lower priority and are slated for later iterations. Concerns about non-functional requirements may not surface until late in the project. An early architecture review can prevent redesign and rework late in the project.

The Project has a System Architect

Projects that are important enough to merit a formal architecture review are important enough to have a system architect. The "architect" may be a person or a small team.

The reviewers rely on the architect to describe the architecture and provide the rationale for the design decisions. The reviewers also assess the team's skills. Does anyone on the team have prior experience with such a system? Are the tools new or known to the team?

10.3.2 Discovery, Deep-Dive, and Retrospective Reviews

The focus of a review varies from project to project and, for a given project, from stage to stage in the life of a project. The following three kinds of reviews are appropriate during the early, middle, and late stages of a project, respectively:

Problem Definition

- How will the customer benefit?
- What is the rationale for choosing this opportunity?

System Architecture

- What are the prioritized requirements?
- What are the main components of the system? How do they support the basic scenario?
- What is the desired performance? Scale? Availability?

Team

- Has the team built something like this before?
- Is the team co-located or distributed?

Constraints and Risks

- Are there any business constraints? Time to market?
- Are there any ethical, social, or legal constraints?
- What are the risks associated with the external technology and services?

Figure 10.4: A short checklist for an architecture review.

- A discovery review for early feedback.
- A deep dive for an evaluation of a specific aspect of the architecture.
- A retrospective for lessons learned.

An *architectural discovery review* assesses whether an architectural approach promises a suitable solution. A discovery review can begin as soon as preliminary design decisions are made, before an architecture fully exists. The reviewers focus on the problem definition, the feasibility of the emerging design, and the estimated costs and schedule. A short checklist of questions for a discovery review appears in Fig. 10.4; a somewhat longer list appears in Appendix A.

The benefit to the project team of an early discovery review is that design issues are uncovered early. The earlier a design issue is uncovered, the easier it is to address.

An *architectural deep dive* evaluates the requirements, the architecture, and high-level design of either the entire project, or of some aspect of the project. The project team may identify specific areas for feedback. The following is a small sample of focus areas from actual architecture reviews:[13]

user experience	performance	interoperability
user interface	security	software upgrades
disability access	reliability	deployment

Deep dives are conducted during the planning phase, before implementation begins.

An *architectural retrospective* is a debriefing to identify lessons learned that could help other projects. The reviewers ask what went especially well and what didn't. Retrospectives are useful for sharing best practices and recommendations for problems that other projects might encounter.

10.4 Software Inspections

An inspection is a formal review by a carefully selected group of independent experts. Software inspections have been widely deployed since Michael E. Fagan's influential 1976 paper. He wrote,

> "Substantial net improvements in programming quality and productivity have been obtained [at IBM] through the use of formal inspections of designs and of code."[14]

The Mars mission described in Section 1.2 had 145 software inspections, which surfaced 10,000 comments that were individually tracked and addressed by the project team.

Inspections are especially useful for non-executable artifacts, such as architecture descriptions, designs, test plans, and documentation. They are also good for identifying omissions, such as missing cases. In effect, inspections are a form of human testing.

For code, there are additional verification techniques, since code can be compiled, analyzed, and executed. Code reviews are discussed in Section 10.5, static program analysis in Section 10.6.

This section describes traditional inspections à la Fagan, before considering enhancements and variations.

10.4.1 Traditional Inspection

An inspection has four main phases (see Fig. 10.5): planning for an inspection; individual preparation by reviewers; a moderated examination of the review materials; and subsequent rework by the project team to address significant issues and findings. These phases are represented by the boxes in Fig. 10.5. To the left of the boxes are the goals of each phase in a traditional Fagan inspection. To the right of the boxes are some questions from empirical studies of the inspection process.

In a traditional inspection à la Fagan, the main event is a group meeting to detect and collect defects. The group meeting corresponds to the moderated examination phase in Fig. 10.5. The earlier phases—planning and preparation—are simply to prepare the review team, so they will be effective in the moderated group meeting.[15]

The basic flow of a traditional inspection appears in Fig. 10.6. The phases of the flow are explored below.

Screening Projects for Inspection

Since the purpose of an inspection is to provide the project team with objective feedback, the request for an inspection must come from the project team. Tthey are the ones to benefit from and act on the findings. As part of the request, the project team may indicate specific areas of focus for the inspection; e.g., usability, security, or reliability.

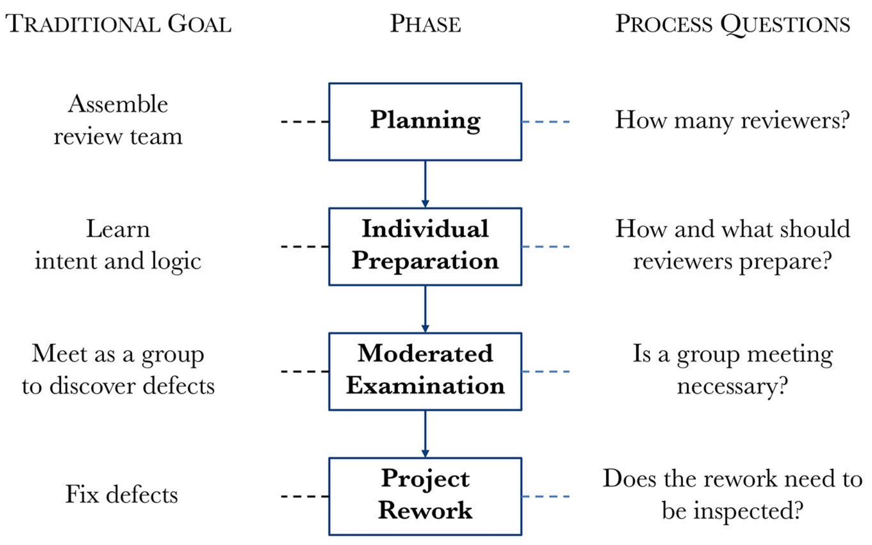


Figure 10.5: Phases of a software inspection. Goals from a traditional inspection are on the left. The questions on the right are based on empirical studies.

The inspection is not for assessing performance or for assigning blame. If the project team is pressured into having an inspection or is resistant to acting on the findings, the inspection could turn out to be a waste of everyone's time.

Companies typically have a screening process to prioritize requests for inspections. Inspections have a cost: they require a time commitment by the reviewers and the project team. Screening is based on the perceived cost effectiveness of an inspection.

Roles in a Traditional Inspection

The main roles associated with an inspection are as follows:

- *Moderator*. The moderator organizes the inspection and facilitates group interactions to maximize effectiveness. Moderators need special training; they need to be objective. Hence they must be independent of the project and its immediate management.
- *Author*. The author may be a person or a small team. The author prepares the materials for review and answers questions about the project.
- *Reviewer*. Reviewers need to be independent, so they can be objective. They can be drawn from other projects within the same company. See also the comments about reviewers in Section 10.3.1.

Planning
A project team requests an inspection.
The moderator assembles a team of independent reviewers.
The moderator confirms that the materials meet entry criteria.

Overview and Preparation
The moderator spells out the objectives.
The author provides an overview of the materials.
The reviewers study the intent and logic individually.

Group Meeting
The moderator facilitates and sets the pace.
The reviewers examine the materials for defects.
The reviewers conclude with preliminary findings.

Rework and Follow Up
The reviewers compile a report with significant findings.
The author reworks the materials to fix defects.
The moderator verifies that that issues are addressed.

Figure 10.6: The basic flow of a traditional software inspection.

The moderator role can be split into two: organizer of the inspection and moderator of the group interactions. Similarly, the author role can be split into two: author of the artifact and reader who paraphrases the content to be reviewed during group meetings.

The Planning Phase

The moderator assembles a team of independent reviewers. The moderator also ensures that the project team provides clear objectives and adequate materials for inspection. For example, for an architecture review, the project must provide a suitable architectural description. For a code inspection, the code must compile without syntax errors.

Overview and Individual Preparation

Individual preparation by the reviewers may optionally be preceded by a briefing session to orient the reviewers. During the briefing, the project team provides an overview of the project, drawing attention to the areas of focus for the inspection.

The reviewers then work on their own to prepare for the group meeting. While they may discover defects during preparation, this emphasis on this phase of a traditional review is on understanding—the group meeting is for defect detection and collection.

The Group Meeting

Ideally, the group meeting is face-to-face. The moderator sets the pace and keeps the meeting on track. The entire review team goes over the materials line-by-line to find defects. The detected defects are recorded: the meeting is for finding defects, not for fixing them. At the end of the review, the reviewers may confer privately and provide preliminary feedback to the project team.

Inspection meetings are taxing, so the recommended length of a meeting is two hours. Meetings for complex projects may take multiple day.

Rework and Follow Up

After the group meeting, the reviewers prepare a report with significant findings. The report classifies issues by severity: major issues must be addressed for the project to be successful; minor issues are for consideration by the project team. Finally, the moderator follows up to verify that the reported issues get addressed. The issues identified by the reviewers may include false positives—a false positive is an reported defect that turns out not to be a defect on further examination. The author must respond to all issues, even if it is to note that no rework is needed.

10.4.2 What Makes Inspections Work?

Software inspections have been studied extensively since they were introduced several decades ago. The cost of an inspection rises with the number of reviewers: how many are enough? A group meeting causes delays: is a meeting necessary? The rest of this section considers some questions about the process for conducting inspections.

How Many Reviewers?

The number of reviewers depends on the nature of the inspection: with too few, the review team may not have the required breadth of expertise; with too many, the inspection becomes inefficient. The fewer reviewers the better, not only for the cost of the reviewers' time, but because it takes longer to coordinate schedules and collect comments from the reviewers.

A typical inspection may have three to six reviewers. For code inspections, there is evidence that two reviewers find as many defects as four.[16]

Is a Group Meeting Really Necessary?

The group meeting is the main event of a traditional inspection. Reviewers are instructed to study the materials prior to the meeting. The meeting is where defect detection is expected to take place.

Experiments at Bell Labs in the 1990s questioned the necessity of a group meeting. In one study, 90% of the defects were found during individual preparation; the remaining 10% were found during the group meeting.[17] This data argues against group meetings. Such meetings are expensive. Schedule coordination alone can take up a third of the time interval for an inspection.[18]

How Should Reviewers Prepare?

Reviewers are often given checklists or scenarios to guide their individual preparation. Such guidance is to avoid two problems: duplication of effort and gaps in coverage. Duplication of effort occurs when multiple reviewers find the same defects. Gaps in coverage occur if defects remain undiscovered: no reviewer finds them.

In one study, reviewers who used scenarios or use cases were more effective and finding defects than reviewers who used checklists.[19]

10.5 Code Reviews

Code reviews have changed along with the tools and methods of software development. Instead of checking for defects, the primary motivation has changed to checking for intent—to checking that the code is readable and that it can be trusted to do what it is intended to do.[20]

Code reviews remain relevant, however. People are better than automated tools at getting to the root cause of a problem and in making judgements about design and style.

10.5.1 What has Changed?

Many of the consistency checks that were once done by human inspectors have been automated. Changes in programming languages and compilers have eliminated the need for questions like[21]

> "Have all variables been explicitly declared?"

> "Are there any comparisons between variables having inconsistent data types (e.g., comparing a character string to an address)?"

Static program analysis (see Section 10.6) eliminates the need for

> "Is a variable referenced whose value is unset or uninitialized?"

Run-time checking can address questions like

> "When indexing into a string, are the limits of the string exceeded?"

Design and style questions, however, still require human review and may never be automated. For example, consider

> "Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate."

	Traditional Inspections	Open-Source Code Reviews
Frequency	per Phase	per Commit
Code Size	Tens of lines	Hundreds of lines
Reviewers	Independent (3-6)	Invested (1-3)
Goal of the Review	Detect defects	Detect and Fix defects
Meet	Face-to-Face	Asynchronously
Elapsed Time	Days	Hours

Figure 10.7: Differences between traditional inspections and open-source code reviews.

10.5.2 Code Reviews Today

The differences between traditional code inspections and modern code reviews touch every aspect of a review. For concreteness the comparison in Fig. 10.7 is with open-source code reviews. Similar comments apply to companies like Google. Note that there may be exceptions to the general observations in Fig. 10.7.[22]

Invested Expert Reviewers

Open-source projects have hundreds of contributors, who are geographically dispersed.[23] A trusted group of core developers is responsible for the integrity of the code base.

Contributions are broadcast to a developer mailing list for review. Reviewers self select, based on their expertise and interests. They respond with comments and suggested fixes. While the number of reviewers for each contribution is small, 1-3, a larger community of developers is aware of the review.[24]

At Google, the main code repository is organized into subtrees with owners for each subtree.

> "All changes to the main source code repository MUST be reviewed by at least one other engineer."[25]

The author of a change chooses reviewers; however, anyone on the relevant mailing is free to respond to any change. All changes to a subtree must be approved by the owner of the subtree.

In general, self-selected reviewers have considerable expertise.

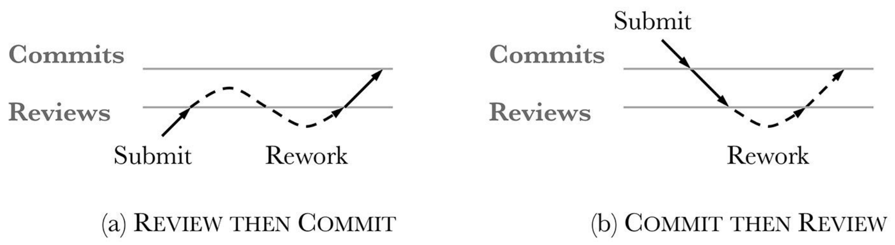


Figure 10.8: The Review-then-Commit and the Commit-then-Review processes.

Review Early and Often

It is a good practice to review all code before it is committed; that is, before it goes live in a production setting. Major companies, like Google, require a review before a commit. Open-source software projects require code from new contributors to be reviewed before it is committed.

Projects that review all code have frequent reviews of small pieces of code: tens of lines, say 30-50, instead of the hundreds of line in a traditional inspection. Code for review is self-contained, so reviewers see not only the change, but the context for the change.

The *Review-then-Commit* process is illustrated in Fig. 10.8(a). A contributor submits code for review. The reviewers examine the code for defects and suggest fixes. The contributor reworks the code and resubmits. Once the reviewers have no further comments, the code is committed and become part of the production code base. The dashed line indicates that rework is conditional on reviewers have comments that need to be addressed.

An alternative review process is followed if a change is urgent or if the contributor is known and trusted. The *Commit-then-Review* process is illustrated in Fig. 10.8(b). A contributor commits the code and notifies potential reviewers, who examine the change. Based on their comments, the commit either holds or the the change is rolled back and reworked. The dashed line represents the case in which the change is reworked until it is approved by the reviewers.

Asynchronous Rapid Responses

With self-selected geographically-distributed reviewers, code reviews are *asynchronous*: there is no group meeting. Reviewer comments are collected by email or though an online tool. Reviewers respond within hours.

10.6 Static Analysis

Static analysis examines a program without running it. The purpose of static analysis is to find anomalies, be they potential defects or deviations from guidelines about programming practices. Static analysis is effective enough that it has become an essential verification technique, along with reviews and testing.

What static analysis cannot do is to prove significant properties of a program, such as correctness, or predict the value of a variable at run time. The problem of proving such properties is equivalent to solving the famous "halting problem," which is known to be undecidable.

For example, we cannot predict whether the value of `x` will be `0` or `1` at the end of the following program fragment:

```
x = 0;
if( f() ) x = 1;
```

There can be no general algorithm to decide whether an arbitrary computation `f()` will halt, so we cannot predict whether control will get to the assignment `x = 1`.

What static analysis can do is to handle special cases. It can identify specific kinds of questionable program constructions. For example, by examining execution paths through a program, a static analyzer might identify lines of code that cannot be reached at run time. As another example, for some loops, a static analyzer might determine whether the loop is infinite—by examining the boolean expressions in the loop, the static analyzer might deduce that once control enters the loop, it can never leave.

In practice, static analyzers can exhaustively identify questionable constructions in millions of lines of code. A complete scan of the source code allows static analyzers to find defects that reviews and testing might miss. With the heightened interest in security, there is renewed interest in exhaustive static checking.

In short, some static analyzers can detect enough of the defects all of the time.

A drawback of static analysis, is that it can raise false alarms, along with flagging critical defects. In the past, the volume of false alarms was a barrier to adoption. Now, static analyzers use heuristics and deeper analysis to hold down the number of false alarms. They also prioritize their warnings: the higher the priority, the more critical the defect.

10.6.1 A Variety of Static Checkers

Automated static analysis tools consist of a set of *checkers* or *detectors*, where each checker looks for specific questionable constructions in the source code. Questionable constructions include the following:

- A variable is used before it is defined.
- A piece of code is unreachable (such code is also known as *dead code*).
- A resource *leaks*; that is, the resource is allocated but not released.
- A loop never terminates; e.g., its control variable is never updated.

This list of constructions is far from complete. New checkers continue to be defined, inspired by real problems in real code. The examples in this section are drawn from real problems in production code.

Static analyzers rely on compiler techniques to trace the flow of control and data through a program. *Control-flow analysis* traces execution paths through a program. *Data-flow analysis* traces the connections between the points in a program where the value of a variable is defined and where that value could potentially be used. The two are related, since data-flow analysis can help with control-flow analysis and vice versa.

Checking for Infinite Loops

The general problem of detecting infinite loops is undecidable, as noted earlier in this section. Many loops have a simple structure, however, where the value of a variable, called a *control variable*, determines when control exits the loop. Simple deductions may suffice for deciding whether such a loop is infinite.

Example 10.4: The following code fragment is adapted from a commercial product:

```
for ( j = 0; j < length; j-- ) {
    ...  // j is not touched in the body of the loop
}
```

If the value of `length` is positive, `j < length` will remain true as `j` takes on successively larger negative values.

An infinite loop is probably not what the programmer intended. □

Checking for Unreachable Code

Control-flow analysis is needed for detecting unreachable code. Such code is typically a sign of a bug in the program.

Example 10.5: For 17 months, between September 2012 and February 2014, a security vulnerability lay undetected in code running on hundreds of millions of devices. The code had been open sourced, available for all to see.

The vulnerability was in the Secure Sockets Layer (SSL) code for the operating systems for iPhones and iPads (iOS), and Macs (OS X). The vulnerability was significant, since it left the door open for an attacker to intercept communication with websites for applications such as secure browsing and credit-card transactions.

The vulnerability was due to a bug, in the form of an extra unwanted `goto`; see Fig. 10.9. The indentation of the second circled `goto` on line 9 is deceptive. Lines 7-11 have the following form:

```
 1) if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
 2)        goto fail;
 3) if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
 4)        goto fail;
 5) if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
 6)        goto fail;
 7) if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
 8)        goto fail;
 9)        goto fail;
10) if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
11)        goto fail;
```

Figure 10.9: A bug in the form of an extra "`goto fail;`" introduced a vulnerability into an implementation of SSL.

```
 7)  if( $condition_1$ )
 8)      goto fail;
 9)  goto fail;              // bug: unwanted goto
10)  if( $condition_2$ )
11)      goto fail;
```

Note that the unwanted goto on line 9 prevents control from reaching the conditional on line 10.

A static analyzer would have detected the unreachable code on lines 10-11. □

Checking for Null-Dereference Failures

Null is a special value, reserved for denoting the absence of an object. A *null-dereference failure* occurs at run time when there is an attempt to use a null value. Static analysis can detect potential null deferences.

The following Java program fragment assigns the value `null` to variable `logger` of class `Logger` and then promptly proceeds to use null value:

```
Logger logger = null;
...          // code that does not change the value of logger
logger.log(message);
```

Using data-flow analysis, we can deduce that the value of `logger` will be `null` when control reaches the last line of the above program fragment. At that point, a failure will occur: there will be no object for `logger` to point to, so the method call `log(message)` will fail.

```
1)  Logger logger = null;
2)  if (container != null)
3)      logger = container.getLogger();
4)  if (logger != null)
5)      logger.log(... + container.getName() + ...);
6)  else
7)      System.out.println(... + container.getName() + ...);
```

Figure 10.10: A program fragment from the open-source Apache Tomcat Server with a null-dereference bug.

Before reading the next example, can you find the potential null dereferences in Fig. 10.10?

Example 10.6 : The real code fragment in Fig. 10.10, avoids a null dereference for `logger`, but it introduces a null dereference for `container`.

Data-flow analysis would discover that `logger` in Fig. 10.10 is defined in two places and used in two places. The two definitions are on lines 1 and 3. The two uses are on lines 4 and 5. As for `container`, there are no definitions; there are four uses, on lines 2, 3, 5, and 7.

A null-dereference failure will occur if `container` is `null` when line 1 is reached. Control then flows from the decision on line 2 to line 4, leaving `logger` unchanged at `null`. From the decision on line 4, control therefore flows to line 7, which has a use of `container`. But `container` is `null`, so we have a null-dereference failure.

There is no null dereference if `container` is non-null when line 1 is reached, even if `container.getLogger()` returns `null`. □

The open-source static analyzer FindBugs would warn about two potential null dereferences for the uses of `container` on lines 5 and 7.[26] From Example 10.6 only the null dereference on line 7 is possible. The FindBugs warning about the use of `container` on line 5 is therefore a false alarm. Such false alarms are called false positives.

10.6.2 False Positives and False Negatives

A warning from a static analyzer about a piece of code is called a *false positive* if the code does not in fact have a defect. False positives arise when a static analyzer errs on the side of safety and flags a piece of code that *might* harbor a defect. If the piece of code does not in fact have a defect, then the warning is a false positive.

A *false negative* is a defect that is not detected by static analysis.

Static analysis tools choose to hold down the number of false positives at the expense of introducing some false negatives. The designers of a commercial static analyzer, Coverity, observe

> "In our experience, more than 30% [false positives] easily cause problems. People ignore the tool. ... We aim for below 20% for 'stable' checkers. When forced to choose between more bugs or fewer false positives we typically choose the latter."[27]

From the above quote, the designers of Coverity choose fewer false positives over fewer false negatives. More false negatives means more bugs missed. Other static analysis tools make the same choice.

10.7 Key Concepts and Terms

- A *static* property of a program can be analyzed without executing the program. By contrast, a *dynamic* property is a run-time property of the behavior of a program.
- *Software quality* is a general term for any of the following forms of quality:
  - *functional quality* is the degree to which a system meets user requirements for functionality;
  - *process quality* refers to the effectiveness of a process in organizing software development activities and teams;
  - *product quality* refers to inherent properties of a product that cannot be altered without altering the product itself;
  - *operational quality* refers to a customer's operational experience after the product is delivered;
  - *transcendental quality* refers to the indefinable goodness of a product;
  - the *value* notion of quality refers to a customer's willingness to pay for a product.
- A *fault* is a flaw in the source code, the design, the documentation, or some other software artifact. A *failure* occurs when the behavior of an implementation does not match the specification. Faults are a measure of product quality. Failures are a measure of operational quality.
- A *defect* is a fault or an omission from a software artifact. *Error* is a broad term for a fault, a failure, or an omission.
- The severity levels of defects are as follows (the lower the number, the more severe the defect):
  1. *critical* for total stoppage;
  2. *high* for major error that cripples some functionality;
  3. *medium* if there is a problem, but there is a workaround; and

4. *low* for a cosmetic error.

- *Validation* refers to checking that a software artifact meets customer requirements; or, "Am I building the right product?" *Verification* refers to checking that an implementation is correct; or, "Am I building the product right?" In terms of overall effort, verification is a much bigger problem than validation.
- A *review* is a process or meeting for examining a software artifact for comment or approval. An *inspection* is a formal review by a carefully selected group of independent experts, with the purpose of finding "anomalies, including errors and deviations from standards and specifications."
- The primary purpose of an *architecture review* is to clarify the goals of the proposed architecture and confirm that a solution based on the architecture will meet those goals. The guiding principles for architecture reviews are:
  - The review is for the project team's benefit.
  - The expert reviewers are independent.
  - The project has a clear problem definition.
  - The architecture fits the problem.
  - The project has a system architect.
- In a *traditional* or *Fagan inspection*, a group of 3-6 independent experts prepare in advance for a group meeting to detect and collect defects in a software artifact. Subsequent studies have shown that two committed expert reviewers may be enough and that a group meeting is not necessary. Instead, reviewer comments can be collected asynchronously by email or through an online tool.
- The primary motivation for *code reviews* has changed from checking for defects to checking for whether the code is clean and whether it can be trusted. It is a good practice to review all code before it is committed; that is, before it goes live in a production setting. *Asynchronous code reviews* are conducted using email and online tools. Small, say 30-50 line, contributions are examined by either named reviewers or by self-selected reviewers, based on their expertise and interest.
  - With a *Review-then-Commit* process, a contribution must be approved before it is committed.
  - If a change is urgent or if a reviewer is trusted, the *Commit-then-Review* process allows a contribution to be committed first and then reworked, if needed.
- *Static analysis* examines a program for defects without running the program. Static analysis is essentially an automated review.

- Automated static analysis tools consist of a set of *checkers* or *detectors*, where each checker looks for specific questionable constructions in the source code. For example, there are checkers for undefined variables, unreachable code, null-dereferences, and infinite loops.
- A warning from a static analyzer about a piece of code is a *false positive* if the code does not in fact have a defect. A *false negative* is a defect that is not detected by static analysis. Developers ignore static analyzers that produce too many false positives, so static analyzers hold down false positives at the risk of missing some defects; that is, at the risk of having some false negatives.

Exercises for Chapter 10

Exercise 10.1 : Explain the distinction between the following pairs of concepts:

a) failure and fault

b) process quality and product quality

c) validation and verification

d) a traditional inspection and an open-source code review

Exercise 10.2 : For each of the following words,

a) anomaly	b) bug	c) defect	d) flaw	e) failure
f) fault	g) error	h) glitch	i) omission	j) problem

- Look up the word in a dictionary and write down its dictionary meaning.
- Classify the dictionary meaning as being closer to that of fault; closer to that of failure; or not a fit with either fault or failure. Explain your answer.

Exercise 10.3 : For each of the following forms of software quality, associate two metrics to measure quality relative to that view: functional, process, product, and operational.

Exercise 10.4 : For a deep-dive architecture review, come up with 10 separate security-related questions.

Exercise 10.5 : For a deep-dive architecture review, come up with 10 separate performance-related questions.

Notes for Chapter 10

[1]in the 1920s, Walter A. Shewhart explored "the various definitions of quality ... to examine the basic requirements of effective specifications of quality." [25, p. 37].

[2]Miller [19] notes that "the accuracy with which we can identify absolutely the magnitude of a unidimensional stimulus variable ... is usually somewhere in the neighborhood of seven." Unidimensional refers to like chunks of information, such as bits, words, colors, and tones. Faces and objects differ from one another along multiple dimensions, hence we can accurately distinguish hundreds of faces and thousands of objects.

[3]Garvin [9] synthesized the varying definitions of product quality into five approaches: "(1) the transcendental approach of philosophy; (2) the product-based approach of economics; (3) the user-based approach of economics, marketing, and operations management; and (4) the manufacturing based and (5) value-based approach of operations management." Kitchenham and Lawrence Pfleeger [16] applied Garvin's model to software. The model in Fig. 10.1 splits the user-based approach into two: functional and operational.

[4]The definition of product quality in terms of properties that cannot be altered without altering the system is from Shewhart [25, p. 38].

[5]Example 10.1 is based on email from Gilman Stevens to Audris Mockus, May 14, 2014.

[6]Example 10.2 is based on a congressional hearing into the loss of Mariner 1 [27. 100-101].

[7]The right-product/product-right characterization of validation and verification is due to Boehm [5].

[8]Humphrey [12, p. 123] writes, "While the classical definition of product quality must focus on customer needs ... removing software defects consumes such a large proportion of our efforts that it overwhelms everything else."

[9]IEEE Standard 10-28-2008 defines five kinds of reviews: management reviews, technical reviews, inspections, walk-throughs, and audits [13].

[10]The edited account in Example 10.3 of the congressional hearing into the loss of Mariner 1 is based on the following exchange [27. 99-103]:

"Mr. Fulton. Does NASA check to see that the computers are correctly fed the equations? Doesn't any outside inspector check ...

"Dr. Morrison. This is a minute detail of the equations, which I agree should be checked. However, in good management practices, if we followed every detail to this point, we would have a tremendous staff.

"Mr. Fulton. ... the loss of up to \$18 or \$20 million, plus the time, plus the loss of prestige in the race with the Russians."

[11]Maranzano et al. [18] conducted over 700 architecture reviews between 1988 and 2005. Of the issues uncovered during the reviews, 29%-49% of the design issues could be categorized under "The proposed solution doesn't adequately solve the problem," and 10%-18% under "The problem isn't completely or clearly defined."

[12]The guiding principles for architecture reviews are adapted from Maranzano et al. [18]. They "estimate that projects of 100,000 non-commentary source lines of code have saved an average of US\$1 million each by identifying and resolving problems early". This estimate is based on reviews at companies that share a Bell Labs heritage.

[13]John Palframan, personal communication, September 2014.

[14]See Fagan [7] and his subsequent paper on software inspections [8]. See also the surveys [17, 2].

[15]Fagan's 1976 paper [7] had five phases: overview, preparation, inspection, rework, and follow-up. Based on experience with "hundreds of inspections involving thousands of programmers," his 1986 paper [8] added an initial planning phase and noted that "Omitting or combining [phases] led to degraded inspection efficiency that outweighed the apparent short-term benefits. Overview is the only [phase] that under certain conditions can be omitted with slight risk."

[16]Porter, Siy, Toman, and Votta found that there "was no difference between two- and four-person inspections, but both performed better than one-person inspections." [22, p. 338]

[17]Eick et al. [6, p. 64] found that 90% of defects were found during individual preparation. This data was collected as part of a study to estimate residual faults; that is, faults that

remain in a completed system.

[18]Votta [28] suggests two alternatives to group meetings: (a) "collect faults by deposition (small face-to-face meetings of two or three persons), or (b) collect faults using verbal or written media (telephone, electronic mail, or notes)."

[19]Porter and Votta [23] report on a study that found reviewers who used checklists were no more effective at finding defects than reviewers who used ad hoc techniques. Reviewers who used scenarios were more effective at finding defects.

[20]In a mid-1990s study of code inspections, Siy and Votta [26] found that 60% of all issues related to readability and maintainability, not to behavior or failure.

[21]The checklist questions in Section 10.5.1 are from a seminal book on testing by Myers [21, p. 22-32].

[22]Rigby et al. [24] review the "policies of 25 [open-source] projects and study the archival records of six large, mature, successful [open-source] projects". The six are Apache httpd server, Subversion, Linux, FreeBSD, KDE, and Gnome.

[23]Mockus, Fielding, and Herbsleb found that 458 people contributed to the Apache server code and documentation [20, p. 311]; 486 people contributed code and 412 people contributed fixes to Mozilla [20, p. 333].

[24]Rigby et al. [24, p. 35:11-35:13] counted a median of two reviewers for Review-then-Commit and one reviewer for Commit-then-Review.

[25]Henderson [10].

[26]David Hovemeyer "developed FindBugs as part of his PhD research "in conjunction with his thesis advisor William Pugh." [1] Example 10.6 is based on [11].

[27]Bessey et al. [3] describe the challenges in commercializing the static analyzer, Coverity.

References for Chapter 10

1. Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software* 25, 5 (September-October 2008) 22-29.

2. Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability* 12 (2002) 133-154.

3. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines fo code later: using static analysis to find bugs in the real world. *Comm. ACM* 53, 2 (February 2010) 66-75.

4. Mike Bland. Finding more than one worm in the apple. *Comm. ACM* 57, 7 (July 2014) 58-64.

5. Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software* (January 1984) 75-88. A 1979 version is available as technical report USC-79-501
`http://csse.usc.edu/TECHRPTS/1979/usccse79-501/usccse79-501.pdf` .

6. Stephen G. Eick, Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. Estimating software fault content before coding. *14th International Conference on Software Engineering (ICSE)* (May 1992) 59-65.

7. Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1876) 258-287.

8. Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering* SE12, 7 (July 1986) 744-751

9. David A. Garvin. What does "product quality" really mean? *Sloan Management Review* 26, 1 (Fall 1984) 25-43.

10. Fergus Henderson. Software engineering at Google. (January 31, 2017).
`https://arxiv.org/ftp/arxiv/papers/1702/1702.01715.pdf` .

11. David Hovenmeyer and William Pugh. Finding more null pointer bugs, but not too many. *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* ACM, New York (June 2007) 9-14.

12. Watts S. Humphrey. *The Watts New? Collection: Columns by SEI's Watts Humphrey.* Software Engineering Institute CMU/SEI-2009-SR-024 (November 2009). `http://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15035.pdf`.

13. IEEE Standard 1028-2008. *IEEE Standard for Software Reviews and Audits* (August 2008).

14. Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering* 3, 1 (1998) 9-35.

15. Stephen C. Johnson. *Lint, a C Program Checker.* Computing Science Technical Report 65, Bell Laboratories (July 26, 1978). `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841`.

16. Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: the elusive target. *IEEE Software* (January 1996) 12-21.

17. Olivier Laitenberger and Jean-Marc DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software* 50, 1 (2000) 5-31.

18. Joseph F. Maranzano, Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss. Architecture Reviews: Practice and Experience. *IEEE Software* (March-April 2005) 34-43.

19. George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* 101, 2 (1955) 343-352.

20. Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11, 3 (July 2002) 309?346.

21. Glenford J. Myers. *The Art of Software Testing.* John Wiley (1979).

22. Adam Porter, Harvey P. Siy, Carol A. Toman, and Lawrence G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997) 329-346.

23. Adam Porter and Lawrence G. Votta, Jr. What makes inspections work? *IEEE Software* 14, 6 (November-December 1997)

24. Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer review on open- source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology* 23, 4 (August 2014) 35:1-35:33.

25. Walter A. Shewhart. *Economic Control of Quality of Manufactured Product.* D. Van Nostrand, New York (1931). Reprinted by American Society for Quality Control (1980).

26. Harvey Siy and Lawrence G. Votta. Does the modern code inspection have value? *IEEE International Conference on Software Maintenance* (2001) 281-289

27. U.S. House of Representatives. *Hearing before the Committee on Science and Astronautics: Ways and Means of Effecting Economies in the National Space Program.* Eighty-Seventh Congress, Second Session (July 31, 1962).

28. Lawrence G. Votta, Jr. Does every inspection need a meeting? *1st ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '93).* Distributed as *Software Engineering Notes* 18, 5 (December 1993) 107-114.

Chapter 11

Software Quality: Testing

"... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."

— *Bill Gates, chairman and chief software architect of Microsoft, during an interview in 2002.*[1]

Software testing is the process of running a program in a controlled environment to check whether the program behaves as expected. The purpose of testing is to improve software quality. If a test fails—that is, the program does not behave as expected—there must be a fault, either in the program or in the specification of expected behavior. Either way, the test has provided feedback that can be used to remove the fault and improve quality.

At one time, coding and testing were distinct phases of software development. Testing accounted for half of the time and cost of development.[2] Coding was done by developers; testing was done by testers. Testers prided themselves on their ability to trigger failures and track down faults. Defects were more likely to be in the code for special or edge cases, since developers tended to pay more attention to the basic functionality.

Since then, test automation has blurred the distinction between coding and testing. Batteries of tests can be run automatically every time a change is made. Developers can thereby produce code of sufficient quality that the code can go straight from development to deployment. Automated tests are run as part of the deployment process to ensure that applications that used to work continue to work.

```
 1)  year = ORIGINYEAR; /* = 1980 */
 2)  while (days > 365)
 3)  {
 4)      if (IsLeapYear(year))
 5)      {
 6)          if (days > 366)
 7)          {
 8)              days -= 366;
 9)              year += 1;
10)          }
11)      }
12)      else
13)      {
14)          days -= 365;
15)          year += 1;
16)      }
17)  }
```

Figure 11.1: Where is the fault?

Although when and how tests are run may have changed, testing remains a significant part of software development. The principal techniques for defect detection and removal have remained the same. They are reviews, static analysis, and testing. Reviews and static analysis were covered in Chapter 10. This chapter explores the process of testing.

11.1 Overview of Testing

The code in Fig. 11.1 is from a digital music and video player. On December 31, 2008, owners of the player awoke to find that it froze on startup. On the last day of a leap year, the code in Fig. 11.1 loops forever.[3]

What went wrong?

Example 11.1: Suppose that the code in Fig. 11.1 is reached with variable `days` representing the current date as an integer. January 1, 1980 is represented by the integer 1; December 31, 1980 by 366 since 1980 was a leap year; and so on.

Variable `year` is initialized to 1980 on line 1. On exit from the while loop on lines 2-17, variable `year` represents the current year. The body of the loop computes the current year by repeatedly subtracting 366 for a leap year (line 8) or 365 for a non-leap year (line 14). Each subtraction is followed by a line that increments `year` (lines 9 and 15). In other words, the body of the loop counts down the days and counts up the years.

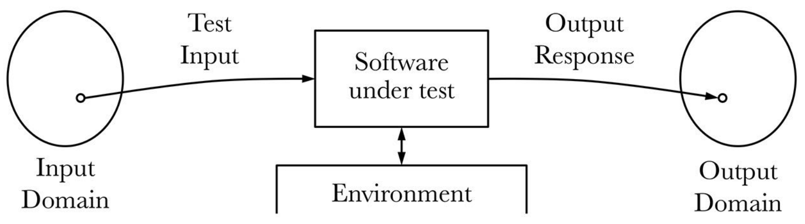


Figure 11.2: Software can be tested by applying an input stimulus and evaluating the output response.

On the 366th day of 2008, a leap year, line 6 is eventually reached with value 366 for `days`. Control therefore loops back from line 6 to line 2 with `days` unchanged. Ad infinitum. □

How is it that "simple" bugs escape detection until there is an embarrassing product failure? The rest of this section explores the process of testing and its strengths and limitations.

11.1.1 Issues During Testing

The issues that arise during testing relate to the four main elements in Fig. 11.2:

- *Software Under Test.* The software under test can be a code fragment, a component, a subsystem, a self-contained program, or a complete hardware-software system.
- *Input Domain.* A tester selects an element of some input domain and uses it as test input.
- *Output Domain.* The output domain is the set of possible output responses or observable behaviors by the software under test. Examples of behaviors include producing integer outputs, as in Example 11.1, and displaying a web page.
- *Environment.* Typically, the software under test is not be self contained, so an environment is needed to provide the context for running the software.

If the software under test is a program fragment, the environment handles dependencies on the rest of the program. The environment also includes the operating system, libraries, and external services that may be running either locally or in the cloud. In the early stages of development, external services can be simulated by dummy or mock modules with controllable behavior. For example, an external database can be simulated by a module that uses a local table seeded with known values.

Example 11.2 : Suppose that the software under test is the code on lines 1-17 in Fig. 11.1. The input domain is the set of possible initial integer values for the variable `days`. The output domain is the set of possible final integer values for the variables `year` and `days`.

The code in Fig. 11.1 cannot be run as is, because it needs a definition for `ORIGNYEAR` and an implementation for function `IsLeapYear()`. These things must be provided by the environment. (We are treating the initial value of variable `days` as an input, so the environment does not need to provide a value for `days`.) □

The following questions capture the main issues that arise during testing:[4]

- How to stabilize the environment to make tests repeatable?
- How to select test inputs?
- How to evaluate the response to a test input?
- How to decide whether to continue testing?

11.1.2 Test Selection

The judicious selection of test inputs is a key problem during testing. Fortunately, reliable software can be developed without exhaustive testing on all possible inputs—exhaustive testing is infeasible.

The Input Domain

The term *test input* is interpreted broadly to include any form of input; e.g., a value such as an integer; a gesture; a combination of values and gestures; or an input sequence, such as a sequence of mouse clicks. In short, a test input can be any stimulus that produces a response from the software under test.

A set of tests is also known as a *test suite.*

The *input domain* is the set of all possible test inputs. For all practical purposes, the input domain is typically infinite. Variable `days` in Fig. 11.1 can be initialized to any integer value and, machine limitations aside, there is an infinite supply of integers.

Some faults are triggered by a mistimed sequence of input events. Therac-25 delivered a radiation overdose only when the technician entered patient-treatment data fast enough to trigger a fault; see Section 1.3. Other faults are triggered by an unfortunate combination of values. Avionics software is tested for interactions between multiple inputs; e.g., a decision may be based on data from a variety of sensors and a failure occurs only when the pressure crosses a threshold and the temperature is in a certain range.

It is important to test on both valid and invalid inputs, for the software must work as expected or valid inputs and do something sensible on invalid inputs. Crashing on invalid input is not sensible behavior.

Input domains can therefore consist a single values, (b) combinations of values, or (c) scenarios consisting of sequences of values.

Black-Box and White-Box Testing

During test selection, we can either treat the software under test as a black box or we can look inside the box at the source code. Testing that depends only on the software's interface is called *black-box* or *functional testing.* Testing that is based on knowledge of the source code is called *white-box* or *structural testing.* As we shall see in Section 11.2, white-box testing is used for smaller units of software and black-box testing is used for larger subsystems that are built up from the units.

Test design and selection is a theme that runs through this chapter.

11.1.3 Test Adequacy: Deciding When to Stop

Ideally, testing would continue until the desired level of software quality is reached. Unfortunately, there is no way of knowing when the desired level of quality is reached because, at any time during testing, there is no way of knowing how many more defects remain undetected. If a test fails—that is, the output does not match the expected response—the tester has discovered that there is a defect somewhere.

But, if the test passes, all the tester has learned is that the software works as expected on that particular test input. The software could potentially fail on some other input. As Edsger Dijkstra put it,

> "Testing shows the presence, not the absence of bugs."[5]

Stopping or Test-Adequacy Criteria

A *test adequacy* criterion is a measure of progress during testing. Adequacy criteria support statements of the form, "Testing is $x\%$ complete." Test adequacy criteria are typically based on three kinds of information: code coverage, input coverage, and defect-discovery data.

- *Code coverage* is the degree to which a construct in the source code is touched during testing. For example, statement coverage is the proportion of statements that are executed at least once during a set of tests. Code coverage is discussed in Section 11.3 on white-box testing.
- *Input coverage* is the degree to which a set of test inputs is representative of the whole input domain. For example, in Section 11.4 on black-box testing, the input domain will be partitioned into equivalence classes. Equivalence-class coverage is the proportion of equivalence classes that are represented in a test set.
- *Defect-discovery data* includes data about the number and severity of the defects discovered in a given time interval. When combined with historical data from similar projects, the rate of defect discovery is sometimes used to make predictions about product quality.

Test adequacy criteria based on coverage and defect-discovery data are much better than arbitrary criteria such as stopping when time runs out or when a certain number of defects have been found. They cannot, however, guarantee the absence of bugs.

While testing alone is not enough, it can be a key component of an overall quality-improvement based on reviews, static analysis, and testing.[6]

11.1.4 Test Oracles: Evaluating the Response to a Test

Implicit in the above discussion is the assumption that we can readily tell whether an output is "correct;" that is, we can readily decide whether the output response to an input stimulus matches the expected output. This assumption is called the oracle assumption.

The *oracle assumption* has two parts:

1. There is a specification that defines the correct response to a test input.
2. There is a mechanism to decide whether or not a response is correct. Such a mechanism is called an *oracle*.

Most of the time, there is an oracle, human or automated. For values such as integers and characters, all an oracle may need to do is to compare the output with the expected value. An oracles based on a known comparison can be easily automated.

Graphical and audio/video interfaces may require a human oracle. For example, how do you evaluate a text-to-speech system? It may require a human to decide whether the spoken output sounds natural to a native speaker.

Questioning the Oracle Assumption

The oracle assumption does not always hold. A test oracle may not be readily available, or may be nontrivial to construct.

Example 11.3: Elaine Weyuker gives the example of a major oil company's accounting software, which "had been running without apparent error for years."[7] One month, it reported $300 for the company's assets, an obviously incorrect output response. This is an example of knowing that a response is incorrect, without knowing the right response.

There was no test oracle for the accounting software. Even an expert could not tell whether "$1,134,906.43 is correct and $1,135,627.85 is incorrect." □

Example 11.4: Consider a program that takes as input an integer n and produces as output the nth prime number. On input 1 the program produces 2, the first prime number; on 2 it produces 3; on 3 it produces 5; and so on. On input 1000, it produces 7919 as output.

Is 7919 the 1000th prime? (Yes, it is.)

It is nontrivial to create an oracle to decide if a number p is the nth prime number.[8] □

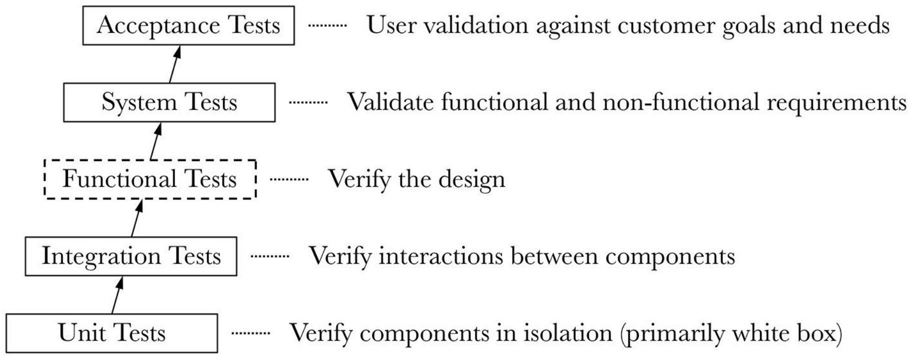

Acceptance Tests
User validation against customer goals and needs
System Tests
Validate functional and non-functional requirements
Functional Tests
Verify the design
Integration Tests
Verify interactions between components
Unit Tests
Verify components in isolation (primarily white box)

Figure 11.3: Levels of testing. Functional tests may be merged into system tests; hence the dashed box.

11.2 Levels of Testing

Testing becomes more manageable if the problem is partitioned: bugs are easier to find and fix if components are debugged before they are assembled into a larger system. A components-before-systems approach motivates the levels of testing in Fig. 11.3. Testing proceeds from bottom to top; the size of the software under test increases from bottom to top.

Each level of testing plays a different role. The terms “verify” and “validate” were introduced as follows in Section 10.2:

Validation:	“Am I building the right product?”
Verification:	“Am I building the product right?”

The top two levels in Fig. 11.3 validate that customer needs and requirements are met. The lower levels verify the implementation. System and functional testing may be combined into a single level that tests the behavior of the system; hence the dashed box for functional tests. The number of levels varies from project to project, depending on the complexity of the software and the importance of the application.[9]

Based on data from hundreds of companies, each level catches about one in three defects.[10]

11.2.1 Unit Testing

A *unit* of software is a logically separate component that can be tested by itself. It may be a module or part of a module. *Unit testing* verifies a unit in isolation from the rest of the system. With respect to the overview of testing in Fig. 11.2, the environment simulates just enough of the rest of the system to allow the unit to be run and tested.

```
public class Date {
   ...
   public int getYear(...) {
      ...
   }
   ...
}
```

(a) Software Under Test

```
public class DateTest {
   @Test
   public void test365() {
      Date date = new Date();
      int year = date.getYear(365);
      assertEquals(year, 1980);
   }
}
```

(a) A JUnit 4 Test

Figure 11.4: A JUnit test.

Unit testing is primarily white box testing, where test selection is informed by the source code of the unit. White-box testing is discussed in Section 11.3.

xUnit: Automated Unit Testing

Convenient automated unit testing profoundly changes software development. A full suite of tests can be run automatically at any time to verify the code. Changes can be made with reasonable assurance that the changes will not break any existing functionality. Code and tests can be developed together; new tests can be added as development proceeds. In fact, automated tests enable test-first or test-driven development, where tests are written first and then code is written to pass the tests.

Convenience was the number one goal for *JUnit*, a framework for automated testing of Java programs. Kent Beck and Erich Gamma wanted to make it so convenient that "we have some glimmer of hope that developers will actually write tests."[11]

JUnit quickly spread. It inspired unit testing frameworks for other languages, including CUnit for C, CPPUnit for C++, PyUnit for Python, JSUnit for JavaScript, and so on. This family of testing frameworks is called *xUnit*.

An xUnit test proceeds as follows:

```
set up the environment;
run test;
tear down the environment;
```

From Section 11.1, the environment includes the context that is needed to run the software under test. For a Java program, the context includes values of variables and simulations of any constructs that the software relies on.

Example 11.5 : The pseudo-code in Fig. 11.4(a) shows a class `Date` with a method `getYear()`. The body of `getYear()` is not shown—think of it as implementing the year calculation in Fig. 11.1.

The code in Fig. 11.4(b) sets up a single JUnit test for `getYear()`. The annotation `@Test` marks the beginning of a test. The name of the test is `test365`.

A descriptive name is recommended, for readability of messages about failed tests. Simple tests are recommended to make it easier to identify faults.

The test creates object `date` and calls `getYear(365)`, where 365 represents December 31, 1980. JUnit supports a range of assert methods; `assertEquals()` is an example. If the computed value `year` does not equal 1980, the test fails, and JUnit will issue a descriptive message.

For more information about JUnit, visit `junit.org`. □

11.2.2 Integration Testing

Integration testing verifies interactions between the components of a subsystem. Integration testing is typically black-box testing.

Prior unit testing increases the likelihood that a failure during integration testing is due to interactions between components instead of being due to a fault within some component.

With *big bang* integration testing, the whole system is assembled all at once from individually unit tested components. *Incremental* integration testing is a better approach: interactions are verified as components are added, one or more at a time, to a previously tested set of components.

Dependencies Between Modules

Incremental integration testing must deal with dependencies between modules.

Example 11.6 : In a model-view-controller architecture, the view displays information that it gets from the model. The view depends on the model, but not the other way around.

The model in Example 9.5 held information about a picture of the Mona Lisa, including a digital photo and the height, width, and resolution of the photo. There were two views: one displayed the digital photo; the other displayed the height, width, and resolution of the photo.

Both views got their information from the model, so they depended on the model. The model, however, was not dependent on the views. □

Dependencies between modules can be defined in terms of a uses relationship: module M *uses* module N, if N must be present and satisfy its specification for M to satisfy its specification.[12] Note that the used module need not be a subcomponent of the using module. In Example 11.6, the views used the model, but the model was not a subcomponent of either view.

During incremental integration, suppose module M uses module N. Then, either M must be added after N or there must be a "stub" that can be used instead of N for testing purposes. More precisely, module N' is a *stub* for N if N' has the same interface and enough of the functionality of N to allow testing of modules that use N.

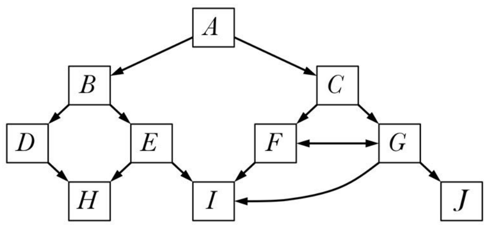


Figure 11.5: Modules to be integrated. The horizontal line between F and G means that they use each other.

Example 11.7: The edges and paths in Fig. 11.5 represent the uses relation between the modules in a system. Module A uses all the modules below it. A uses B and C directly; it uses the other modules indirectly.

A uses B, so B must be present and work for A to work. But, B uses D and E, so D and E must also be present and work for A to work. □

Top-Down Integration Testing

With stubs, integration testing can proceed top down. Testing of module A in Fig. 11.5 can begin with stubs for B and C. Then, testing of A and B together can begin with stubs for C, D, and E. Alternatively, testing A and C together can begin with stubs for B, F, and G.

A disadvantage with top-down integration testing is that stubs need to provide enough functionality for the using modules to be tested. Glenford J. Myers cautions that

> "Stub modules are often more complicated than they first appear to be."[13]

Bottom-Up Integration Testing

With bottom-up integration testing, a module is integrated before any using module needs it; that is, if M uses N, then M is integrated after N. If two modules use each other, then they are added together.

Bottom-up integration testing requires drivers: a *driver* module sets up the environment for the software under test. Automated testing tools like the xUnit family set up the environment, so there is no need for separate drivers.

Example 11.8: Consider the modules in Fig. 11.5. Any ordering that adds a child node before a parent node can serve for incremental integration testing, except for modules F and G, which use each other. They must be added together.

Here are two possible ordering for bottom-up testing:

H, D, I, E, B, J, F and G, C, A
J, I, F and G, C, H, E, D, B, A

□

11.2.3 Functional and System Testing

Functional testing verifies that the overall system meets its design specifications. *System testing* validates the system as a whole with respect to customer requirements, both functional and non-functional. The overall system may include hardware and third-party software components.

Functional testing may be merged into system testing. If the two are merged, then system testing performs a combination of verification and validation.

System testing is a major activity. Non-functional requirements include performance, security, usability, reliability, scalability, serviceability, documentation, among others. These are end-to-end properties that can be tested only after the overall system is available.

Testing for properties like security, usability, and performance are important enough that they may be split off from system testing and conducted by dedicated specialized teams.

11.2.4 Acceptance Testing

Acceptance testing differs from the other levels of testing since it is performed by the customer organization. Mission-critical systems are usually installed in a lab at a customer site and subjected to rigorous acceptance testing before they are put into production. Acceptance tests based on usage scenarios ensure that the system will support the customer organization's business goals.

11.2.5 Case Study: Test Early and Often

Testing of a component can begin as soon as the component is coded, while other components are still in an earlier stage of development.

The following example is motivated by the development process for a highly reliable communications product. The development team for the product was made up of small subteams that worked independently on separate components called features. The code for the features was then integrated to form the product. Testing was a priority. The development process called for extensive unit, functional, system, and performance testing.

Example 11.9 : The process in Fig 11.6 focuses on the building phase of a project. The process does not show the initial activities to identify customer needs and opportunities and to define a product.

The process in Fig. 11.6 begins with system design. During system design, the components of the system are designed and plans are drawn to develop the

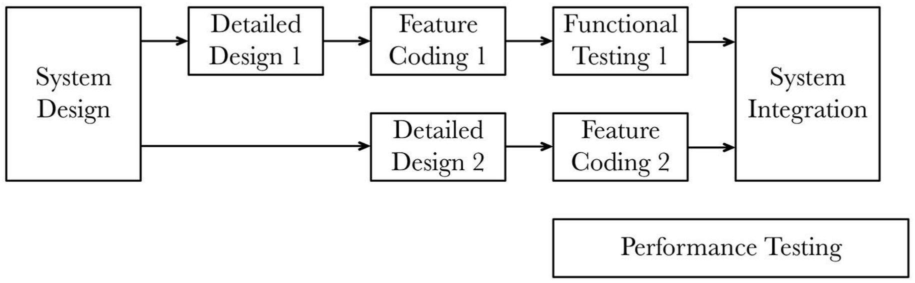


Figure 11.6: Simplified version of a plan-driven process with an emphasis on testing.

components independently. Between initial system design and final system integration are two parallel subprocesses for developing the components, referred to as Feature 1 and Feature 2.

The development of the two features is staggered in time: coding for Feature 1 overlaps with the detailed design of Feature 2; functional testing of Feature 1 overlaps with the coding of Feature 2.

The roles for the process in Fig. 11.6 are project manager, designer, coder, and tester. Parallel development of features allows the same designers and coders to work on both features. After the initial system design, the designers work first on the detailed design for Feature 1 and then on the detailed design for Feature 2. Similarly, the coders work sequentially on coding for Feature 1 and Feature 2. Functional testing of Feature 1 begins as soon as the code for the feature is available.

Once both features have been coded, the testers begin system integration and testing. Functional testing of Feature 2 is combined with system integration. Another group of testers begins performance testing once the first feature has been coded.

The project manager is responsible for assembling the team and coordinating the work of designers, coders, and testers. □

Instead of two features, the product that motivated Example 11.9 had multiple features. Overall planning and system design were done up front, before detailed design, coding, and testing.. Careful planning ensured that the independently developed components came together as designed during system integration. Not only did the project meet its functionality, reliability, and performance goals, it was on time and under budget.

11.3 Testing for Code Coverage

Since program testing cannot prove the absence of bugs it is unrealistic to expect that testing must continue until all defects have been found. The test-adequacy criteria in this section are based on code coverage, which is the degree to which a construct in the source code is executed during testing. In practice, the adequacy of test sets is measured relative to coverage targets.

Code coverage is closely, but not exclusively, associated with white-box testing, where test selection is based on knowledge of the source code.

11.3.1 Control-Flow Graphs

A *control-flow graph*, or simply *flow graph*, represents the flow of control between the statements in a program. Flow graphs have two kinds of nodes:

- *basic nodes* with one incoming and one outgoing edge; and
- *decision nodes* with one incoming and two or more outgoing edges.

Basic nodes represent assignments and procedure calls. Decision nodes result from Boolean expressions, such as those in conditional and while statements. The two outgoing edges from a decision node are called *branches* and labeled T for *true* and F for *false*. If the Boolean expression in the decision node evaluates to *true* control flows through the T branch; otherwise, control flows through the F branch.

For convenience, a sequences of one or more basic nodes may be merged to form a node called a *basic block*. As a further convenience, assume that a flow graph has a single *entry* node with no incoming edges and a single *exit* node with no outgoing edges.

Example 11.10 : The flow graph in Fig. 11.7 represents the flow of control between the statements in Fig. 11.1. The flow graph has three basic blocks, B_1-B_3, and three decision nodes, D_1-D_3. Basic block B_1 has one assignment, which initializes `year`. (We take the liberty of initializing `year` to `1980`, instead of `ORIGINYEAR`.) B_2 has two assignments:

```
days -= 365;
year += 1;
```

Decision node D_1 corresponds to the decision in the while statement

```
while (days > 365) { ... }
```

D_2 corresponds to the decision in the conditional statement

```
if (IsLeapYear(year)) { ... } else {...}
```

□

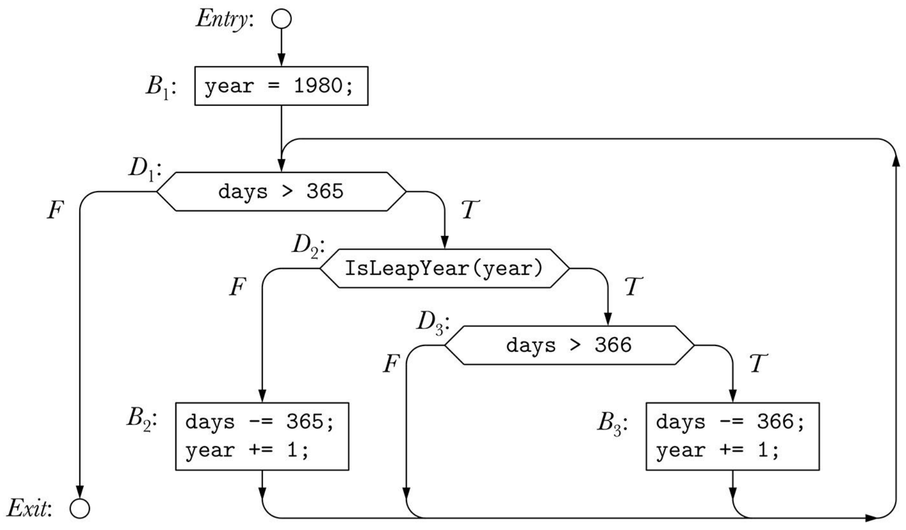


Figure 11.7: Control-flow graph for the code fragment in Fig. 11.1.

The flow graph for a program can be constructed by applying rules like the ones in Fig. 11.8. The rules are for a simple language that supports assignments, conditionals, while loops, and sequences of statements. Variables E and S represent expressions and statements, respectively. The rules can be applied recursively to construct the flow graph for a statement. It is left to the reader to extend the rules to handle conditionals with both then and else parts.

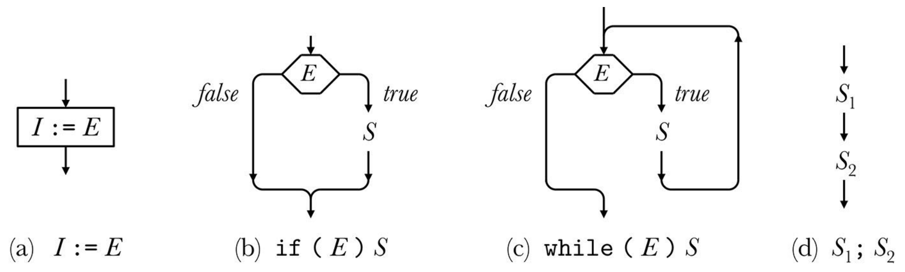


Figure 11.8: Rules for constructing a conrrol-flow graph.

Paths Through a Flow Graph

A test corresponds to a path through a flow graph. Testing is done by running a program, so a test corresponds to an execution of the program. Each execution corresponds to a path through the flow graph for a program. Hence the statement that a test corresponds to a path through a flowgraph.

More precisely, a *path* through a flow graph is a sequence of contiguous edges from the entry node to the exit node. Two edges are *contiguous* if the end node of the first edge is the start node of the second edge. A path can be written as a sequence of nodes $n_1, n_2, ...$, where there is an edge from node n_i to node n_{i+1}, for all i.

A *simple path* is a path with no repeated edges. A simple path through a loop corresponds to a single execution of the loop. If there was a second execution, one or more of the edges in the loop would repeat.

Example 11.11 : This example relates tests of the code in Fig. 11.1 with paths through the flow graph in Fig. 11.7. A test of the code consists of an execution, where the test input is an initial value for variable `days`.

Consider the test 365; that is, the initial value of `days` is 365. The Boolean expression `days > 365` evaluates to false, so control skips the body of the while-loop. This execution corresponds to the simple path

$$\mathit{Entry}, B_1, D_1, \mathit{Exit}$$

With test 367, control goes once through the body of the while loop, before exiting. Variable `year` is initialized to `1980`, a leap year, so this test traces the simple path

$$\mathit{Entry}, B_1, D_1, D_2, D_3, B_3, D_1, \mathit{Exit}$$

Although node D_1 appears twice, this path is simple because no edge is repeated. □

11.3.2 Control-Flow Coverage Criteria

The correspondence between tests and paths is one-to-one. Each test of a program corresponds to a path through the program's flow graph, and vice versa. Code-coverage criteria can be therefore be expressed either in terms of program constructs like statements, decisions, and executions or in terms of nodes, branches, and paths, respectively.

Coverage tracking is a job best done by automated tools that build flow graphs and keep track of coverage information as tests are run. Given the close correspondence between flow graphs and the source code, the tools display coverage information by annotating the source code. For example, statement coverage is typically displayed by showing the number of times each line of code is executed by a test set. The tools also produce reports that summarize the coverage achieved by a test set.

All-Paths Coverage: Strong But Unrealistic

With *all-paths coverage*, each path is traced at least once.

The set of all paths through a flow graph corresponds to the set of all possible executions of the program. All paths coverage therefore corresponds to exhaustive testing. Exhaustive testing is the strongest possible test-adequacy criterion: if a program passes exhaustive testing, then we know that the program will work for all inputs.

Exhaustive testing is also infeasible. A flow graph with loops has an infinite number of paths: given any path that goes through a loop, we can construct another longer path by going one more time through the loop.

Since all-paths coverage is an unattainable ideal, many more or less restrictive coverage criteria have been proposed. The rest of the this section considers some widely used code coverage criteria.

Node or Statement Coverage

Node coverage, also known as *statement coverage*, requires a set of paths (tests) that touch each node at least once. Node coverage is the weakest of the commonly used coverage criteria.

Example 11.12 : For the flow graph in Fig. 11.7, 100% node coverage can be achieved without triggering the known leap-year bug: on the last day of a leap year the code loops forever.

From Example 11.11, the paths traced by the test inputs 365 and 367 are as follows (for readability, the entry and exit nodes are omitted):

TEST INPUT	PATH
365	B_1, D_1
367	$B_1, D_1, D_2, D_3, B_3, D_1$

The test set $\{365, 367\}$ covers the nodes

$$B_1, B_3, D_1, D_2, D_3$$

Note that node B_2 is not yet covered. For B_2 to be covered, control must take the F branch of decision node D_2 in Fig. 11.7. The first time through D_2, control will take the T branch, since the initial value of `year` is 1980, a leap year.

The path for test 732 takes the F branch to B_2 the second time the path reaches D_2, The path is

$$B_1, D_1, D_2, D_3, B_3, D_1, D_2, B_2, D_1$$

The singleton test set $\{732\}$ happens to covers all nodes in Fig. 11.7. In general, multiple tests are needed to achieve the desired level of node coverage. □

While 100% node coverage is a desirable goal, it may not be achievable. If the software under test has dead code, by definition, the dead code cannot be reached during any execution. No test can therefore reach it. While dead code can be removed by refactoring, legacy systems are touched only as needed.

In practice, companies set stricter node coverage thresholds for new than for legacy code.

Branch Coverage is Stronger than Node Coverage

Branch coverage, also known as *decision coverage*, requires a set of paths (tests) that touch both the true and false branches out of each decision node. Branch coverage is a stronger criterion than node coverage. (As we shall see, branch coverage does uncover the leap-year bug in Fig. 11.7.)

Specifically, branch coverage *subsumes* node coverage, which means that any test set that achieves 100% branch coverage also achieves 100% node coverage. The converse is false, as the next example demonstrates.

Example 11.13: From Example 11.12, the tests 365, 367, and 732 correspond to the following paths

Test Input	Path
365	B_1, D_1
367	$B_1, D_1, D_2, D_3, B_3, D_1$
732	$B_1, D_1, D_2, D_3, B_3, D_1, D_2, B_2, D_1$

The branch coverage of these tests is as follows:

Test Input	D_1	D_2	D_3
365	F		
367	T, F	T	T
732	T, F	T, F	T

These tests achieve 100% node coverage, but they do not achieve 100% branch coverage because they do not cover the F branch from D_3 to D_1. (In fact, test 732 alone covers all branches covered by the other tests.)

This F branch out of D_3 is covered only when the code is in an infinite loop. Here's why. For the branch (D_3, D_1) to be taken, the following must hold:

`days > 365`	*true*	at node D_1
`IsLeapYear(year)`	*true*	at node D_2
`days > 366`	*false*	at node D_3

Together, these observations imply that, when the branch (D_3, D_1) is taken, `days` must have value 366 and `year` must represent a leap year. With these values for `days` and `year`, the program loops forever.

Thus, a test suite designed to achieve 100% branch coverage would uncover the leap-year bug.

The smallest test input that triggers the infinite loop is the value 366 for `days`—the corresponding date is December 31, 1980. □

Other Control-Flow Coverage Criteria

In practice, node and branch coverage are the most popular control-flow-based adequacy criteria. Branch coverage subsumes node coverage: any test set that achieves 100% branch coverage also achieves 100% node coverage. A number of other control-flow-based criteria have ben proposed; e.g.,[14]

- *Loop count coverage*: exercise each loop up to k times, where k is a parameter.
- *Length-n path coverage*: cover all subpaths of length n.

11.3.3 MC/DC: Modified Condition/Decision Coverage

Branch coverage is adequate for the vast majority of decisions. Branch coverage is not enough, however, for decisions involving complex Boolean expressions containing operators such as `&` (logical and) and `|` (logical or) For example, suppose the value of the following decision is *false*:

```
(pressure > 32) & (temperature <= LIMIT)
```

Is it *false* because `pressure` is less than or equal to `32` or is it *false* because `temperature` is greater than `LIMIT`?

In discussing Boolean expressions, it is helpful to distinguish between atomic expressions which do not contain Boolean operators and general expressions which could. A *condition* is an atomic Boolean expression; e.g., `days > 365` or `IsLeapYear(year)`. A *decision* is a Boolean expression formed by applying Boolean operators to one or more conditions.

Branch coverage is adequate for decisions involving a single condition, as in

```
while (days > 365) { ... }
```

One study found that almost 85% of the decisions in a collection of software tools were based on just one "condition."[15]

For system safety and security, however, it is not enough for a coverage criterion to be adequate 85% of the time. All a hacker needs is one security vulnerability. One flaw may be enough to jeopardize a mission-critical system. Avionics software can have complex Boolean expressions with multiple conditions. The same study found that 17% of the decisions in a flight-instrumentation system had two or more conditions; over 5% had three or more conditions.[15]

For avionics software, the US Federal Aviation Administration recognizes a criterion called MC/DC, which is stronger than branch coverage.[16] MC/DC has also been recommended for detecting security backdoors.[17]

Condition and Decision Coverage are Independent

Let T and F denote the Boolean truth values, *true* and *false*, respectively. Let the lowercase letters $a, b, c, ...$ denote conditions; e.g.,

Condition a: `pressure > 32`
Condition b: `temperature <= LIMIT`

The expression $a \,\&\, b$ is a decision, representing

```
(pressure > 32) & (temperature <= LIMIT)
```

Condition coverage requires that each condition in a decision take on both truth values, T and F. *Decision coverage* requires each decision to take on both truth values. As we shall see, condition coverage does not ensure decision coverage, and vice versa.

While discussing condition and decision coverage, it is convenient to summarize tests by writing tables like the following for decision $a \,|\, b$:

TEST	a	b	$a \,\vert\, b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Each row of this table represents a test. The columns represent the values of the conditions a and b and the decision $a \,|\, b$. In test 2, a is T and b is F, so $a \,|\, b$ is T.

Example 11.14: Tests 2 and 3 above provide condition coverage, but not decision coverage. Condition coverage follows from the observation that in tests 2 and 3, a is T and F, respectively; b is F and T, respectively. But, the two tests do not provide decision coverage, since $a \,|\, b$ is T in both tests.

Tests 2 and 4 provide decision coverage, but not condition coverage. While the value of $a \,|\, b$ flips from T to F in these tests, b is not covered, since b is F in tests 2 and 4. □

MC/DC Pairs of Tests

MC/DC is a stronger criterion than condition and decision coverage put together. The following two tests provide both condition and decision coverage:

TEST	a	b	$a \,\vert\, b$
1	T	T	T
4	F	F	F

The limitation of these tests is that both a and b vary together: they both flip from T to F together.

Modified Condition/Decision Coverage (MC/DC) requires each condition to independently affect the outcome of the decision. Independently means that tor each condition x, there is a pair of tests such that

- the outcome of the decision flips from T to F, or vice versa,

TEST	a	b	$a \mid b$	a	b
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	F	F	F	2	3

TEST	a	b	$a \,\&\, b$	a	b
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

Figure 11.9: Pairs tables for decisions $a \mid b$ and $a \,\&\, b$.

- the value of condition x also flips, and
- the values of the remaining conditions stay the same.

Such a pair of tests is called an *MC/DC pair* of tests for x.

Example 11.15 : For decision $a \mid b$, the following tests form a pair for a:

TEST	a	b	$a \mid b$
2	T	F	T
4	F	F	F

The following tests form a pair for b:

TEST	a	b	$a \mid b$
3	F	T	T
4	F	F	F

□

A *pairs table* for a decision succinctly identifies the MC/DC pairs for the conditions in the decision. The pairs tables for $a \mid b$ and $a \,\&\, b$ appear in Fig. 11.9. A pairs table is an extension of a truth table for the decision. A truth table has a column for each condition and a column for the decision. The rows of the truth table show all combinations of values for the conditions and the corresponding value for the decision. A pairs table adds a column for each condition x. If (i, j) is a pair for x, then the added column for x has j in row i and i in row j.

Example 11.16 : The pairs table for the decision $(a \,\&\, b) \mid c$ appears in Fig. 11.10.

There is only one MC/DC pair for a: it is $(2, 6)$. Since $(2, 6)$ is a pair, there is a 6 in row 2 and a 2 in row 6. In the other pairs for a the outcome of the decision does not flip. The decision remains T for the pairs $(1, 5)$ and $(3, 7)$; it remains F for the pair $(4, 8)$.

The only pair for b is $(2, 4)$. There are three pairs for c: they are $(3, 4)$, $(5, 6)$ and $(7, 8)$. □

Test	a	b	c	$(a \,\&\, b) \mid c$	a	b	c
1	T	T	T	T			
2	T	T	F	T	6	4	
3	T	F	T	T			4
4	T	F	F	F		2	3
5	F	T	T	T			6
6	F	T	F	F	2		5
7	F	F	T	T			8
8	F	F	F	F			7

Figure 11.10: Pairs tables for the decision $(a \,\&\, b) \mid c$.

MC/DC Tests

The MC/DC tests for a decision can be deduced from its pairs table. The fewer the tests the better.

A direct approach is

```
for each condition $x$, pick a pair for $x$, any pair.
```

The resulting number of tests is at most twice the number of conditions. The actual number may be smaller, since some tests may appear in more than one pair. With the direct approach, the number of tests grows linearly with conditions, since each condition adds at most two tests. The number of possible tests grows exponentially, however, since the number of possible tests doubles with each condition.

Fewer tests result from the following approach

```
for each condition with only one pair, pick that pair;
for the remaining conditions $x$,
    pick the pair that adds the fewest tests;
```

Example 11.17 : For the decision $(a \,\&\, b) \mid c$ in Fig. 11.10, $(2, 6)$ is the only pair for a and $(2, 4)$ is the only pair for b, so they must be picked. Of the three pairs for c, either $(3, 4)$ or $(5, 6)$ would be a better choice than $(7, 8)$, since they overlap with the tests needed for a and b. Choosing the pair $(3, 4)$ for c, we get four tests:

Test	a	b	c	$(a \,\&\, b) \mid c$
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
6	F	T	F	F

□

11.3.4 Data-Flow Coverage

11.4 Testing for Input-Domain Coverage

From the overview of testing in Section 11.1, testing proceeds roughly as follows:

- Select a test input for the software under test.
- Evaluate the software's response to the input.
- Decide whether to continue testing.

Test inputs are drawn from a set called the input domain. The input domain can be infinite. If not infinite, if is typically so large that exhaustive testing of all possible inputs is infeasible. Test selection is therefore based on some criterion for sampling the input domain.

The selection criteria in this section are closely, but not exclusively, associated with black-box testing, which treats the software under test as a black box that hides the source code. Test selection is based on the software's specification.

11.4.1 Equivalence-Class Coverage

Equivalence partitioning is a heuristic technique for partitioning the input domain into subdomains with inputs that are equivalent for testing purposes. The subdomains are called equivalence classes. If two test inputs are in the same equivalence class, we expect them to provide the same information about a program's behavior: they either both catch a fault or they both miss the fault.

A test suite provides *equivalence-class coverage* if the set includes a test from each equivalence class.

There are no hard and fast rules for defining equivalence classes—just guidelines. The following example sets the stage for considering some guidelines.

Example 11.18 : Consider a program that determines whether a year between 1800 and 2100 represents a leap year. Strictly speaking, the input domain of this program is the range 1800-2100, but let us take the input domain to be the integers, since the program might be called with any integer. Integers between 1800 and 2100 will be referred to as valid inputs; all other integers will be referred to as invalid inputs.

As a first approximation, we might partition the input domain into two equivalence classes, corresponding to valid and invalid integer inputs. For testing, however, it is convenient to start with three equivalence classes: integers up to 1799; 1800 through 2100; and 2101 and higher.

These equivalence classes can be refined, however, since some years are leap years and some years are not. The specification of leap years is as follows:

> "Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400."[18]

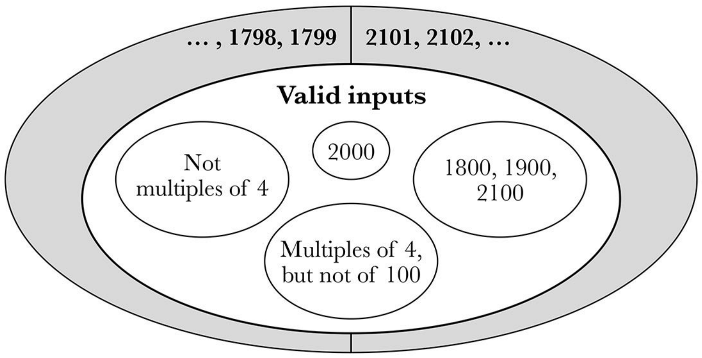


Figure 11.11: Equivalence classes for a leap-year program. The two shaded regions are for invalid test inputs.

This specification distinguishes between years divisible by 4, 100, 400, and all other years. These distinctions motivate the following equivalence classes (see Fig. 11.11):

- Integers less than or equal to 1799,
- Integers between 1800 and 2100 that are not divisible by 4.
- integers between 1800 and 2100 that are divisible by 4, but not by 100.
- The integers 1800, 1900, and 2100, which are divisible by 100, but not by 400.
- The integer 2000, which is divisible by 400.
- Integers that are greater than or equal to 2101.

The leap-year program can now be tested by selecting representatives from each equivalence class. □

Equivalence classes are designed. Two people working from the same specification could potentially come up with different designs. The following are some guidelines for getting started with equivalence partitioning:[19]

- If the inputs to a program are from a range of integers m-n, then start with three equivalence classes. The first class contains invalid inputs that are less then or equal to $m - 1$; the second contains the valid inputs $m, m-1, ..., n$; the third contains the invalid inputs greater than or equal to $m + 1$. This guideline can be generalized from integers to other data types.
- If an equivalence class has two or more inputs that produce different outputs, then split the equivalence class into subclasses, where all of the inputs in a subclass produce the same output.

- If the specification singles out one or more test inputs for similar treatment, then put all inputs that get the same "same" treatment into an equivalence class.
- For each class of valid inputs, define corresponding classes of invalid inputs.

Once equivalence classes are designed, tests can selected to cover them; that is, select a test input from each equivalence class.

11.4.2 Boundary-Value Coverage

Suppose that the input domain is ordered; that is, for two inputs i and j, it makes sense to talk of i being less than or before j, written $i < j$. Then, a value is a *boundary value* if it is either the first or the last—alternatively, the least or the greatest—with respect to the ordering.

Based on experience, errors are often found at boundary values. For example, the date-calculation code in Fig, 11.1 fails on December 31st of a leap year, a boundary value.

Boundary-value testing leverages the equivalence classes defined during equivalence partitioning.

Example 11.19 : In Fig. 11.11, consider the equivalence classes for valid and invalid inputs. The valid inputs are the years between 1800 and 2100. The boundaries of this equivalence class are 1800 and 2100. There are two classes for invalid inputs: the smaller class of years less than 1800 and the bigger class of years greater than 2100. The upper boundary for the smaller class is 1799, but there is no lower boundary, since there are an infinite number of integers smaller than 1800. Similarly, the lower boundary for the bigger class is 2101 and there is no upper boundary.

For the equivalence class of years between 1800 and 2100 that are not multiples of 4, lower boundary is 1801 and the upper boundary is 2099. □

A test suite provides *boundary value coverage* if it includes the upper and lower boundaries of each of the equivalence classes. For an equivalence class with one element, the lower and upper boundaries are the same. In Fig. 11.11, the year 2000 is in a class by itself.

11.4.3 Combinatorial Testing

Combinatorial testing is an efficient technique for detecting failures that result from a combination of factors—a *factor* is a quantity that can be controlled during testing. At the unit level, failures can result from complex decisions involving multiple conditions. Two factors, pressure and temperature, are involved in the following pseudo-code:

```
if (pressure > 32 & temperature > LIMIT) {
    // faulty code
} else {
    // good code
}
```

The faulty code is reached at specific combinations of pressure and temperature.

At the system level, failures can result from combinations of components, as in the following example.

Example 11.20: Consider the problem of testing a software product that must support multiple browsers (Chrome, Safari, Explorer, Firefox), run on multiple platforms (Linux, Windows, Mac OS), and interface with multiple databases (MongoDB, Oracle, MySQL).

With 4 browsers, 3 platforms, and 3 databases, the number of combinations is $4 \times 3 \times 3 = 36$. That's 36 test sets, not tests, since the full test set must be run for each combination. The problem gets worse if the product must support not only the current version of a browser, but previous versions as well. $\Box$

A software failure that results from a specific combination of factors is called an *interaction failure.* A 2-way interaction involves two factors; a 3-way interaction involves three factors; and so on. An empirical study found that roughly 60% of web-server failures involved two or more factors. Incidentally, web-server and browser failures involved more complex interactions than either medical devices or a NASA application.[20].

Combinatorial testing is based on the empirical observation that

> "most failures are triggered by one or two factors, and progressively fewer by three, four, or more factors, and the maximum interaction degree is small."[21]

Pairwise Interactions

Pairwise testing addresses 2-way interactions. The idea is to test all combinations of values for each pair of factors. For the system in Example 11.20 the pairs of factors are

$$(\text{browser}, \text{platform}),\ (\text{browser}, \text{database}),\ (\text{platform}, \text{database})$$

Some conventions will be helpful in organizing sets of tests. Let the letters $A, B, C, \ldots$ represent factors; e.g, A represents browser, B represents platform, and C represents database. Let the integers $0, 1, 2, \ldots$ represent factor values; e.g., for factor B (platform), 0 represents Linux, 1 represents Windows, and 2 represents Mac OS. Let two-letter combinations $AB, AC, BC, \ldots$ represent the pairs of factors $(A, B),\ (A, C),\ (B, C), \ldots$, respectively.

Consider tests involving two factors A and B, each of which can take on the two values 0 and 1. With two factors and two values, there are four possible combinations of values for the two factors. A test consists of a specific

Test	A	B	C
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

(a) All combinations

Test	A	B	C
2	0	0	1
3	0	1	0
5	1	0	0
8	1	1	1

(b) A 2-way covering array

Figure 11.12: Tables for three factors, each of which can have two possible values. See Example 11.21.

combination of values. The following table represents an exhaustive set of tests involving the two factors:

Test	A	B
1	0	0
2	0	1
3	1	0
4	1	1

In such tables, columns represent factors, rows represent tests, and table entries represent values for the factors.

A set of tests is a *2-way covering array* if the tests include all possible combinations for each pair of factors. The next two examples illustrate covering arrays.

Example 11.21 : Consider a simplified version of Example 11.20, where each factor can have one of two values. Factor A (browser) can take on the two values 0 and 1 (Chrome and Safari); factor B (platform) the values 0 and 1 (Linux and Windows); and factor C (database) the values 0 and 1 (MongoDB and Oracle).

The table in Fig. 11.12(a) shows all possible combinations for three factors, where each factor can have one of two values. Tests 1-8 therefore constitute an exhaustive test set for three factors.

The four tests in Fig. 11.12(b) constitute a covering array for pairwise testing of three factors. The three pairs of factors are AB, AC, and BC.

Let us verify that the four tests cover all combinations of values for each of these pairs. Consider the pair AC. All possible combinations of values for AC are 00, 01, 10, and 11. These combinations are covered by tests 3, 2, 5, and 8, respectively. Test 3 has the form $0x0$, where both A and C have value 0. The

A	B
0	0
0	1
0	2
1	0
1	1
1	2

A	C
0	0
0	1
1	0
1	1

B	C
0	0
0	1
1	0
1	1
2	0
2	1

Test	A	B	C
1	0	0	1
2	0	1	0
3	0	2	1
4	1	0	0
5	1	1	1
6	1	2	0

Figure 11.13: All combinations for pairs AB, AC, and BC, and a covering array. See Example 11.22.

value of B is ignored for now, since we are focusing on the pair AC. Tests 2, 5, and 8 have the form $0x1$, $1x0$, and $1x1$, respectively.

The combinations of values for the pair AB, are covered by the tests 2, 3, 5, and 8. For the pair BC, consider the tests 5, 2, 3, and 8. □

Example 11.22: Now, suppose that factor B can take on three values 0, 1, and 2 (for Linux, Windows, and Mac OS), and that factors A and C can have two values, as in Example 11.21. The total number of combinations for the three factors are $2 \times 3 \times 2 = 12$.

For pairwise testing, 6 tests are enough. All combinations for the pairs AB, AC, and BC appear in Fig. 11.13, along with a covering array with 6 tests. For pair AB, tests 1-6, in that order, correspond to the rows in the combinations-table for AB. For pair AC, see tests 2-5, in that order. For pair BC, tests 4, 1, 2, 5, 6, and 3, correspond to the rows in the combinations-table for BC. □

Multiway Covering Arrays

The discussion of 2-way interactions generalizes directly to the interaction of three or more factors. More factors need to be considered since 2-way testing finds between 50% and 90% of faults, depending on the application. For critical applications, 90% is not good enough. Three-way testing raises the lower bound from 50% to over 85%.

The benefits of combinatorial testing become more dramatic as the size of the testing problem increases. The number of possible combinations grows exponentially with the number of factors. By contrast, for fixed t, the size of a t-way covering array grows logarithmically with the number of factors.[22] For example, there are $2^{10} = 1024$ combinations of 10 factors, with each factor having two values. There is a 3-way covering array, however, that has only 13 tests; see Fig. 11.14.[23]

Algorithms and tools are available for finding covering arrays. The general problem of finding covering arrays is believed to be a hard problem. A naive heuristic approach is to build up a covering array by adding columns for the

TEST	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Figure 11.14: 3-way covering array for 10 factors.

factors, one at a time. Entries in the new column can be filled in by extending an existing test, or by adding a row for a new test.

```
start with an empty array;
for each factor $F$:
    add a column for $F$;
    mark $F$;
    for each 3-way interaction $XYF$, where $X$ and $Y$ are marked:
        for each combination in the combinations table for $XYF$:
            if possible:
                fill in a blank in an existing row to cover the combination.
            else:
                add a row with entries in the columns for $X$, $Y$, and $F$;
                comment: leave all other entries in the row blank
```

11.5 Key Concepts and Terms

Exercises for Chapter 11

Exercise 11.1 : The diagram in Fig. 11.15 represents the development process for the SAGE air-defense system. The dashed arrows go from specification phases to the testing phases. What is the best fit between the testing phases and unit, functional, integration, system, and acceptance testing? Explain your answer.

Here are brief descriptions of the testing phases:

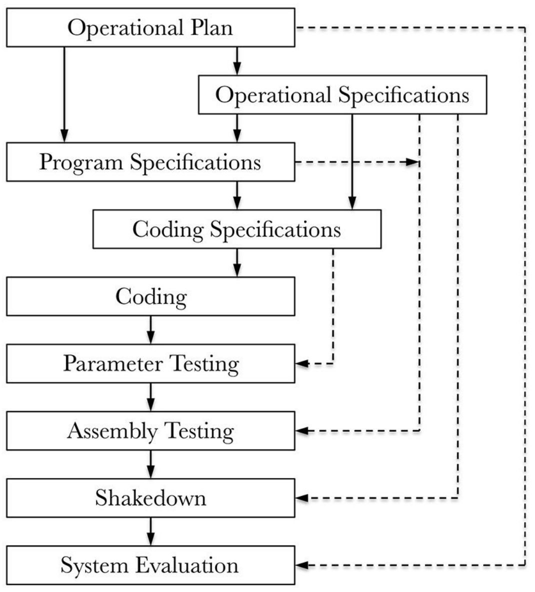


Figure 11.15: The development process for the SAGE air-defense system.

- *Parameter Testing.* Test each component by itself guided by the coding specification.
- *Assembly Testing.* As parameter testing completes, the system is gradually assembled and tested using first simulated inputs and then live data.
- *Shakedown.* The completed system is tested in its operational environment.
- *System Evaluation.* After assembly, the system is ready for operation and evaluation.

Exercise 11.2 : For each of the following, define a minimal set of MC/DC (Modified Condition/Decision Coverage) tests. Why is your answer a minimal set?

a) `(!a)|(!b)|c`

Exercise 11.3 : Suppose that there are 4 possible effects for text formatting and that you want to test all two-way interactions.

a) Create no more than 6 tests to test all 2-way interactions. Explain why your solution works.

b) Find a solution that requires no more than 5 tests.

Notes for Chapter 11

[1]Bill Gates interview with Information Week [8].

[2]In his 2002 interview [8], Bill Gates noted, "When we do a new release of [Microsoft] Windows, which is, say, a billion-dollar effort, over half that is going into the quality." The classic 1979 book on software testing by Glenford J. Myers [15, p. vii] begins with, "It has been known for some time that, in a typical programming project, approximately 50% of the elapsed time and over 50% of the cost are expended in testing the program or system being developed."

[3]The defective code in Fig. 11.1 is from the clock driver for the Freescale MC 13783 processor used by the Microsoft Zune 30 and Toshiba Gigabeat S media players [21]. The root cause of the failure on December 31, 2008 was isolated by "itsnotabigtruck" [11].

[4]For more on the issues that arise during testing, see the "practice tutorial" by James A. Whittaker [19].

[5]This version of Edsger W. Dijkstra's famous quote about testing is from the 1969 NATO Software Engineering Techniques conference [5, p. 16].

[6]Capers Jones provides empirical data on the "defect removal efficiency" of various combinations of reviews, static, analysis, and testing. Hackbarth, Mockus, Palframan, and Sethi [9] describe a software-quality improvement program based on reviews, static analysis, and testing that led to improvements in post-delivery operational experience.

[7]Weyuker [18].

[8]Primality testing is the problem of deciding whether a number n is a prime number. In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena showed that there is a deterministic polynomial algorithm for primality testing [1].

[9]SWEBOK 3.0 merges system and functional testing into a single level [4, page 4-5.]. The levels of testing in Section 11.2 assign the validation and verification roles to system and functional testing, respectively. The classic text on testing by Glenford J. Myers separates system and functional testing [15].

[10]Capers Jones has published summary data from 600 client companies [13]: "Many test stages such as unit test, function test, regression test, etc. are only about 35% efficient in finding code bugs, or find one bug out of three. This explains why 6 to 10 separate kinds of testing are needed."

[11]The xUnit family began with Kent Beck's automated testing frameworks for Smalltalk. In 1997, Smalltalk usage was on the decline and Java usage was on the rise, so Kent Beck and Erich Gamma created JUnit for Java. They had three goals for JUnit: make it natural enough that developers would actually use it; enable tests that retain their value over time; and leverage existing tests in creating new ones [2].

[12]David L. Parnas [16] introduced the "uses" relation as an aid for designing systems "so that subsets and extensions are more easily obtained." Incremental bottom-up integration corresponds to building a system by extension.

[13]Myers [15, p. 99-100] notes that a comparison between top-down and bottom-up integration testing "seems to give the bottom-up strategy the edge."

[14]Zhu, Hall, and May survey test coverage and adequacy criteria [22].

[15]Chilensky and Miller 1994

[16]MC/DC is included in RTCA document DO-178C [20]. FAA Advisory Circular 20-115C, dated July 19, 2013, recognizes DO-178C as an "acceptable means, but not the only means, for showing compliance with the applicable airworthiness regulations for the software aspects of airborne systems."

[17]While discussing industry trends in 2017, Ebert and Shankar [**??**] recommend MC/DC for detecting security backdoors.

[18]The leap-year specification is from the US Naval Observatory]17].

[19]Myers [15, p. 46-47] provides heuristic guidelines for equivalence partitioning.

[20]hagar-2015-testing

[21]Kuhn 2014 keynote

[22]Cohen, Dalal, Fredman, Patton [6].

[23]Kuhn keynote. It's also in the Hagar paper.

References for Chapter 11

1. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in *P*. *Annals of Mathematics*, Second Series 160, 2 (September 2004) 781-793.
2. Kent Beck and Erich Gamma. JUnit: a cook's tour. (circa 1998). `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`.
3. Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software* (January 1984) 75-88. A 1979 version is available as a technical report USC-79-501 `http://csse.usc.edu/TECHRPTS/1979/usccse79-501/usccse79-501.pdf`
4. Pierre Borque and Richard E. Fairley (eds) *Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0.* IEEE Computer Society (2014) `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1265988` .
5. John N. Buxton and Brian Randell (eds), *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, October 1969. (published April 1970). `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`
6. David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: an approach to testing based on combinatorial designs. *IEEE Transactions on Software Engineering* 23, 7 (July 1997) 437-444.
7. Christof Ebert and Kris Shankar. Industry trends 2017. *IEEE Software* (March-April 2017) 112-116.
8. Gates, Bill. Q&A: Trustworthy Computing. *Information Week* (May 16, 2002). `http://www.informationweek.com/qanda-bill-gates-on-trustworthy-computing/d/d-id/1015083?`
9. Randy Hackbarth, Audris Mockus, John Palframan, and Ravi Sethi. Improving software quality as customers perceive it. *IEEE Software* (July-August 2016) 40-45. `https://www.computer.org/cms/Computer.org/ComputingNow/issues/2016/08/mso2016040040.pdf`
10. Jon D. Hagar, Thomas L. Wissink, D. Richard Kuhn, and Raghu N. Kacker. Introducing combinatorial testing in a large organization. *IEEE Computer* (April 2015) 64-72.
11. itsnotabigtruck. Cause of Zune 30 leapyear problem ISOLATED! `http://www.zuneboards.com/forums/showthread.php?t=38143`
12. Capers Jones. Software Quality in 2013: A Survey of the State of the Art. *35th Annual Pacific NW Software Quality Conference* (October 2013). The following website has a link to his video keynote: `https://www.pnsqc.org/software-quality-in-2013-survey-of-the-state-of-the-art/`.
13. Capers Jones. Achieving software excellence. *Crosstalk* (July-August 2014) 19-25.
14. D. Richard Kuhn. Combinatorial testing: rationale and impact. Keynote at *IEEE Seventh International Conference on Software Testing, Verification, and Validation* (April 2, 2014) Presentation available at `http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-icst-14.pdf`
15. Glenford J. Myers. *The Art of Software Testing.* John Wiley (1979).
16. David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 2 (March 1979) 128-138.
17. US Naval Observatory. Introduction to Calendars. `http://aa.usno.navy.mil/faq/docs/calendars.php`.
18. Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal* 25, 4 (1982) 465-470.
19. Whittaker, James A. What is software tesing? And why is it so hard? *IEEE Software* (January-February 2000) 70-79.

20. Wikipedia. Modified condition/decision coverage. `https://en.wikipedia.org/wiki/Modified_condition/decision_coverage`.

21. Wikipedia. Zune 30. `https://en.wikipedia.org/wiki/Zune_30`.

22. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys* 29, 4 (December 1997) 366-427.

Appendices

Index

Appendix A

Architecture Reviews: Sample Questions

The questions in this appendix are representative of questions that may arise during an architecture discovery review. System architects can prepare answers to such questions ahead of time. The list appears long, but it should not take the architects too long to fill out.

Variants of this appendix have been classroom tested for project proposals. Students form teams of four and prepare a proposal for a semester-long project of their own choosing. The proposal is a narrative—not bullet points—with an introduction and sections that mirror the sections of this appendix. A typical proposal has half a dozen pages or less.

Architecture reviews that focus on specific areas of an architecture may benefit from additional questions that dig deeper into that area. For example, a review that focuses on security might include questions about the privacy of customer information, authentication, and on the information flows between components in the architecture. What does each component really need to know?

See also the lists of questions can in the references.

A.1 Customer Needs and Wants

A.1.1 Target Customer

- Who is the target customer for this project?
- What does the customer want and why?
- What is their desired overall experience?
- Based on the overall experience, what are their unmet needs?

A.1.2 Other Stakeholders

- Who are the additional stakeholders?
- What is their desired experience?

A.2 Problem Definition

A.2.1 Proposed Benefit

- What customer opportunities have you chosen to address?
- In implementation-free terms, how will the customer benefit?
- Will they be able to do something new?
- Or, will their experience be much better, faster, ...?
- What are some enhancements that stakeholders would want?
- What alternative customer opportunities did you consider?
- What is the rationale for the chosen opportunity?

A.2.2 Measures of Success

- Who have you tested the idea on?
- How will you know that the customer got the proposed benefit?
- What are your customer-centric measures of success?

A.3 Technology

A.3.1 System Architecture

- What is the prioritized list of requirements?
- Do you have a high-level block diagram of the architecture?
- What are the message flows for providing the basic functionality?

- Do you have a module guide for the main components and what they do?
- What is a minimal system that will have some value for the customer?
- What future enhancements could the architecture handle?
- How will your system fit into its environment? The major interfaces?

A.3.2 Non-Functional Requirements

- How will the architecture handle performance and scalability?
- How will it handle security?
- Reliability and availability?
- How will the components handle unexpected input/stimuli?
- Can the system crash if the user makes a mistake?
- How will the system be deployed?
- How will it be maintained?

A.3.3 Tools and Plaforms

- What will you use to build your system?
- Are there available platforms, services, or components you can leverage?

A.4 Team

A.4.1 Development Process

- What is your planned development process?
- Does the team have a system architect or team of architects?
- Is the team co-located or distributed?
- Does the team have the right resources in the right places?

A.4.2 Skills

- Has the team built something like this before?
- Is the technology known to the team? To someone on the team?

A.5 Constraints and Risks

A.5.1 Constraints

- Are there any business constraints? Time to market? Budget?
- Are there any social, ethical, policy, or legal constraints?

A.5.2 Risks

- Are there risks associated with external technology or services?
- Any other risks?

%addcontentslinetocsectionNotes

References for Appendix A

1. James Cusick. *Architecture and Production Readiness Reviews in Practice* (December 2014) `https://arxiv.org/abs/1305.2402` .
2. Mikael Lindvall and Roseanne Tesoriero Tvedt. *Software Architecture Inspection Checklist* Fraunhofer Center for Experimental Software Engineering, College Park, Maryland (2003) `http://www.fc-md.umd.edu/sites/default/files/EBS/INSPECTIONS/checklists/common/4_2_Arch.pdf` .
3. Joseph F. Maranzano, Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss. Architecture reviews: practice and experience. *IEEE Software* (March-April 2005) 34-43.