Programming by Contract (PBC) Class Utility

Programming/design by contract (PBC/DBC) pioneered by Bertrand Meyer (refer to references [2] and [3]) is widely acknowledged to be a powerful technique for writing reliable software.

PBC is based on the notion of a contract between a client and supplier. This contract is expressed as pre-conditions, post-conditions and invariants.

This article outlines an approach for enforcing contacts by using the Java assertion feature and a dedicated PBC class utility.

Robustness vs Correctness

Correctness is the ability of a system to perform its job according the specification. Correctness is of most concern during the design and development stages (this includes unit testing). Only programming errors within the system being developed result in the system being incorrect.

Robustness is the ability of a system to handle abnormal conditions. Java runtime exceptions can be used to implement fault tolerance boundaries.

PBC is concerned with correctness, not robustness.

Enforcing Contracts with Assertions

Many references indicate that assertions should not be used to check the validity of input parameters on public methods. For example, reference [1] provides the following argument:

Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException). An assertion failure will not throw an appropriate exception.

Referring to the second aspect to this argument, generally unchecked exceptions are used to document these preconditions (the example exceptions quoted above are all unchecked exceptions). This means that there is no guarantee that the caller will catch the exceptions. Moreover, experience has shown that exceptions such as these are rarely trapped. There are two reasons for this:

-
It is harder to write appropriate logic in the catch block than it is to check that the corresponding conditions are impossible to reach prior to making the call. Put another way, it is easier to handle the problem closest to where it occurs.

-
Adding try catch blocks throughout the code makes the code significantly harder to read.

The solution is to treat these conditions as a programming error. Methods should be documented to indicate the conditions required, however, they will NOT declare exceptions that can be used to trap these conditions. The onus is on the client to ensure the conditions are met prior to making the call.

This solution allows the Java assertion feature to be used since the semantics of the methods indicate the erroneous conditions, yet do not guarantee particular exceptions being raised. Using the Java assertion feature will allow these semantics to be met regardless of whether assertions are enabled or disabled.

Reference [4] provides a more detailed analysis which also justifies this approach

PBC Class Utility

A class utility can provide a variety of methods supporting the enforcement of contracts using assertions. For example:

```
/**
 * This method asserts that a specified object is not null.
 * @param object - the object that should not be null.
 * @param name - the name of the object to be used for error reporting.
 */
public final static boolean notNull(Object object, String name) {
 assert object != null: name + " must not be null";
 return true;
}

/**
```

```
 * This method asserts that an int value is within a specified range.
 * @param value - the value that must be within the specified range.
 * @param min - the minimum value specified of the specified range.
 * @param max - the maximum value of the specified range.
 * @param name - the name of the value to be used for error reporting.
 */
public final static boolean inRange(int value, int min, int max, String name) {
 assert value >= min && value
```