COMP 401

Prasun Dewan[1]

20. Assertions and Visitors

Assertions are useful to convince ourselves that our programs work correctly. They can be used in formal correctness proofs. A more practical application is to encode them in programs to catch errors. An even more practical example, illustrated in ObjectEditor, is to use them to automatically enable/disable user-interface items. They extend the notion of unchecked or runtime exceptions.

Language-supported vs. application-specific assertions

Assertions are statements about properties of programs. We have been making them implicitly. Each time we cast an object to a type, we assert that the object is of that type. For example, in the class cast below:

```
((String) next ())
```

we assert that the object returned by next () is a String. In this example, the programming language understands this assertion and provides a way to express it. However, a programming language can provide us only with a fixed number of application-independent assertions. Some assertions are application-specific. For example, we might wish to say that the string returned by nextElement() is guaranteed to begin with a letter. The programming language does not understand the notion of letters (though the class Character does) and thus cannot provide us with a predefined way to express this assertion. More important, even if it did understand letters, there are innumerable assertions involving them to be burn into the language. For example, we may wish to say the second element of the string is guaranteed to be a letter, or the third element of the string is guaranteed to be a letter and so on. What is needed then is a mechanism to express arbitrary assertions.

Assertions support in Java 1.4

After Java was first designed, James Gosling, its inventor said in a conference that his biggest mistake was not supporting assertions. Java's extensive runtime checking reduces the impact of this mistake – it automatically catches many inconsistencies that would have to be explicitly asserted in other programming languages. The most common assertions I have seen for C code are subscript checking assertions that indicate that an index variable takes values that are within the bounds defined by the array it indexes. Subscript checking and many other internal errors are automatically checked by Java. However, as demonstrated by the examples above, there is still need for programmer-defined exceptions.

[1] © Copyright Prasun Dewan, 2009.

Though it does not support pre-conditions, post-conditions, invariants, and quantifiers, Java (version 1.4) now supports basic assertions, illustrated in the statement above. It defines the AssertionError error, which is like an Exception in that it can be thrown and caught. The gene

assert <boolean expression>

and

assert <boolean expression>: <value>

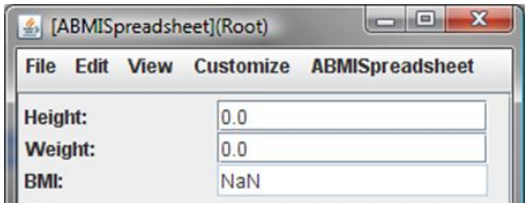
statements. Both statements throw an AssertionError if the Boolean expression is false. The second one sets the message of AssertionError to <value>.toString(). Thus, we execute these statements to tell Java tell Java we expect some condition, expressed as a Boolean expression, to hold true. If the condition is false, Java throws an AssertionError, which is normally handled by terminating the program and printing a default message, if the first statement is executed, and <value>.toString() in the second case. You might have seen assertion failed messages before because it is not uncommon for some product to fail with a message of the form "internal assertion failed".

BMI with Assertions

To concretely illustrate the nature and use of assertions, consider again our original `BMISpreadsheet` shown in the figure below.

It has the problem that the height and weight have illegal values. As a result, the BMI value makes no sense. We saw that we could use constructors to prevent the initial values of these variables from being inconsistent. But how can we ensure that subsequent illegal changes to these values result in an error message? The following code shows how assertions can be used to avoid illegal values of BMI:

```
public class ABMISpreadsheet {
      double height, weight;

      public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
            setHeight( theInitialHeight);
            setWeight( theInitialWeight);
      }
```




```
      public double getHeight() { return height; }
```

```
	public void setHeight(double newHeight) { height = newHeight; }
	public double getWeight() {return weight; }
	public void setWeight(double newWeight) {weight = newWeight; }
	public boolean preGetBMI() { return weight >= 0 && height >= 0;}
	public double getBMI() {
		assert (preGetBMI());
		return weight/(height*height);
	}
}
```

Here `getBMI` calls a method called, `preGetBMI`, which, in turn, makes asserts the following Boolean expression:

```
weight > 0 && height > 0
```

If this condition is false, the assertion fails and the Java program terminates. An expression such as this that must be true at the start of the method is called a precondition of the method. A method precondition is much like a course pre-requisite. If you don't meet a course prerequisite, you are not able to completely understand the course. Similarly, if a method prerequisite is not met, the method will not execute correctly.

It is a good idea, as we have done above, to check the precondition of some method `M`, in a separate Boolean public method, called `preM`. This way, another method, `P`, that wishes to call `M` can first call `preM` first to see if the precondition of M is met. If `preM` returns false, then P should not call `M` to halt the program. Of course, you can give any name to the method that checks the precondition. However, a standard naming strategy such as the one we have used above helps makes it possible to find the precondition method without reading documentation. Such a naming strategy is consistent with the idea of using Bean conventions to name getters and setters.

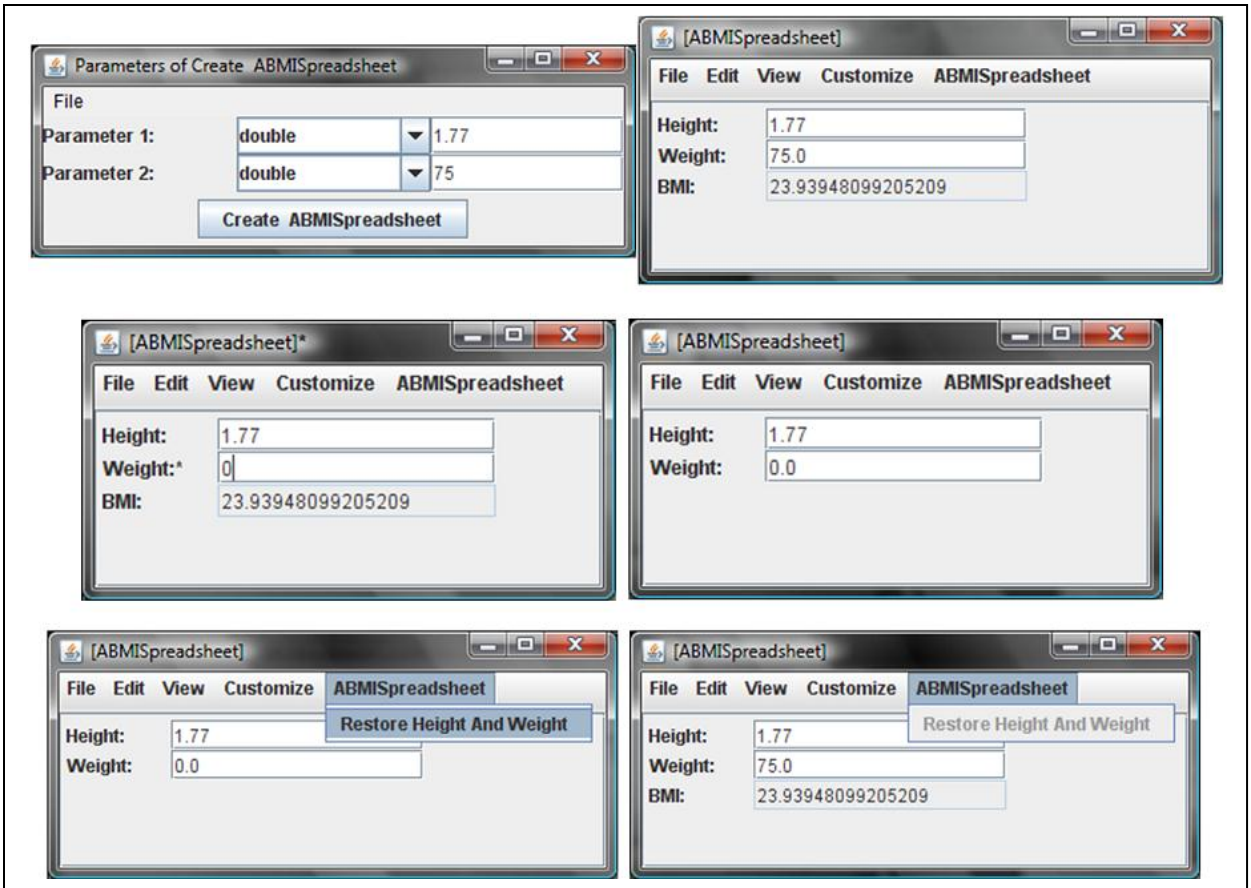


Figure 2. ABMISpreadsheet with preconditions

Like Bean conventions, the above naming strategy is used by ObjectEditor to determine if a getter method should be called. This is illustrated in the interaction shown in Figure 2. Suppose we create a new instance of `ABMISpreadsheet` with legal values (Figure 2 top left and right) and then try to enter the illegal value, 0.0, for the weight (Figure 2 middle left). If we had used the previous version of the class, ObjectEditor would have called `getBMI` to refresh the display of BMI. However, as we now have a precondition method for `getBMI`. ObjectEditor calls this method before it tried to call `getBMI`. As this method returns false, it does not call `getBMI`. Since it does not have a legal value for the BMI field, it simply removes the field from the display (Figure 2 middle right). It will redisplay the field when we enter a legal value for weight.

As the above example illustrates, at times we might want to restore the values of instance variables to their original values. We can provide a special method to do so, as shown in (Figure 2 bottom left). Notice that after we execute the command to invoke the method, it is disabled by ObjectEditor (Figure 2 bottom right). This makes sense because the variables already have their initial values. In general, ObjectEditor disables the menu command for a method whenever its precondition is not met. This implies that the newly added method has a precondition method that checks if the current values of the instance variables are their initial values. This, in turn, implies that the object stores the initial values of

the variables. As initial values are given in the constructor, we must change this method to store these values.

The following code gives the new version of the class:

```
public class ABMISpreadsheet {
	double height, weight;
	double initialHeight, initialWeight;
	public ABMISpreadsheet(
		double theInitialHeight, double theInitialWeight) {
		setHeight( theInitialHeight);
		setWeight( theInitialWeight);
		initialHeight = theInitialHeight;
		initialWeight = theInitialWeight;
	}
	public double getHeight() {return height; }
	public void setHeight(double newHeight) { height = newHeight; }
	public double getWeight() {return weight; }
	public void setWeight(double newWeight) {weight = newWeight; }
	public boolean preGetBMI() { return weight > 0 && height > 0;}
	public double getBMI() {
		assert preGetBMI();
		return weight/(height*height);
	}
	public boolean preRestoreHeightAndWeight() {
		return height != initialHeight || weight != initialWeight;
	}
	public void restoreHeightAndWeight() {
		assert preRestoreHeightAndWeight();
		height = initialHeight;
		weight = initialWeight;
	}
}
```

In the above code, we have predefined preconditions for only two of the methods. Should the other methods also have preconditions?

Consider first `setWeight` and `setHeight`. It does not make sense to give a negative or zero value to the weight and height, hence we can assert preconditions that check this condition as illustrated below:

```
		public boolean preSetWeight (double newWeight) {
			return newWeight > 0;
		}
		public void setWeight(double newWeight) {
			assert preSetWeight(newWeight);
			weight = newWeight;
		}

		public boolean preSetHeight (double newHeight) {
```

```
        return newHeight > 0;
    }
    public void setHeight(double newHeight) {
        assert preSetHeight(newHeight);
        height = newHeight;
    }
```

We can similarly add preconditions for the getters:

```
    public boolean preGetWeight () {
        return weight > 0;
    }
    public double getWeight() {
        assert preGetWeight();
        return weight
    }
    public boolean preGetHeight () {
        return weight > 0;
    }
    public double getHeight() {
        assert preGetHeight();
        return height
    }
```

However, once we have checked the preconditions for the two setter methods, we don't need to write preconditions for any of the three getter methods. This is because the setters have made sure that illegal values cannot be assigned to the variables.

Expressing Assertions

In the examples above, we have used executable code to express assertions. This is one of three approaches typically used to express assertions, with their pros and cons:

- Write assertions informally using a natural language such as English. These are easy to read but potentially ambiguous. (e.g all elements of this array are either odd or positive or all array elements are not odd.)
- Write them in a programming language such as Java or Turing. These are executable and can thus be used to raise runtime exceptions. They can be supported by a library or special language constructs. In either case, they are language-dependent, and can thus not be used as a specification of an application before we have decided the programming language in which it will be coded. Perhaps more important, there is no programming language that allows us to conveniently express the kind of assertions we would like to make. Some languages do a better job than others. For instance, Turing offers more support for assertions than Java.
- Write them in a mathematical language. Mathematical languages tend to be thorough, carefully thought out, and "time-tested". However, they are not executable and thus cannot be used for testing.

Propositional Calculus and Quantifiers

There exists a mathematical language, called Propositional Calculus, developed for expressing the kind of assertions we would like to make. Propositional calculus describes a language for creating *propositions.* A proposition is a Boolean expression. The expression can refer to Boolean constants, *true* and *false*. In addition, it can refer to *propositional variables*. These variables include the variables of the program about which we are making assertions. They can also include assertion variables, such as the recording variables we saw earlier.

Thus, the following are propositions:

true
false
x > 6

The result of a proposition is determined by the values assigned or *bound* to the propositional variables to which it refers. Thus, if x has been assigned the value 7, then the last proposition is true.

The calculus defines three Boolean operations: *not, and,* and *or.* All other Boolean operations such as *xor* can be described in terms of these operations. It does not define arithmetic and relational operations such as *+,* square*,* and *>,* these are defined by the *algebra* on which the calculus is based.

For the purposes of this course, we will assume that the logical and arithmetic operations are defined by Java. Moreover, we will assume that the Boolean operations are also defined by Java. This will allow us to use Java syntax for these operators, which will make it easier to convert propositional assertions into something Java can automatically check. Thus, we will consider the following to be a proposition in propositional calculus:

(x > 6) && (y > 6)

Also we will use *standard* mathematical functions such as *min, max,* and Σ when Java does not directly implement these operations.

By this definition, we have already been using propositional calculus for expressing assertions, since all of our assertions have used Java operations. The reason for bringing up propositional calculus is that we have not used the full power of this calculus. A proposition can be a *simple proposition* or a *quantified proposition*. A simple proposition is a Boolean expression involving individual variables of the program we are asserting about. The assertions we have seen so far are simple since they involve statements about individual variables such as x and y.

We may want to make assertions about collections of variables. For instance, we might want to say:

All elements of b are not null.

At least one element of b is not null.

Quantified propositions allow us to express such assertions formally:

$\forall$j: 0 < j < b.size() : b.get(j) != null

∃j: 0 < j < b.size(): b.get(j) != null

The symbols ∀ and ∃ are called *quantifiers* in Propositional Calculus. The former is called a *universal quantifier* while the latter is called an *existential quantifier.* The *j* in the examples is a variable *quantified* by the quantifier preceding it. The term:

0 < j < b.size()

describes the *domain* of the quantified variable. A domain is a collection of values such as b.get(0), … b.get(b.size() – 1). The Boolean expression:

b.get(j) != null

describes a (simple or quantified) proposition in which the quantified variable j can be used as a proposition variable in addition to the program variables. We will call this proposition the *sub-proposition* of the quantified proposition. As we see above, this subproposition is evaluated for each element of the domain.

Thus, the general form of a quantified assertion is:

Qx:D(x):P(x)

where Q is either ∀ or ∃, x is a quantified variable, D(x) describes its domain using x, and P(x) is a predicate or Boolean expression involving *x* (and other variables).

Both quantifiers *bind x* to each value of D(x) and calculate P(x) with this bound value. If Q is ∀, then the quantified assertion is *true*, if P(x) is true for *each element*, *x,* of D(x). If Q is ∃, then the quantified assertion is true if P(x) is true for *at least one element*.

Thus, the first assertion above is *true* if for every value of *j* in the range 0 < j < B.length, B[j] >= 0. The second assertion is true if there exists a value of *j* in the range *0 < j < B.length*, for which *B[j] >= 0*.

Implementing Quantified Asserions

Consider now quantified propositions:

Qx:D(x):P(x)

Ideally, what we would like are *quantifier functions* that automatically do the quantification for us and return boolean results.

In order to write such functions, we need to be able to encode information about the three parts of a quantified proposition: the quantifier, the domain, and the sub-propositition.

. The most convenient encoding approach would be to allow us to pass String expressions for the three components:

Asserter asserter = **new** AnAsserter();
assert asserter.checkQuantified("∀j: 0 <= j < b.size(): b.get(j) != null"): "some element of b is null" ;

That approach makes the functions methods complex, as they must do some of the work of a compiler/interpreter. More important, the methods cannot access the non-public variables referenced by the expression as names of these variables, such as b, declared in our scope, have no meaning to a library.

We will encode the quantier in the name of the quantfying function, defining separate methods for each quanitifier. Moreover, we will separate the task of defining the domain and proposition, by associating them with different arguments of the methods. Thus, our interface becomes:

```
public interface Asserter {
    public boolean checkUniversal (…, …);
    public boolean checkExistential (…, …);
}
```

Since the domain is a sequence of elements, we could model it using the standard Java List interface.

```
    public boolean checkUniversal (List<ElementType> domain, …);
    public boolean checkExistential (List<ElementType> domain, …);
```

However, this approach does not allow us to model sequences such as histories, hashtables, and streams that are not lists. The functions above do not need several operations provided by List such as remove() and contains(). In fact, they do not need to even assume that the sequence is indexable. All they need is a way to access all elements of the sequence. All they need is a way to iterate the elements of the sequence. Thus, the standard Java Iterator interface is far more suitable:

```
    public boolean checkUniversal (Iterator<ElementType> domain, …);
    public boolean checkExistential (Iterator<ElementType> domain, …);
```

The implementation of these functions would simply iterate through the iterated elements, as shown below:

```
import java.util.Iterator;
public class AnAsserter<ElementType> implements Asserter<ElementType> {
    public boolean checkUniversal (Iterator<ElementType> elements, …) {
        while (elements.hasNext())
            if (…) return false;
       return true;
    }
    public void checkExistential (Iterator<ElementType> elements, …) {
        while (elements.hasNext())
```

```
            if (…) return true;
        return false;
    }
}
```

The two functions use the domain argument to look at each of the elements of the domain. A user of the function is responsible for mapping an actual collection to an iterator object. Our example assertion, thus, would be of the form:

assert asserter.checkUniversal(b.iterator(), ….): “some element of b is null” ;

The code above shows … to indicate that we must next worry about how a sub-proposition is to be encoded.

Encoding the sub-proposition is trickier. Unlike the proposition argument of the **assert** statement, it is not an evaluated boolean expression. Instead, it is a function of the quantfied variable that can return a different value each time it is bound to a new member of the domain.

What we need to do is pass a Boolean function as a parameter to each of the two methods that can be called by them. For example, we can write the following function:

```
boolean nonNullChecker(Object element) {
   return element != null;
 }
```

and execute the following assertion:

```
assert asserter.checkUniversal(b.iterator(), nonNullChecker, “some element of b is null” )
```

Here, in addition to the domain object, a reference to the function is passed. A quantifier method now passes each element of the domain to the function received as a parameter:

```
public boolean checkUniversal (Iterator<ElementType> elements, (<ElementType→
boolean)elementChecker ) {
      while (elements.hasNext())
            if (!elementChecker(elements.next())) return false;
       return true;
}
```

As shown above, the `checkUniversal`() function returns false if the subproposition evaluates to false for any element, and true otherwise. The checkExistential() function, on the other hand, returns true if the function evaluates to true for any element of the domain, and false otherwise.

As it turns out, Java does not support a typed method parameter, where user of the parameter can constrain the parameters and return type of the method parameter. The Method object, as we saw before, is not a typed Method, as it can be assigned any method.

However, as we know, Java does allow object parameters; and an object is a collection of methods (and data). So instead of defining a method parameter that evaluates the sub-proposition, we can define an object parameter on which we can invoke a method that evaluates the sub-proposition. Our quantifier function now changes as follows:

```
public boolean checkUniversal (Iterator<ElementType> elements, ElementChecker
elementChecker ) {
      while (elements.hasNext())
            if (!elementChecker.check(elements.next())) return false;
       return true;
}
```

As we see above, a quantifier method is passed, instead of a sub-proposition function, a sub-proposition object on which the function can be invoked. The implementation does not call the function directly as it did in the previous version. Instead, it calls the function on the sub-proposition object passed as a parameter, which visits each element of the domain. If each sub-proposition object implements the same interface, then the quantifier methods can be defined in terms of this interface.

Here we assume that the interface implements the check() method:

```
package util.assertions;
public interface ElementChecker<ElementType> {
   public boolean check (ElementType element);
}
```

Now, we must put the function we wrote above in the class of the sub-proposition object:

```
import util.assertions.ElementChecker;
public class ANonNullChecker implements ElementChecker<Object> {
  public boolean check(Object element) {
      return element != null;
  }
}
```

Our example assertion thus becomes:

assert asserter.checkUniversal(b.iterator(), **new** ANonNullChecker())): “some element of b is null” ;

Instead of passing the function directly as a parameter, we have passed an object encapsulating it.

I

Let us complicate matters by considering how the following assertion can made:

$\forall j: 0 < j < b.size() : b.get(j) != a.get(0)$

How should the check method, which receives only the domain element as an argument, reference a.get(0)? This problem arises because these assertions reference variables other than the one describing

the domain. In general, sub-proposition P(x) can not only reference the quantified variable, x, but also reference any other visible variable. Thus, its general form is:

P(x, v1, … vn)

We can imagine adding another argument to the check function that contains an array of the values of v1, … vn. The caller of the quantifier functions passes this array to them, and they in turn pass them to the check function. However, a simpler approach is possible, which relies on the fact that the same values are used in each visit to the domain element. As a result, the caller of the quantifier function can pass them to the constructor used to create the sub-proposition object, which can then store them in instance variables of the object. The check() function can refer to these instance variables instead of the array parameter. This allows us to use meaningful names for these values.

Thus, for the above assertions, we implement the following class:

```
import util.assertions.ElementChecker;
public class AnInequalityChecker implements ElementChecker<String> {
        Strring testObject;
        public AnInequalityChecker(String theTestObject) {
                testObject = theTestObject;
        }
        public boolean check(String element) {
                return !element.equals(testObject);
        }
}
```

and pass an instance of it to our assertion:

```
AnAsserter.assert(AQuantifier.forAll(b.elements(), new AnInequalityChecker(a.get(0))), "Some element of b is equal to a.get(0)");
```

```
assert asserter.checkUniversal(b.iterator(), new AnInqualityChecker()): “some element of b == a.get(0)” ;
```

The asserter passed the value of a.get(0) to the constructor of the subproposition object. This value is stored in an instance variable. The visit function now becomes an impure function using both its parameter and the value of the global variable.

Visitor Pattern

We have seen above an example of the visitor pattern, so called because it allows a method to be invoked or “visit" a sequence of elements. The pattern has several components:

- A collection interface C providing a way to access elements of type T. In our example, the interface was java.util.Iterator, but in general it can be any interface providing a way to access the elements.

- An interface, called a visitor interface, defining a method, called a visit method, that takes a single argument of type T. This method is invoked on each element of the sequence. In our example, the method returned a Boolean result. In general, its return type can be arbitrary including **void**.

  ```
  public interface V {public T2 m (T p);}
  ```

  A visitor method can perform many useful activities involving the elements on which it is invoked such as printing the elements, gathering statistics about them, and modifying them.

- One or more traverser methods that use the collection and visitor interface to pass one or more collection elements to the visitor method.

  ```
  traverser1 (C c, V v) { …v.m(element of C)…}
  ```

- One or more implementations of the visitor interface whose constructors take as arguments external variables that need to be accessed by the visitor method

  ```
  public class AV1 implements V {
    public AV1 (T1 p1, … Tn pN) { …}
    public T2 m (T p) { … }
  }
  ```

A client or user of the pattern uses instantiates the visitor implementations and passes them as arguments to a traverser method.

The following figure shows the general pattern and its implementation in the example above.

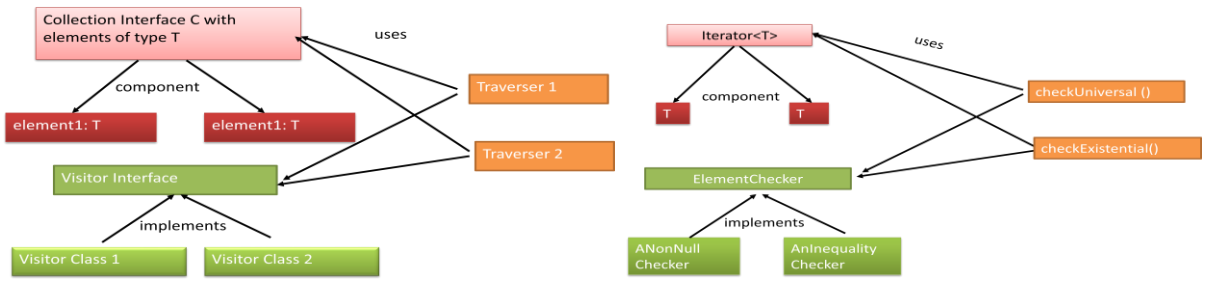


Figure 3 General Visitor Pattern and Example

A client of the pattern passes a traverser an instance of the visitor and collection interfaces:

```
traverser1(c, new AV1(a1,.. aN))
```

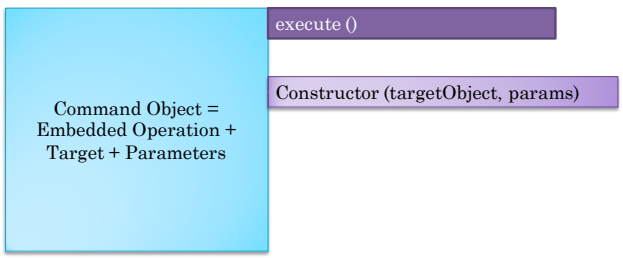
This pattern is useful in a compiler. As mentioned before, a compiler builds a tree representing the program. Several kinds of operations can be done on the tree nodes such as printing, code generation, refactoring, and type checking. Instead of the bundling all of these complex operations in the program object, it is more modular to put in them in separate visitor classes.The program object or some other class can implement one or more traverser methods that take as arguments instances of these classes, and call their visitor methods on the program elements.

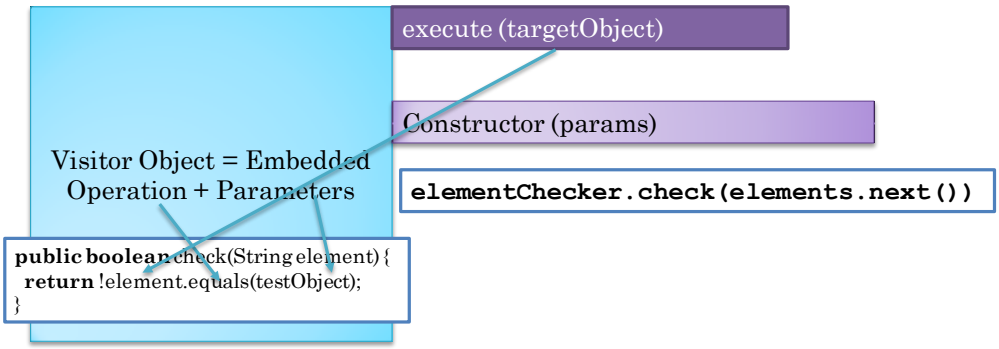
The pattern is also used in ObjectEditor to perform several operations on logical components of an object as printing it and associating it with widgets, registering listeners of the object, and processing annotations in the class of the object.

Action vs. Command vs. Visitor

A visitor object is related to an action and command object in that it is associated with an embedded operation and provides an execute method to invoke the operation, which we refer to as a visit method. However the constructor and execute operations of command, action, and visitor objects are different:

- *Action Object = embedded operation:* An action object is not bound to any object or parameters. It is bound only to the embedded operation. As a result, the execute method provided by it takes parameters speciffying the object on which the embedded operation must be executed and the parameters of the operation. For example, the getBMI() method object is bound to the getBMI() embedded operation provided by the interface BMISpreadsheet. The invoke() (execute()) method provided by the method object takes as parameters the BMISpreadsheet object on which the embedded opertation must be executed and the (empty) array of parameters of the operation.
- *Command object = embedded operation + parameters + target:* A command object is an object that is bound to not only an embedded operation, but also the parameters and target of the embedded operation, which are passed to the object in the constructor. As a result, the execute operation provided by it takes no parameters. For example, the ASetWeightCommand() command object is bound to the embedded setWeight() operation defined by BMISpreadsheet. Its constructor takes as arguments the instance of BMISpreadsheet on which the embedded operatio must be executed and the parameters of the operation.
- *Visitor object = embedded operation + parameters:* A visitor object is in between an action object and a command object. It is bound to both an embedded operation and parameters of the operation, which are passed to it in the constructor. It's execute operation, thus, takes as a parameters the target object on which the embedded operation must be invoked. This, like an action object, it is not bound to a target object, and like a command object, it is bound to parameters of the operation. For example, AnInequalityChecker() is bound to the embedded operation, equals(), provided by String. The parameter of the operation is passed to it in its constructor. The target of the operation is passed to it as an argument to its check() (execute()) method.





The figure above shows the difference between the three kinds of objects.

Nested Quantification and Complex Visitors

To really complicate matters, consider the following assertion:

∀j: 0 <= j < b.size(): ∀k: 0 <= k < b.elementAt(j).size(): b.elementAt(j).elementAt(k) != null

This assertion is made for a list b whose elements are themselves lists. It says no element of an element of b is null. In other words, in this example, b is a matrix or a table, and the assertion says no cell in the matrix/table is null.

Our assertion library assumes one-level elements. It makes no assumptions about the structure of each element. Thus, it cannot directly traverse the children of the elements. This task must be carried out by the visit method, which calls another visit method to traverse the inner elements. This is shown below:

```
public class AListChecker implements ElementChecker<List> {
   public boolean check(List element) {
      Iterator children = element.iterate();
      return asserter.checkUniversal(children, new ANonNullChecker()));
   }
}
```

Thus, the check() method in this example visits an element by traversing the children of the element using another visitor object. The method assumes the outer elements are List objects. If they are other kinds of sequences, we must write different visit methods.

Language vs. Library Support for Assertions

Why bother with language support when we can easily provide library support to throw programmer-defined exceptions when some condition is not true? In particular, we can define an assert(<Boolean Expression>) method in some predefined class that throws a special exception denoting assertion error.

Java's language support allows a programmer to turn on or off assertion checking without changing the source code. This is useful as we might want assertions to be turned off when performing slow or time critical computations. In fact, by default, assertion checking is turned off. The –ea and –da options to the Java interpreter indicate that assertions should be enabled and disabled, respectively. In fact, it is

possible to enable and disable assertion checking for specific packages and classes, as shown below, which disables assertion checking for ObjectEditor (as it is so error free ☺) but enabled assertions for the class examples.assertions.ABMISpreadsheet, when executing the main method in class <MainClass>.

java –da:bus.uigen… –ea:examples.assertions.ABMISpreadsheet <MainClass>

Interestingly, AssertionError, thrown when an assertion fails, is a subclass of Error rather than RunTimeException. The reasoning given is that convention dictates that RunTimeException should be caught (even though it may not be acknowledged). Java documentation says that we should "discourage programmers from attempting to recover from assertion failures." Apparently, this decision was controversial. Java does allow assertions to be caught to do custom reporting such as a mail error report.

Actual Library Code

The actual library code provided with ObjectEditor is a bit different from what has been described above and is given below:

```
package util.assertions;
import java.util.Iterator;
public class AnAsserter<ElementType> implements
Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType>
elements, ElementChecker elementChecker, String message) {
        while (elements.hasNext())
            if (!elementChecker.check(elements.next())) throw
new AssertionError (message);
    }
    public void assertExistential (Iterator<ElementType>
elements, ElementChecker elementChecker, String message) {
        while (elements.hasNext())
            if (elementChecker.check(elements.next()))
return;
        throw new AssertionError (message);
    }
}
```

Here checking and throwing of an AssertionError have been combined together. Thus, instead of writing :

assert asserter.checkUniversal(b.iterator(), nonNullChecker, "some element of b is null");

you would say:

asserter.assertUniversal(b.iterator(), nonNullChecker, "some element of b is null");

These two are not equivalent statements. The former throws an AssertionError on failure only when assertion checking is enabled. The latter always throws an AssertionError on failure – so you don't have to change your Eclipse settings.

The Importance of Being Earnest

To illustrate the importance of using assertions, consider the following code to manage bank transactions.

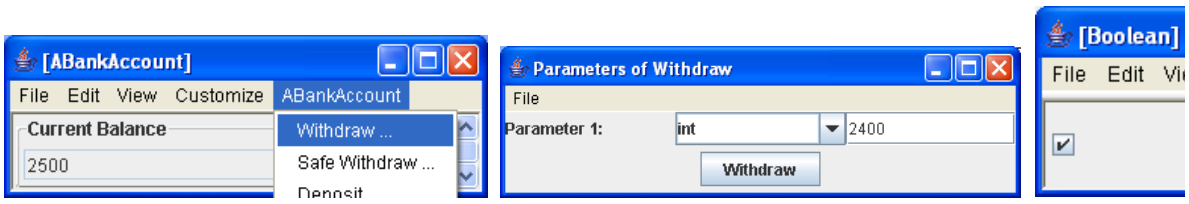
```
public class ABankAccount implements BankAccount {
        int currentBalance = 0;
        public static final int MIN_BALANCE = 100;
        public  ABankAccount (int initialBalance) {
                currentBalance = initialBalance;
        }
        public int getCurrentBalance () {return currentBalance; }
        public void deposit (int amount) {currentBalance += amount;}
        public boolean withdraw (int amount) {
                int minNecessaryBalance = MIN_BALANCE + amount;
                if (minNecessaryBalance  <= currentBalance) {
                        currentBalance -= amount;
                        return true;
                } else return false;
        }

}
```

The object priovides operations to get the current balance and add and withdraw money. The most complicated operation is the withdraw operation, as it must ensure that the account holder cannot withdraw an amount that will make the minimum balance below the value MIN_BALANCE. The operation is coded without assertions. If we were to use assertions, we would saty that the precondition of the operation is that the amount is not negative (which would also be the precondition of the deposit operation) and the post condition is that the currenBalance is at least MIN_BALANCE. Thus we might have the following alternative implementation of withdraw:

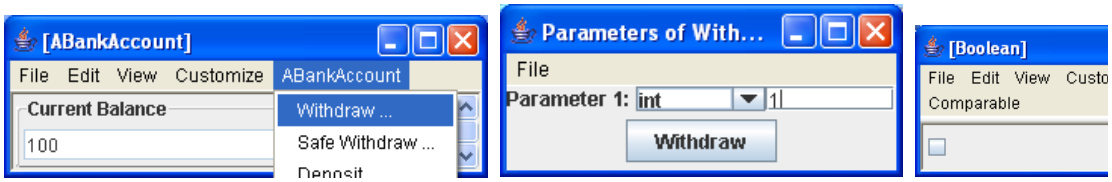
```
        public boolean safeWithdraw (int amount) {
                assert amount > 0: "amount < 0";
                boolean retVal = withdraw(amount);
                assert currentBalance >= MIN_BALANCE: "currentBalance
< MIN_BALANCE");
                return retVal;
        }
```

Perhaps the assertions seem an overkill as the operation seems to be coded correctly. But, in fact, it has problems that the assertion code can catch. To illustrate, consider an implementation of the object with both the withdraw() and safeWithdraw() operations.

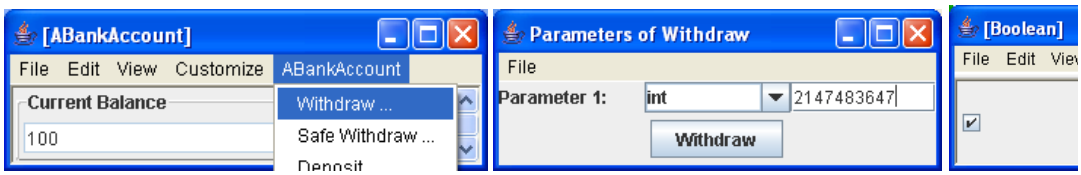
Assume that that I have a ciurrent balance of $2500 and I withdraw $2400. The withdraw() operation lets me correctly withdraw the money, as shown below.



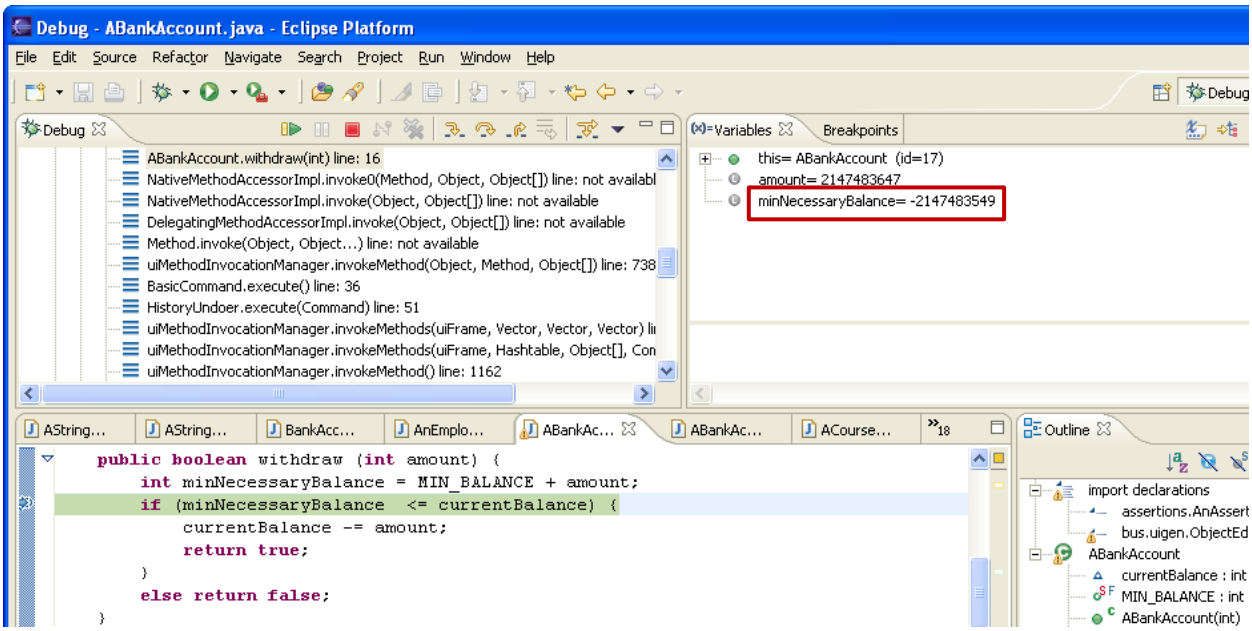
Next, if I try to withdraw $1, the operation correctly fails, as that would make my current balance below the threshold, as shown below.



Surprsingly, though, if I try to withdraw a much larger amount, an amount that is the biggest **int** value supported by Java, my operation succeeds!

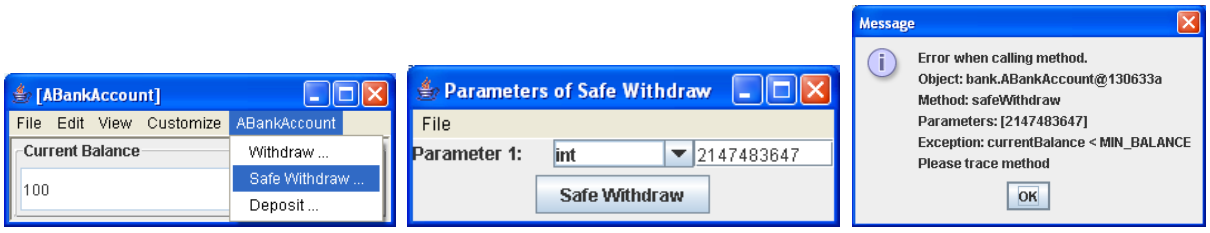


So I cannot withdraw $1 but I can withdraw $2147483647! To understand what is going on, let us trace the program that checks if the withdrawal is safe using the Eclipse debugger.



The highlighted area shows what went wrong. Because the variable, amount, holds the maximum int value, when we added MIN_BALANCE to it, the int value overflowed and became the negative number: -2147483549. An int value is a 32 bit number. Half of these numbers are positive numbers, and the other half are negative numbes. The most significant bit in the binary representation of such a number determines if the remaining bits encode a positive or negative numbers. This bit is 1 for negative numbers and 0 for positive. When we add something to the highest positive number, the 0 bit becomes 1, and the number becomes negative.

This is obviously a very subtle problem; so a programmer not familiar with overflow problems cannot be blamed for writing the withdraw() method above. The fault is in not asserting the post condition, as in safeWithdraw(). Such an assertion would have prevented the large amount of money from being withdrawn, as shown below:



Here, ObjectEditor catches the error and shows the error message associated with the assertion.

Exercises

1. Write boolean expressions that are equivalent to the English assertions given below. That is, they should return true if the assertions are true, and false otherwise. An example assertion and corresponding expression are given below. Assume i is an int variable, c1 and c2 are char variables, and b1 and b2 are boolean variables.

   (e.g.) i is 2.

   Answer: i == 2

   (a) i is greater than 5

   (b) i is in the range 2 to 5 inclusive.

   (c) i is an odd number.

   (d) exactly one of b1 and b2 is true.