JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Enhancing Design by Contract with Knowledge about Equivalence Partitions

Per Madsen, Department of Computer Science, Aalborg University, Denmark

Software testing seems to be a huge struggle for most developers. This is presumably because of the amount of manual work involved in software testing. In this article we propose a testing approach that makes the testing process less manual. The approach combines three well-known concepts *Design by Contract*, *Unit Testing* and *Equivalence Partitioning* into a new approach named *Testing by Contract*. The first step is to define a new programming language that supports an extended form of Design by Contract. The idea is to enhance the contracts with knowledge of Equivalence Partitions. The second step is to design a tool that exploits the enhanced contracts to build quality test cases automatically. The test cases are built as JUnit test cases, which allow them to be combined with manually written test cases. A prototype Java-like language with support for this extended form of Design by Contract has been developed. A compiler to regular Java-code as well as a tool for automatic generation of test cases is currently being built. This prototype language/tool will be used to further examine and evaluate the ideas described above.

1 INTRODUCTION

The process of doing proper software testing seems to be a huge struggle for software developers. Even though a lot of time and money is spent on testing we are still experiencing problems with all kinds of software ranging from software for small embedded systems to software for large application servers. Somehow it seems like developers have a negative attitude against software testing. One of the reasons for this is that software testing often is connected with a great deal of manual work for the developer. This article will focus on how to make the testing process less manual.

Over the years a lot of different testing techniques has been suggested. On the conceptual level testing seems to be a well-understood topic. We are able to talk about well-established concepts like white- and black-box testing techniques as well as unit testing and integration testing, but on the operational level it seems like we are missing something. How the individual developer should handle the concrete testing challenges is not always clear.

Take the idea of Equivalence Partitions (Equivalence Classes) [3] as an example. The concept that we can split both input and output of a function into a number of Equivalence Partitions and then just test one or a few representatives from each partition is well-accepted. But how to do this on the concrete operational level is

Cite this article as follows: Per Madsen: "Enhancing Design by Contract with Knowledge about Equivalence Partitions", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pages 5–21, http://www.jot.fm/issues/issue_2004_4/article1

an open question for many developers. There is a large gab between understanding the key idea and actually applying it to a concrete program.

Extreme Programming and Unit Testing

Extreme Programming [1] has addressed the problem described above in a number of ways. First of all Extreme Programming builds upon the idea of a test-driven development process, where testing is an integrated part of the software development process. Testing in this context is not just something you do at the end of the development process, but something that has to be carefully considered from the start.

Secondly Extreme Programming is based on the idea of using tools to make the testing process less manual. JUnit [2] is a regression testing framework for Java. It allows the programmer to write test cases as regular Java programs and then having a framework to take care of the test execution.

Frameworks like JUnit gives some tool-support for software testing, but still the programmer has to write the actual test cases manually. This leaves the programmer with two non-trivial tasks:

- Selecting the scenarios for the actual test cases. In which context should each method be tested? When do we have enough test cases? These are questions left for the programmer to answer.
- Finding a way to evaluate whether a test case succeeds or fails. The automation in JUnit is based on the assertions that the programmer specifies manually. The quality of a test case will never be better than the quality of the assertions that is put into the test case.

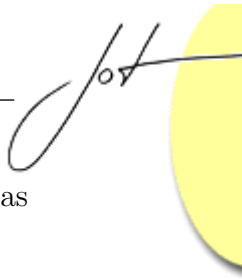
Even though Extreme Programming and JUnit provide us with some automation there is still a lot of manual work to be done by the programmer.

Design By Contract

Bertrand Meyer introduced the idea of Design by Contract [11]. The main idea is to use a contract to describe the responsibilities among the classes in an object oriented program.

The contract is specified using assertions directly in the program. Design by Contract includes three main types of assertions:

- **Precondition**: Describes the conditions that must hold before a method can be called. The caller is responsible for these conditions.



- **Postcondition**: Describes the conditions that must hold after a method has been called. The callee is responsible for these conditions.
- **Class Invariant**: Describes the conditions that should always hold for objects of a given class. All methods should preserve the class invariant i.e. the callee is responsible for these conditions.

Design by Contract might not be considered as test technique at first, but Design by Contract and software testing share the goal of ensuring software quality and robustness.

Design by Contract can be thought of as a built-in test case evaluations mechanism. If a number of test cases are executed and a violation of an assertion is spotted during the execution a potential bug has been detected. Another argument for considering Design by Contract when dealing with testing is that Design by Contract in some sense can limit the number of test cases we need to execute. Whenever a precondition is specified the number of needed test cases drops because it is not necessary to make test cases for situations where the precondition is not satisfied.

The rest of this paper is structured in the following way. In section 2 we introduce the concept of Testing by Contract. Section 3 describes how Design By Contract can be enhanced with knowledge about Equivalence Partitions and section 4 describes the concrete design of a prototype programming language. In section 5 the concept of partition coverage is discussed. Section 6 discusses how a tool can exploit the enhanced contracts for generating test cases. Section 7 gives an overview of related work and section 8 summarizes the perspectives of the ideas suggested in this paper. Finally section 9 describes the future work planned in this area.

2 TESTING BY CONTRACT

In this paper we propose a testing approach where the ideas of JUnit and Design by Contract are combined. The goal is to achieve a more automated approach by using the automatic test case execution feature of JUnit and the automatic test case evaluation feature of Design by Contract. We have named this combination "Testing by Contract" [7]. If we can achieve automated test case execution and evaluation the remaining hurdle is to find a way to generate the actual test cases. In theory we should generate a test case for every combination of possible sequences of method calls with every possible combination of parameters for each class we would like to test. As this is of course not a realistic approach, the real challenge here is to find the appropriate subset of all these possible infinite number of test cases.

A number of different approaches for finding the right subset of test cases could be imagined. In the rest of this paper we will investigate the idea of using information about Equivalence Partitions for selecting test cases.

The general concept of Equivalence Partitions is that we divide the unit under test into a number of partitions, based on the criterion that testing a single member

of a partition should be as good as testing all members of a partition. This is to say that if a test case using one member of a partition fails, all test cases using members of this partition should fail, and similar if a test case using one member of a partition succeeds, all test cases using members of this partition should succeed. The success of this approach is obviously very close connected to the quality of the partitions specified and it is therefore hard to imagine a tool that could automatically find the appropriate partitions.

It is our hypothesis that a programmer is more or less aware of Equivalence Partitions when doing the design and implementation of a program. If this knowledge can be expressed as a concrete part of the program, a testing tool can use this knowledge.

On one hand it can by argued that a programmer is aware of Equivalence Partitions when creating the program. Instead of throwing this information away and then have to re-discovering it in the testing phase; why not keep the information as part of the contract specification. On the other hand, if the programmer is not aware of Equivalence Partitions encouraging him to start thinking about Equivalence Partitions will raise the quality of the program.

3 ENHANCING DESIGN BY CONTRACT

In this section we discuss how the concept of Design by Contract can be enhanced with knowledge about Equivalence Partitions.

Regular Design By Contract

In regular Design by Contract the class invariant is described as a boolean expression. For very simple classes the invariant can be just a simple expression concerning one instance variable (e.g. $a > 0$). For more complex classes the invariant is often described as a conjunction of sub-invariants. Each sub-invariant can then deal with some particular aspect of the class.

Figure 1 shows the idea of regular Design by Contract invariants in terms of the set of all possible objects of a given class. The overall class invariant is given by:

$inv = inv_1$ and inv_2 and $inv_3 \ldots$ and inv_n, where $n =$ number of sub-invariants

Equivalence Partitions

We can sketch the concept of Equivalence Partitions in a similar way. Figure 2 shows the idea of Equivalence Partitions in terms of the set of all possible objects of a given class.

The two figures show some similarities between a class invariant and a partition.

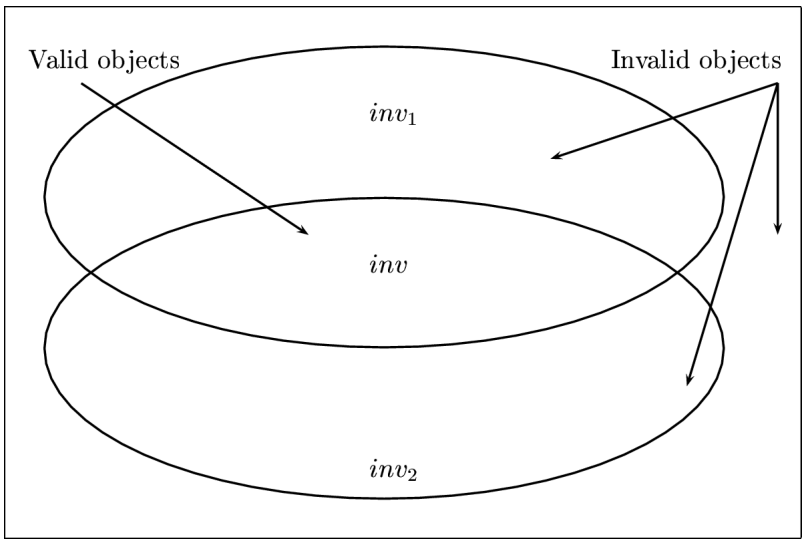


Figure 1: From the set of all possible objects the invariant describes the valid objects in terms of the conjunction of a number of sub-invariants. In this example the class has two sub-invariants (inv_1 and inv_2)

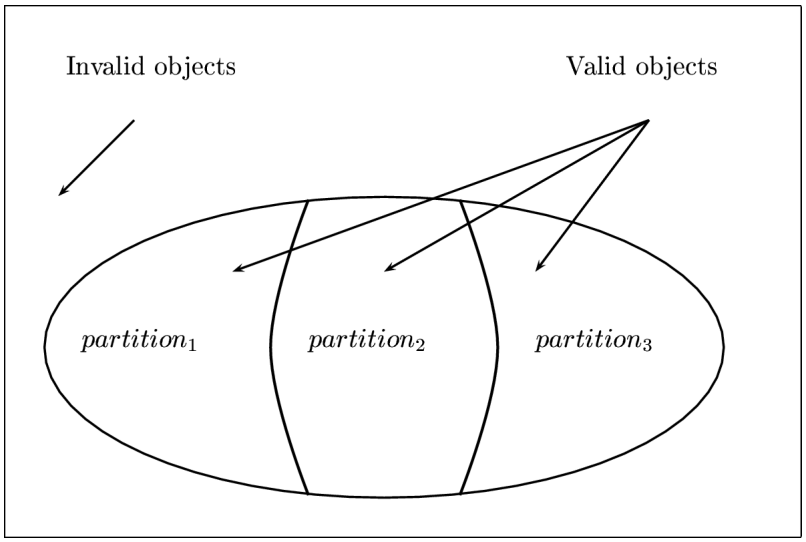


Figure 2: A class with three partitions. The set of valid objects is divided into three partitions $partition_1$, $partition_2$ and $partition_3$.

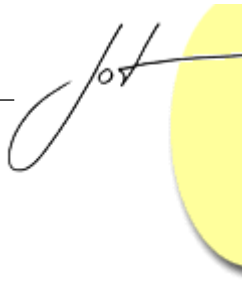
If the Equivalence Partitions can be expressed in terms of invariants we will have an easy way to formalize the knowledge of Equivalence Partitions and express this knowledge in a concrete way in a programming language. Figure 3 shows a part of the implementation of some container in a Java-like language with support for Design by Contract. Figure 4 shows the same example extended with knowledge of Equivalence Partitions. In this example the class has three partitions: empty, oneElement and severalElements. The reasoning behind this partitioning is that when the programmer writes the code he will realize that an empty container and a container with only one element are special cases that need special treatment. If something calls for a special treatment in the programming it should also be given special attention when it comes to testing.

This gives a general rule for which Equivalence Partitions the programmer should specify. Whenever the programmer makes special considerations about certain situations, these special situations should be signaled to the outside world in form of Equivalence Partitions.

```
public class Container
{
   int count;
   int capacity;

   classinvariant:
      count>=0 and count<=capacity;

   public int insert(Object a)
   {
      precondition: a!=null;

      /* actual implementation */
      return count;

      postcondition: this.contains(a) == true;
      postcondition: old count + 1 == Result;
   }
}
```

Figure 3: This is part of the implementation of some container in a java-like language with support for Design by Contract. `Result` is a special keyword referring to the return value of the method. `Old` is also a special keyword referring to the original value of an instance variable before the method call.

In figure 4 the knowledge of Equivalence Partitions is expressed as partition invariants. They have the same semantic as class invariants with the exception that they should not hold for every object of the class, but only for the objects belonging to a particular partition.



```
public class Container
{
   int count; int capacity;

   partitioninvariant empty
      count==0;

   partitioninvariant oneElement
      count==1;

   partitioninvariant severalElements
      count>1,

   classinvariant:
      count>=0 and count<=capacity,
      in[empty oneElement severalElements];

   public int insert(Object a)
   {
      precondition: a!=null;

      /* actual implementation */
      return count;

      postcondition: this.contains(a) == true;
      postcondition: old count + 1 == Result;
      postcondition: notin[empty];
   }
}
```

Figure 4: The same example as figure 3, but the code has been extended with partition invariants. The keywords `in`, `notin` and `partitioninvariant` are new. The full specification of this language can be found in section 4.

Referring to partition invariants

Defining and naming the Equivalence Partitions of a class has some nice side effects. In figure 4 the class invariant has been extended with the expression `in[empty oneElement severalElements];` This basically expresses that any object of this class should always be in one of the three partitions. The keyword `in` gives a short way of writing `(count==0) or (count==1) or (count>1)`. In a similar way the `insert` method has been given an extra postcondition `postcondition: notin[empty]`. This postcondition expresses that at this point the object cannot belong to the `empty` partition. This is in other words the same as writing `(count==0)==false` as a post-condition.

At first this can be seen as a simple macro-expansion like feature, but later we will see that these expressions can play an important role in generating test cases.

Overlapping partitions

In the container example the partition invariants are defined in nice way such that any object will always satisfy one and exactly one partition invariant. If we look at more complex examples this might not always be the case. Figure 5 sketches an example where three overlapping partitions are defined. Imagine that we would like to add more partitions to the container example. First we add the partitions `sorted` and `notSorted` to express whether the elements in the container are sorted or not. Next we add the partitions `sameType` and `differentTypes` to express whether all the elements in the container have the same type or different types. Clearly these two properties for the container are independent and their corresponding partitions invariants will be overlapping.

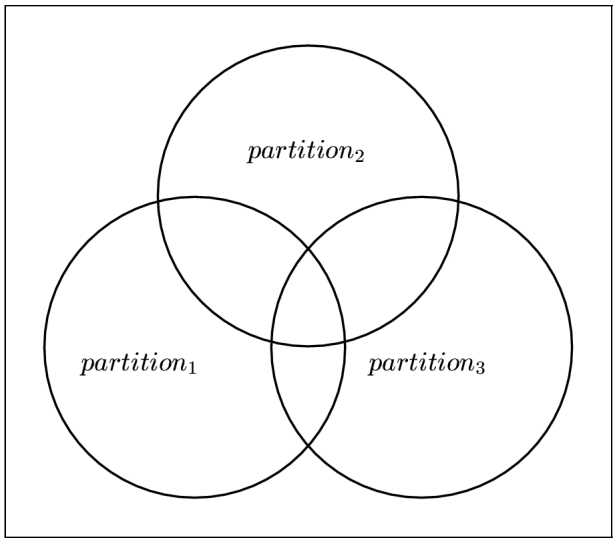
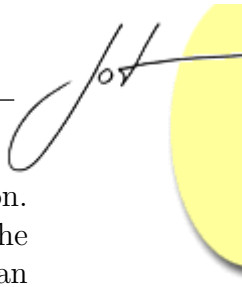


Figure 5: In more complex examples partitions can overlap each other.



The concept of overlapping partitions calls for a variant of the `in` expression. We will use `in[`$p1..p_n$`]` to describe that an object satisfies at least one of the partition invariants $p1..p_n$, and we will use `onlyin[`$p1..p_n$`]` to describe that an object satisfies one and only one of the partition invariants $p1..p_n$.

Who should specify the partitions?

So far we have argued that the programmer should specify the partitions. This is obviously a white box approach to testing. The programmer knows the internal structure of the program and exploits this knowledge when defining the partitions. This might seem odd as we normally connect Equivalence Partitions with black box testing.

We believe that it makes sense to use the Equivalence Partitions concept for both white- and black box testing. Imagine the following scenario. At the first level of testing the programmer specifies partitions based on his internal knowledge of the design and implementation of a class. At a second level of testing an external tester (e.g. a programmer that have not created this class, but wants to use it) specifies external partitions based on the interface of the class and based on his knowledge of the context in which the class is to be used.

In our current work we focus on the partition specified by the programmer, but future work should address how external partition can be specified and how all the partitions can be combined.

4 DESIGNING THE LANGUAGE

This section describes the design of a programming language that supports Design by Contract enhanced with knowledge of Equivalence Partitions. As we do not want to start from scratch we have selected Java as a starting point. It might seem that it would have made more sense to start out with Eiffel that already supports Design by Contract, but since the syntax of the assertion language had to be changed anyway and since the author of this paper felt more familiar with the Java-environment, the Java language was picked as a starting point. The general concepts should apply to any object oriented language.

Figure 6 shows the grammar for the language extension in BNF-format. The new language is equivalent to Java with the following exceptions:

- In a class declaration, before any variables or methods are declared a number of partition invariants and a class invariant can be declared.
- At the beginning of any method a number of preconditions can be declared.
- At the end of any method a number of postconditions can be declared.

In the top of a class declaration:

```
<declaration> = <partition_invariant_declaration>* <class_invariant_declaration>?

<partition_invariant_declaration> = partitioninvariant <identifier> <partition_invariant>

<partition_invariant> = <label> : <expression> ;

<class_invariant_declaration> = classinvariant <class_invariant>+

<class_invariant> = <label> : <assertion> ;

<assertion> = <expression> | <in_exp> | <notin_exp > | <onlyin_exp>| <nochange_exp>

<in_exp> = in <partition_list> ;

<notin_exp > = notin <partition_list> ;

<onlyin_exp> = onlyin <partition_list>;

<nochange_exp> = nopartitionchange;

<partition_list> = [ <identifier>+ ]

<label> = <identifier>
```

In the top of a method declaration:

```
<precondition_declaration> = <precondition>*

<precondition> = precondition <assertion> ;
```

In the bottom of a method declaration:

```
<postcondition_declaration> = <postcondition>*

<postcondition> = postcondition <assertion> ;
```

Figure 6: BNF for the extensions of the Java language. The nonterminals <identifier> and <expression> are normal java constructs.

All the new language constructs have already been explained in the previous section except for `nopartitionchange`, which can be used in postconditions to specify that a method cannot make an object change partition. A typical example is simple "get methods" that just returns a value.

A prototype of this language has been implemented as a Java pre-compiler. This means it is possible to write a program in this language and compile it to a regular Java program where all assertions are explicitly checked and exceptions are thrown if any assertions are violated. The prototype compiler is available for download at our homepage [9].

5 TEST COVERAGE

The introduction of partition invariants as expressions that can be evaluated during program execution brings a new form of coverage analysis: partition coverage. We can now execute a set of test cases and get statistics about which methods have been called and which partition invariants were satisfied before the call.

Figure 7 shows an example of partition coverage statistics for the container example. Statistics like this gives us a way to measure the quality of a given set of test cases. This concrete example raises two important results. The first is that we have not tried to call the `remove` method on an object from the empty partition. This is OK since the precondition states that this is not allowed and thus should not be tested. The second result is that the method `get` has not been called with an object from the oneElement partition. This is a more interesting result as it reveals that this set of test cases is missing this particular test, which might be relevant.

Methods / Partitions	empty	oneElement	severalElements
insert	5	5	7
remove	**0**	3	5
get	4	**0**	4

Figure 7: An example of partition coverage statistics

As with other kinds of test coverage analysis partition coverage should be used with great care [10]. That a method has been invoked on objects belonging to every specified partition do not necessarily mean that the method has been fully tested. Coverage analysis can be used to identify test cases that otherwise would have been missed, but coverage analysis do not say anything about the quality of the individual test cases. In our case the quality of the individual test cases depend on the quality of the assertions written in the code.

As an interesting side note it should be noticed that an advanced form of coverage analysis might be used for evaluating the way a class has been divided into partitions. If we generate two test cases with objects belonging to the same partition and afterwards do a standard path coverage analysis [3], we can compare the results. If

two objects from the same partition behave in completely different manners when it comes to path coverage, we can conclude that there must be some problem with the way the partitions have been defined. If the partitions were "correct" the two objects should behave not exactly the same, but at least in a similar fashion. This idea will not be discussed further in this paper, but will be addressed in future work.

6 A TOOL FOR TEST CASE GENERATION

The reason for enhancing Design by Contract with knowledge about Equivalence Partitions in the first place was the idea that it should give us an automatic way of generating test cases. In this section we will sketch the how to implement a tool like that.

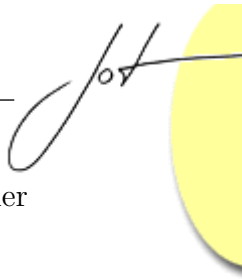
Deriving a partition transition graph

In section 3 we claimed that the concept of referring to partition invariants in pre- and postconditions could play a role in the test case generation. The idea is that these pre- and postconditions actually form a model for the possible method interaction of a class. We can illustrate this by looking at an example. Figure 8 shows part of an implementation of a class for handling socket connections. Four Equivalence Partitions have been declared:

- NonInit: The object has not yet been initialized with any information about IP-address or port number.
- Ok: The object has been initialized, the connection has been established and the status is OK.
- Error: A communication error has occurred.
- Close: The connection has been closed.

As the figure does not show the full implementation the partition invariants have only been specified in terms of the instance variables `_address` and `_errorcode`. The interesting part of this example is the pre- and postconditions in the constructors and methods. The class has two constructors, one with no parameters and one that takes an address and a port number as parameters. The first constructor specifies as part of its contract that objects created using this constructor will belong to the `nonInit` partition. Similar the second constructor specifies that object created using this constructor will belong to either the `ok` or the `error` partition.

If we look at the method `open` we will see that the precondition specifies that it only makes sense to call `open` on an object belonging to the `nonInit` partition, and



that the postcondition specifies that after the call the object will belong to either the `ok` or the `error` partition.

Analyzing all the methods in this way we end up with the partition transition graph shown in figure 9. Graphs like this one is very valuable for generating test cases. In the next section we will sketch an algorithm for generating test case from a partition transition graph.

Generating test cases

The goal is to automatically generate test cases that ensures that all methods have been called on objects belonging to each partition. This comes from the definition of Equivalence Partitions. If we test one member of a partition it should be as good as testing any member of the partition.

The basic idea is to use the graph as a recipe for creating objects belonging to the different partitions. First we build test cases for the partitions that can be reached directly from the "start" node. In the socket example this will be the `nonInit`, `ok` and `error` partitions. Since the graph is nondeterministic an automatic tool will either need to do some guessing (e.g. based on random data) or need some user guidance to figure out how to get to the `ok` and `error` partitions. At any point the test cases can be executed and coverage statistics can reveal whether all the intended partitions was reached. When a certain partition is actually reached, the information about how to reach it is saved for later reuse.

When the basic set of partitions has been reached we continuously tries to reach the rest of the partitions by applying the methods from the graph.

7 RELATED WORK

Java 1.4 has introduced a simple assertion facility as an integrated programming language construct [12]. This is indeed not a complete implementation of Design by Contract. This facility only allows the programmer to specify a simple assertion that should be evaluated at a given point in the program.

The ideas of Design by Contract has only been fully implemented in Eiffel, but a number of extensions that give the same functionality are available for other programming languages. One of these is Jass (**J**ava with **ass**ertions) [13], which is a pre-compiler that allows the use of Design by Contract in Java. JML (**J**ava **M**odelling **L**anguage) [6] is a behavior interface specification language for Java. JML uses Eiffel-style assertions combined with a model-based approach for specification.

Kolowa [5] has suggested the idea of using Design by Contract for testing purpose, but without actually stating how this could be done. Basically Kolowa states that Design by Contract gives a good starting point for doing testing because the pre- and postconditions and class invariants give a way to automatically detect when a

```
public class Socket
{
   partitioninvariant nonInit
      a1: _address == null,
      e1: _errorCode == null;

   partitioninvariant ok
      a2: _address != null,
      e2:_errorCode == null;

   partitioninvariant error
      a3: _address != null,
      e3: _errorCode != null;

   partitioninvariant closed
      a4: _address != null;

   classinvariant
      consistency: in[nonInit ok error closed];

   private String _address;
   private int _port;
   private Exception _errorCode;

   public Socket()
   {
      post in [nonInit];
   }

   public Socket(String adr, int port)
   {
      pre (adr!=null && port != 0);

      post in [ok error];
   }

   public void open(String adr, int port)
   {
      pre in [nonInit];
      pre (adr!=null && port != 0);

      post in [ok error];
   }

   public void close()
   {
      pre in [ok error];

      post in [closed error];
   }

   public byte get()
   {
      pre in [ok];

      post in [ok error];
   }
}
```

Figure 8: Part of an implementation of a class for handling socket connections.

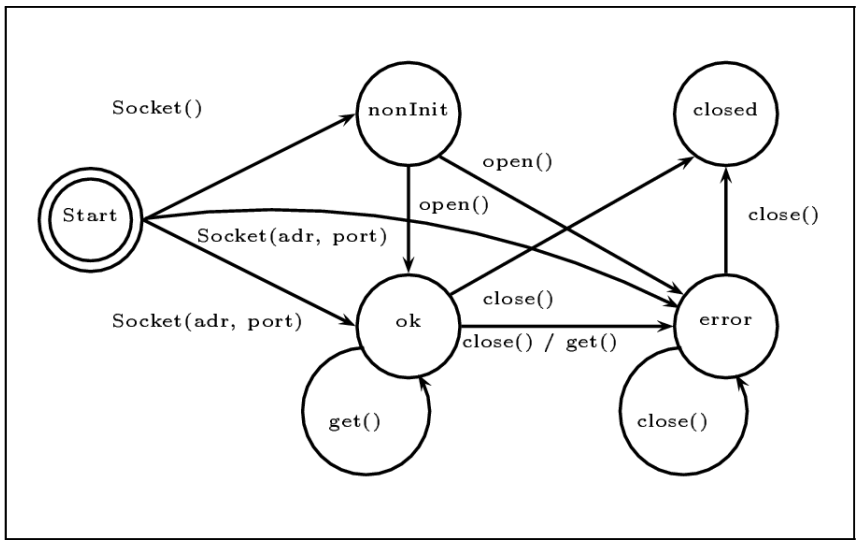


Figure 9: A partition transition graph for the Socket example. The graph shows which methods that can make an object change from one partition to another partition. "Start" is not an actual partition, but just the starting node of the transition graph.

test case fails. Cheon and Leavens [4] has described how JML and JUnit can be combined. This idea is very similar to the idea of Testing by Contract, but their approach relies on hand written test data.

The early ideas of this work have been described in [7] and [8]. To the best of our knowledge no one else has tried to include the concept of Equivalence Partitioning in Design by Contract.

8 CONCLUSIONS

In this paper we have suggested a testing approach where Design by Contract is enhanced with knowledge about Equivalence Partitions. Even though the work is in an early phase where no final conclusions can be drawn yet some preliminary conclusions are worth mentioning.

We have shown how the concept of Equivalence Partitions can be incorporated into Design by Contract in terms of partition invariants. We have argued that the programmer is or at least should be aware of Equivalence Partition during the implementation. If this hypothesis holds it makes good sense to maintain the knowledge about partitions from the implementation phase to the testing phase.

We have sketched that it is possible to build partition transition graphs by analyzing the information about partitions in a program. Furthermore we have argued that it is possible to generate quality test cases from these graphs.

Finally we believe the concept of using Design by Contract for testing purposes will encourage more people to use Design by Contract, which can be seen as a result in itself.

9 FUTURE WORK

In the nearest future we plan to work on the implementation of a tool for generating test cases based on the partition transition graphs. After that we plan to implement a few larger examples for evaluation of both the language and the test case generation tool.

In the not some near future there are a number of topics that needs to be addressed. In this paper we have already mentioned the idea of evaluating the quality of partitions based on the comparison of a partition coverage analysis and a more traditional coverage analysis like basic branch coverage. We have also mentioned the idea of external specified partitions that would support a more black box oriented testing approach. Finally it is important to evaluate that our extension of Design by Contract do not in any way diminishes the value of regular Design by Contract.

REFERENCES

[1] Kent Beck. Extreme programming explained. Addison Wesley, 2000.

[2] Kent Beck and Eric Gamma. Junit is a regression testing framework for java. Version 3.8.1 is freely available at http://www.junit.org, 2003.

[3] Boris Beizer. Software testing techniques. Van Nostrand Reinhold, 1990. Second Edition.

[4] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. Technical Report 01-12, Iowa State University, Department of Computer Science, Nov 2001.

[5] Adam Kolawa. Automating the development process. Software Development, July, 2000. Article available at http://www.sdmagazine.com/.

[6] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. Technical Report 98-06i, Iowa State University, Department of Computer Science, 2000.

[7] Per Madsen. Testing by contract - combining unit testing and design by contract. Proceedings of The Tenth Nordic Workshop on Programming and Software Development Tools and Techniques, August 2002.

[8] Per Madsen. Unit testing using design by contract and equivalence partitions. Research abstract published in XP2003 conference proceedings, Genova 2003, Springer's Lecture Notes in Computer Science Series, Volume 2675/2003, May 2003.

[9] Per Madsen. Project homepage for the friends project: http://www.cs.auc.dk/~madsen/Friends/Doc/, January 2004.

[10] Brian Marick. How to misuse code coverage, 1997. Paper available at http://www.testing.com/writings/coverage.pdf.

[11] Bertrand Meyer. Object-oriented software construction. Prentice Hall, 1997. Second Edition.

[12] Sun Microsystems. Programming with assertions, 2002. Documentation available at http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html.

[13] Detlef Bartetzko; Clemens Fischer; Michael Moeller; Heike Wehrheim. Jass - java with assertions. Electronic Notes in Theoretical Computer Science, Vol. 55 (2) (2001), Elsevier Science Publishers, 2001.

ABOUT THE AUTHORS

Per Madsen is a PhD student from Department of Computer Science, Aalborg University, Denmark. He can be reached at madsen@cs.auc.dk. See also http://www.cs.auc.dk/~madsen.