CSc 110, Autumn 2016

Lecture 33: Inheritance

Adapted from slides by Marty Stepp and Stuart Reges
The software crisis

- **software engineering**: The practice of developing, designing, documenting, testing large computer programs.

- Large-scale projects face many issues:
  - programmers working together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, reusing existing code

- **code reuse**: The practice of writing program code once and using it in many contexts.
Law firm employee analogy

• common rules: hours, vacation, benefits, regulations ...  
  • all employees attend a common orientation to learn general company rules  
  • each employee receives a 20-page manual of common rules

• each subdivision also has specific rules:  
  • employee receives a smaller (1-3 page) manual of these rules  
  • smaller manual adds some new rules and also changes some rules from the large manual
Separating behavior

• Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?

• Some advantages of the separate manuals:
  • maintenance: Only one update if a common rule changes.
  • locality: Quick discovery of all rules specific to lawyers.

• Some key ideas from this example:
  • General rules are useful (the 20-page manual).
  • Specific rules that may override general ones are also useful.
Is-a relationships, hierarchies

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
  - every marketer *is an* employee
  - every legal secretary *is a* secretary

- **inheritance hierarchy**: A set of classes connected by is-a relationships that can share common code.
Employee regulations

• Consider the following employee regulations:
  • Employees work 40 hours / week.
  • Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  • Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  • Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

• Each type of employee has some unique behavior:
  • Lawyers know how to sue.
  • Marketers know how to advertise.
  • Secretaries know how to take dictation.
  • Legal secretaries know how to prepare legal documents.
An Employee class

# A class to represent employees in general (20-page manual).
class Employee:
    def get_hours(self):
        return 40  # works 40 hours / week

    def get_salary(self):
        return 40000.0  # $40,000.00 / year

    def get_vacation_days(self):
        return 10  # 2 weeks' paid vacation

    def get_vacation_form(self):
        return "yellow"  # use the yellow form

• Exercise: Implement class Secretary, based on the previous employee regulations. (Secretaries can take dictation.)
Redundant Secretary class

# A redundant class to represent secretaries.

class Secretary:
    def get_hours(self):
        return 40  # works 40 hours / week

    def get_salary(self):
        return 40000.0  # $40,000.00 / year

    def get_vacation_days(self):
        return 10  # 2 weeks' paid vacation

    def get_vacation_form(self):
        return "yellow"  # use the yellow form

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)
Desire for code-sharing

• *take_dictation* is the only unique behavior in *Secretary*.

• We'd like to be able to say:

```python
# A class to represent secretaries.
class Secretary:
    copy all the contents from the Employee class

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)
```
Inheritance

• **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  • a way to group related classes
  • a way to share code between two or more classes

• One class can *extend* another, absorbing its data/behavior.
  • **superclass**: The parent class that is being extended.
  • **subclass**: The child class that extends the superclass and inherits its behavior.
    • Subclass gets a copy of every field and method from superclass
Inheritance syntax

class name (superclass) :

• Example:

class Secretary (Employee) :
    ...

• By extending Employee, each Secretary object now:
  • receives a get_hours, get_salary, get_vacation_days, and get_vacation_form method automatically
  • can be treated as an Employee by client code (seen later)
# A class to represent secretaries.
class Secretary (Employee):
    def take_dictation(self, text):
        print("Taking dictation of text: " + text)

• Now we only write the parts unique to each type.
  • Secretary inherits get_hours, get_salary, get_vacation_days, and getVacationForm methods from Employee.
  • Secretary adds the take_dictation method.
Implementing Lawyer

• Consider the following lawyer regulations:
  • Lawyers who get an extra week of paid vacation (a total of 3).
  • Lawyers use a pink form when applying for vacation leave.
  • Lawyers have some unique behavior: they know how to sue.

• Problem: We want lawyers to inherit most behavior from employee, but we want to replace parts with new behavior.
Overriding methods

• **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  • No special syntax required to override a superclass method. Just write a new version of it in the subclass.

    ```python
    class Lawyer(Employee):
        # overrides get_vacation_form method in Employee class
        def get_vacation_form():
            return "pink"
        ...
    
    • Exercise: Complete the Lawyer class.
      • (3 weeks vacation, pink vacation form, can sue)```
Lawyer class

# A class to represent lawyers.
class Lawyer(Employee):
    # overrides get_vacation_form from Employee class
def get_vacation_form(self):
    return "pink"

    # overrides get_vacation_days from Employee class
def get_vacation_days(self):
    return 15 # 3 weeks vacation

def sue(self):
    print("I'll see you in court!")

• Exercise: Complete the Marketer class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.
# A class to represent marketers.
class Marketer(Employee):
    def advertise():
        print("Act now while supplies last!")

    def get_salary():
        return 50000.0  # $50,000.00 / year
Levels of inheritance

• Multiple levels of inheritance in a hierarchy are allowed.
  • Example: A legal secretary is the same as a regular secretary but makes more money ($45,000) and can file legal briefs.

```python
class LegalSecretary(Secretary):
    ...
```

• Exercise: Complete the LegalSecretary class.
LegalSecretary class

# A class to represent legal secretaries.
class LegalSecretary(Secretary):
    def file_legal_briefs(self):
        print("I could file all day!")

    def get_salary(self):
        return 45000.0  # $45,000.00 / year
Calling overridden methods

• Subclasses can call overridden methods with `super`

```
   super(ClassName, self).method(parameters)
```

• Example:

```python
class LegalSecretary(Secretary):
    def get_salary(self):
        base_salary = super(LegalSecretary, self).get_salary()
        return base_salary + 5000.0
...
```
Inheritance and constructors

• Imagine that we want to give employees more vacation days the longer they've been with the company.
  • For each year worked, we'll award 2 additional vacation days.

  • When an Employee object is constructed, we'll pass in the number of years the person has been with the company.

  • This will require us to modify our Employee class and add some new state and behavior.

• Exercise: Make necessary modifications to the Employee class.
class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_hours(self):
        return 40

    def get_salary(self):
        return 50000.0

    def get_vacation_days(self):
        return 10 + 2 * self.__years

    def get_vacation_form(self):
        return "yellow"
Problem with constructors

• Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
TypeError: __init__() missing 1 required positional argument: 'initial_years'
```

• The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
Modified Marketer class

# A class to represent marketers.
class Marketer(Employee):
    def __init__(self, years):
        super(Marketer, self).__init__(years)

    def advertise(self):
        self.print("Act now while supplies last!")

    def get_salary(self):
        return super(Marketer, self).get_salary() + 10000.0

• Exercise: Modify the Secretary subclass.
  • Secretaries' years of employment are not tracked.
  • They do not earn extra vacation for years worked.
Modified Secretary class

```python
# A class to represent secretaries.
class Secretary(Employee):
    def __init__(self):
        super(Secretary, self).__init__(0)

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)

• Since Secretary doesn't require any parameters to its constructor, LegalSecretary compiles without a constructor.
  • Its default constructor calls the Secretary constructor.
```
Inheritance and fields

- Try to give lawyers $5000 for each year at the company:

```python
class Lawyer(Employee):
    ...
    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * years
    ...
```

- Does not work; the error is the following:

```
AttributeError: 'Lawyer' object has no attribute '_Employee__years'
```

- Private fields cannot be directly accessed from subclasses.
  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?
Improved Employee code

Add an accessor for any field needed by the subclass.

class Employee:
    self.__years

    def __init__(self, initial_years):
        self.__years = initial_years

        def get_years(self):
            return self.__years

    ...

class Lawyer(Employee):
    def __init__(self, years):
        super(Lawyer, self).__init__(years)

    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * get_years()
Revisiting Secretary

- The Secretary class currently has a poor solution.
  - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  - If we call get_years on a Secretary object, we'll always get 0.
  - This isn't a good solution; what if we wanted to give some other reward to all employees based on years of service?

- Redesign our Employee class to allow for a better solution.
Improved Employee code

• Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```python
class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_vacation_days(self):
        return 10 + self.get_seniority_bonus()

    def get_seniority_bonus(self):
        return 2 * self.__years

    # vacation days given for each year in the company
    def get_vacation_days(self):
        return 10 + self.get_seniority_bonus()
```

• How does this help us improve the Secretary?
Improved Secretary code

- Secretary can selectively override `get_seniority_bonus`; when `get_vacation_days` runs, it will use the new version.
- Choosing a method at runtime is called *dynamic binding*.

```python
class Secretary(Employee):
    def __init__(self, years):
        super(Secretary, self).__init__(years)

        # Secretaries don't get a bonus for their years of service.
    def get_seniority_bonus(self):
        return 0

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)
```