McGill University
School of Computer Science
Sable Research Group

Sable API Design Guidelines

Sable Technical Note No. 1

Raja Vallée-Rai

March 3, 1998

Guideline 1: Use proper capitalization

1. Package names must be completely in lower case. e.g.

```
ca.mcgill.sable.jimple
com.sun.happy.applet
```

2. The first letter of class names and of each of their internal word must be capitalized. All other letters must be in lower case, except in the case of abbreviations. e.g.

```
JavaClass
JClass
MalformedURLException
```

3. Method names must begin with a lower case letter, but the first letter of internal words must be capitalized. The rest are to be in lower case, with an exception for abbreviations. e.g.

```
getComponent()
getMalformedURL()
```

Guideline 2: Avoid packaging unnecessary structure in base objects

Suppose you are creating an class, `Stmt`, which represents Java statements, and you know this object will be contained in a list. You might be tempted to give it `getNext()` and `getPrevious()` methods in order to navigate the list. That would unfortunately tie each `Stmt` to an implicit `List`. What happens when you want a `Stmt` object to part of a control flow graph instead? The methods `getNext()` and `getPrevious()` might become meaningless in that context.

The correct (and cleaner) strategy is to remove all the methods which pertain to structure from this class, and then place the `Stmt` instances in a particular `List`. This gives you more flexibility in building structures involving `Stmt` and simplifies its meaning.

However In some cases it is desirable to have methods relating to structure explicit in the contained objects. For example, when designing control flow graphs, it makes sense for the `BasicBlock` class to have methods like `getPreds()` and getSuccs(). This is because control flow graphs are used purely for their structure and basic blocks will only be used in conjunction with them.

Guideline 3: Pluralize references to objects stored in sets

References to a set of objects by the object name should be pluralized. e.g.

```
getElement()   should be    getElements()
getPred()      should be    getPreds()
```

Guideline 4: Avoid public variables

Unless if a class is a pure data structure, like a C `struct`, all accesses to variables should be through `getXXX()/setXXX()` methods. e.g. if you want to give public access to a local variable maxColors in an object, then it should be implemented with the methods:

```
getMaxColors()
setMaxColors()
```

In the case of boolean variables, you should use the `isXXX()/setXXX()` method pair

```
isExternal()
setExternal()
```

There are two reasons for this convention:

1. Accesses to local variables might be accompanied by validity checks. It is much easier to package these checks in a method then to have the invoker perform them every time the local variable is accessed.
2. Provides more flexibility for future modifications to the class (like inserting validity checks.)

Guideline 5: Use the prefixes `get()/set()` to access properties

Some aspects of objects can be termed *properties*. We define a property to be a value which, when retrieved through repeated method calls at a given program point, is always the same. For example, the `java.lang.Class` object has the following properties:

- fields
- name
- superclass

And so, to access these properties, the proper method names would be:

```
getFields()
setFields()
getName()
setName()
getSuperClass()
setSuperClass()
```

If a method which returns has a value has a side effect, then you should not use the get prefix. For example, a method which returns the bytes from a file, sequentially, should not be called `getBytes()`, but `readBytes()` since repeated calls to `getBytes()` may generate different values. The stream of bytes is not to be considered a *property* of the file.

Boolean properties are manipulated through variants of the `get() / set()` pair:

```
declaresField()

isExternal()
setExternal()
```

By using these conventions, learning how to use the API is simplified. For example, the programmer knows that `get()` methods have no side effects and can feel confident about how they will behave.

Note In C it is common practice to name functions by the values they return. This practice should be avoided. e.g. if you are returning a length of an object, it should not be `length()`, but `getLength()`.

Guideline 6: Choose simple names

Simpler names are better. They are easier to understand and remember. Here are some case examples on name simplification:

1. If you have a control flow graph method which returns all of the graph's basic blocks, and that is the only method which returns basic blocks, then call it `getBasicBlocks()` instead of `getAllBasicBlocks()`.
2. If there is only one notion of graph around, then instead of naming a class `ControlFlowGraph`, it may be appropriate to just name it `FlowGraph`, `ControlGraph` or `Graph`.

Although compound class names give the impression that they more accurately reflect the object being represented, like `ControlFlowGraph`, they, in fact, might hinder its manipulation due to its visual complexity and length. Simpler names which seem imprecise often recover their precision through usage. For example, a double floating point precision real numbers is just called a `double`. It is perfectly clear today what a `double` stands for and yet could have seemed simplistic and inaccurate when it was first introduced.

Guideline 7: Avoid abbreviations

Abbreviations increase the learning curve of an API because they are visually complex and users need to know what they stand for. Entire words are usually preferred for this reason, unless if the abbreviation is more often used than the complete name, like URL.

Class names like `MethodCFG`, which represent the control flow graph of a method, should be changed to something simpler like `MethodGraph`. Method names like `getBB()` should be expanded in full to `getBasicBlocks()`, which is much clearer.

However Abbreviations are acceptable in contexts where the full name does not provide a clearer meaning and may be cumbersome. In Jimple, I abbreviated the class names which represent statements to:

```
AssignStmt,
AddExpr,
ValueExpr,
```

Writing expressions such as

```
new AssignStmt(a, new AddExpr(new ValueExpr(b), new ValueExpr(c)))
```

is somewhat clearer than the complete

```
new AssignStatement(a, new AddExpression(new ValueExpression(b),
new ValueExpression(c)))
```

Similarly, when manipulating basic blocks, the successors and predecessors of a given block are often desired. Strictly speaking the methods to retrieve these should be named:

```
getPredecessors()
getSuccessors()
```

but

```
getPreds()
getSuccs()
```

seem to convey it just as well.

Guideline 8: Overload method names when appropriate

It is a good idea to overload method names when retrieving objects from containers by different aspects. Suppose methods are contained in a `JClass` object, and that these methods can be retrieved by name or by ID. One way of naming the retrieval methods is:

```
Method getMethodByName(String s)
Method getMethodByID(ID id)
```

It is convention however, to retrieve contained objects of class X with simply `getX()`. So, the names can be simplified to

```
Method getMethod(String s)
Method getMethod(ID id)
```

which uses overloading.

Guideline 9: Minimize your API

The best APIs are the simplest ones. More fields, methods, and classes in an API might provide more functionality, but they make the API more complex and harder to use. Your API should only contain the classes, fields and methods which are strictly necessary.

For example, suppose you want the end programmers to be able to build house objects, and these houses are built in three phases with the methods:

```
buildHousePhaseOne()
buildHousePhaseTwo()
buildHousePhaseThree()
```

You coud let them use these methods as well as `buildHouse()`. But if they will not find partially built houses useful, then you should only really make `buildHouse()` method available.