Design by Contract for Java

Jim Weirich

What is Design by Contract? ... and Why Should I Care?

Design by Contract (DbC for short) is way of specifying interfaces such that the client and the supplier have a clear understanding of their responsibilities. The responsibilities are explicitly set forth in a list of preconditions and postconditions to a procedure.

Although Design by Contract could be applied to any programming style, it is especially useful in specifying the behavior of a object in OO programming.

A DbC Example

Assume you are writing a class that implements a stack object. You would probably provide the following 4 methods in some form or another.

- **int count()** -- Return the number of items on the stack.
- **int top()** -- Return the top of the stack.
- **void pop()** -- Remove the top of the stack.
- **void push(int n)** -- Push item *n* onto the stack.

  NOTE: *In our example, pop() is a void function and does not return the item it removes.*

Now that we have a list of methods for our class, let's pause a minute to think about how they are used. Lets start with **top()**. Is there anytime when it does not make sense for the client to make a call to **top()**? Yes! If the stack is empty! We can capture that constraint as a precondition to the **top()** method ...

- **int top()** -- Return the top of the stack.
  - Require: *count() > 0*

A requirement is a condition that the client of Stack must fulfill. If I am writing code that uses a stack, I must make sure that there are items in the stack before I make a call to **top()**. If I call **top()** on an empty stack, that is a *bug* in the client and should be fixed before shipping. Let's continue by looking at **pop()**. It turns out that **pop()** has the same precondition as **top()**, it must not be called on an empty stack. We can also make a statement about the effects of **pop()**, the stack size will decrease by 1. We can capture this in a contract ...

- **void pop()** -- Remove the top of the stack.
  - Require: *count() > 0*
  - Ensure: *count() == old count() - 1*

  NOTE: *The expression **old count()** refers to the value of count() at the beginning of the call to **pop()**.*

The ensure clause lays out the responsibilities of the object. After **pop()** is called, the number of items in the stack

will be reduced by 1, assuming the precondition holds.

That assumption of the precondition is important. If **pop()** is ever called without the precondition being true, then it is under no contractual obligations to fulfill its part of the bargain.

Lets look at **push(int n)**. It has two obvious postconditions (1) the stack has grown by one, and (2) the argument to push becomes the new top item. This gives the following summary ...

- **void push(int n)** -- Push item *n* onto the stack.
  - Require: *?*
  - Ensure: *count() == old count() + 1*
  - Ensure: *top() == n*

But what about a precondition for **push(int)**? It is at this point we have to make some decisions about our Stack object. Will it be fixed sized or grow-able?

If we decide on a grow-able stack, then we don't need a precondition for **push(int)**. You will always be able to push another item on the stack (assuming we have enough memory).

If we decide on a fixed size stack, then we realize our interface is incomplete. **push(int)** should have precondition that says you can't call it if the stack is full. But how can the client tell if stack is full. We realize that our Stack class is incomplete and is missing some way of telling when a stack is full. If the client can't tell if the stack is full, its seems very unfair to make it a requirement of the contract as a precondition (did someone say "small print"?).

To fix this problem we will add a new query function called **capacity()** to our stack interface, and a requirement that **count() < capacity()** to **push(int)**. For clarity, let's add **isFull()** and **isEmpty()** as additional queries.

The Stack Interface

Here is the complete (fixed-size) stack interface in one place for your reference.

- **int count()** -- Return the number of items on the stack.
- **int capacity()** -- Return the maximum allowed number of items.
- **boolean isEmpty()** -- Is the stack empty?
  - Ensure: *Result == (count() == 0)*
- **boolean isFull()** -- Is the stack full?
  - Ensure: *Result == (count() == capacity())*
- **int top()** -- Return the top of the stack.
  - Require: *!isEmpty()*
- **void pop()** -- Remove the top of the stack.
  - Require: *!isEmpty()*
  - Ensure: *count() == old count() - 1*
- **void push(int n)** -- Push item *n* onto the stack.
  - Require: *!isFull()*
  - Ensure: *count() == old count() + 1*
  - Ensure: *top() == n*

DbC and Inheritance

We spend a lot of time in OO design creating interfaces and deriving new classes from those interfaces. There is a rule, called the Liskov Substitution Principle, that tells us we can substitute objects from our derived class where ever an object of the appropriate base class is expected, provided the derived class acts sufficiently like the base class so that the client software doesn't care about the switch.

The Liskov principle is very good; but, as stated above, it's a little vague. What does "acts sufficiently like the base class" mean exactly. It means that the derived class supports the *same contract* as the base class. Since it supports the same contract, the client software gets exactly what it expects.

Actually, its a little better that that. We are allowed to vary the contract in the derived class in interesting ways. First of all, we may loosen the preconditions. A class that requires less can certain be used with a client the provides more. Secondly, we can tighten up the postconditions. A class that provides stronger postconditions can certainly be used by a client that expects less. This can be summarized by the slogan:

> *Require no more, promise no less.*

Isn't This a Lot of Work?

I can tell you are thinking "All this Design by Contract stuff *sounds* really great, but it *looks* like a lot of work."

If you are going to spend any time at all thinking about the interface you are designing (and if you don't think about it, how will your interface be any good?), then the extra amount of time needed to formalized the contracts are very minimal. In fact, you may have noticed that in the process of formalizing our stack contract we discovered holes and omissions in our initial specification for a Stack. Taking the time to do DbC may actually save time in the long run.

Now, On To The Code ...

Up to this point in time, we have been talking about pure design ... figuring out what the interface should be and how clients will be using the interface. Just taking the time to write down and publish the contract for an object gives you great benefits. However, you can take it one step further and actually implement contract enforcement in your code. Consider this possible implementation for the **push(int)** method.

```
void push (int n) {
    // Preconditions ...
    if (isFull()) {
        throw new PreconditionViolation();
    }
    int oldCount = count();

    // Actual Code ...
    itsData[itsCount] = n;
    itsCount++;

    // Postconditions ...
    if (count() != oldCount+1) {
        throw new PostconditionViolation();
    }
    if (top() != n) {
```

```
            throw new PostconditionViolation();
        }
    }
```

This is kind of cool. Imagine using this version of **push()** in a real system. Anytime a client violates a contract by calling a method with inappropriate data, a precondition violation exception will be thrown. Anytime a method is improperly implemented a postcondition violation exception will be thrown. Now, its true that this won't catch all your software errors, but consistent use of executable pre and post conditions will make errors in your code *much* more evident and, in turn, easier to fix.

All this precondition and postcondition checking tends to obscure the actual logic of the code. Furthermore, while all this checking is good for debugging, the extra baggage may to too much of a burden for production code. We will address these two issues by first, transforming the code into the following.

```
    void push (int n) {
        Require.that (!isFull());
        int oldCount = count();

        itsData[itsCount] = n;
        itsCount++;

        Ensure.that (count() != oldCount+1);
        Ensure.that (top() != n);
    }
```

Ah, that's a little better. Its very clear what part of the code is the contract, and what part of the code actually implements the method. The Require and Ensure classes each have a static method called **that(boolean)** that takes a boolean argument. If the argument is not true, then an exception will be thrown (Require throws a Precondition exception, Ensure throws a Postcondition exception). There is also an overloaded version of **that(String,boolean)** that allows a descriptive name to be assigned to the assertion. The assertion name will appear in stack dumps caused by that exception. For example, our first precondition could be written ...

```
        Require.that ("not full", !isFull());
```

There is also a Check class that allows you to specify arbitrary assertions that are not part of the contract. Use it like this ...

```
        Check.that (isConsistent());
```

Disabling Assertions

But what about performance. Do we really want to perform all the checking all time? As it turns out, by using this form it becomes very easy to turn assertion checking on and off. Suppose we are done with testing and we are sure that our code is working (and obeying all contracts). We wish to produce a version of our code that will run without the safety net of assertion checking. We can execute the command ...

```
    jassert -n Stack.java
```

to turn off all assertion checking in the file Stack.java. Our **push()** method now looks like ...

```
    void push (int n) {
```

```
        //Require.that (!isFull());
        int oldCount = count();

        itsData[itsCount] = n;
        itsCount++;

        //Ensure.that (count() != oldCount+1);
        //Ensure.that (top() != n);
    }
```

Same code, but with the assertions commented out!

C.A.R. Hoare once said that using array bounds checking when debugging and turning it off on production code is like wearing a life jacket on the shore, but taking it off when you go to sea. Some people feel the same about contract assertions. Fortunately, there is some middle ground that turns out to be very useful.

If a postcondition is violated, then we can be sure that the problem is in the method implementing the contract. Once an object is well debugged, disabling postcondition assertions carries little risk.

On the other hand, if a precondition is violated, the problem is in the client code. Because we cannot control who uses our objects, leaving preconditions enabled seems to be a useful thing to do. This can be accomplished using jassert ...

```
jassert -r Stack.java
```

The *-r* option indicates that the require assertions should be enabled, but ensure and check assertions will be disabled.

By using jassert, you can carefully tune the amount of contract checking to meet your needs on a module by module basis.

Limitations

The simple contract assertions described above go a long way in helping to do design by contract in Java. However, there are some limitations to this approach.

1. *Contracts aren't inherited.* In a real DbC system, derived classes would automatically inherit the contract of the base class. In our system, we have to manually cut and paste the contract of the base class into the derived class.

2. *Interfaces can't have executable contracts.* Java interfaces are sterile declarations of method names and argument types. We cannot put executable contracts in the interface. Actually, because we don't inherit contracts in Java, this is more of an irritation than a real failing.

   I just leave the contract in a Java interface as a comment. Since I don't want jassert to uncomment the code, I leave the ".that" portion off of the assertion. For example, an interface that handled ages might look like:

```
    interface Age {
```

```
        int age();
        //Ensure: (Result >= 0);

        void setAge (int newAge);
        //Require: (newAge >= 0);
        //Ensure:  (age() == newAge);
    }
```

When I write a class that implements this interface, it is easy to cut and paste the contracts and add the ".that" string.

3. Contracts are part of the interface of a class and should appear in any printed documentation. In Java, this means that the javadoc output should include the preconditions and postconditions. Unfortunately, we have no mechanism in place to make that happen.

You may want to checkout iContract, which is a Java Design by Contract system from Reliable Systems that overcomes some of the limitations of this simple approach.

Software

The following software is available ...

- Complete source code for Require/Ensure/Check, related exceptions and the jassert perl script in the following formats.
  - tar'ed and gzip'ed
  - Jar File
- Example stack class with assertions.
- jassert man page.

NOTE: *The Java software is made available under the LGPL license.*

Things to Remember

Keep these things in mind when developing your contracts.

1. *Queries generally have no precondition.* It should be legal to ask for the capacity of a stack at any time. Exceptions would be queries on attributes that only exist when the object is in particular states. For example, asking for the file name of a file object might only be valid if the file is open.

2. *Don't include fine print.* In other words, don't require something that the client cannot determine. If the client can't tell if a precondition is satisfied, then it is impossible for the client to call the method safely. This also implies that all functions used in the preconditions should be publicly available.

   On the other hand, postconditions that guarantee private, internal state is OK.

3. *Use real code where possible.* The contract should be written in terms of actual code if possible. We expressed the precondition for **pop()** as "**!isEmpty()**" rather than "*the stack cannot be empty*". By expressing the conditions in terms of real code in our example we discovered that the interface was not complete and we had to add additional queries to support the precondition for **push(int)**. In addition, by

using real code we it makes it easy to make our contracts executable.

If it is impossible to express an assertion in real code, then use a comment. This at least informs the author of the client software about your intentions.

4. *The pre/post conditions don't completely specify the object.* If you examine the contract for our stack object very carefully, you will not find anything that specifies that the stack operates in a Last In/First Out manner. In fact, a queue class would fulfill almost the exact same contract. A full and complete semantic specification in the contract would require predicate calculus in the conditions and is a little beyond the goals of Design by Contract.

5. *Separate commands from queries.* Since the contract is executable code that can be turned on and off, we need to make sure that the class acts the same with contract checking on as it does with contract checking off. This generally means that queries used in pre and post conditions are side-effect free. Programmers in Eiffel, a language that supports Design by Contract from its inception, have developed a style they call "Command/Query Separation" (or CQS). In CQS, all functions are written in a side effect free manner. State is only changed when calling commands (i.e. **void** functions in Java). This means functions like **readLine()** would be avoided when using CQS. When CQS is followed rigorously, any function is allowed to be used in a contract.

   Notice that in our stack design, we have **top()** which is a pure, side effect free function and **pop()** which is a command and changes the state of the stack. This is an example of CQS.

   Because of the heavy influence of C and C++ on Java, I doubt I'll see CQS adopted in the near future. That means you have to be careful when design contracts so that you stay away from these "dangerous" side effect functions.

References

You can read more about Design by Contract at the following sites.

- Design by Contract Overview -- A paper on *Design by Contract* by Bertrand Meyer.

- *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall.
  OOSC is one of the best books on OO programming. In this book, Meyer develops his principles for good OO design (including Design by Contract). Although the notation used in the book is Eiffel, much of what he says can be applied to other OO languages.

- iContract -- A Design by Contract tool for Java. iContract puts pre- and post-conditions into the java-doc comment for each method. A preprocessor will insert the proper code into each method. Derived classes will properly inherit their parents contract. This is a close as it gets to full Eiffel-like Design by Contract in Java (without a language change).

If you are really interested in Design by Contract, I encourage you to look over the Eiffel language where the DbC concepts are integrated directly into the language. The GNU SmallEiffel compiler is a free (both no cost and freely available source code) compiler and is available on both Linux (unix) and Windows platforms.

[Jim's Java and OO Page]

Jim Weirich / jweirich@one.net
Visitors: