**Architecture – Contexts**

- A *context* supplies a means of loading and editing classes. All contexts implement the `EditorContext`. This interface supplies the generic means of committing edits, editing classes, and a utility to load classes.
- `PersistentBloatContext` and the `CachingBloatContext` both act as repositories for all of the edits made to java class files.
- `PersistentBloatContext` keeps every piece of data relevant to your work.
- The `CachingBloatContext` on the other hand manages your edits so that when a editor is no longer dirty it drops it.

**Architecture – Editors**

- Editors consist of the `ClassEditor`, `MethodEditor`, and `FieldEditor`.
- It is through these editors that you will make actual changes to byte code.

**BLOAT**

- `BLOAT` allows you to load a class, iterate through the methods and fields, change methods, add new methods and fields, etc.
- `BLOAT` is really a bytecode *optimizer*. In order to do code optimization `BLOAT` performs many program analyses. These can be used to do code *obfuscation* as well.
- `BLOAT` performs many traditional program optimizations such as constant/copy propagation, constant folding and algebraic simplification, dead code elimination, and peephole optimizations.
Creating a Class from Scratch

- To create a new class we call the `newClass` method in a `BlobContext`:
  ```java
  newClass(int modifiers, java.lang.String className,
            TypesuperType, Type[] interfaces); 
  ```

  ```java
  EDU.purdue.cs.bloat.file.ClassFileLoader cfl = ...; 
  EDU.purdue.cs.bloat.context.PersistentBlobContext bc = ...; 
  EDU.purdue.cs.bloat.editor.ClassEditor classEditor = 
      bc.newClass(
          EDU.purdue.cs.bloat.reflect.Modifiers.SUPER, 
          "MyNewClass", 
          EDU.purdue.cs.bloat.editor.Type.OBJECT, null); 
  classEditor.setPublic(true); 
  classEditor.setSuper(true); 
  bc.commit(); 
  ```

Editing a Method

- The `BlobContext` provides a method, `editMethod`, to modify methods. The `editMethod` method takes a `MethodInfo` object as a parameter. You can retrieve this from a `ClassEditor` with the method:

  ```java
  EDU.purdue.cs.bloat.reflect.MethodInfo[] methods = 
      classEditor.methods(); 
  EDU.purdue.cs.editor.MethodEditor methodEditor = 
      bc.editMethod(methods[0]);
  ```

Loading Classes

```java
EDU.purdue.cs.bloat.reflect.ClassInfo info = null;
try {
    info = cfl.loadClass("ClassToTest");
} catch (ClassNotFoundException ex) {...}
EDU.purdue.cs.bloat.file.ClassFile classFile =
    (EDU.purdue.cs.bloat.file.ClassFile)info;
```
Editing Fields

- The FieldEditor is used to edit attributes of a class.
- To edit fields of an existing class call the editField method of the BloatContext. This method takes a FieldInfo object.
- FieldInfo is obtained from the ClassEditor by calling the fields method.

```java
EDU.purdue.cs.bloat.reflect.FieldInfo[] fields =
    classEditor.fields();
EDU.purdue.cs.bloat.editor.FieldEditor fieldEditor =
    bc.editField(fieldInfo[0]);
```

Editing a Method...

- You can retrieve a specific method by obtaining its name with the name method in the MethodEditor object.
- To create a new MethodEditor use this constructor:

```java
MethodEditor(
    ClassEditor editor,
    int modifiers,
    Type returnType,
    java.lang.String methodName,
    Type[] paramTypes,
    Type[] exceptionTypes);
```

- Note that the first “instruction” in a method has to be a label:

```java
methodEditor.addLabel(methodEditor.newLabel());
```

Committing Changes

- After you have edited a class, field, or method, make sure to call the relevant commit methods so that the changes are reflected in the class file.
- The ClassEditor, FieldEditor, MethodEditor, and the BloatContext all have several commit methods.

Adding Instructions

- Once you have created a method you can begin adding instructions to the method. This done by calling the method addInstruction in the MethodEditor.

```java
EDU.purdue.cs.bloat.editor.LocalVariable L =
    methodEditor.newLocal(
        EDU.purdue.cs.bloat.editor.Type.INTEGER);
methodEditor.addInstruction(
    new EDU.purdue.cs.bloat.editor.Instruction(
        EDU.purdue.cs.bloat.editor.Opcode.opcx ldc,
        new java.lang.Integer(65)));
methodEditor.addInstruction(
    new EDU.purdue.cs.bloat.editor.Instruction(
        EDU.purdue.cs.bloat.editor.Opcode.opcx_istore, L));
```
**Class Hierarchy**

- **BLOAT** allows us to manipulate the class hierarchy.
- The **BloatContext** must first have knowledge of all of the classes relevant to the hierarchy.
- The **ClassHierarchy** (get it by calling the **getHierarchy** method in the **BloatContext**) object allows you to view all of the classes and interfaces in a hierarchy as well as edit the hierarchy by adding or removing classes and interfaces.

**Class Hierarchy...**

```java
// Add a class
EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy = bc.getHierarchy();

String className = "Test";
classHierarchy.addClassNamed(Test);
// extract the Type of classes in the hierarchy:
Collection classes =
    classHierarchy.classes();

// extract Type of classes that implement a specific interface:
Collection classes =
    classHierarchy.implementors(
        EDU.purdue.cs.bloat.editor.Type.SERIALIZABLE);
```

**Inlining**

- Create a **BloatContext** and load your classes as usual. Create an **Inline** object. An integer argument determines the maximum number of instructions a method can grow to.
- Using the method **maxCallDepth** you can specify with an integer the maximum number of nested method calls inlined.
- Bloat attempts to cut down the time to inline by providing a method, **setMaxInlineSize**, that sets the maximum size method that will be inlined.
- You may also specify whether to inline method that throw exceptions or not. This is simply set passing a boolean value to the method **setInlineExceptions**.

```java
EDU.purdue.cs.bloat.context.PersistentBloatContext bc = ...;
EDU.purdue.cs.bloat.inline.Inline inliner =
    new EDU.purdue.cs.bloat.inline.Inline(bc, 5);
inliner.setMaxCallDepth(2);
EDU.purdue.cs.bloat.file.ClassFile classFile = ...

EDU.purdue.cs.bloat.editor.ClassEditor classEditor =
    new EDU.purdue.cs.bloat.editor.ClassEditor(bc, info);
EDU.purdue.cs.bloat.reflect.MethodInfo[] methodInfo =
    classEditor.methods();
for(int i=0; i<methodInfo.length; i++){
    EDU.purdue.cs.bloat.editor.MethodEditor methodEditor =
        new EDU.purdue.cs.bloat.editor.MethodEditor(
            classEditor, methodInfo[i]);
inliner.inline(methodEditor);
}
```

```java
classEditor.commit();
```
Class Hierarchy...

- The `MemberRef` returned by `methodInvoked` represents the method you are interested in and class from where it was invoked.

```java
EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy = bc.getHierarchy();
EDU.purdue.cs.bloat.editor.NameAndType nameAndType =
  new EDU.purdue.cs.bloat.editor.NameAndType("test",
  EDU.purdue.cs.bloat.editor.Type.STRING);
MemberRef memRef =
  classHierarchy.methodInvoked(
    EDU.purdue.cs.bloat.editor.Type.OBJECT,
    nameAndType);
System.out.println(memRef);
```

Class Hierarchy...

- In order to determine if a method has been overridden you call the `methodIsOverridden` method:

```java
EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy =
  bc.getHierarchy();
EDU.purdue.cs.bloat.editor.NameAndType nameAndType =
  new EDU.purdue.cs.bloat.editor.NameAndType("test",
  EDU.purdue.cs.bloat.editor.Type.STRING);
boolean value =
  classHierarchy.methodIsOverridden(
    EDU.purdue.cs.bloat.editor.Type.OBJECT,
    nameAndType);
```

Class Hierarchy...

- The `ClassHierarchy` may also be used to determine what method will be invoked at any depth in the hierarchy.

```java
// simulates dynamic dispatching:
MemberRef methodInvoked(
  Type receiver,
  NameAndType method);
EDU.purdue.cs.bloat.editor.NameAndType(
    java.lang.String name,
    Type type);
```

The `Type` argument determines the receiver of the call. The `NameAndType` class describes an a method by its name and descriptor.
Control Flow Graph...

- The above example extracts the blocks from a CFG.
- It then accesses the Tree for each Block. A Tree represents the expression tree for each block.
- Using the tree you can view the instructions contained within a Block as well as edit them by adding and deleting instructions.

Control Flow Graph (CFG)

- BLOAT allows a developer to create a control flow graph for any given method:

```java
  // printing cfg in different orders
  EDU.purdue.cs.bloat.editor.MethodEditor methodEditor = ...
  EDU.purdue.cs.bloat.cfg.FlowGraph cfg =
      new EDU.purdue.cs.bloat.cfg.FlowGraph(methodEditor);
  cfg.preOrder();
  cfg.postOrder();
  cfg.print();
  cfg.printGraph();
```

Readings and References

- These slides are based on documentation written by Richard Smith (rsmith@cs.arizona.edu) for the SandMark project.
- The BLOAT Book can be found at http://www.cs.purdue.edu/s3/projects/bloat. At the time of writing this how-to this book referred to BLOAT-0.8a which lacks many of the features of BLOAT-1.0.
- To see how to use BLOAT in practice, have a look at the smbloat2 CVS directory.

Control Flow Graph...

- Once the graph is created you can extract the basic blocks using a variety of methods. Most of these methods produce a List or Collection of blocks that can be iterated across.

```java
  // extracting blocks and manipulating blocks:
  EDU.purdue.cs.bloat.cfg.FlowGraph cfg = ...

  //Blocks returned in trace order:
  java.util.List list = cfg.trace();
  for (int i=0;i<list.size();i++){
      EDU.purdue.cs.bloat.tree.Tree tree = list.tree();
      EDU.purdue.cs.bloat.editor.Instruction instruction =
        new EDU.purdue.cs.bloat.editor.Instruction(EDU.purdue.cs.bloat.editor.Opcode.ops_aaload);
      tree.addInstruction(instruction);
  }
```