Software Engineering 4 (F22HX3)

Dynamic Analysis Techniques Part 2

Andrew Ireland

School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh

Review

- Functional testing:
  - Equivalence partitioning
  - Boundary value analysis
  - State based testing
- Structural testing:
  - Code coverage analysis
  - Data definition-use testing
- Hybrid testing:
  - Error guessing
  - Built-in test

Error Guessing

- Error guessing is driven by experience and will typically cover both functional and structural perspectives.
- To illustrate, given a software component to test that exploits dynamic storage allocation, then an experienced tester will typically focus upon the correct use of resource deallocation, *e.g.* seek out memory leakage defects.
- By its very nature, error guessing is hard to teach. However, check lists designed around language features and related defects can play a useful role in the transfer of experience.

Built-in Test

- Auxiliary code that integrates testing and code execution, typically achieved via **assertions**.
- An assertion is a boolean valued expression, *i.e.* evaluates to `true` or `false`.
- Assertions allow a programmer to express what they believe **should** be true at given points within their code.
- Note that here we only consider assertions within the context of dynamic analysis, *i.e.* assertions that are executed.
- Assertions will be represented differently depending upon your implementation language – we will focus upon assertions within Java.

Assertion Based Object Oriented Testing

- **Encapsulation:** gaining access to states and hidden instance variables is crucial if effective testing is to be conducted.
- **Inheritance and polymorphism:** overriding does not mean local testing is enough. That is, an inherited method that has be fully tested within the superclass must be re-tested within the context of the subclass.
- **Unit testing**: treating methods as the basis for unit testing will not work in general, a class perspective is called for, *i.e.* class states, class modality etc.

Design by Contract

- Strong analogy with legal contracts, *i.e.* an explicit statement about the benefits (rights) and obligations that exist between a client and supplier.
- Each supplier method has an associated precondition assertion that checks the correctness of client requests – client responsible for establishing precondition.
- Each supplier method has an associated postcondition assertion that checks the correctness of the supplier's:
  - response to the client and
  - suppliers state after the response is calculated.

  supplier responsible for establishing postcondition.

Note: intermediate method assertions may also be desirable, *e.g.* loop invariants.

Obligations and Benefits

	Obligations	**Benefits**
Client	Must ensure preconditions, *e.g.* *Ensure space in table for given key/value pair*	May benefit from postconditions, *e.g.* *Obtain new table updated with given key/value pair*
Supplier	Must ensure postconditions, *e.g.* *Ensure given key/value pair has been correctly added to the table*	May assume preconditions, *e.g.* *No need to consider if table is full*

Java Support for Assertions

- J2SE (Java 2 Platform, Standard Edition) 1.4 supports a simple assertion facility:
  - new keyword, *i.e.* `assert`
  - new class, *i.e.* `AssertionError`
  - some additional methods added to `java.lang.ClassLoader`
- J2SE 1.4 is the default on the departmental Linux PCs.

Java Assertions: Syntax & Semantics

Syntax: `assert` statements come in 2 forms:

- `assert expression1;`
- `assert expression1 : expression2;`

where `expression1` and `expression2` must be of type `boolean` and `String` respectively.

Semantics: execution requires evaluation of `expression1`:

- If `expression1` evaluates to `true` then no effect.
- If `expression1` evaluates to `false` then
  - default `AssertionError` constructor invoked (if no `expression2`).
  - otherwise evaluate `expression2` and use result with single-parameter `AssertionError` constructor.

A Simple Example

```
class Foo
{
  public static void even(int value)
  {
      assert 0 <= value;
      if (value % 2 == 0) System.out.println("Even");
                     else System.out.println("Odd");
  }
  public static void main(String[] args)
                                   throws IOException
  {   Foo foo = new Foo();
      even(2);
      even(3);
      even(-1);
  }
}
```

Compilation and Execution

- By default the J2SE 1.4 compiler does not allow `assert` statements, so you must tell it via a command line option, *e.g.*

  ```
  javac -source 1.4 Foo.java
  ```

- By default the Java run-time system ignores assertions, so again a command line option to enable assertions is required, *e.g.*

  ```
  java -enableassertions Foo
  ```

  fortunately this can be abbreviated to:

  ```
  java -ea Foo
  ```

- Note that default behaviour a little backward, developers are being very cautious.

Example Revisited

- **Compile:**

  ```
  $ javac -source 1.4 Foo.java
  ```

- **Execute with assertions disabled:**

  ```
  $ java  Foo
  Even
  Odd
  Odd
  ```

- **Execute with assertions enabled:**

  ```
  $ java -ea Foo
  Even
  Odd
  Exception in thread "main" java.lang.AssertionError
          at Foo.even(Foo.java:12)
          at Foo.main(Foo.java:23)
  ```

Assertion Messages

```
class Foo
{
  public static void even(int value)
  {
    assert 0 <= value : "Value must not be negative:
                         value= " + value;
    ...
  }
  ...

  $ javac -source 1.4 Foo.java
  $ java -ea Foo
  Even
  Odd
  Exception in thread "main" java.lang.AssertionError:
  Value must not be negative: value= -1
          at Foo.even(Foo.java:12)
          at Foo.main(Foo.java:23)
```


A More Interesting Example

```
class Book
{
    class Entry
    {
        String key;
        String value;
        Entry(String key, String value)
        {
            this.key = key;
            this.value = value;
        }
    }
    private int capacity;
    private int index;
    private Entry [] book;

    public Book(int capacity)
    {
        book = new Entry [capacity];
        this.capacity = capacity;
        index = 0;
    }
...
```

Specification via Comments

```
public void insertData(String key, String data)
{
  // assume data count is less than capacity

  ... your array insertion algorithm goes here ...

  // book now contains given data and
  // is associated with the given key and
  // data count has been increased by 1

}
```


Specification via Assertions

```
public void insertData(String key, String data)
{
    int initialCount = count();

    assert initialCount < capacity

  ... your array insertion algorithm goes here ...

    assert contains(data) &&
           iskey(data) == key &&
           count() == initialCount + 1
}
```

Specification & Implementation

```
public void insertData(String key, String data)
{
    int initialCount = count();

    assert initialCount < capacity

    Entry e = new Entry(key, data);
    book[index] = e;
    index++;

    assert contains(data) &&
           iskey(data) == key &&
           count() == initialCount + 1
}
```


Auxiliary Methods

```
boolean contains(String data)
    {
        int i=0;
        boolean found = false;
        while (i < index && !found){
            if (book[i].value == data) found = true;
            else i++;
        }
        return found;
    }
```

Exercise: Define the `count` and `iskey` methods.

Class Invariant for Book

```
boolean BookClassInvariant()
{
    int cnt = count();
    return cnt >= 0 && cnt <= capacity;
}
public Book(int capacity)
{
    ... code ...

    assert BookClassInvariant()
}
public void insertData(String key, String data)
{
    assert BookClassInvariant()

    ... code ...

    assert BookClassInvariant()
}
```


Design by Contract (more)

- Each class has an associated invariant assertion that characterizes the space of correct instance variable values for the class. Invariant assertions are checked when method precondition and postcondition assertions are checked.
- Class contracts must be consistent across subclass/superclass definitions, *i.e.* a subcontractor (supplier) that is employed to perform a specialized task takes on part of the responsibility of the main contractor (supplier):
  - Preconditions of subclass method must be the same (or weaker) than the associated superclass method.
  - Postconditions of subclass method must be the same (or stronger) than the associated superclass method.
  - Subclass invariant must be the same (or stronger) than the superclass invariant.

Contracts, Inheritance and Robustness

- Contract consistency across subclass/superclass definitions is not mechanized within Java, *i.e.* needs to be checked off-line by hand.
- The **Eiffel** object-oriented programming language provides greater support for the Design-by-Contract approach - see `http://www.eiffel.com/`
- Input validation checks should not be confused with assertions, *i.e.*
  - Input validation checks are about **robustness**
  - Assertions are about **correctness**
- Input validation checks and assertions are complementary.

Evaluation of Assertion Based Testing

- Advantages:
  - Client faults typically appear as supplier failures – supplier preconditions eliminate this level of indirection.
  - Class invariants can detect faulty states that arise from internal class level faults.
  - Assertions give access to encapsulated/hidden information.
  - Assertions encourage well-documented source code and are likely to remain the most consistent form of documentation.
- Disadvantages:
  - Does not directly address issues of class modality defects.
  - Assertions can be faulty!

Summary

- The importance of error guessing.
- The value of built-in test via executable assertions – "active comments".
- Assertions within the context of object-oriented programming and Java in particular.

References

- "Testing Object-Oriented Systems", Binder, R.V. Addison-Wesley, 1999.
- "Design By Contract", Meyer, B. IEEE Computer 25(10):40-51, 1992.
- BCS (1995). Standard for Software Component Testing. Working Draft 3.4, BCS Specialist Interest Group in Software Testing (BCS SIGIST), April 2001. `http://www.testingstandards.co.uk`
- ANSI/IEEE standard 1008-1987: IEEE standard for software unit testing. IEEE 1987.

References

- Java assertions –
  `http://www.java.sun.com/j2se/1.4/docs/guide/lang/`
- Assertion Definition Language (ADL) C and C++ compatible:
  `http://adl.opengroup.org/`
- AssertMate – a code assertion system for Java, RST Corp:
  `http://www.cigital.com/products/`
- Test Coverage Tool Venders:
  `http://www.iplbath.com`
  `http://www.reflex-tech.co.uk`