Using Design by Contract™ to Automate Java™ Software and Component Testing

ParaSoft®

Abstract

Design by Contract (DbC) is a beneficial software development practice that, until now, was too cumbersome to actually incorporate into most development processes. Jtest and Jcontract make it possible for you to realistically implement DbC. Jtest helps you add specification information into your code using DbC contracts. This is beneficial because it ensures that the code and specification are always kept together. From the DbC contracts, Jtest automatically creates test cases that verify class/component functionality. Because this relieves you from having to create black-box test cases, you save a tremendous amount of time and resources. When the classes/components that contain DbC contracts are added to the system, Jcontract automatically verifies whether the system uses them correctly. A misused class/component might appear to perform fine, but might actually spur subtle errors that can trigger system-wide problems.

1. Introduction

Design by Contract™ (DbC) is a formal way of using comments to incorporate specification information into the code itself. Basically, the code specification is expressed unambiguously using a formal language that describes the code's implicit contracts. These contracts specify such requirements as:

- Conditions that the client must meet before a method is invoked.
- Conditions that a method must meet after it executes.
- Assertions that a method must satisfy at specific points of its execution.

On its own, DbC can be an effective way to prevent errors throughout the development process. It can be even more powerful when used with ParaSoft's Jtest® and Jcontract™, two Java™ development tools that understand and take full advantage of the specification information recorded in the DbC contracts.

Jtest is a Java unit testing tool that tests any Java class or component; it completely automates black-box testing (functionality testing), white-box testing (construction testing), and regression testing, as well as static analysis. For more information on Jtest's general features, see our paper "Automatic Java Software and Component Testing: Using Jtest to Automate Unit Testing and Coding Standard Enforcement." Jtest has been extended to work with the DbC language: when used with classes or components that contain DbC contracts, Jtest automates black-box testing, a process that few people ever thought could be automated. Jtest reads the specification information built into a class, then automatically creates and executes test cases that check the functionality described in the specification. It also tailors its unit-level white-box test creation to the specifications contained in the contract.

Jcontract is a new Java development tool that checks DbC contracts at runtime; it can be run independently of Jtest, but the two tools are complementary. After you have used Jtest to thoroughly test your class or component at the unit level, use Jcontract to instrument and compile the DbC-commented code. Once a class or component is instrumented, Jcontract automatically checks whether its contracts are violated at runtime. Jcontract is particularly useful for determining whether an application misuses specific classes or components.

Together, these tools will help you improve the quality and speed of unit-level and system-level testing. This paper will introduce you to the DbC language, briefly explain how Jtest and Jcontract can leverage DbC information, then describe in detail how using DbC along with these tools can improve your development process.

1.1 About Design by Contract

DbC originated in Eiffel. Eiffel classes are components that cooperate through the use of the contract, which defines the obligations and benefits for each class. For an excellent introduction to DbC and a description of how it can be applied to Eiffel, see “Building bug-free O-O software: An introduction to Design by Contract™” at http://www.eiffel.com/doc/manuals/technology/contract/page.html

DbC is not yet commonly a part of programming languages such as C, C++, and Java, but ideally it should be. After all, any piece of code in any language has implicit contracts attached to it. The simplest example of an implicit contract is a method to which you are not supposed to pass `null`. If this contract is not met, a `NullPointerException` will occur. Another example is a component whose specification states that it only returns positive values. If it occasionally returns negative values and the consumer of this component is expecting the functionality described in the specification (only positive values returned), this contract violation could lead to a critical problem in the application.

Applying DbC to your code has significant benefits even before you start using Jtest and Jcontract. These benefits include:

- The code’s assumptions are clearly documented (for example, you assume that `item` should not be `null`). Design concepts are placed directly in the code itself.
- The code’s contracts can be checked for consistency because they are explicit.
- The code is much easier to reuse.
- The specification will never be lost.
- When you see the specification while writing the code, you are more likely to implement the specification correctly.
- When you see the specification while modifying code, you are much less likely to introduce errors.

Once you start using Jtest and Jcontract, the benefits of using DbC also include:

- Black-box test cases are created automatically. If you currently create your black-box test cases manually, this means fewer resources spent creating test cases and more resources you can dedicate to more complex tasks, such as design and coding. If you do not currently perform black-box testing, this will translate to more reliable software/components.
- Black-box test cases are automatically updated as the code’s specification changes.

- Class/component misuse is automatically detected.
- The class implementation can assume that input arguments satisfy the preconditions, so the implementation can be simpler and more efficient.
- The class client is guaranteed that the results will satisfy the postconditions.

How difficult is it to use the DbC language in Java development? Not very. There have been several efforts to make DbC available in Java. Most of the efforts involve using `Javadoc` comments to specify the contract's conditions. A simple example is:

```
public class ShoppingCart
{
    /**
     * @pre item != null
     * @post $result > 0
     */

    public float add (Item item) {
        _items.addElement (item);
        _totalCost += item.getPrice ();
        return _totalCost;
    }
    private float _totalCost = 0;
    private Vector _items = new Vector ();
}
```

The contract contains the following conditions:

1. A precondition (`@pre item != null`) which specifies that the item to be added to the shopping cart shouldn't be `null`.
2. A postcondition (`@post $result > 0`) which specifies that the value returned by the method is always greater than `0`.

Preconditions and postconditions can be thought of as sophisticated assertions. Preconditions are conditions that the method's client needs to satisfy before the method can execute; a violation of a precondition indicates a problem with the client (the client is misusing the method). Postconditions are conditions that the implementor of the class guarantees will always be satisfied after a method completes; a postcondition violation indicates a problem within the method.

Other contract elements that Jtest and Jcontract understand and use include:

Element	**Purpose**
`@invariant`	Conditions (similar to postconditions) that apply to all of the methods in the class. An invariant violation indicates a problem with the class's implementation.
`@assert`	Boolean expressions about the state of the software. Each `@assert` expression is executed at the point in the program where the `@assert` tag is located. An assertion violation indicates a problem within the method.
`@exception`	Tags used to indicate that the code is expected to throw a certain exception.
`@concurrency`	Tags used to specify the concurrency mode in which the method can be called.
`@verbose`	Tags that allow you to add verbose statements to the code.

For more information about DbC see:

- Geoff Eldridge, "Java and 'Design by Contract'" (http://www.elj.com/eiffel/feature/dbc/java/ge/)
- Adam Kolawa, "Automating the Development Process" *Software Development*, July 2000 (http://www.sdmagazine.com)

1.2 About Jtest

Jtest is a Java unit testing and static analysis tool; its general purpose is to help you ensure that any class or component is solid and correct before it is integrated into an application. Jtest's DbC-related functionality (just one part of Jtest's total functionality) helps you create a DbC contract, then automatically verifies if the class or component under test is built according to the specification incorporated into the contract. In other words, it completely automates the black-box (functionality) testing process. This DbC-specific functionality is described below.

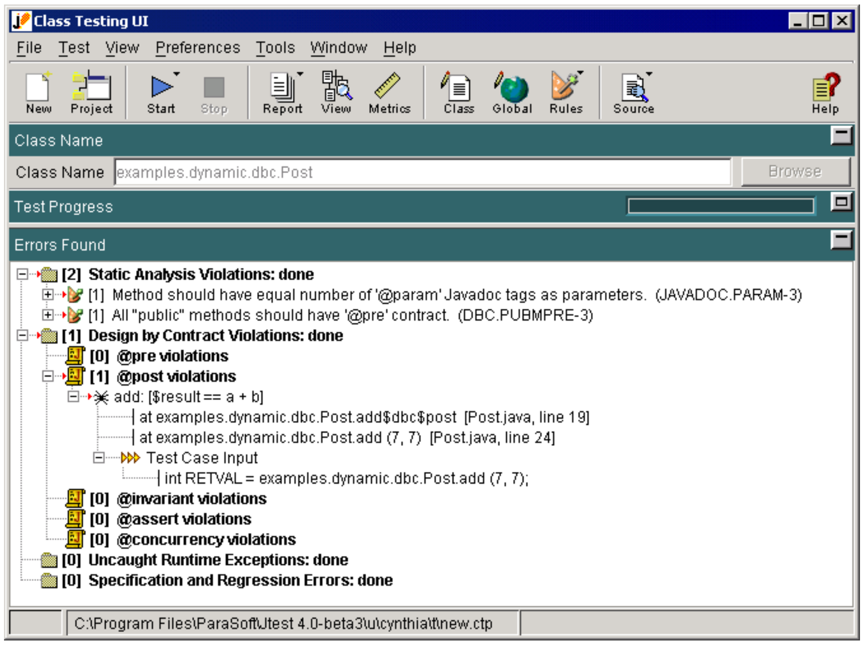
As you create code, you incorporate specification information into it using the DbC language. Next, you compile your class as normal, then tell Jtest to test it. With the click of a button, Jtest performs the following DbC-related tasks in addition to its normal white-box (construction) testing, black-box testing, regression testing, and static analysis tasks:

1. Instruments the code's specification information and recompiles the class with extra bytecodes that describe how the class is supposed to work and be used.
2. Checks if the class is missing any necessary DbC comment tags.
3. Examines the specification information contained in the contract, then creates and executes test cases that test whether the class functions as specified.

4. Suppresses any problems found for inputs that violate the preconditions of the class under test.
5. Suppresses any exception that is documented in the contract.
6. Reports problems found in its UI.

For a more detailed description of this functionality, see “Unit Testing” on page 7.

Once all of the problems that Jtest uncovered (functionality problems, construction problems, and static analysis violations) are repaired, the class is ready to be integrated into the system. At that point, Jcontract can be used to monitor whether the class’s contracts are met at runtime.


Class Testing UI
File Test View Preferences Tools Window Help
New Project Start Stop Report View Metrics Class Global Rules Source Help
Class Name
Browse
Test Progress
Errors Found

Class Name examples.dynamic.dbc.Post

```
[2] Static Analysis Violations: done
    [1] Method should have equal number of '@param' Javadoc tags as parameters. (JAVADOC.PARAM-3)
    [1] All "public" methods should have '@pre' contract. (DBC.PUBMPRE-3)
[1] Design by Contract Violations: done
    [0] @pre violations
    [1] @post violations
        add: [$result == a + b]
            | at examples.dynamic.dbc.Post.add$dbc$post [Post.java, line 19]
            | at examples.dynamic.dbc.Post.add (7, 7) [Post.java, line 24]
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Post.add (7, 7);
    [0] @invariant violations
    [0] @assert violations
    [0] @concurrency violations
[0] Uncaught Runtime Exceptions: done
[0] Specification and Regression Errors: done
```

C:\Program Files\ParaSoft\Jtest 4.0-beta3\u\cynthia\t\new.ctp

Figure 1: Jtest UI

1.3 About Jcontract

Jcontract is used to check a class or component’s contracts at runtime, after you have used Jtest to verify that the class or component is solid and correct. It is complementary to Jtest, but the respective tools can be used independently of one another.

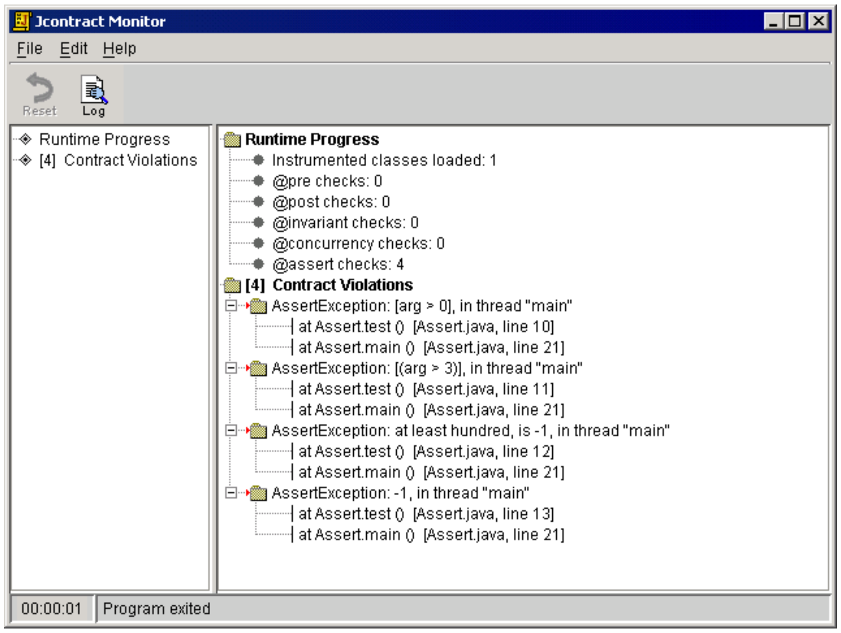
When you have thoroughly tested a class or component and are ready to integrate it into the system, recompile it with Jcontract by simply calling Jcontract’s `dbc_javac` compiler instead of `javac`. For example, to instrument and compile `Example.java`, you would enter the following command:

```
dbc_javac Example.java
```

Jcontract then instruments the code’s specification information and recompiles the class with extra bytecodes that describe how the class is supposed to work and be used.

Next, integrate the instrumented class into the system, and run the system. Jcontract watches the system as it runs, and like an X-ray machine, automatically detects contract violations. For example, if a component’s DbC specification said that a particular method required positive integer inputs, Jcontract would report a violation if the system passed that method any negative inputs.

By default, contract violations found are reported in the Jcontract monitor. This monitor displays the nature of each violation as well as stack trace information.



```
Runtime Progress
    Instrumented classes loaded: 1
    @pre checks: 0
    @post checks: 0
    @invariant checks: 0
    @concurrency checks: 0
    @assert checks: 4
[4] Contract Violations
    AssertException: [arg > 0], in thread "main"
        at Assert.test () [Assert.java, line 10]
        at Assert.main () [Assert.java, line 21]
    AssertException: [(arg > 3)], in thread "main"
        at Assert.test () [Assert.java, line 11]
        at Assert.main () [Assert.java, line 21]
    AssertException: at least hundred, is -1, in thread "main"
        at Assert.test () [Assert.java, line 12]
        at Assert.main () [Assert.java, line 21]
    AssertException: -1, in thread "main"
        at Assert.test () [Assert.java, line 13]
        at Assert.main () [Assert.java, line 21]
```

```
00:00:01    Program exited
```

Figure 2: Jcontract Monitor

Jcontract’s degree of program interference is completely customizable. By default, Jcontract uses a non-intrusive runtime handler that reports violations found, but does not alter program execution. You can also choose a runtime handler that throws an exception when a violation occurs, choose a runtime handler that logs violations in a file, or create a customized runtime handler that is specially tailored to your needs.

Jcontract also adapts to your needs by letting you select which contract conditions you want it to instrument. This way, you can optimize program performance by having Jcontract focus on the conditions that are most important at your current stage of the development process. For example, after a well-tested class is integrated into an application, you might only want to instrument and check preconditions that verify whether the application uses the class correctly.

For a more detailed description of Jcontract's functionality, see "System-Level Testing" on page 12.

2. Improving the Quality and Speed of Testing

At this point, we have given you a brief overview of the DbC language and how Jtest and Jcontract leverage the information provided in DbC-format contracts. The remainder of this paper is dedicated to describing in detail how Jtest and Jcontract use DbC information to make the entire testing phase faster and more efficient.

2.1 Unit Testing

Often, developers hear about unit testing and think the term refers to module testing. In other words, developers think they are performing unit testing when they take a module, or a sub-program that is part of a larger application, and test it. Module testing is important and should certainly be performed, but it is not the technique that we want to concentrate on here. When we use the term "unit testing," we are talking about testing the smallest possible unit of an application; in terms of Java, unit testing involves testing a class as soon as it is compiled.

Jtest automates all steps of the otherwise time-consuming unit testing process: it automatically creates a harness and any necessary stubs for the class under test, then performs white-box testing, black-box testing, regression testing, and static analysis. This section focuses on how Jtest uses DbC contracts in the unit testing process.

2.1.1 Adding Comments

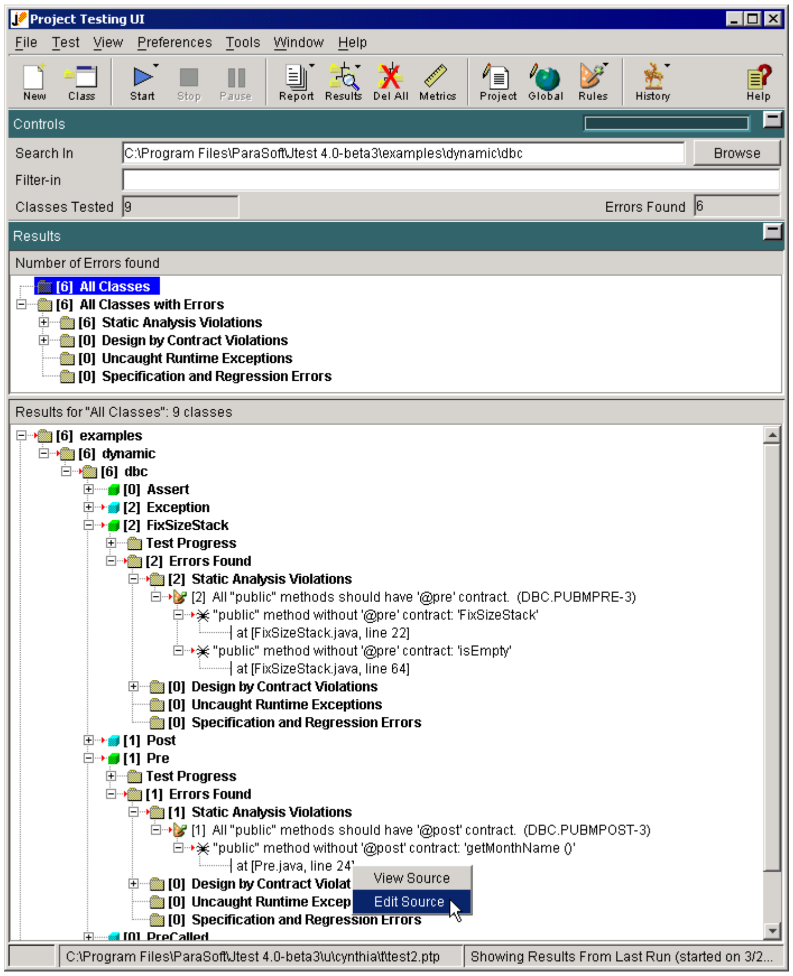
Before your tools can use DbC information, someone needs to add the contract to the code. That's why Jtest uses static analysis to guide you through the process of adding DbC comments. When you let Jtest statically analyze the `.java` file you are working on, Jtest applies a special set of coding standards that determine if any critical DbC comments are missing; when Jtest finds a missing comment, it reports what type of comment is missing and where each comment should be added.

The coding standards that Jtest applies include:

- All "protected" classes should have an `@invariant` contract.
- All "protected" methods should have an `@post` contract.
- All "protected" methods should have an `@pre` contract.
- All "public" classes should have an `@invariant` contract.
- All "public" methods should have an `@post` contract.
- All "public" methods should have an `@pre` contract.
- All package-private classes should have an `@invariant` contract.
- All package-private methods should have an `@post Javadoc` tag.

- All package-private methods should have an `@pre Javadoc` tag.
- All “private” classes should have an `@invariant` contract.
- All “private” methods should have an `@post` contract.
- All “private” methods should have an `@pre` contract.

For example, when Jtest finds a public method without an `@post` condition, it reports a static analysis violation message that tells you that an `@post` condition should be added, and at what line it should be added. Such a violation is shown below in Figure 3.



Search In C:\Program Files\ParaSoft\Jtest 4.0-beta3\examples\dynamic\dbc
Filter-in
Classes Tested 9 Errors Found 6

Number of Errors found

```
[6] All Classes
[6] All Classes with Errors
  [6] Static Analysis Violations
  [0] Design by Contract Violations
  [0] Uncaught Runtime Exceptions
  [0] Specification and Regression Errors
```

Results for "All Classes": 9 classes

```
[6] examples
  [6] dynamic
    [6] dbc
      [0] Assert
      [2] Exception
      [2] FixSizeStack
        Test Progress
        [2] Errors Found
          [2] Static Analysis Violations
            [2] All "public" methods should have '@pre' contract.  (DBC.PUBMPRE-3)
              "public" method without '@pre' contract: 'FixSizeStack'
                at [FixSizeStack.java, line 22]
              "public" method without '@pre' contract: 'isEmpty'
                at [FixSizeStack.java, line 64]
          [0] Design by Contract Violations
          [0] Uncaught Runtime Exceptions
          [0] Specification and Regression Errors
      [1] Post
      [1] Pre
        Test Progress
        [1] Errors Found
          [1] Static Analysis Violations
            [1] All "public" methods should have '@post' contract.  (DBC.PUBMPOST-3)
              "public" method without '@post' contract: 'getMonthName ()'
                at [Pre.java, line 24]
          [0] Design by Contract Violat
          [0] Uncaught Runtime Excep
          [0] Specification and Regression Errors
      [0] PreCalled
```

C:\Program Files\ParaSoft\Jtest 4.0-beta3\u\cynthia\ttest2.ptp | Showing Results From Last Run (started on 3/2...

Figure 3: Jtest helps you add DbC contracts

By automatically checking contracts as part of static analysis, Jtest makes it easier to implement DbC as a team-wide or company-wide coding standard.

2.1.2 Automating Unit-Level Black-Box (Functionality) Testing

As we said earlier, Jtest uses the DbC specification information to automatically create and execute test cases that verify whether a class or component functions as its DbC specification says it should function. When you test a class or component with Jtest, Jtest automatically creates and executes black-box test cases as part of its normal battery of tests. Jtest designs its black-box test cases as follows:

- If the code has postconditions, Jtest creates test cases that verify whether the code satisfies those conditions.
- If the code has assert conditions, Jtest creates test cases that try to make the assertions fail.
- If the code has invariant conditions, Jtest creates test cases that try to make the invariant conditions fail.
- If the code has preconditions, Jtest tries to find inputs that force all of the paths in the preconditions.
- If the method under test calls other methods that have specified preconditions, Jtest determines if the method under test can pass non-permissible values to the other methods.

For example, the following code contains a postcondition that describes part of the method's specification:

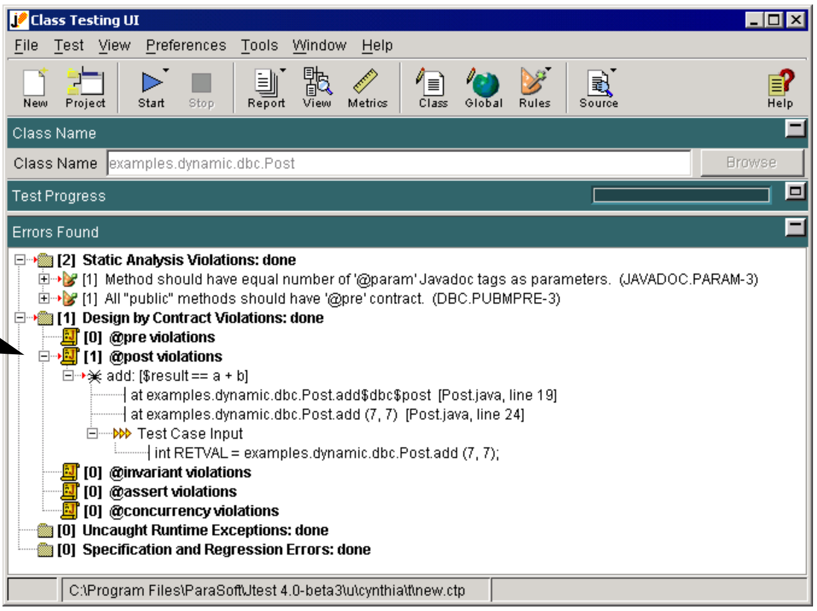
```
package examples.dynamic.dbc;

class Post
{
    /** @post $result == a + b */

    public static int add (int a, int b)
    {
        return a - b;  //BUG: note it should be '+' not '-'
    }
}
```

According to the specification, the method should return `a+b`. However, it actually returns `a-b`.

When we test this class, Jtest instruments the comments, compiles the class, analyzes the specification information, then creates and executes test cases that check whether its functionality is implemented correctly

.

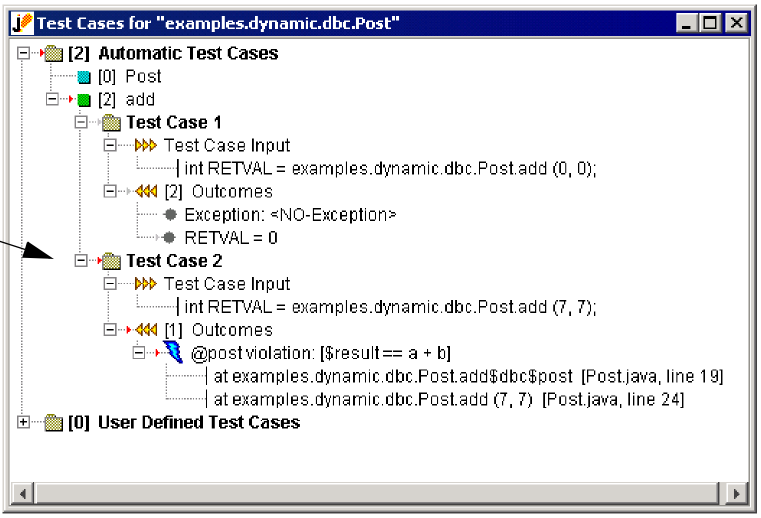
Failed postcondition


Class Testing UI
File Test View Preferences Tools Window Help
New Project Start Stop Report View Metrics Class Global Rules Source Help
Class Name
Browse
Test Progress
Errors Found

```
Class Name  examples.dynamic.dbc.Post

[2] Static Analysis Violations: done
    [1] Method should have equal number of '@param' Javadoc tags as parameters.  (JAVADOC.PARAM-3)
    [1] All "public" methods should have '@pre' contract.  (DBC.PUBMPRE-3)
[1] Design by Contract Violations: done
    [0] @pre violations
    [1] @post violations
        add: [$result == a + b]
            | at examples.dynamic.dbc.Post.add$dbc$post [Post.java, line 19]
            | at examples.dynamic.dbc.Post.add (7, 7) [Post.java, line 24]
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Post.add (7, 7);
    [0] @invariant violations
    [0] @assert violations
    [0] @concurrency violations
[0] Uncaught Runtime Exceptions: done
[0] Specification and Regression Errors: done

C:\Program Files\ParaSoft\Jtest 4.0-beta3\u\cynthia\t\new.ctp
```

Figure 4: Jtest automatically finds a functionality problem

Test case that exposes failed postcondition



```
[2] Automatic Test Cases
    [0] Post
    [2] add
        Test Case 1
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Post.add (0, 0);
            [2] Outcomes
                Exception: <NO-Exception>
                RETVAL = 0
        Test Case 2
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Post.add (7, 7);
            [1] Outcomes
                @post violation: [$result == a + b]
                    | at examples.dynamic.dbc.Post.add$dbc$post [Post.java, line 19]
                    | at examples.dynamic.dbc.Post.add (7, 7) [Post.java, line 24]
[0] User Defined Test Cases
```

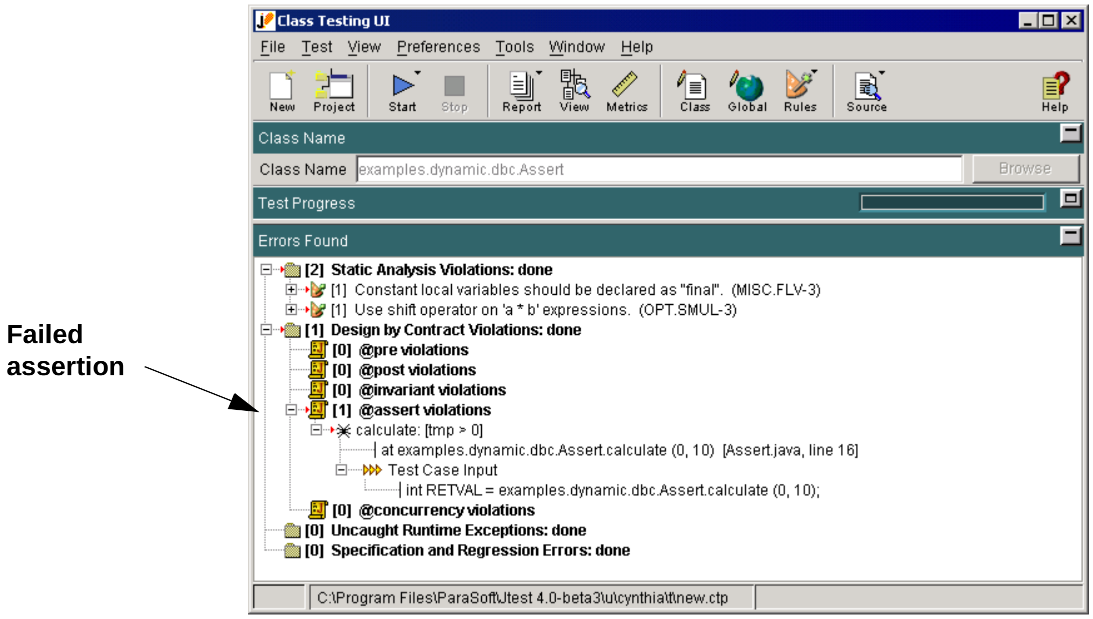
Figure 5: Jtest automatically designs test cases that verify functionality

The test results reveal that the functionality is not implemented correctly. Jtest's Test Cases window displays selected test cases that were created automatically (only the test cases that do something new [e.g., increase coverage, throw a new exception, etc.] are shown). Test Case 1 shows that the method functions as specified when the value of `0` is assigned to both `a` and `b`. However, Test Case 2 uncovers values for `a` and `b` that violate the method's postcondition contract. When the value of `7` is assigned to both `a` and `b`, the method's functionality flaw (it subtracts `b` from `a` rather than add the two values) is exposed. If the method was working correctly, Jtest would not be able to find a test case that violated the postcondition.

For a second example, let's look at a class that has a simple assertion.

```
package examples.dynamic.dbc;

public class Assert
{
    public static int calculate (int size1, int size2)
    {
        int tmp = size1 * size2 - 10;

        /** @assert tmp > 0 */

        return tmp * 2;
    }
}
```

When Jtest tests this class, it tries to create test cases that make the assertion fail.

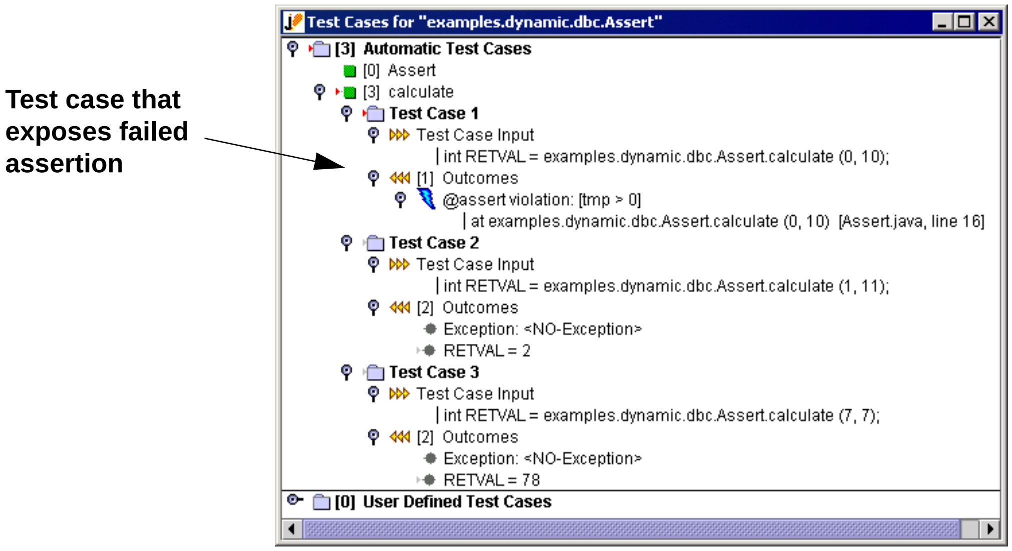

Failed assertion
Class Testing UI
File Test View Preferences Tools Window Help
New Project Start Stop Report View Metrics Class Global Rules Source Help
Class Name
Test Progress
Errors Found
Browse

Class Name: examples.dynamic.dbc.Assert

```
[2] Static Analysis Violations: done
    [1] Constant local variables should be declared as "final". (MISC.FLV-3)
    [1] Use shift operator on 'a * b' expressions. (OPT.SMUL-3)
[1] Design by Contract Violations: done
    [0] @pre violations
    [0] @post violations
    [0] @invariant violations
    [1] @assert violations
        calculate: [tmp > 0]
            | at examples.dynamic.dbc.Assert.calculate (0, 10) [Assert.java, line 16]
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Assert.calculate (0, 10);
    [0] @concurrency violations
[0] Uncaught Runtime Exceptions: done
[0] Specification and Regression Errors: done
```

C:\Program Files\ParaSoft\Jtest 4.0-beta3\u\cynthia\t\new.ctp

Figure 6: Jtest automatically finds a failed assertion



Test Cases for "examples.dynamic.dbc.Assert"

```
[3] Automatic Test Cases
    [0] Assert
    [3] calculate
        Test Case 1
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Assert.calculate (0, 10);
            [1] Outcomes
                @assert violation: [tmp > 0]
                    | at examples.dynamic.dbc.Assert.calculate (0, 10)  [Assert.java, line 16]
        Test Case 2
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Assert.calculate (1, 11);
            [2] Outcomes
                Exception: <NO-Exception>
                RETVAL = 2
        Test Case 3
            Test Case Input
                | int RETVAL = examples.dynamic.dbc.Assert.calculate (7, 7);
            [2] Outcomes
                Exception: <NO-Exception>
                RETVAL = 78
[0] User Defined Test Cases
```

Figure 7: Jtest automatically creates test cases that make the assertion fail

Once again, Jtest automatically creates test cases that test the specified functionality and expose functionality problems. After the code is repaired, these same test cases can be replayed to determine whether or not the modifications actually repaired the problem.

2.1.3 Optimizing Unit-Level White-Box (Construction) Testing

DbC can also help you optimize your unit-level white-box (construction) testing with Jtest. You can use DbC comments to filter out error messages that are not relevant to the class under test.

If you document expected exceptions in the code using the `@exception` tag, Jtest will suppress any occurrence of that particular exception.

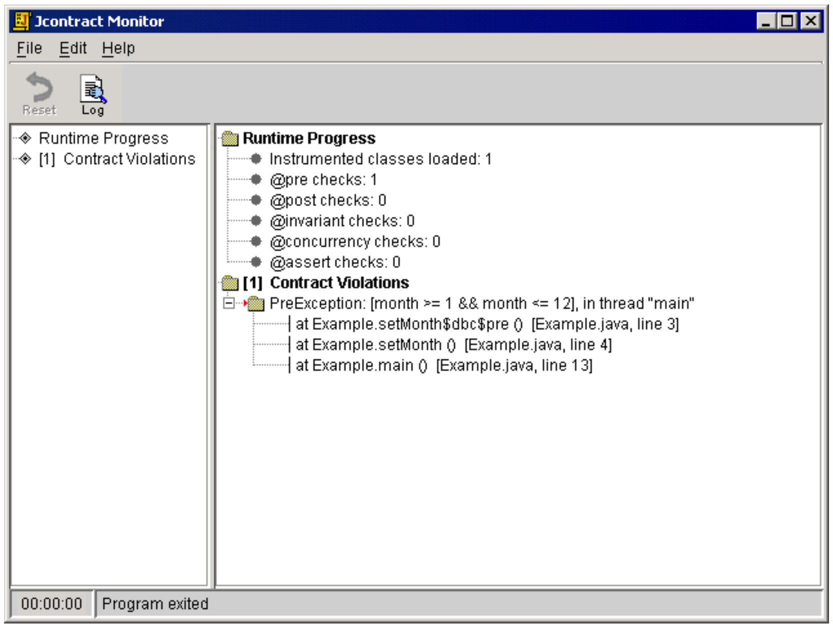
If you document the permissible range for valid method inputs using the `@pre` tag, Jtest will suppress any errors found for inputs that do not satisfy those preconditions. In addition, after you integrate the instrumented class or component into the application, Jcontract will watch the system execute and alert you if the system passes the class/component any inputs that the `@pre` tags describe as being “not permissible” (for more information on this functionality, see “System-Level Testing” below).

2.2 System-Level Testing

System-level testing can uncover different types of errors than unit testing: it can expose instances where the system misuses a class or component as well as instances where complex system interactions cause problems that were not apparent at the unit-level.

After you are confident that your class or component works correctly, recompile it with Jcontract's `dbc_javac` compiler, then integrate it into the system where it will be used. Next, run the system using your normal test suite. For example, if the instrumented component is part of a Web application, you would want to test all facets of the application— including its interactions with databases and other business logic components— to ensure that a wide range of possible interactions and uses are tested. At runtime, Jcontract checks two main things:

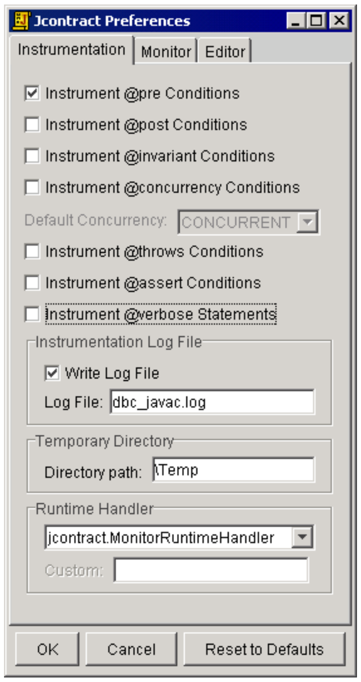
- Does the rest of the system use the instrumented class correctly? (Do other parts of the system pass the instrumented class inputs that violate the requirements specified in the preconditions?)
- Do the instrumented class's interactions with the rest of the system lead to functionality problems that could not be exposed at the unit level? (Does a certain chain of reactions cause an assertion to fail, or a method to return a value that violates its postcondition?)



```
Runtime Progress
    Instrumented classes loaded: 1
    @pre checks: 1
    @post checks: 0
    @invariant checks: 0
    @concurrency checks: 0
    @assert checks: 0
[1] Contract Violations
    PreException: [month >= 1 && month <= 12], in thread "main"
        | at Example.setMonth$dbc$pre ()  [Example.java, line 3]
        | at Example.setMonth ()  [Example.java, line 4]
        | at Example.main ()  [Example.java, line 13]
00:00:00    Program exited
```

Figure 8: Jcontract checks contracts at runtime

If performance speed is an issue, you can easily configure Jcontract so that it only instruments the contract types that you are most concerned with. This makes it easy for you to get the precise checking that you need without sacrificing performance. For example, say that you have thoroughly tested a performance-critical class at the unit level and are fairly certain that it functions correctly. You want to determine if it is used correctly within the system, but you do not want the performance hit that might arise from checking all of the class's postcondition and assertion contracts. Your best solution would be to have Jcontract instrument only the preconditions when it recompiled the class; this way, you would gain the precondition checking without a significant impact on that critical class's performance.



- [✓] Instrument @pre Conditions
- [ ] Instrument @post Conditions
- [ ] Instrument @invariant Conditions
- [ ] Instrument @concurrency Conditions

Default Concurrency: CONCURRENT

- [ ] Instrument @throws Conditions
- [ ] Instrument @assert Conditions
- [ ] Instrument @verbose Statements

Instrumentation Log File

- [✓] Write Log File

Log File: dbc_javac.log

Temporary Directory

Directory path: \Temp

Runtime Handler

jcontract.MonitorRuntimeHandler

Custom:

Figure 9: Jcontract preferences are easily customizable

3. Conclusion

Design by Contract is a beneficial software development practice, but until now, the time required to implement it prevented the majority of developers from incorporating it into their development process. Jtest and Jcontract make it possible to realistically implement DbC and reap the benefits that DbC has to offer. When you use DbC along with Jtest and Jcontract:

- Jtest's static analysis feature helps you incorporate specification information into the code. When formalized specification information is included in the code, you (and other developers) will be able to easily locate the specification and understand what the code is supposed to do.
- Jtest automatically creates and executes test cases that check class/component functionality. Because this relieves you from having to create black-box test cases, this saves you a tremendous amount of time and resources.
- Jtest and Jcontract automatically alert you when methods are misused. A misused method might appear to perform fine, but might actually spur subtle errors that can trigger problems throughout the application it is used in.
- Jtest and Jcontract facilitate component exchange and software reuse. Jtest helps the component's producer incorporate specification information into the component and verify that the component performs according to specification. Jtest helps the component's consumer immediately test if the component works as the producer claims it does, then

Jcontract helps him determine if his system is using the component as it is designed to be used.

With these tools, a small upfront investment in writing specifications (which should be written anyway) allows you to produce more reliable and reusable software in less time than ever before.

4. References

Interactive Software Engineering, “Building Bug-Free O-O Software: An Introduction to Design by Contract™.” http://www.eiffel.com/doc/manuals/technology/contract/page.html

Eldridge, G. “Java and ‘Design by Contract.’” http://www.elj.com/eiffel/feature/dbc/java/ge/

Kolawa, A., “Automating the Development Process.” *Software Development*, July 2000.

ParaSoft Corporation, “Automatic Java Software and Component Testing: Using Jtest to Automate Unit Testing and Coding Standard Enforcement.” http://www.parasoft.com/jtest/papers/jtestwp_4.htm

5. Availability

To learn more about how Jtest, Jcontract, and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041, or visit www.parasoft.com.

6. Contacting ParaSoft

USA

2031 S. Myrtle Ave.
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626)305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 1 64 89 26 00
UK: Tel: +44 171 288 66 00
Germany: Tel: +49 (0) 78 05 95 69 60
Email: info-europe@parasoft.com

Last Updated 3/28/01