Automated Refactoring of Java Contracts

Thesis for MSc in Advanced Software Engineering

Iain Hull

A thesis submitted in part fulfilment of the degree of MSc in Computer Science with the supervision of Dr. Joseph Kiniry.

School of Computer Science and Informatics

University College Dublin

15 April 2010

Abstract

Modern agile practices and techniques help programmers construct working software while responding to changing requirements. The requirements are encoded into automated tests, which ensure the software is correct and remains correct while it evolves. Design by Contract is another method that ensures software is correct. This uses formal specifications to define the expected behaviour of each module. These specifications can then be used to verify the system at runtime and statically.

There are a lot of synergies between these two methodologies, such as their emphasis on working software, however when used together they can also complement each other. This is especially true of Design by Contract and the agile practice of Test Driven Development. Design by Contract must also respond well to changing requirement if agile programmers are to be encouraged to adopt it. It must also be able to work well with their other agile practices, especially refactoring. The specifications used in Design by Contract are written with the code they describe. Therefore to respond to change they must able to change the same way and at the same time as the code, which is via refactoring.

This dissertation demonstrates the feasibility and power of automated specification refactoring by implementing an Eclipse plug-in that automates the *Pull Up Specification* and *Push Down Specification* refactoring operations for Java Contracts, a derivative of the Java Modelling Language (JML). The refactoring operations are defined and compared to their code counterparts. The process of gathering the requirements and implementing the plug-in are explained. Based on this work the role of refactoring formal specifications is explored as well as the synergy between Design by Contract and Test Driven Development with regard to refactoring. Finally future research that could extend these work and ideas are also discussed.

Acknowledgements

I would like to thank my supervisor Dr. Joe Kiniry, for suggesting Automated Refactoring of Java Contracts as a research topic. I should also acknowledge the time, advice and encouragement he gave during this dissertation. His passion for the subject is infectious, and as a result I thoroughly enjoyed our discussions and researching this topic.

I would also like to thank David Graham for reviewing multiple copies of this text as it evolved.

Finally none of this would have been possible without the support of my loving wife Karina and son Oisín. They have given up a lot to allow me the time to work on this for which I am very grateful.

Table of Contents

1 Introduction

Over the past decade the software industry has crossed a rubicon in how it deals with the prospect of change. Previously it tried to manage or control change. It put a lot of effort into plans and designs to avoid it. Recently the industry has started to accept change as inevitable, and moved its focus onto how to live with it and even embrace it. Agile processes and practices have been developed and reworked to help software organisations cope with change.

Design by Contract is a methodology to build systems and verify they are correct. It is a slightly older concept pioneered by Meyer and demonstrated in the language Eiffel [**1**,**2**]. In Design by Contract the module's interface and its semantics are specified before it is implemented. The semantics in one module can then be relied upon to implement the next. This simplifies the implementation as it removes the need for defensive programming which increases the complexity of the system. Its use and importance are growing beyond its traditional domain of mission critical software and formal methods. This is especially true as support of Design by Contact has spread to the tools and languages most developers use. This includes the Java Modelling Language, JML, which adds support for Design by Contract to the Java language using special annotated comments with a Java-like syntax [**3**,**4**,**5**].

The benefits of Design by Contract and formal methods have been lauded by some in the Java community [**6**]. However a lot of people in the agile community have associated them with big up front design and the waterfall process. More recently there has been a move afoot to bring the best of agile methods and formal methods together [**7**,**8**,**9**,**10**].

The recent popularity of agile methods and practices has brought an explosion in tool support to these programmers. Of these practices, Refactoring, is geared specifically at changing systems safely, and is now so ubiquitous that modern integrated development environments, like Eclipse, include automated support for common refactoring operations [**11**,**12**,**13**]. These automated operations guarantee that the code's behaviour will not change as a result of the refactoring. This enables programmers to write the simplest possible code now safe in the knowledge that the structure can easily be changed later if required.

For the benefits of Design by Contract and formal specifications to reach the agile community it must work well with these agile practices and tools. Programmers must be able to refactor specifications as easily as code. This means refactoring specifications, and modifying specifications when code refactoring changes the meaning of the code.

1.1 Goals

The goal of this dissertation is to demonstrate the infrastructure required to implement automated refactoring of formal specifications. The refactoring operations *Pull Up Specification* and *Push Down Specification* are used as candidates for this. These operations were chosen because they are practical and non-trivial to implement, but the code variants are also well understood [**12**].

The *Pull Up Specification* and *Push Down Specification* refactorings are practical operations that a Design by Contract programmer might expect to carry out while improving the design of their system. While these operations like all refactorings can be performed manually, there is benefit to the programmer of automating the steps required. The tool can verify the refactoring is sound before completing and the programmer can rely on the consistency and correctness of the operation to improve the system more quickly.

The operations are non-trivial: they have to work with the semantic properties of the system as well as just the syntactic properties. For example the *Rename* family of refactoring operations operate on the syntactic properties of code and specifications [**12**]. Whereas the *Pull Up Specification* and *Push Down Specification* have to take account of the context and semantics of the specifications they are modifying as well as their syntactic properties.

The code versions of these operations are well understood. As well as providing a firm foundation to work from, it enables the code and specification variants of these operations to be compared and contrasted. From this we can distil the important aspects of refactoring specifications as opposed to code and how this might affect the practices of Design by Contract and Test Driven Development [**14**].

1.2 Layout

Section 2 Literature Review:

Reviews the main works which influenced this dissertation, they explore applied formal methods and Design by Contract's relation with agile practices, Test Driven Development and refactoring.

Section 2 Refactoring:

Introduces the core aspects of refactoring referred to throughout this dissertation.

Section 4 Formal Specifications:

Introduces Formal Specifications and how they are usually practiced. This includes a description of the Java Modelling Language and Java Contracts.

Section 5 Eclipse:

Introduces the Eclipse platform and how it supports the development of refactoring tools.

Section 6 Refactoring Formal Specifications:

Discusses the issues surrounding refactoring formal specifications, including specification smells and semantics. The similarities between Design by Contract and Test Driven Development are also discussed as this leads to some insight into the role of refactoring for specifications.

Section 7 Requirements:

Describes the details of the refactoring operations and explores the requirements for their automation.

Section 8 Construction:

Describes the construction and implementation details of the automated refactoring operations as Eclipse plug-ins

Section 9 Conclusion:

Explains the importance of automated refactoring tools for the adoption of Design by Contract and formal specifications by Java programmers and the larger Agile community.

2 Literature Review

The combination of formal specifications with agile software development is a relatively new area of research. Below are the sources in this area that influenced this dissertation.

2.1 Extreme Design by Contract

One of the earliest papers to discuss the merger of Design by Contract and Extreme Programming practices is "Extreme Design by Contract" by Feldman [**9**]. This paper briefly mentions Design by Contract's synergy with unit testing and refactoring. It notes that the inclusion of formal specifications can simplify unit tests. The implications to specifications of refactoring Java classes are then demonstrated. A sequence of refactoring operations is applied to a reverse polish notation calculator class to extract its stack implementation. The steps to achieve this refactoring are performed manually based on Fowler's method with the addition of the specification concerns. The concept of Specification only Refactoring operations are then introduced, including *Pull Up Assertion*, *Push Down Assertion*, and *Create Abstract Precondition*. This dissertation implements the first two of these refactoring operations as *Pull Up Specification* and *Push Down Specification*. Finally an analysis of the specification implications of Fowler's 68 refactoring operations is summarised [**12**]. This breaks down the refactoring operations into: ones that have no specification implications; ones that require the addition of specifications; and ones that require specification modification. The automation of these refactorings is also briefly discussed.

2.2 Refactoring with Contracts

Goldstein, Feldman and Tyszberowicz expand on the concepts of refactoring code with specifications in "Refactoring with Contracts" [**10**]. This introduces an Eclipse plug-in called Crepe which refactors code with specifications. A number of refactoring operations are detailed: Extract Superclass and Add Inheritance. These are code refactoring operations but Crepe is able to reason about the specifications. The specifications can be moved to a parent class; combined and simplified; and redundant ones removed. The Crepe system is very advanced however it does not operate on JML specifications and code was never released publicly. No further work seems to have been carried out since this paper was published in 2006.

2.3 Formal Versus Agile: Survival Of The Fittest?

Sue Black et al discuss integrating formal methods into the agile process in "Formal Versus Agile: Survival of the Fittest?" [**7**]. They note that both share the common aim of producing reliable software. This is the primary motivation for formal methods, while Agile strives to cope with change as this is a common cause of unreliability. It is argued that formal methods can cope with change too and summarises what is required for a formal method to be agile. The four areas formal methods add value to agile processes are enumerated as: testing, requirements, refactoring and documentation. For refactoring it is noted that as a human activity it is error prone and while tests help catch mistakes assertions add extra safety. It is also noted that the tools for formal

method must be fast and available to be included as part of an agile process. The time and effort required for formal and agile projects are also compared. Finally the myths surrounding the cultural divide between both approaches are broken, indicating their combination shows promise.

2.4 Integrating Extreme Programming and Contracts

Heinecke and Noack use the novel approach of user stories to justify the inclusion of Design by Contract with Extreme programming, in "Integration Extreme Programming and Contracts" [**8**]. This is on the bases that all value must be expressed in user stories before being added to the system. One user story enables large teams to be divided into smaller teams along subsystem boundaries. Specifications are then used to maintain consistency between these boundaries. The second user story details the documentation benefits, noting that manually documenting specifications outside of the code results in incomplete and outdated documentation. It is shown that Design by Contract enhances the Extreme Programming values of communication, feedback, and courage. The relationship between specifications and unit tests is explored, arguing that specifications and unit tests supplement each other. While unit tests serve as documentation for the code they test, it can be hard to infer a method's preconditions and post conditions from its tests. It also highlights that unit tests check a number of example situations where as specifications cover the universal desired semantics of the method. This fits well with the "Once and Only Once" principle of Extreme Programming. Finally it is pointed out that some existing practices perform some of the roles of Design by Contract but not as rigorously.

3 Refactoring

The definitive introduction to refactoring for software professions is the book "Refactoring Improving the Design of Existing Code" by Martin Fowler [**12**]. In this he defines:

> **Refactoring (noun)**: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.
>
> **Refactor (verb)**: to restructure software by applying a series of refactorings without changing its observable behaviour.

These definitions describe refactoring very well once you already know what it is, but highlight an important aspect of refactoring while the definition is informal the practice is rigorous and well defined.

The goal of refactoring is to improve the structure and design of code. This stems from the fact that most programming languages enable even simple problems to be solved in many different ways. While all can provide the same results, some solutions are intrinsically better than others. For example some are more concise, or describe the problem they solve better; while others are easier to extend and add new features. What constitutes an improvement is deliberately left vague and subjective. It is the programmer's responsibility to decide this based on the fact that they are the person with full knowledge of the context required to make the decision. For example they know the intent of the code; the skill level of the other programmers who will work on the code; and the coding practices the team follows. While metrics can guide the programmer only a person can know all this information. Fowler does categorise a number of code and design issues called smells that can be improved. These serve as an example of when and how refactorings can be applied.

While what constitutes an improvement is vague, the process of applying the improvement is not. The improvement must not change the behaviour of the system. This is not possible if the current behaviour cannot be verified. This should be via a suite of automated tests to prove the functionality of the part of the system being refactored. If such tests do not exist they have to be created before the refactoring can be applied. Adding tests where there are none not only helps verify this refactoring but can be relied upon by future refactorings as well.

A refactoring operation is performed as a number of small steps. Each step is completed and verified before the next step is performed. This is very important because if a mistake is made or the operation is not appropriate, not only is it found as soon as possible, but it can easily be reversed if required.

While programmers have always improved the design of code and systems while they worked, what makes refactoring unique and worthy of its own name is the rigorous way it is applied.

3.1 Smell

Refactorings are initiated by the discovery of a smell in the code or system. Kent Beck coined the term "Smell" for code that could have its structured improved. "If it stinks, change it" [**12**]. Smells are not defined by metrics, though this can play a part, but are defined by informed human intuition. The purpose of defining smells and listing the refactoring options is not to provide concrete steps to improving the design of the code, as in *if smell 'A' then do steps 1, 2 & 3*, but

guide the programmer's intuition and give them a terminology to discuss bad design with their colleagues.

Smells or rather their absence also serve as a stopping point for refactoring. If there are no smells there is nothing to improve and refactoring in this case is a sign of over engineering and should be avoided.

The term smell has since been expanded from just code to refer to software design, architecture, and documentation. It can also be used to refer to formal specifications.

3.2 Agile Practice

Refactoring is an integral part of agile software development, for example it is one of Extreme Programming's 12 practices [**15**]. This importance is seen by how many of the agile principles it embraces: accepting change, frequent deliverables, working software, technical excellence, simplicity, and emergent design [**16**].

Refactoring enables programmers to welcome changing requirements because they know they can quickly refactor the system into a form that makes the change easy to implement. It is performed in small steps and does not change the behaviour of the system supporting frequent deliveries and working software. Refactoring encourages programmers to improve the design of the software they develop, other programmers also get share its results, and this promotes learning and technical excellence in the team. When programmers know they can safely refactor the system to fit their needs in the future, they can keep the current design as simple as possible; they don't feel the need to design for future extensibility as it can be added later when it is required. By only designing the system for today and refactoring only as required tomorrow the system's design is emergent it does not have to be planned out in advance.

Frequent refactoring also helps to solve the problem of "Design Debt" [**17**]. This is where a system grows organically, but the design is not modified to support this growth. As a result new features take longer to implement. This happens when programmers try to add features as quickly as possible and don't modify the design to support these new features. Often this is because it would take too much time or the risk of breaking something is too high. By concentrating on the short term goals of implementing the feature but neglecting the long term goals of maintaining the design of the system, programmers are effectively borrowing time from tomorrow's features and using it today. By concentrating on only delivering the next feature as quickly as possible, agile software development could quickly suffer from Design Debt, however this is solved by its use of refactoring to continually integrate the new features into a cohesive system and improve its design.

3.3 Refactoring Tools

Refactoring operations are defined in a number of simple steps which can safely be applied to code. The order and process of these steps is important, they have been practiced and refined to make the process as safe as possible. However the steps often involve small changes to lots of source files. The processes are designed to rely on the compiler and unit tests as much as possible to prevent simple coding errors. Programmers can miss this subtlety and can change the steps or their order by mistake or when for looking a shortcut. This can reduce the safety of the refactoring and possibly allow the code's behaviour to change.

The steps involved in refactoring operations are always the same and relatively simple, this makes them ideal candidates for automation. This removes the chances of programmer error and speeds up the refactoring process. When the tools are fast and deliver consistent results, programmers can safely perform a number of refactoring operations between unit test runs. As Fowler says, "The main purpose of a refactoring tool is to allow the programmer to refactor code without having to retest the program." [**12**]. This encourages programmers to use refactoring more freely thus improving the design of the system, and allowing programmers to keep the current design as simple as possible safe in the knowledge that they can refactor it later.

3.4 Refactoring Tools Criteria

Fowler outlines the main criteria for a refactoring tool as: access to a program database, and code parser, then accuracy, speed, undo and integration [**12**]. The program database provides the ability to search for program entities such as classes, methods calls and variables. The code parser enables the tool to understand the code's syntax and modify it. Accuracy means the tool must provide reliable and predictable results, this enables the user understand the affects of an operation before executing it. The tool must also be able to perform the operations quickly, so the user can chain operations together to form compound refactoring operations. If the tests fail or the programmer does not like the results of a refactoring they must be able to easily back-out the changes restoring the system to its previously valid state. Finally the refactoring tool must be a part of the user's tool chain or integrated development environment; the user should not have to switch focus to use refactoring tools.

4 Formal Specifications

Formal specifications control how software components collaborate with one another by explicitly specifying the rules they must follow to talk to one another. Each module describes its semantics as well as its syntax. By formally describing the semantics of each module it is easier to verify that the system is correct. The system can then be verified by verifying each module in the system. Each module is verified by verifying that it fulfils the semantic requirements of the other modules it uses, and verifying the module provides the semantics it promises. Breaking the system down in this way enables large and complicated systems to be verified in a manageable way, be that manual desk checking, runtime checking or static checking.

The semantics describe the rewards and responsibilities of using a module. When the client meets its responsibilities by complying with the interface's preconditions, it is rewarded by the guarantees in the postconditions and the class's invariants. The postconditions describe the class's state immediately after the operation. The invariants describe properties of the class's objects that are always true. These postconditions and invariants can then be relied upon to provide the preconditions of the next module.

These contracts, or specifications, are used to document the system's design and aid its implementation. Once in place specifications aid further development and reuse by clearly explaining the semantics of each module, testing and debugging by quickly showing the location and reason for an error, and verifying the system as a whole is correct.

Formal specifications are usually written for one of two reasons: to verify a system as it is built or to enable the verification of an existing system. These usage patterns are commonly called "Design by Contract" and "Contracting the Design" respectively [**1**,**18**].

4.1 Design by Contract

In Design by Contract the specifications are a stepping stone between the requirements and implementation. Requirements are broken down into the modules required to implement them. The contract of each module is formally specified before it is implemented. The module's logic is simplified by removing the need for defensive programming. This is possible because it can rely on its preconditions and does not have to explicitly check them. As new requirements are implemented, the specifications and the implementation evolve together. This gives the programmer more freedom, since both specifications and code can change. The code can change to meet the specifications or to make the specifications easier to write. Likewise, the specification can change to make the code easier to write.

Overly complicated specifications highlight a problem or a smell in the system's design. Once discovered the system can be refactored to make the specifications easier to write, and make the module easier to use. Without specifications this smell might never have been discovered, and the chance to improve the design of the code missed.

Modules tend to be broken down into a number of small procedures, as a result of writing specifications, since this makes them easier to write. Typically each procedure does exactly one thing; it either queries or modifies the module. This simplification makes modules easier to implement and use.

The main drivers of refactoring in Design by Contract are extension and code/design smells, but specification smells are also possible. Refactoring operations have more freedom to modify the

design of the system and more scope to modify the semantics (see section 6.3 below). Ideally specification semantics should not change but they can change when the structure they describe changes. The synergy between code and specifications in Design by Contract is very similar to the synergy between code and unit tests in Test Driven Development (see section 6.4 below).

4.2 Contracting the Design

In contracting the design programmers are more restricted. The system is already implemented and they are adding specifications to each module. This is either to verify the system or to enable new systems built upon it to be verified.

The programmer cannot modify the code, so must adjust the specifications to match the code and its documentation. This can lead to more complicated specifications, since the design cannot be changed to simplify them. If the specifications had been written first or at the same time the system could have been designed differently.

In contracting the design, specification smells are the main driver for refactoring; design smells and extension have no bearing as the system cannot be modified. Since refactoring has no freedom to modify the design, it has less scope to modify the semantics of the specification. The semantics can only be modified if they are wrong. In this case, since the specification is the deliverable, it can be argued that modifications are fixes and not refactorings.

4.3 Java Modelling Language

The Java Modelling Language (JML) is a formal behavioural interface specification language for Java. It provides Design by Contract capabilities to the Java programming language, as well as more formal model based semantics [**3**]. It describes both the details of a module's interface, and its behaviour from the client's point of view [**5**].

JML was specifically designed to be powerful enough for full verification yet simple enough to be easily understandable by Java programmers. To achieve this JML takes the best aspects of other formal specification technologies Eiffel and Larch. Like Eiffel JML declarations are expressed in their native programming language, Java, and stay as close as possible to Java's syntax and semantics. Unlike Eiffel, JML specifications are model based which enables more complete specification of complicated modules. This is like Larch, however model specifications do not require Larch's mathematical syntax [**5**].

JML specifications are written with special annotation comments. These are standard Java comments whose first character is an @ (at-sign). Note that JML annotations are not the same as Java 5 annotations. JML does not impose any particular design methodology on its users, but includes a suite of tools to enable programmers to choose their preferred methodology [**19**].

JML comes with many tools. The JML checker, jml, parses and type checks the specifications. Specifications are added to the systems API documentation using the jmldoc; this enables programmers to quickly browse and discover an API's semantics. The runtime checker, jmlc, adds code to verify each module's specifications. Whenever a method is called, an error is raised if any of these specifications are broken. This can be used in conjunction with jmlunit, which generates unit tests from specifications, these unit tests are run with jmlc. The tests include cases that meet and break a module's specifications. It a case meets the modules preconditions an error is a failure. If a case breaks the modules preconditions an error is expected and not a failure. There are a number of static checkers, ESC/Java2, Krakato and Sireum/Kiasan. These verify that

the system meets its specifications by proving the semantics of the code semantic match the semantics of the specifications. The automatic invariant generator, Daikon, analyses running code to propose new class invariants. Most of these tools include Eclipse IDE integration [**19**].

```
public class BankAccount {
    private int balance;

    public /*@ pure @*/ int getBalance() {
        return balance;
    }

    /*@ public invariant this.getBalance() >= 0; @*/

    /*@
      @ requires amount > 0;
      @ requires this.getBalance() - amount >= 0;
      @ ensures this.getBalance() = \old(this.getBalance()) - amount;
      @ ensures target.getBalance() = \old(target.getBalance()) + amount;
      @*/
    public void credit(BankAccount target, int amount) {
        balance -= amount;
        target.balance += amount;
    }
}
```

Example 1: JML Specifications

4.4 Java Contracts

Java Contracts are a form of JML specification written completely in Java [**20**]. The specifications are not in special annotated comments, but written as regular java statements. They use either static to the `JC` class or special Java 5 annotations. Java Contracts is based on Microsoft Research's Code Contracts for .Net [**21**]. It applies Code Contract's techniques to JML and Java. Java Contract uses some of the annotations defined in JML 5 [**22**].

Java Contracts aim to remove the requirement for a separate JML processor that has to be maintained as Java continues to evolve. Java Contracts tools can be processed using existing parsers. These can operate at the source level using existing Java Source parsers, like the one contained in the Eclipse JDT; or at the byte code level using existing byte code parsers like Apache's Byte Code Engineering Library (BCEL) or ObjectWeb's ASM [**23**,**24**]. Being able to leverage existing parsers makes Java Contracts ideal for creating and experimenting with new tools for JML.

```
public class BankAccount {
    private int balance;

    @Pure
    public int getBalance() {
        return balance;
    }

    @Invariant
    public boolean jcInvPositiveBalance() {
        return this.getBalance() >= 0;
    }

    public void credit(BankAccount target, int amount) {
        JC.requires(amount > 0);
        JC.requires(this.getBalance() - amount >= 0);
        JC.ensures(this.getBalance() == JC.old(this.getBalance()) - amount);
        JC.ensures(target.getBalance()
                   == JC.old(target.getBalance()) + amount);

        balance -= amount;
        target.balance += amount;
    }
}
```

Example 2: Java Contract Specifications

As can be seen in Example 2, Java Contracts specify preconditions and postconditions as calls to the methods `JC.requires()` and `JC.ensures()` respectively inside the method they apply to. Also note that the pure method `getBalance()` uses the JML 5 `@Pure` annotation, which indicates the method does not modify any data and so can be used in specifications. Invariants are specified as java methods with the `@Invariant` annotation; the return value of this method is the invariant's expression.

The methods on the `JC` class do not implement any functionality, their only purpose is to mark up the java code with the specifications. This mark up can then be used to add runtime checking code, verify the code with static checkers or could be removed completely for production.

```
@JC.ContractClasses(BankAccountContract.class)
public interface BankAccountInterface {
    @Pure
    public int getBalance();

    public void credit(BankAccountInterface target, int amount);
}

class BankAccountContract implements BankAccountInterface {
    @Override
    public void credit(BankAccountInterface target, int amount) {
        JC.requires(amount > 0);
        JC.requires(this.getBalance() - amount >= 0);
        JC.ensures(this.getBalance() == JC.old(this.getBalance()) - amount);
        JC.ensures(target.getBalance()
                   == JC.old(target.getBalance()) + amount);
    }

    @Override
    public int getBalance() {
        return JC.result();
    }

    @Invariant
    public boolean jcInvPositiveBalance() {
        return this.getBalance() >= 0;
    }
}
```

Example 3: Java Contracts Specifications on Interfaces

Java interfaces cannot contain any fields or method implementations; as a result Java Contracts cannot be defined inline as for classes. Interface specifications are defined in a class that implements the interface, the interface uses the `JC.ContractClasses` annotation to link the contract class.

5 Eclipse

Eclipse is an open source suite of integrated development environments (IDE) for Java and other languages [**13**]. It is lead by the Eclipse Foundation and developed by a community of companies and individuals with a stake in its success. Eclipse was initially developed by IBM in 1998, they later open sourced it in 2001 [**25**].

Eclipse is made up of a number of separate projects or features which are bundled together into a number of different Eclipse based products, each tailored for different types of language and technology. The Eclipse Foundation releases 10 different Eclipse based products, many third parties also release their own Eclipse based products [**13**]. For example the Eclipse IDE for Java developers is composed of the Eclipse Platform, Java Development Tools, CVS source control, XML tools and other projects. As well as using these pre-packaged products users can also extend their environments by downloading and installing other Eclipse and third party plug-ins and features.

These products and extensibility are made possible by the open plug-in architecture and philosophy of the Eclipse [**26**]. The Eclipse Platform provides the functionality and services common to all integrated development environments; this includes file and project management and a basic editor. The platform then provides extension points which enable higher layers to reuse and extend this functionality. The Java Development Tools use these extension points to define the Java Development Environment, including compiler, Java source editor, and other tools. However the platform is not the only thing that can publish extension points. All plug-ins can. This enables later plug-ins to extend earlier ones. The platform is just a set of plug-ins that provides the common IDE functionality built on extension points. Plug-ins are encouraged to implement their own features through extension points, thus extending themselves. This ensures that all plug-ins are first class citizens when they try to extend features. The result is not simply an IDE that is extensible, but an extensible platform used to build an IDE.

Eclipse with the Java Development Tools is one of the most widely used Java IDEs today. Its plug-in architecture has enabled it to develop the features most developers want very fast. It was the first free Java IDE to include Refactoring and other automated source modification operations. This has been present since June 2002 with version 2.0 [**27**]. It is also the basis for a number of commercial Java IDEs, for example IBM's WebShere Studio.

The Eclipse code base is open source, allowing programmers to study it, learn from it and extend it. This when combined with its architecture and philosophy enable quick innovation. New tools are easily built by adding just what is required to an existing IDE that is designed from the ground up to be extended. There is no need to reinvent the wheel. As plug-ins provide services that others want to build on, these services can be refined and generalised and moved to lower layers, enabling more plug-ins to reuse them.

This happened to the Eclipse refactoring infrastructure [**28**]. Originally refactoring was implemented in the Java Development Tools, then the language-neutral refactoring logic was separated to a new plug-in, the Language Refactoring Toolkit (LTK), and made part of the platform. Now the Eclipse C/C++ Development Tools uses the LTK to implement its refactoring operations [**29**].

The Eclipse Platform together with the LTK provides four of Fowler's six criteria for refactoring tools [**12**]. All the Java code in the Eclipse workspace is searchable through a built-in program database. The compiler's Java AST is available to parse source files. Changes made by the

refactoring operations are automatically added to the platforms undo stack, enabling their changes to be easily rolled back by the user if required. Finally, all this is integrated into the Eclipse development environment. This just leaves refactoring authors responsible for the accuracy and speed of their operations. This combined with the well designed and abstracted Language Refactoring Toolkit and Eclipse's open source nature make Eclipse an ideal platform to research refactoring tools.

6 Refactoring Formal Specifications

This section explores the issues in refactoring formal specifications: specification smells, the differences between refactoring code and specifications, how to treat semantics and the similarity between Design by Contract and Test Driven Development.

6.1 Specification Smells

A refactoring operation is driven by the discovery of a smell. That smell might be the problem or a symptom of a larger problem. A simple code smell, like duplicate code, could simply be the result of lazy programming or could be caused by bad design making the code hard or impossible to reuse. In the first case the code smell is simply a code problem, and in the second it is a symptom of an underlying design problem.

Specification can have smells too, and these can are trigger specification refactorings, but just like code smells, specification smells can also be a symptom of a greater malaise in the system.

Below is an exploration of basic specification smells and their possible refactorings.

6.1.1 Over demanding precondition

Clients are responsible for ensuring that a method's preconditions are met before calling it. If the precondition is not intrinsically met by the client then it must test the precondition before making the call, and handle the case where the precondition is not met. It is clear that each precondition adds extra responsibility to the client. An over-demanding precondition is where a large number of clients use a method and the majority provide special handling when the precondition is not met [**1**]. This smell highlights a design flaw where a class does not provide easy access to its most common use cases. Sometimes the preconditions are hard to comply with and add extra complexity to the clients each time they call the method.

Other times the preconditions are not fulfillable, and the client does not have access to the information required to ensure they are met, or does not know how to change the object's state to meet them. A set of methods might have to be called in a specific sequence to meet the preconditions of the final method, if the client only plays a part in this sequence how can it guarantee the previous methods have been called correctly. For example a value has to be pushed onto a stack before the stack can be popped; if the client wants to pop a value to print it on the screen it must be able to check that a value has already been pushed. In this case a method that returns the size of the stack or whether it is empty would be sufficient as long as the pop precondition was expressed in terms of one of these methods. Even if the client can sense that the preconditions are not met, it might have to know how to modify the object to meet them. For example an airline ticket can only be printed once it is finalised, the finalise step can happen at any stage prior to the request to print the ticket. The print request should not fail just because the ticket is not been finalised, the system should finalise the ticket if required then print the ticket. In this case the print request must be able to call a method to test whether the finalise step has been completed. The return value of this method must also be guaranteed by the postcondition of the finalise method. Then the print precondition of the print method can use this method to express the requirement that the ticket be finalised before printing. It is clear that a precondition can have knock on affects for other methods in a class to ensure that it a client can fulfil its obligations.

6.1.1.1 Possible Refactorings

- Remove the offending precondition and provide the special handling required in the method being called.

- Add a new method without the offending precondition which delegates to the original method and provides the special handling required.

- If the preconditions are unfulfillable ensure a query method can be used to test the precondition. If a command method is required to meet the precondition ensure that its postcondition guarantees the first method's precondition. Both precondition and postcondition should be written in terms of the query method.

- If a group of methods have the same over-demanding preconditions for their parameters, use the *Introduce Parameter Object* refactoring to wrap the affected parameters with an object [**12**]. The preconditions become invariants in the new class.

- If the parameter's values referred to in the preconditions originally come from the same class use the *Preserve Whole Object* refactoring to replace the group of parameters with a single object [**12**].

6.1.2 Non Complementary Preconditions

Heavyweight specifications can contain a number of cases: a normal case when the method behaves as expected and a set of exceptional cases for errors. The preconditions for each case should be mutually exclusive as they are used to select the expected outcome of calling the method. If the preconditions allow more than one case at a time then for some inputs the client does not know what outcomes to expect, in this case the client can only rely on the postconditions that are common to these cases, their lowest common denominator.

For example a naive method which reads a file whose name is passed in as a parameter might specify two exception cases one when the file name is invalid and the other when the file cannot be found. However it is also true that an invalid file name cannot be found, so in this case when a client passes an invalid file name which outcome can they expect. A programmer might think that the first case is obvious however a machine verifier might not be able to distinguish this case. In this the second case should include the precondition: *the file name is valid*; which excludes it from the first case.

As the number of cases a methods supports increases and their preconditions become more complicated, modifying the conditions for each case becomes harder and the preconditions can become non complementary. The primary cause of this, like *Over Demanding Precondition*, is complicated preconditions. Each time a condition is added to a specification case the complement of the condition should be added to the other cases. If the conditions on a case are later merged removing the condition can be difficult. Another complicating factor is that preconditions are inherited, so ensuring the specification cases of all the subclasses are complimentary can be very difficult.

6.1.2.1 Possible Refactorings

- Extract some of the preconditions that refer to the class's state into new query methods, and then refer to these query methods in the preconditions. This will help simplify the logic in the preconditions making them easier to keep consistent.

- Use the *Introduce Parameter Object* refactoring to wrap the effected parameters with an object [**12**]. The preconditions become invariants in the new class.

6.1.3 Inconsistent Use of Data Groups

A data group specifies a set of class fields that are typically modified together. They are used with the assigns clause to specify which fields a method modifies. If a number of methods modify the same set of fields they should abstract the set of fields with a data group. Data groups are used inconsistently: if one method uses a data group to refer to a set of fields in its assigns clause but another method refers to a field in the group by name.

6.1.3.1 Possible refactorings

- Replace references to fields with references to the data group the field belongs to.

- If a method references a data group's fields because it only modifies some of the fields in the group, split the data group into smaller groups reflecting their use.

6.1.4 Constructor Specifications

Constructor specifications are not inherited, so subclass constructors must manually repeat the specifications causing duplication. As a result constructor specifications should be kept to a minimum. Constructors have no object state to rely upon, so preconditions only refer to parameters. The result of calling a constructor should be an object that meets its invariants, so postconditions should not be required except to indicate assignment for a parameter to a field.

6.1.4.1 Possible Refactorings

- Promote postconditions to invariants or initially clauses.

- Reduce the preconditions by using the *Introduce Parameter Object* refactoring [**12**].

- Extract the preconditions into a static method that returns true if the preconditions are met, subclasses and clients can then use this method to test the preconditions.

6.1.5 Invariants as Postconditions

A group of methods in a class share the same postcondition. This is a sign that these methods are more closely related than the others. The duplication of the postconditions makes it difficult to change the postconditions or add new methods, as all other methods must be checked and maybe modified. This is a sign that the class is doing too much.

6.1.5.1 Possible Refactorings

- Use the *Extract Class* refactoring to move the affected methods and their data to a new class, and promote the postconditions to invariants of the new class [**12**].

- If the methods are in two groups, one with common postconditions and the other group's postconditions are the complement of the first, specify which set of postconditions are active with a new boolean field, then promote the postconditions to invariants of the form: *new field implies postconditions one otherwise postconditions two*. This is very common when an initialise method is used to finish constructing the class.

6.1.6 Missing Mention of Model Fields

Superclasses use model fields in specifications to refer to concepts and fields that only exist in subclasses. Each subclass then declares how it implements the model field. This is very useful for interfaces, which cannot contain fields, but model field behaviour. This enables superclasses to contain common specifications without forcing implementation details on their subclasses.

When subclass fields implement a superclass's model field all specifications should refer to the model field and not its implementation. Sometimes the field which implements its parent's model field is used in specifications, in this case the specification should use the model field not the implementation field. Normally the implementation field has a lower visibility than the model field and the specification, for example when a private or protected field is referenced in a public specification. This can only be done by increasing the visibility the implementation field to the same level as the specifications, which silently creates a new local model field. When two model fields refer to the same implementation it can be very difficult for machine verifiers to know that they are aliases.

6.1.6.1 Possible Refactorings

- Replace references to the implementation field with references to their model.

6.1.7 General Specifications in Subclasses

Sometimes specifications on a class belong to its superclass or interface. Specifications should be on the most general class or interface possible in a class hierarchy. This makes it easier for clients of the hierarchy to substitute different implementations. It also simplifies comprehending the responsibilities and rewards of the hierarchy.

Sometimes a specification is placed on a subclass so it can refer to a field or query method which is not available in the superclass or interface. In this case a model field should be used to enable the specification to be moved up the class hierarchy.

When classes are refactored, with operations such as *Extract Class*, *Extract Interface* or *Push Down Method*, a specification belonging higher up the class hierarchy can end up on the subclass instead of the superclass [**12**].

6.1.7.1 Possible Refactorings

- Use *Pull Up Specification* to move the specification to the correct class.

- Introduce a model field to abstract any implementation details, and then use *Pull Up Specification*.

6.1.8 Breaking the Liskov Substitution Principle

The *Liskov Substitution Principle* states that any important property of a type should also hold for its subtypes, so any client code written for the type should work equally well on its subtypes [**30**,**6**]. This is called Behavioural Subtyping, and results in two rules for preconditions and postconditions: subclasses can only loosen the preconditions of a method; and subclasses can only tighten the postconditions of a method.

As systems evolve new classes can be added that require new specifications that break the Liskov Substitution Principle. If the system must change to support these conflicting goals, one possibility is to refactor the specifications to support all subclasses.

6.1.8.1 Possible Refactorings

- If a precondition of a method on a new class tightens the base method's precondition then if all implementations of the method and their clients can also support the precondition use the *Pull Up Specification* refactoring on the precondition.

- If a postcondition or invariant are loosened by a subclass, the class is probably in the wrong part of the class hierarchy. Move the class up the hierarchy so its base class's postconditions or invariants are looser. If such a class does not exit the *Extract Class* or *Extract Interface* refactoring can be used to create one [**12**].

6.1.9 Duplication

Duplication is a catch-all smell, and is also present in many other smells. When classes provide a lot of similar functionality or when the specifications become large, it is easy for specifications to contain duplicate clauses. When this happens once or twice in a small class it might not be a problem. However as the system evolves there is always a possibility of one clause being modified, without the appropriate change being made to the others.

6.1.9.1 Possible Refactorings

- The common clauses can be extracted to a model method. If the common clauses are preconditions a regular method might be a better solution as the clients can use this method too while ensuring the call is safe.

- Reduce the preconditions by using the *Introduce Parameter Object* refactoring [**12**].

- If the specifications are shared across a class hierarchy they can be pulled up to the common base class.

6.2 Refactoring Specifications

The list of specification smells above, together with the more general goals of refactoring can be used to predict when they will be discovered and when they might lead to specifications being refactored.

"The goals of refactoring are: improve the design of software, make software easier to understand, help to find bugs, help the engineer program faster" [**12**].

These goals highlight a number of ways specification smells can be discovered: while trying to understand a difficult specification; before extending the system; after extending the system; during a code review; during static checking; or while adding specifications to an existing class. These are explained below.

When working with complicated specifications either as a client or as in implementer it can take time to understand the specifications and how they affect your code. Other programmers in the future will probably have a similarly hard time. It is worth pausing to see if refactoring the specifications or code would make them easier to understand for the next programmer. Use the new understanding gained from studying the specifications, including how difficult they were to comprehend, to refactor them into a simpler form.

When implementing new or changing requirements it is common to have to extend the system in ways not envisioned when it was created. Before extending the system, refactor it to make it easier to extend. Test the system still behaves as expected before adding the extension. The new structure could require the specifications to change also.

If, after extending a system a smell is discovered in the specifications, the specifications and/or the code are refactored to remove the smell.

After modifying the system a colleague points out that the intention of a specification is hard to understand during a code review. The specification and/or the code are refactored to make its intention more obvious and the class easier to use.

A benefit of formal specifications is that a static checking tool like *ESC/Java2* can be used to verify that the code meets all the obligations the specifications describe [**31**]. This can uncover bugs that could take years to discover through testing alone. Static checkers prove code meets its specifications by checking all the possible states in the code satisfy the specifications. This is a computationally expensive task and suffers from combinatorial explosion as the number of possible states in a method or class grows. If the code or specifications are too complicated the checker will not have the resources to prove the code is correct. Refactoring the specifications or the code to enable the static checker to verify the class or method, guarantees the code is correct and improves the design of the system.

Refactoring can also be applied while adding specifications to existing code, as in Contracting the Design. Before writing the specifications for a subclass refactor the superclass's specifications to simplify adding the new specifications. Or after writing the specifications for an existing class a specification smell is discovered, the specification is refactored to remove the smell. When refactoring specifications during Contracting the Design the specification's existing semantics should be preserved (see section 6.3 below).

As evident from this list, specifications can be refactored before or after modifying the system. Specification smells can be a symptom of bad design as well as bad specifications. Likewise, specification smells can also trigger code, design and specification refactorings.

Most smells require more than one refactoring operation to solve. It is rare that a refactoring operation is performed on its own, especially now with tools that quickly automate the common refactoring operations. It is more common that refactoring operations are applied in a sequence to reach a desired design goal, for instance modifying the code to follow a particular Design Pattern, these sequences of refactorings are called composite refactorings [**17**]. Since specifications do not provide any functionality, but only verify the functionality of the system's code, it is expected that specification refactorings will normally be performed as part of a composite refactoring. For example first modify the specification to allow the code to be refactored in the desired way; or after refactoring the code the specifications are refactored to make better use of the new structure.

6.3 Semantics

Refactoring is defined as a behaviour preserving or semantic preserving operation. However formal specifications are also described as semantics. This section tries to disambiguate the role of semantics in both, by examining what refactoring preserves and how semantics are defined for refactoring. Then it examines how others view specification refactoring and the semantics that should be preserved. Finally, it shows the semantics of refactoring and specifications to be different.

6.3.1 Refactoring and Behaviour

The definition of refactoring above is quite loose. The primary criteria are that the change is internal, that it improves the structure, and that it preserves the external behaviour. By only changing the internal structure the refactoring operation has a boundary separating the internal structure from the rest of the system. The system outside of this boundary is not affected by the operation. The structure is improved, the improvement is deliberately left vague, and subjective. Finally the operation does not change the observable behaviour.

This makes no reference to semantics. This definition allows refactoring operations to modify specifications that are inside the operation's boundary and preserve the behaviour outside the boundary.

6.3.2 Refactoring and Semantics

Opdyke introduced the idea of refactoring and their preserving semantics:

> *"As noted above, saying that refactorings are behaviour preserving is more than saying that they produce legal programs. The versions of the program before and after a refactoring must also produce semantically equivalent references and operations. Semantic equivalence is defined here as follows: let the external interface to the program be via the function main. If the function main is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same.*
>
> *"This definition of semantic equivalence allows changes throughout the program, as long as this mapping of input to output values remains the same. Imagine that a circle is drawn around the parts of a program affected by a refactoring. The behaviour as viewed from outside the circle does not change. For some refactorings, the circle surrounds most or the entire program. For example, if a variable is referenced throughout a program, the refactoring that changes its name will affect much of the program. For other refactorings, the circle covers a much smaller area; for example, only part of one function body is affected when a particular function call contained in it is inline expanded. In both cases, the key idea is that the results (including side effect) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle."* [**11**]

The boundary separating the parts of the system affected by a refactoring operation from the rest of the system starts at the programs main function and contracts to just the extent of the refactoring. This seems to suggest that the semantics preserved by a refactoring transformation are only the semantics outside the transformation's boundary. For example when the *Inline Class* refactoring is applied the class is removed destroying its semantics, but the external semantics of the class's clients is preserved [**12**].

6.3.3 Specifications and Semantics

While an interface describes the syntax required to use an object the specifications provide the semantics of how to use it and the results to expect [**1**]. These semantics describe the behaviour of their module. It has not been made clear, formally, whether refactoring operations should preserve specification semantics or not.

One of the first to discuss refactoring and Design by Contract was Feldman [**9**], he introduces the concept of specification only refactoring operations such as *Pull Up Assertion*, *Push Down Assertion*, and *Create Abstract Precondition*. These clearly modify the specifications involved and their semantics. He also performed an analysis of the specification implications of Fowler's refactoring operations and found that 3 of the 68 or 4 % require specification modification.

Goldstein, Feldman and Tyszberowicz later discuss the implementation of some specifications aware refactoring operations in a tool called Crepe [**10**]. In this Crepe automatically modifies, moves, combined and simplified specifications.

Gabriel Falconieri Freitas and Márcio Cornélio take a programming laws approach to refactoring Java with JML annotations [**32**]. They demonstrate a pull up invariant refactoring, which preserves the original specification semantics by adding an extra clause to the invariant which only fires for the subclass originally holding the invariant. They then use programming laws to prove that these are specification and code semantic preserving operations.

6.3.4 Two Different Semantics

The semantics as described in formal specifications are different to the semantics that refactoring intends to preserve. While both describe the behaviour of the system, the semantics in formal specifications describe the behaviour of individual classes and methods and cover the whole system. The semantics that are preserved by a refactoring operation are the set of all behaviours outside the boundary of the refactoring operation; that is the whole system minus the parts changed by the refactoring.

It is clear from Opdyke's definition that he is not suggesting that refactoring preserves all the semantics described by the system's formal specifications. Limiting specification refactoring to operations that do not change the semantics of the specification constrains these operations much more than the constraints placed on code refactoring. The parallels between Test Driven Development and Design by Contract however can be used to find guidelines on the appropriate constraints [**14**]. (See section 6.4.4 below)

In systems without formal specifications the semantics of a class or method are encoded informally by all the client code that uses it and by their behaviour at run time. In systems with formal unit tests the semantics are also encoded in the unit tests, and these test the semantics of classes and methods that can be relied upon in the system.

Fowler offers the following advice for refactoring unit tests:

> When refactoring *"...you only change tests when you absolutely need to in order to cope with a change in an interface*" [**12**].

This advice can be applied to refactoring code and/or formal specifications:

> *When refactoring formal specifications you only change their semantics when you absolutely need to in order to cope with the change in structure.*

Allowing refactoring operations to modify the semantics of formal specifications is pragmatic advice to when applying Design by Contract and working on an evolving system. However this is not so pragmatic when adding specifications to an existing system, as in Contracting the Design. In this case the semantics of the system are already set and the goal of refactoring the specifications is to simply rephrase, so preserving the specification semantics is important. Then

restricting the refactoring operations and using programming laws to prove they preserve semantics is useful [**32**].

6.4 Design by Contract and Test Driven Development

The similarities between Design by Contract and Test Driven Development quickly become apparent when considering refactoring formal specifications[**8**,**14**,**9**]. Both methodologies improve the correctness, design, and documentation of the system. Additionally contracts and tests are specifications, both seek to transform requirements to machine readable form as quickly as possible, both methods are incremental, and both emphasise quality first in terms of a units of functionality [**33**].

6.4.1 Correctness

Test Driven Development relies solely on automated unit tests to ensure correctness. First, the test is written for the required interface. Then only enough of the implementation is written to let the test compile. The test is run to ensure that an empty implementation fails the test. The implementation is written as simply as possible to get the test to pass. Finally the code is refactored to improve its design [**14**]. With practice large portions of code can be written between system tests, by unit testing alone. This improves productivity, especially for server software when starting and stopping the system can take a longer than verifying the new functionality. If a bug is discovered during system testing, a unit test is written before the bug is fixed; this then verifies the bug is fixed and prevents a later regression.

Design by Contract improves the correctness of a system by specifying what is required by each unit. The specification is also in a machine readable and executable form. While this aids the correct implementation of the system, the fact the specifications are executable enables verification of the system. This verification can be run statically on the whole system, but typically happens at runtime. For runtime checking to be methodical each run must be consistent and repeatable; this requires automated unit tests or feature tests for the specifications to be checked [**33**]. However the unit tests required to exercise the specification are simpler than those for Test Driven Development as the assertions are already in the code reducing the number required in the tests [**9**]. The specifications themselves can be used to generate the unit tests [**19**].

With both Design by Contract and Test Driven Development a proportion of the errors discovered are errors in the specifications and tests respectively. These must be fixed before they can help verify the code. When Design by Contract discovers an error it is easy to assign blame; it is either in code where the specification failed, the client that called it or the or the specification itself [**3**]. This is similar to well designed unit tests; the error is either with the code just before the failed assert or in the logic of the test. One property of well designed unit tests is that each test is small. However for unit tests to assign blame well, requires more skill on the part of the programmer.

6.4.2 Documentation

The unit tests in Test Driven Development are an integral part of the documentation of the system [**14**]. They describe how the components plug together, what are valid operations, what are not, and the sequence operations are usually called in. However, tests do not provide precise documentation of class interfaces. They only capture traces of valid behaviour for scenarios of the system, and may miss the big picture, i.e., the architecture and component views. Thus, tests cannot be described as complete documentation or requirements [**11**].

Design by Contract also provides good documentation for the system's interfaces [**3**]. When using an interface it informs the programmers what is expected of the client and what is provided to the client. When maintaining the interface it informs the programmer of the behaviour that must be preserved and any constraints caused by behavioural subtyping [**11**].

Both unit tests in Test Driven Development and specifications in Design by Contract document the system. It is argued that specifications provide more complete documentation. It can be hard to infer a method's preconditions and post conditions from unit tests alone, with formal specifications these are explicitly declared and do not have to be inferred [**8**]. Also where tests provide precise specification for instances of calls to a module, contract specifications are precise and complete specification [**11**].

6.4.3 Design

With Test Driven Development design starts before the unit test is written. The interface to the new code is imagined, then a test is created to exercise this interface, valid and invalid parameters are passed to the interface and the results tested. It has been shown that code developed this way tends to exhibit high cohesion and low coupling, and that this could be down to the frequent refactoring [**11**]. It is the author's opinion, as a practitioner of Test Driven Development, that refactoring is only a part of the reason for this good design. Designing and using the module's interface before implementing it enables the programmer to see how clients will use it before the implementation details change their perspective. The code has to be modular to enable the unit to be tested on its own. If it has too many dependencies the test will be hard to write, so writing the tests first reduces these dependencies. This can be seen in the difficulty writing unit tests for existing code, where more often than not, dependencies have to be broken before an interface can be exercised and the results verified [**34**]. With Test Driven Development the code is written to be easy to test and the tests are written to make the code easier to write. In effect every interface has two clients the system and the unit tests. The greater the number of clients the more cohesive the code must be. It is this extra reuse that most improves the design of code in Test Driven Development.

Design by Contract also starts with a module's interface which is specified before the implementation is started. As with Test Driven Development, the programmer takes the client's perspective first, designing the interface and specifications, before writing the implementation. The specifications also have input into the design of the system; if they are difficult to write or contain Specification Smells then the programmer can modify the design making the specifications easier to write. If an interface's specifications are hard to write, the it is likely that the interface will be hard to use correctly. This duality between specifications and code is very similar to the duality between unit tests and code in Test Driven Development. There are differences though; while the specifications add meaning to the interface they do not play the role of an extra client. The tests are easier to write as the specifications enable runtime checking to verify the code on the interface boundaries. This enables higher level unit tests that are less sensitive to refactoring, which is good, but also allows modules with tightly coupled dependencies to be tested.

6.4.4 Semantics

With Design by Contract the semantics of the system are formally described as the system is created. The semantics for each module are complete. Even modules that are never used are described and can be shown to be correct.

With Test Driven Development the semantics of the system are informally described by the code that uses it, including the unit tests. It can be argued that the unit tests describe the only semantics that count, as these are verified by the tests themselves. If a module is never used or tested, it has no known semantics.

This lack of semantics can be seen with the *Push Down Method* refactoring, which only works if all references to the method are made via subclasses [**12**]. The method being refactored has semantics in the base class that it currently belongs to. However since nothing uses this method via the base class these semantics are lost, and they do not exist as far as the system is concerned. This is why refactoring the method is a semantic preserving operation. If this method has formal specifications the change in semantics caused by the refactoring are obvious. However the behaviour of the system remains the same, implying that these semantics are superfluous to the system.

With Design by Contract the specifications describe the full semantics of a module, however unit tests only describe specific instances of a module's semantics [**8**]. This leaves the onus on the programmer to decide which instances are important to test. Ideally by testing first; all the paths are exercised and all the boundary cases tested so tests should verify the modules full semantics. However this increases the skill required of the programmer and unfortunately since semantics are not formally recorded and verified, it doesn't ensure that the tests remain pertinent as the system and its semantics evolve. The system becomes harder to maintain when the system's semantics are not obvious from the tests or when they change over time.

Test Driven Development uses unit tests and refactoring to replace up-front design (sometimes pejoratively called "big up front design") [**33**]. The design is not planned in advance but rather discovered as requirements are implemented, this is sometimes referred to as Emergent Design. Design by Contract supports up-front design, but is usually used with an iterative approach, where each module is specified and then implemented. It can also support the finer grained emergent design of Test Driven Development and other Agile methodologies. Writing the specifications first does not require the whole system or even the whole module be specified before any code is written. Specifications, like tests, can be written as required and refactored when new features require a different design.

6.4.5 Duplication of Effort

Both Test Driven Development and Design by Contract required a larger initial investment than simply creating the interface without tests or specifications. However they both reduce the cost of getting the interface to work and keeping it working over time. This investment starts to pay for itself quickly as the system gets larger, is maintained longer or the cost of errors is high [**7**].

6.4.6 Refactoring

Refactoring plays a pivotal role in Test Driven Development [**14**]. The system is designed in an emergent way. There is just enough design and just enough system to implement the test cases. The system is not designed with the future in mind; and as a result, extending the system usually involves some refactoring to adjust the design to incorporate the changes.

While refactoring has not been as pivotal in Design by Contract, this does not mean that systems developed using Design by Contract have not been refactored. In fact, systems of all types were refactored before the practice was formalised into a technique. What is new is the confluence of Design by Contract, agile practices and emergent design. This results in the greater importance of refactoring and refactoring support for Design by Contract.

A system cannot be refactored without tests to ensure that the observed behaviour is not changed. Often refactoring can lead to the creating of new tests. For example if there is no test for a part of the system being refactored then a test must be created before refactoring, or if a refactoring splits a class in two, new tests can be created for each one, while the original tests show that the pair together continue to function as before.

Just like tests, new specifications can be added to the system during refactoring operation. For example, following an *Extract Method* operation new pre- and postconditions can be derived from the context of the extracted code [**12**]. One advantage that specifications have over tests is that these can be derived automatically by the refactoring tool [**10**]. As well as deriving new specifications from the original context it might even be possible in the future to derive extra meaning from the intent of the refactoring operation. For example, the *Self Encapsulate Field* refactoring can be used to introduce lazy initialisation. If this is the case the refactoring operation could change invariants and the initially clause to reflect this [**12**].

In Test Driven Development the tests make up part of the documentation for the system [**14**]. As a result the test code must be clean and well designed. This encourages programmers to refactor their test cases as well as their tests. When tests are refactored, the code they are testing should not be modified, as the code does not depend on the tests. When the code is refactored the tests should not change either. However since tests do depend on the code this is not always possible. For example when changing the external interface of a class all clients of the class with have to change, including the tests. Sometimes a smell in a unit test is a symptom of a deeper problem in the code, so the code is refactored, which then modifies the test.

Likewise, specifications form part of a system's documentation and should also be clean. While code refactorings might introduce new specifications or modify existing ones, specification refactorings should not modify code. This does not mean that they cannot change the meaning of the specifications, just that the code should meet the new semantics already. If the code has to change the refactoring is characterised as a code refactoring not a specification one. For example, the *Pull Up Postcondition* refactoring can only pull up the postcondition if the method containing the postcondition is available in the destination class or a superclass of the destination class. If the method is not available the *Pull Up Method* operation must be used first, but this is a code refactoring not a specification one [**12**]. And as pointed out above specification smells just like unit test smells can be symptoms of design problems in the code.

7 Requirements

Before implementing automated refactorings for Java Contracts the refactoring operations have to be defined and their requirements analysed. This section defines the *Pull Up Specification* and *Push Down Specification* refactoring operations. These refactorings were then applied to a large number of test cases by creating before and after refactoring examples (see Appendix 11.2 below). The results of examining these test cases highlight the roles played by specification type, specification structure, java syntax, and call sites in automating these operations.

Note: the refactoring recipes use the term "class" generally to refer to a Java class or a Java interface, where this is not the case the terms "concrete class", "abstract class" and "interface" will be used.

Note: the recipes are described in the same style as Fowler's, that is as if they were being performed manually by a programmer [**12**].

7.1 Pull Up Specification

Safely move the selected specification to a superclass. The specification is still available to the subclass.

7.1.1 Motivation

Occasionally general specifications are placed in a subclass and not in the more general superclass or interface where they belong. This is often because the information required to define the specification is present in the implementation class. This causes problems for *Behavioural Subtyping*, as clients must refer to the implementation class or classes containing the specification to make use of its guarantees, rather than refer to the more general superclass or interface [**30**]. This increases the coupling in the system and the complexity of clients. Pulling up the specification to a class higher up the hierarchy enables clients to rely on its behaviour polymorphically, reducing the coupling and complexity.

Another variant on this is where specifications are duplicated on two or more classes in the same hierarchy. Some small duplication might be acceptable, but it can lead to code and specifications that are hard to maintain in the future. Duplicate specifications also cause similar *Behavioral Subtyping* problems as general specifications. Pulling up the specification has similar benefits too.

A specification can break the *Liskov Substitution Principle*, by strengthening its preconditions or weakening its postconditions or invariants [**30**]. Pulling up the specification concerned can make the class hierarchy comply with the principle again.

Before pulling up a specification all fields and methods referred to by the specification must be checked to ensure they are available in the superclass. This can be accomplished by *Leaning on the Compiler*: move the specification quickly and rebuild; the compiler will highlight the errors [**34**]. When specifications use references that are not available in the superclass they can be abstracted using a model field. The pulled up specification then refers to the model field while the source class declares that the original reference implements the field. This could be achieved with an *Introduce Model Field* refactoring.

Invariant specifications are syntactically straight forward to pull up. They can be moved from the subclass to superclass verbatim once the above check has passed. Preconditions and

postconditions are not so straight forward as they require a method to host them. This method must be defined in the superclass. If the method is not defined, the *Pull Up Method* refactoring is required, to create it or an abstract version before the specifications can be pulled up. When the method is defined in the destination it can be present, not present, abstract, or not abstract. If the method is present and not abstract the specification must be added to the destination method's existing specifications. If the method is present and abstract the specification must be added to the method in the destination's contract class. If the method is not present and not abstract a method must be created in the destination class, this method simply calls `super()`, then the specifications are added to the method. If the method is not present and abstract the specifications are added to the method in the superclass's contract class (because of inheritance, the method can exist in the contract class without existing in the abstract class or interface).

Sometimes it is not possible to pull up a specification straight away, the specification might clash with those of other subclasses, external runtime or static checking is required to prove this. When this is the case the *Extract Subclass* or *Extract Interface* refactorings can be used first to create a class or interface to serve as the destination for the pulled up specification. In this case the clashing subclasses would not share the new class or interface [**12**].

7.1.2 Mechanics for Invariants

- Search for identical or similar invariants in the rest of the class hierarchy.
    - If the invariants are not the same use algorithm substitution on each one to make them the same.
- Copy one of the invariants to the destination class.
    - Use the compiler to find any references that are missing in the superclass,
    - Replace each missing reference with a model field.
- Delete one subclass invariant.
- Compile, test and verify
    - Declare the implementation for each model field that is missing as they are found.
- Keep deleting subclass invariants and testing and verifying until they are gone.
- Take a look at the callers of all the effected subclasses to see if they can be changed to reference the superclass instead.

7.1.3 Mechanics for Preconditions and Postconditions

- Search for identical or similar specifications in the host methods of rest of the class hierarchy.
    - If the specifications are not the same use algorithm substitution on each one to make them the same.
- Ensure the superclass has a host method. If not create one based on these rules:

    - If the host method does not exist further up the hierarch create an abstract version (This is like pulling up an abstract method). Create an implementation of the method in the superclass's contract class.

    - If the host method does exist further up the hierarchy and is abstract, create an implementation of the method in the superclass's contract class.

    - If the host method does exist further up the hierarchy and is not abstract, create a host method which calls and returns its super implementation.

- Copy one of the specifications to the host method on the superclass.

    - Use the compiler to find any references that are missing in the superclass

    - Replace each missing reference with a model field.

- Delete one subclass specification.

- Compile, test and verify.

    - Declare the implementation for each model field that is missing as they are found.

- Keep deleting subclass specification, testing and verifying until they are gone.

- Take a look at the callers of all the effected subclasses to see if they can be changed to reference the superclass instead.

    - If a precondition was pulled up ensure that all callers of the method meets the precondition.

7.1.4 Example

Consider a class hierarchy of `Vehicles`, the `wheels()` method returns an array of the wheels associated with the `Vehicle`. The size of this array is guaranteed by the postconditions of each subclass. Both `Car` and `Truck` return arrays with four wheels, thus they share the same postcondition. This is a candidate for *Pull Up Specification* but the postcondition cannot be moved to the `Vehicle` class as this would clash with the `Hovercraft` and `Motorcycle` classes.

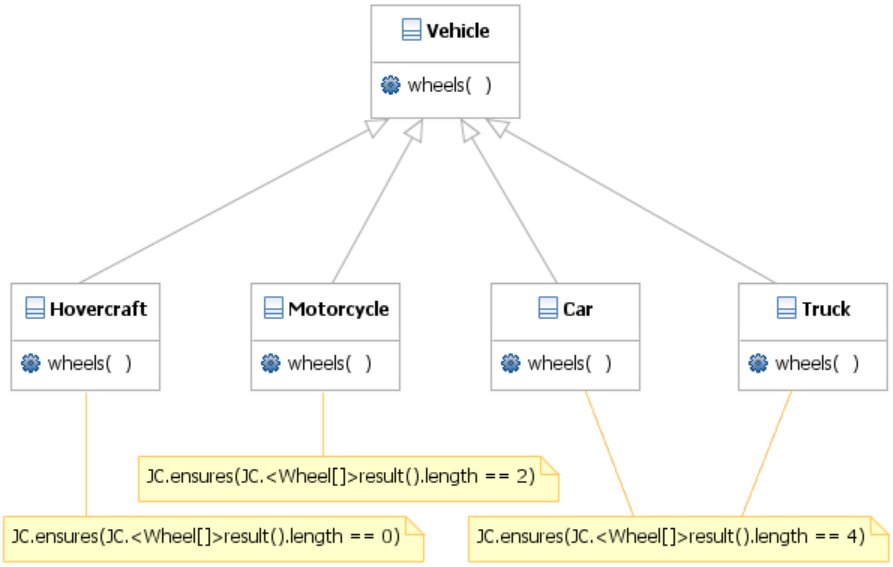


Figure 1: Before Pull Up Refactoring

The clash is solved with the *Extract Subclass* refactoring which introduces the new class `QuadVehicle`, a subclass of `Vehicle`, both `Car` and `Truck` now inherit from this instead of `Vehicle`. Note if `Vehicle` was an interface the *Extract Interface* refactoring could also have been used.

The postconditions are then pulled up to this class.

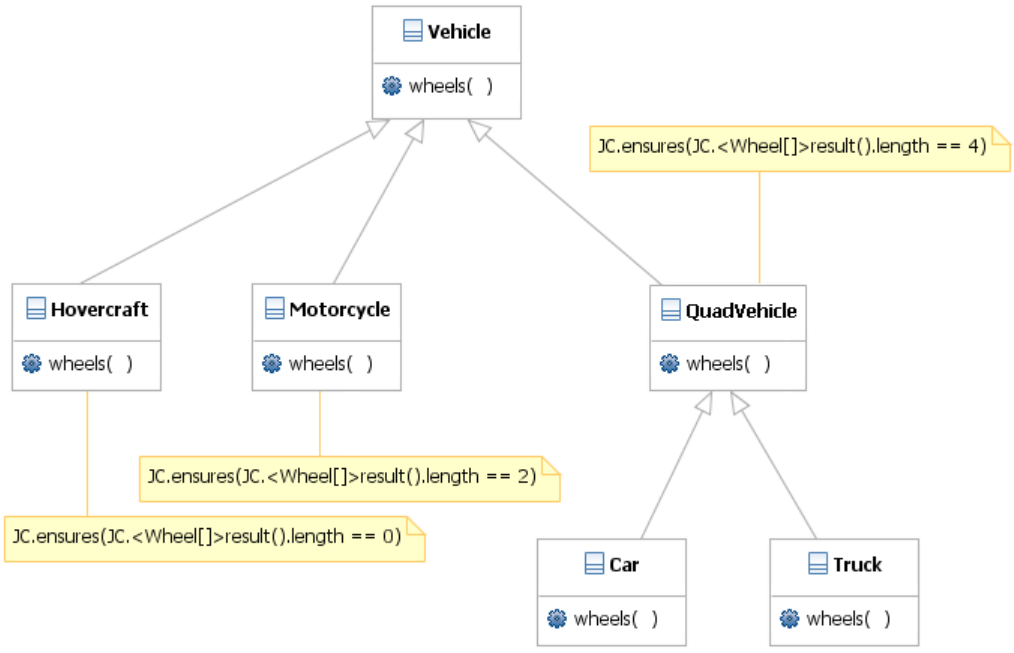


Figure 2: After Pull Up Refactoring

7.2 Push Down Specification

Safely move the selected specification to one or more subclasses. The specification will be available to the subclasses but not the superclass.

7.2.1 Motivation

Occasionally the specifications on a class are too strong for its subclasses. This naturally happens when extending a class hierarchy to support new features not originally planned. This situation can either be discovered by the programmer while adding a new subclass or by verification software after the subclass has been added.

Another variant of this is when two classes in a hierarch evolve independently of each other; they might no longer be able to meet all their specifications.

Pushing down specifications is easier than pulling them up. As all fields and methods referenced in the specification must already be available to the subclasses, removing the need to check them. Private and package private fields cannot be referenced in public and protected method specifications. The implementation details in heavyweight specifications, which can contain references to private members, are not inherited.

Invariant specifications are syntactically straight forward to push down, it can be moved from the source class to destination class verbatim. Preconditions and postconditions are not so straight forward as they require a method to host them. While the method is defined for subclasses, an implementation must exist for the specification to be moved to. This implies that the method cannot be final. When the method is defined in the subclass it can be present, not present, abstract, or not abstract. If the method is present and not abstract the specification is be added to the subclass method's existing specifications. If the method is present and abstract the specification must be added to the implementation method in the subclass's contract class. If the method is not present and not abstract a method must be created in the subclass class, this method simply calls `super()`, then the specifications are added to the method. If the method is not present and abstract the specifications are added to the method in the subclass's contract class.

Ordinarily it is not safe to push down preconditions as this will create subclasses with stronger preconditions than their parent and thus break the *Liskov Substitution Principle*.

7.2.2 Mechanics for Invariants

- Copy the method implementing the invariant from the superclass to all subclasses.
- Delete the method implementing the invariant from the superclass.
- Compile, test and verify.
- Take a look at the callers of all methods on the superclass, if they depend on the invariant then change then to refer to the correct subclass containing the invariant.
- Remove the invariant from subclasses that don't need it.
- Compile, test and verify.

7.2.3 Mechanics for Postconditions

- Ensure all subclasses have a method to host the postcondition. If not create one based on these rules:
    - If the host method does exist further up the hierarchy and is abstract, create an implementation of the method in the subclass's contract class.
    - If the host method does exist further up the hierarchy and is not abstract, create a host method which calls and returns its super implementation.
- Copy the postcondition to the host methods on all the subclasses.
- Delete the postcondition from the superclass.
- Compile, test and verify.
- Take a look at the callers of all methods on the superclass, if they depend on the postcondition then change them to refer to the correct subclass containing the postcondition.
- Remove the postcondition from subclasses that do not need it.
- Compile, test and verify.

7.2.4 Example

A bank's accounting system is being extended to support overdrafts on current accounts. Currently all accounts must have a positive balance, this is enforced with the invariant `jcInvPositiveBalance`.

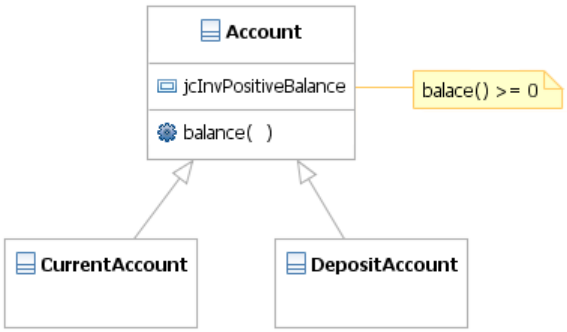


Figure 3: Before Push Down Refactoring

This invariant prohibits a negative balance on a `CurrentAccount` with an overdraft so is pushed down and modified and before the overdraft functionality is added.

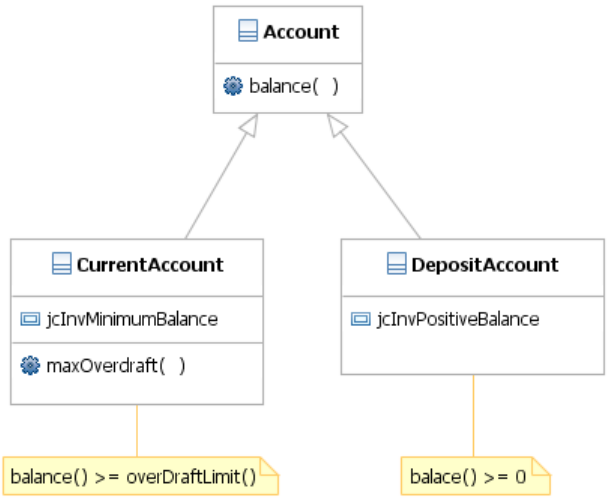


Figure 4: After Push Down Refactoring

7.3 Specification Type

The specification type is one of the primary variables in defining automated refactoring operations for Java Contract. There are two types of Java Contract specifications: expressions and declarations.

An expression can either be a Java statement for a precondition or postcondition, or a method for an invariant. Expressions have full access to the fields and methods of their host class at the same visibility level as themselves (see section 7.5.2 below). The Java Contract expressions supported are assert, assignable, invariant, precondition and postcondition.

In some circumstances specification expressions have to use abstract concepts that are not part of the Java interface of their host class. These abstract concepts are described in declarations which are then referenced in expressions. Subclasses later specify how they implement the declaration; these are like abstract methods for specifications. The Java Contract declarations supported are model field, model method and model specification

All specifications use expressions while some difficult cases use declarations to abstract common code. The refactoring tool concentrates on expressions as these cover the majority of the cases in Design by Contract. Declarations are also covered where it is easy to do so, however a future version could be more intelligent when refactoring these, for example by introducing model fields as required when applying *Pull Up Specification*.

7.4 Specification Structure

All specification expressions that apply to a Java method are implemented as Java statements in those methods. These statements are static calls methods on the `org.jmlspecs.jc.JC` class. The statement specifications are assert, assigns, precondition and postcondition and specification case.

A specification case is used to group a set of expressions together, marking their visibility and whether they are for normal operation or for exceptional circumstances.

When statements are refactored with the *Pull Up Specification* or *Push Down Specification* operations they are moved to the beginning of the appropriate method in the superclass or subclass respectively. When a specification case is refactored the statements associated with it are moved also. If a simple statement within a specification case is refactored it is moved to the appropriate case in the appropriate method; if one does not exist in the destination one is created.

Expressions that apply to the class and all declarations are implemented as methods marked with Java Contract annotations. The method specifications are invariant and all declarations. When method specifications are refactored with the *Pull Up Specification* and *Push Down Specification* operations they are moved to the appropriate class.

Some other specification details pure and spec-public are coded using JML 5 Annotations (these cannot be refactored) [**22**].

The structure of the specification, whether it is a statement or method based specification is the primary difference between Java Contracts when refactoring. For example operations on method based specifications behave the same whether the specification is an invariant or a declaration.

7.5 Syntax

The Java syntax of specifications as well as the methods and classes that host them also affect the technical requirements for the refactoring operations.

7.5.1 Static or Instance

There can be static or instance specifications just like methods and fields. For example, an invariant for static data must be declared static; in this case it affects the static data of the class.

Static invariants are implemented in static methods. These can be pulled up and pushed down just as regular invariants can.

Static methods do not follow the same overriding rules as instance method. Where an instance method in a subclass overrides or replaces a method with the same name and signature in its superclass, a static method in a subclass with the same name and signature as its base class is a completely different method and is referenced explicitly when called. This means that static methods do not inherit preconditions and postconditions from their base classes, so the *Pull Up Specification* and *Push Down Specification* refactoring operations are not available for preconditions and postconditions of static methods.

7.5.2 Visibility

Java Contracts that are encoded as statements or annotations share the visibility of their associated method or field. Method based Java Contracts also have a visibility, which is the same as the visibility of the method that specifies them. The visibility rules effect what code or specifications can rely on a specification. For example, external client code can only rely of the public specifications of a class and a public postcondition cannot refer to a protected model field.

The following rules are applied based on visibility and assume that all members accessed by the specification are accessible in the destination.

Public: Statement and method specifications can be transformed with no changes to the visibility.

Protected: Statement and method specifications can be transformed with no changes to the visibility.

Private: Statement specifications cannot be transformed as private methods do not support overriding, but method based specifications can be transformed. Pushed down methods are left private as the visibility does not affect their availability to the source. Pulled up methods are changed to protected to be available to the source.

Package private: Statement specifications can be transformed if both source and destination are in the same package, otherwise they cannot as package private methods only support overriding for classes in the same package. Method specifications can be transformed. The visibility of the transformed specification is unchanged for push down operations as the destination specification is not available to the source. The visibility of the transformed specification is changed to protected if the source and destination are not in the same package.

7.5.3 Source and Destination Type

Java Contracts are specified differently for Class, Interface or Abstract Class. This affects the parsing and rewriting rules for the contracts.

Class: The specifications are inline with the Java code for the class. Invariants are stored as methods in the class. Precondition and postconditions are stored in the method they affect. The current implementation concentrates on classes

Interface: The specifications cannot be inline with the interface definition as interfaces do not allow fields or methods implementations. To work around this fact Java Contracts use an annotated implementation to specify the interface specifications.

Abstract Class: These are somewhere between regular classes and interfaces. Some specifications are inline but the specifications for abstract methods are coded like interface methods via an annotated implementation class.

The addition of an annotated implementation class to code specifications for interfaces and abstract classes adds a lot of overhead to the parsing and rewriting rules for the refactoring operations. While ideally these rules would be included it was felt that their addition did not add enough to the initial version to justify the cost.

As a result, ordinary classes are fully supported, interfaces are not supported and abstract classes only support invariants, statement specifications where neither the target or destination method is abstract.

Additionally when refactoring a precondition or postconditions if the method is not present in the destination class a new method is added, this method includes the specification then calls super implementation, this only works if the super method is not abstract

7.6 Calling Site Support

The *Pull Up Specification* and *Push Down Specification* refactoring operations modify the specifications of classes and methods. These changes will affect the client code that uses these classes and methods. The parts of the client code that use a refactored class or method are called the calling site. The calling sites will have to be analysed to decide if changes to the semantics of the class or method require changes at the calling site too.

If a method's preconditions are strengthened after a *Pull Up Specification* refactoring adds a new precondition from a subclass, all call sites must ensure that they meet the new preconditions. Likewise, if a method's postconditions are weakened after a *Push Down Specification* refactoring removes a postcondition from a superclass, all the call sites must ensure that they do not rely on the missing postcondition. Finally if a class's invariant is removed from a superclass after a *Push Down Specification* refactoring, all the call sites must ensure that they do not rely on the missing invariant.

The call sites for the first and second cases above are all the client code that calls the affected method, via a reference to the affected class. The affected method is the method containing the specifications that are refactored. The affected class is the source class for postconditions and the destination class preconditions.

The call sites for the third case above (invariants) are all of the client code that calls any method of the affected class. The affected class is the source class of the *Push Down Specification* refactoring operation.

7.6.1 Static Checking

Ideally, the refactoring tool would use a static checking tool like ESC/Java2 to highlight only the call sites where the specifications are broken, and optionally provide fixes for these cases [**31**]. However this would be a large and difficult task requiring changes to ESC/Java2.

7.6.2 Manual Checking

Another option is to rely on manual checking by the programmer to find and fix the call sites after each refactoring operation. To aid the programmer, the refactoring tool would highlights all the affected call sites. This is done by adding a comment at the end of each line describing the reason the call site must be analysed. This uses the standard Eclipse "TODO" style comments which adds an entry to the "Tasks" view in Eclipse. This enables the programmer to quickly examine all the affected call sites.

Support for manual checking of the call site is planned for a future version.

7.7 Plain Java Refactorings and Java Contracts

Since Java Contracts are expressed in plain Java code, it is possible to refactor them using standard Java refactoring operations. These are already available in Eclipse. While invariants can be pulled up and pushed down like this, the Eclipse Java refactoring tools are not capable of pulling up or pushing down preconditions and postconditions.

Plain java refactorings take no account for the semantic and methodological implications of Java Contracts. For example they do not support the concepts of model fields and call sites.

8 Construction

This section describes the details of constructing the refactoring tool for Java Contracts. The first steps begin with understanding the Eclipse Platform. Then the structure of the plug-in, its actions and operations are explained,

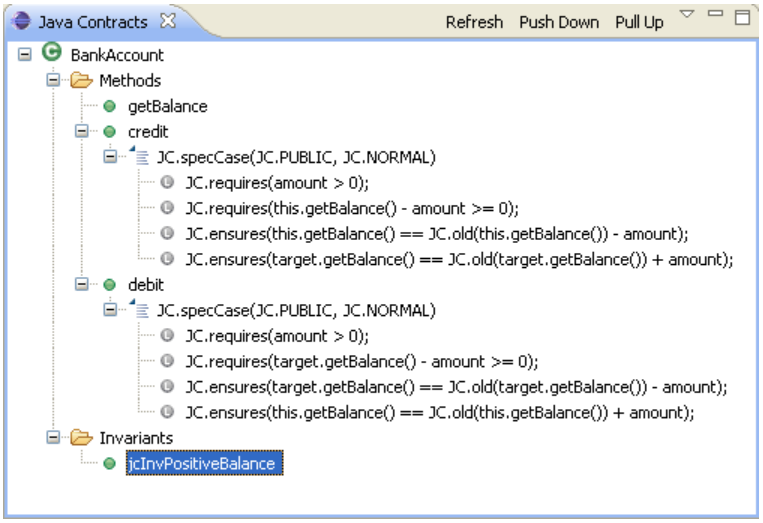
8.1 Understanding Eclipse

The first process to constructing an Eclipse Plug-in is to understand the Eclipse platform and the services it offers developers.

8.1.1 First View

Most Eclipse Plug-in tutorials start with adding a menu item or creating a simple view [**35**,**26**]. A view is a window other than an editor window that provides information and other operations. One of the requirements is to provide users with a higher level view of the Java Contracts specified in the source file.

This provides an opportunity to learn the basics of the Eclipse plug-in architecture: the Plug-in manifest file; creating an Eclipse user interface; adding actions to tool bars and menus; and accessing the current selection.



The Java Contracts view provides an outline of the Java Contracts in the currently selected source file. This is similar to the standard Java Outline, but includes details of all the Java Contract specifications in the source file. This is useful for getting an overview of the specifications in a class and seeing the results of refactoring operations. The view also provides the Pull Up and Push Down actions that commence specification refactoring operations.

8.1.2 Test Driven Plug-in Development

The Eclipse Plug-in Development Environment (PDE) is an extension to the main Java Development Environment with added features for plug-in development [**36**]. One of these features is a launcher to quickly run and debug plug-ins without installing them into a separate Eclipse installation. It also includes a custom JUnit launcher to run and test plug-ins in a full Eclipse environment [**37**]. This automatically launches the Eclipse platform, loads the plug-in and its dependencies and executes the unit tests in the environment and closes Eclipse when the

tests are finished. This enables the unit tests to depend on all the Eclipse services the plug-in requires, which makes the tests easier to write.

The book *Contributing to Eclipse* describes a test driven approach to plug-in development [**26**]. The authors demonstrate how to test plug-in manifest entries, user interface elements such as views and actions as well as java classes. This approach was applied to the construction of the Refactoring Plug-in.

The test cases for the Refactoring plug-in reused the `TestProject` code demonstrated in the book. This class simplifies Java project creation and manipulation for unit tests. The Refactoring Plug-in unit tests use this to create fixtures for each test. For example the fixture for the `AbstractAstVisitorTest` creates a new Java project, adds the Java source files for the Bounded Thing example [**5**,**20**]. Now the tests can rely on the structure of these files while testing the visitor. When the tests are finished the source files and project are deleted to prevent them interfering with other tests.

8.1.3 Java AST

Once the basic view was constructed it had to be populated with a high-level view of the Java Contracts in the source file. The user interface for this view is an extended version of the standard Java outline view provided by the JDT showing the Java Contract outline of the source file rather than the Java code outline.

The *Java Outline* view, like the rest of the JDT user interface, uses the lightweight Java code model provided by the JDT. This is a coarse gained model that includes the high-level Java structures such as classes, methods and fields, but not statements. This model is easy to create from both Java source files and classes and light on CPU and memory resources. However it is not rich enough to build the information for the Java Contract view.

The Java Contract Outline must parse the contents of methods for statement based contracts and resolve the annotations bound to methods for method based contracts. This requires the richer Java Abstract Syntax Tree (AST). The Java AST is also the model used by the refactoring tools to understand and rewrite Java code. Building and processing the AST is much slower and requires more resources than the simple Java model. It is recommended that only one parser is used at a time; if more than on file is to be parsed they should reuse the same parser and be processed one after the other.

The AST contains nodes for every element of a statement or declaration. For example the following statement:

```
JC.requires(data != null);
```

Results in the following AST node hierarchy:

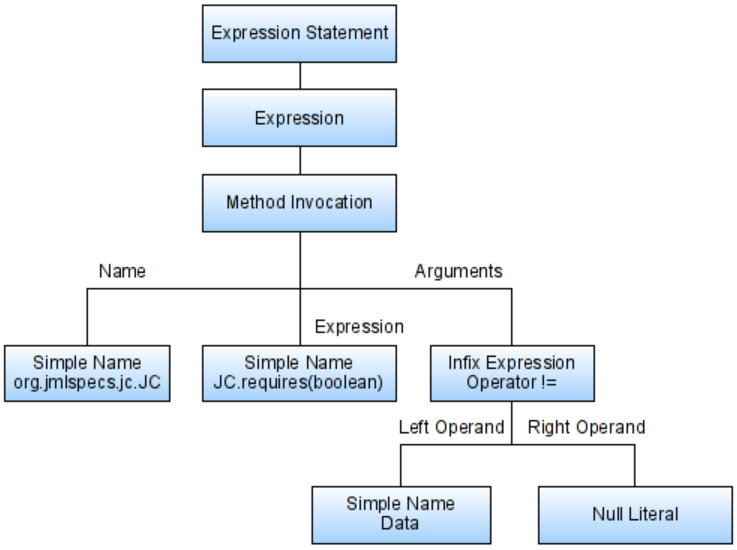


Figure 5: Simple Java Contracts AST

8.1.4 Language Tool Kit Refactoring Library

Eclipse refactoring operations are orchestrated by the Language Toolkit (LTK), which is part of the core Eclipse Platform [**29**]. This API glues the three stages of refactoring operations together. These are: invocation, user interaction and performance. The invocation is triggered by eclipse actions in views, editor or menus. The user interaction is captured with a wizard dialog box. Finally the refactoring is performed by refactoring controller class. Most of the common functionality is provided by the LTK, once the action has started the process the LTK manages it until the refactoring operation is completed or cancelled.

The refactoring operations that ship with the Eclipse Java Tools are similarly orchestrated by the LTK, however they also leverage other internal utility classes to abstract some more of the details specific to Java refactoring. Most of the refactoring operations share common refactoring controllers class which delegate specifics of the operations to other classes. While this design shows the flexibility of the LTK the extra complexity involved made learning from the Java refactoring source code more difficult. Online articles demonstrating simple refactoring operations were found to be more useful [**28**,**29**].

8.1.5 Extending the Java Refactoring Operations

Ideally the specification refactoring operations would be integrated with the standard refactoring operations supplied with Eclipse. For example in the refactoring menu on the main menu bar and in the context menu of Java code editors. This was attempted, but unfortunately JDT does not always follow the open extensible model practiced by lower layers such as the Eclipse Platform. This is a deliberate decision by the JDT programmers who renege on extensibility if it compromises user experience.

This is not a large inconvenience as the Java Contracts view is a natural starting point for specification refactoring but tighter integration with the JDT would be nice in the future.

8.2 Structure

The refactoring plug-in is broken down into a number of components. These components are grouped into logic components (green) and user interface components (blue). The user interface components can access and manipulate the logic components, but the logic components cannot access the user interface. The logic components perform most of the work.

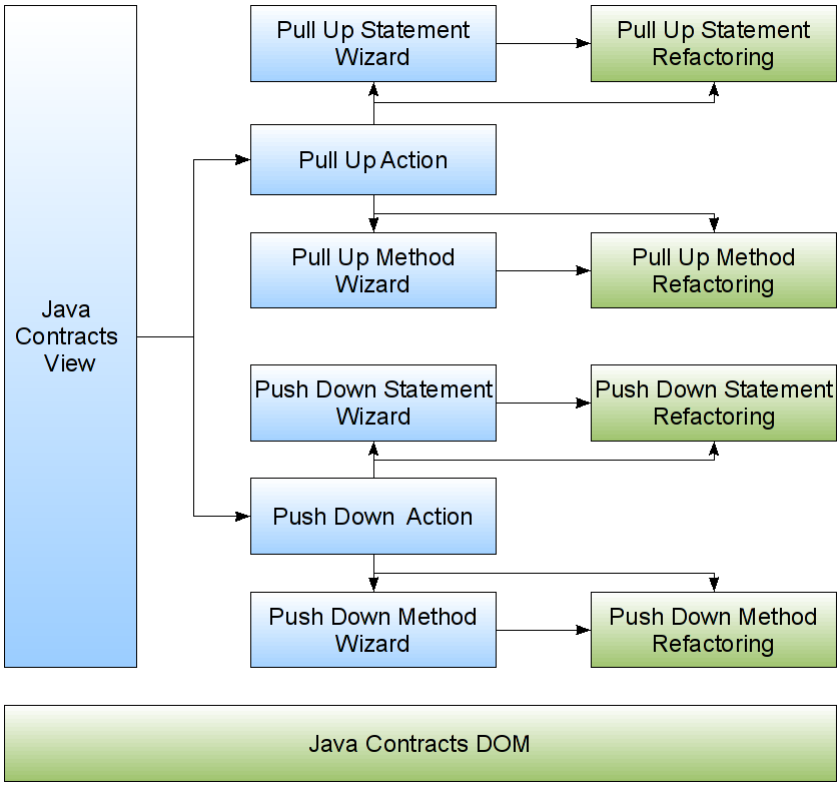


Figure 6: Refactoring Plug-in Structure

The Java Contracts DOM uses the Java AST to construct a Java Contracts document object model. This is then used by the other components. The Java Contracts view constructs a DOM for the active source file and displays it in a tree. The actions are triggered from within the view. Each action decides whether to executed statement or method refactoring operations based on the selected elements in the view. The actions construct a wizard and refactoring object and pass these to the LTK for execution. The wizard prompts the user for input and updates the refactoring object. When the wizard completes the refactoring object executes the refactoring.

8.3 Actions

All refactoring operations are triggered by actions. Actions respond to changes in selection in the Eclipse workspace and trigger their operation when they are invoked. The actions associated with a view are added to the view's toolbar.

Each action associated with a view is added to the view's toolbar. The action class must implement the `IViewActionDelegate` interface. This interface provides the init, `selectionChanged` and run operations. The init method initialises the action when the view is

created. The `selectionChanged` method is called each time the user selects an object or text in the user interface, the selection information can used to enable or disable the action, and can be stored as parameters for the run method. The run method is called when the user invokes the action, and executes its function.

The four refactoring operations are mapped to two actions, *Pull Up* and *Push Down*. The *Pull Up* action decides whether to invoke the *Pull Up Specification Method* or *Pull Up Specification Statement* operation based on the selection, the *Push Down* action does likewise. This grouping of refactoring operations is common in the Eclipse IDE. It simplifies the user interface reducing the options displayed to the user. The IDE selects the appropriate action based on the context the action is called from.

The infrastructure to support this operation grouping is common to both actions. To reduce duplication this commonality is abstracted using the *Factory Method* and *Template Method* design patterns [**38**]. `AbstractRefactoringActionDelegate` is the base class for both actions. It implements the `IViewActionDelegate` interface, the subclasses implement methods to test the selection and create the refactoring operations logic and user interface components.

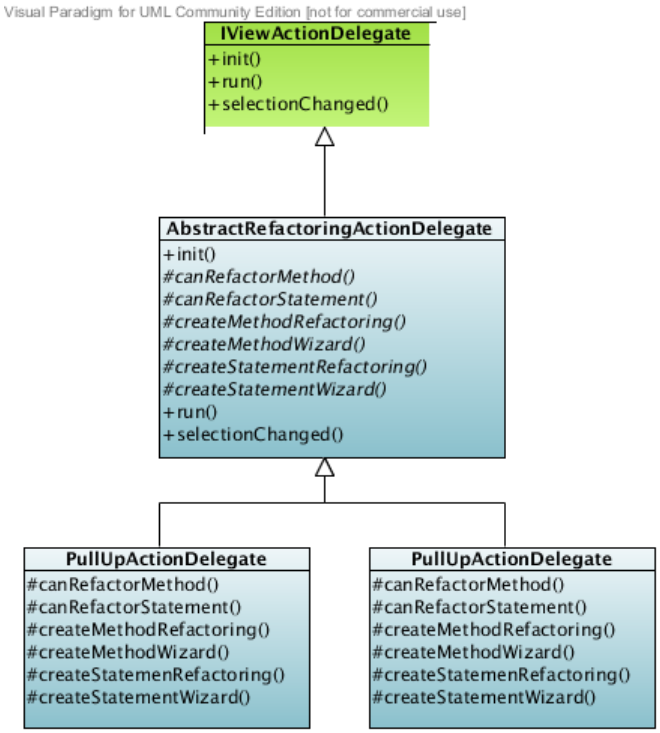


Figure 7: Action Classes

When the selection changes the `selectionChanged` method on `AbstractRefactoringActionDelegate` is executed, it examines the selection and configures the action. If the selection only contains a class or interface the source type is set and method specification refactoring operation is enabled. Otherwise if the selection contains methods, the method specification refactoring operation is enabled; the selected methods are stored; and the source type to is set the parent class of the selected methods (all methods must share a common enclosing type). Each method is passed to the `canRefactorMethod` of the subclass enabling the subclass to decide if it knows how to refactor the method. Otherwise if the selection contains

statements, the statement specification refactoring is enabled, the selected statements are stored, the source method of the operation is set to the method enclosing the statements and the source type to the enclosing type (all statements must share a common enclosing method). Each statement is passed to the `canRefactorStatement` enabling the subclass to decide if it knows how to refactor the statement. Finally if the selection contains none of these disable the action.

When the action is invoked the run method on `AbstractRefactoringActionDelegate` is executed. If the method specification refactoring is enabled the refactoring operation is created by calling `createMethodRefactoring`, the wizard user interface is created by calling `createMethodWizard`. The refactoring operation is initialised with the source type and selected methods. If the statement refactoring is enabled the refactoring operation is created by calling `createStatementRefactoring`, the wizard user interface is created by calling `createStatementWizard`. The refactoring operation is initialised with the source type, source method and selected statements. In both cases the wizard is bound to the refactoring operation and then launched to guide the user through the refactoring operation. If no refactoring is enabled the action does nothing.

8.4 Refactoring Operations

The refactoring operations are generalised to those that operate on methods or statements. The basic algorithm for handling Pull Up method and Push Down method are the same, likewise for statement based specifications.

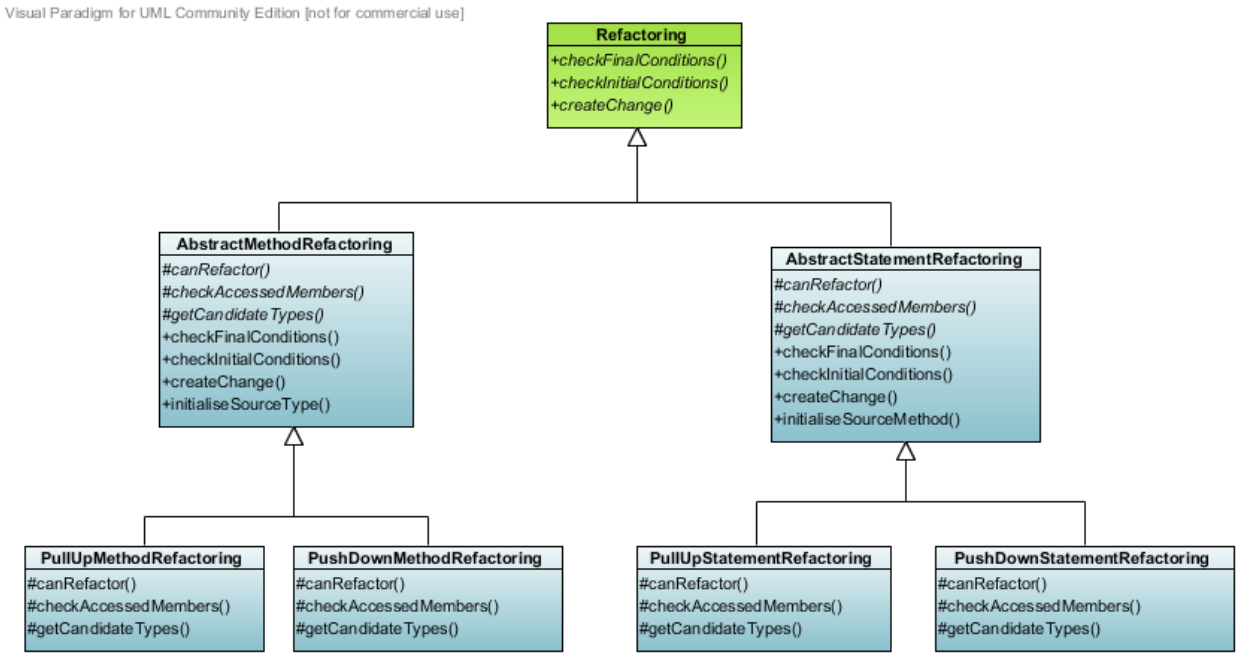


Figure 8: Refactoring Classes

The `AbstractMethodRefactoring` class implements the core algorithm for the Pull Up and Push Down Method Specification. It delegates the Pull Up and Push Down specific parts to its

concrete subclasses. The refactoring algorithm centres on the management of the source type, candidate types, available methods, destination type and selected methods. The source type is the class containing the method specifications to be refactored, this is set be the action when the Refactoring is created. The available methods are the list of method specifications that can be moved, these are initialised after the source type. The candidate types are the set of all classes that the method specification can be moved to, this is initialised after the initial checks are performed, and the logic for initialising this is implemented by the concrete subclasses. The destination type is the class the method specifications will be moved to, this is selected by the user in the wizard. The selected methods are chosen from the list of available methods by the user in the wizard.

The `AbstractMethodRefactoring` class implements the core methods on the `Refactoring` class: `checkInitialConditions`, `checkFinalConditions`, and `createChange` as well as providing the utility method `initialiseSourceType` used by both the Push Up and Push Down actions.

Once the actions have created their Refactoring instance they call the `initialiseSourceType` method to initialise the instance with their selection context. This method clears all previous settings, generates a Java Contract AST for the compilation unit, and uses the AST to populate the list of available Java Contract method based specifications that can be refactored. Source

Once the action has passed the Refactoring and wizard to the LTK, the LTK calls the `checkInitialConditions` method. This method checks the source type, and then asks the concrete subclass to populate the list of candidate destination types. The candidate destination types for a *Push Up* refactoring are the list of super-types of the source, and for a *Push Down* refactoring are a list of the subtypes of the source.

Once the initial conditions have been checked the wizard is displayed to the user. This enables the user to select the destination type from the candidate list and selects the methods to be refactored. The wizard passes this information back to the refactoring instance. Once the user finishes the wizard in the LTK calls the `checkFinalConditions` method and if this is successful the `createChange` method

Most of the refactoring logic is performed in the `checkFinalConditions` method. This performs the final checks. These can take a long time to execute and are too slow to perform in real time while the user configures the refactoring wizard. Once these checks are performed this method also performs the source code rewriting required for the refactoring, the results are then stored as a set of source file deltas. Performing the rewriting in this method is a common idiom adopted by refactoring operations as the final checks can provide input to the rewriting and the rewriting can provide meaningful feedback if an error is discovered [**28**,**29**].

The `checkFinalCondition`, clears any previous file changes, checks the selected nodes, checks the destination type, checks the accessed methods and fields accessed, and then rewrites the files. All the methods and fields accessed by the selected specifications are then checked to ensure they are available in the destination type. This uses the advances search capability of the JDT's Java Model, a quicker and lighter weight solution to iterating the code with an AST. The source and destination files are then rewritten.

The rewrite step creates the AST Parser and rewriter objects, parsers the source file moves the modifies the source file, parses the destination file if it is different to the source file then modifies the destination file. To rewrite the source the JDT AST nodes are found for each selected JML AST Method node. Each JDT node is stored and marked for removal. All the imports required

by the each node are then stored. To rewrite the destination a `ListRewriter` is used to modify the declarations in the destination type. The selected specification methods are added to the type's declarations. Then the imports required are merged into the destinations imports. Finally are the rewriting steps are stored.

If these checks pass the LTK calls the `createChange` method, this simply batches the stored rewriting steps constructed in the `checkFinalCondition` step returns them as a set of source file deltas which the LTK safely applied to the files, taking special care of files open in editors and also any source control considerations.

9 Conclusion

Others have demonstrated the feasibility of automated formal specification refactoring, this dissertation adds to their work. This is the first tool to add refactoring support to Java Contracts or the Java Modelling Language, also as far as the author is aware this is also the first specification refactoring tool made publically available.

Implementing this tool enabled the details of refactoring specifications to be studied more closely. The discovery of specification smells and their description, highlights another facet of the improvements Design by Contract can make to a system's design. The difference between the semantics preserved by refactoring operations and the semantics described in formal methods is explored and advice given on when formal method specifications semantic should and should not be preserved. Design by Contract and Test Driven Development are compared and contrasted in depth with regard to refactoring. Not only does this show how the two practices can complement each other it also provides valuable insight into refactoring formal specifications.

9.1 Combining Formal Specification with Agile Practices

Design by Contract and formal specifications are very powerful methods in their own right. The tools and techniques that have been created on top of these are of significant benefit to programmers across the industry, not just programmers of critical systems. The challenge now is how to make these techniques and tools more amenable to a wider audience. JML does this for Java programmers, by reusing Java syntax and semantics thus enabling Java programmers to express their specifications in a form that is natural to them. The JML community are building a strong suite of tools that make it an attractive addition to the Java programmer's toolbox.

For Design by Contract and formal specifications to be adopted by this wider audience they must work with and complement existing common methodologies. Agile methodologies with their suite of complementary practices and techniques currently have the most momentum in the industry. So for formal specifications to be embraced they must work well with these methodologies.

It is easy to be convinced that merging the two is simply about adapting formal principles to agile techniques or vice versa. However agile methodologies ability to respond to change does not simply come from its set of practices and techniques, but from the self awareness built into the processes. Everything is taken in small steps and predicated on feedback which is used to judge the business value at each step. Practices and the process as a whole follow this pattern. For example after each step in a refactoring the tests are run, if the tests pass the programmer moves to the next step. If a test fails the programmer then decides whether to fix the tests or abandon the refactoring. Likewise after each project and project iteration there is a retrospective, at which the process is discussed to determine if it can be improved and the process adapted.

The self aware and emergent nature of the agile process raises the bar for the adoption of Design by Contract and formal specifications. They cannot simply demonstrate the business value they add and their ability to respond to change through feedback. They have to dovetail with the existing practices and techniques and cannot impede their use.

Refactoring and automated refactoring support for specifications are a key part of this. For example an agile team adopting Design by Contract could find it improves their design, simplifies their unit tests and reduces their bugs. However if it impedes their ability to refactor the system and respond to change by forcing them to examine and modify specifications by hand after each

refactoring, Design by Contract won't make it past the first retrospective. Refactoring is so important because it enables the system's design to be as simple as possible, since it is easy to change in the future. Without this ability the design naturally becomes more complicated as programmers try to pre-empt change. This breaks one of the fundamental values of agile software development: Responding to change over following a plan [**16**].

This possibility is similar to the fate of pair programming one of the least practiced agile techniques. While there might be long term benefits, most programmers find it hard and feel it slows them down so teams tend to remove it from their process. Note that small steps and continual feedback make agile processes susceptible to local minima. Attempts to unite formal specifications and agile practices must ensure they are not hindered by this, by making sure even small steps add value and do not clash with other practices.

Adding automated formal specification refactoring tools to Java programmers' common development environments is a natural next step to the adoption of formal specifications and Design by Contract. Providing support for the tools and techniques programmers use every day is following the path laid by JML adding formal specifications to a common everyday language and reusing that language's syntax and semantics.

9.2 Future Work

The area of specification smells is potentially a very interesting one. The smells listed here are only the tip of the iceberg, there are a lot more to be discovered and possible refactorings suggested. As noted specification smells can be symptoms of design issues in the code. So, how the design of a system can be improved by finding and resolving specification smells is an open question.

The synergies and similarity between Design by Contract and Test Driven Development have been discussed from a number of angles, including here, however the similarities between testing anti-patterns and specification anti-patterns has not.

It is well known that specifications make unit testing easier. The tests require less explicit asserts as the specifications can be verified at runtime. This also enables the tests to be more coarse-grained, which allows tightly coupled systems to be tested. Dealing with tight coupling is one of the biggest challenges when adding unit tests to legacy code that has never been tested like this before. Adding specifications to legacy code could enable it to be efficiently refactored. This is especially true when used with tools like Dikon that can automatically add class invariants from runtime information [**19**].

The refactoring tool should be extended to cover more operations. And also include some operations that refactor both code and specifications. Currently Java Contracts are an experimental technology; the future for JML is OpenJML, a reworking of the JML tools suite to support the latest Java compilers and features. Java Contracts should integrate with OpenJML but this might not have priority with other OpenJML projects. Native OpenJML refactoring support would make JML refactoring available to a larger audience.

The tool would be a lot more powerful if it could reason about the specifications. Having access to the reasoning components of a static checker like ESC/Java2 would enable the tool to check the effects of call sites before operations are performed or deduce and add new specifications to refactored code. Opening up a static checker could encourage a similar amount of innovation that opening the Java compiler to the Eclipse refactoring tools has had. Finally the reason programmers refactor their code contains meaning, if the tool could deduce or ask their intent when performing a refactoring it could be smarter adding or modifying specifications when refactoring.

10 Bibliography

[1] Bertrand Meyer, "Applying Design by Contract," vol. 25, no. 10, 1992.

[2] Bertrand Meyer, *Eiffel: The Language*.: Prentice Hall, 1990.

[3] G. Leavens and Y. Cheo. Design by Contract with JML. [Online]. http://www.jmlspecs.org/jmldbc.pdf

[4] Gary Leavens, Albert Baker, and Clyde Ruby, "JML: A Notation for Detailed Design".

[5] Gary Leavens, Albert Baker, and Clyde Ruby, "Preliminary Design of JML: A Behavioral Interface Specification Language for Java," 2001.

[6] Joshua Bloch, *Effective Java*, 1st ed.: Addison Wesley, 2001.

[7] Sue Black, Paul Boca, Jonathan Bowen, Jason Gorman, and Mike Hinchey, "Formal Versus Agile: Survival of the Fittest?," *Computer*, vol. 42, no. 9, pp. 37-45, September 2009.

[8] Hasko Heinecke and Christian Noack, "Integrating Extreme Programming and Contracts," in *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP 2001*, Villasimius, Sardinia, Italy, 2001, pp. 24–27.

[9] Yishai Feldman, "Extreme Design by Contract," in *Fourth Int'l Conf. Extreme Programming and Agile Processes in Software Engineering (XP 2003)*, Genova, Italy, 2003, pp. 261–270.

[10] Maayan Goldstein, Yishai A. Feldman, and Shmuel Tyszberowicz, "Refactoring with Contracts," in *AGILE*, 2006, pp. 53 - 64.

[11] William Opdyke, "Refactoring Object-Oriented Frameworks," 1992.

[12] Martin Fowler, *Refactoring: Improving the Design of Existing Code*.: Addison-Wesley, 1999.

[13] (2010) The Eclipse Project. [Online]. http://www.eclipse.org

[14] Kent Beck, *Test Driven Development: By Example*.: Addison Wesley, 2002.

[15] Kent Beck, *Extreme Programming Explained*.: Addison-Wesley, 1999.

[16] Agile Manifesto. [Online]. http://agilemanifesto.org/

[17] Joshua Kerievsky, *Refactoring to Patterns*.: Addison-Wesley, 2004.

[18] Daniel Zimmerman and Joseph Kiniry, "A Verification-Centric Software Development Process for Java," in *Ninth International Conference on Quality Software*, 2009, pp. 76-85.

[19] Lilian Burdy et al., "An Overview of JML Tools and Applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 212-232, December 2004.

[20] Patrice Chalin and Robby. (2009) Java Contracts: A Quick Overview. [Online]. http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/org.jmlspecs.javacontract/trunk/doc/overview.doc

[21] Code Contracts. [Online]. http://research.microsoft.com/en-us/projects/contracts/

[22] Kristina Boysen Taylor, "A specification language design for the Java Modeling Language (JML) using Java 5 annotations," 2008.

[23] The Byte Code Engineering Library. [Online]. http://jakarta.apache.org/bcel/

[24] ASM. [Online]. http://asm.ow2.org/

[25] Gary Cernosek. (2005) IBM Developer Works. [Online]. http://www.ibm.com/developerworks/rational/library/nov05/cernosek/index.html

[26] Eric Gamma and Kent Beck, *Contributing to Eclipse*.: Addison-Wesley, 2003.

[27] Eclipse Project Downloads. [Online]. http://archive.eclipse.org/eclipse/downloads/index.php

[28] T. Widmer. (2007) Unleashing the Power of Refactoring. [Online]. http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html

[29] Leif Frenzel. (2006) The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. [Online]. http://www.eclipse.org/articles/Article-LTK/ltk.html

[30] Barbara Liskov, "Data Abstraction and Hierarchy," in *Object Oriented Programming Systems Languages and Applications*, Orlando, Florida, United States, 1987, pp. 17-34.

[31] David Cok and Joseph Kiniry, "ESC/Java2: Uniting ESC/Java and JML".

[32] Gabriel Falconieri Freitas and Márcio Cornélio, "Tenth International Workshop on Rule-Based Programming (RULE 2009)," , Brasília, Brazil, 2009, pp. 65–76.

[33] Jonathan Ostroff, David Makalsky, and Richard Paige, "Agile Specification-Driven Development".

[34] Michael Feathers, *Working Effectively with Legacy Code*.: Prentice Hall, 2004.

[35] Dave Springgay. (2001) Creating an Eclipse View. [Online]. http://www.eclipse.org/articles/viewArticle/ViewArticle2.html

[36] Wassim Melhem and Dejan Glozic. (2003) PDE Does Plug-ins. [Online]. http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[37] (2010) JUnit. [Online]. http://www.junit.org/

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*.: Addison-Wesley, 1994.

[39] (2010) The Java Modelling Language (JML). [Online]. http://www.eecs.ucf.edu/~leavens/JML/

[40] Jan Phillips and Bernhard Rumpe, "Roots of Refactoring".

[41] Exploring Eclipse's ASTParser.

[42] Robby, "An Evaluation of The Eclipse Java Development Tools (JDT) as a Foundational Basis for JML Reloaded," 2007.

[43] Abstract Syntax Tree.

[44] Tom Mens and Tom Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, 2004.

[45] Don Roberts, John Brant, and Ralph Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, 1997.

[46] Check Style. [Online]. http://checkstyle.sourceforge.net

[47] PMD. [Online]. http://pmd.sourcefourge.net

11 Appendix

11.1 Project Source Code

The source code for this project is open-source and available from the jmlspecs project on http://sourceforge.net.

For more details of the project see:

http://sourceforge.net/apps/trac/jmlspecs/wiki/RefactoringTools

For access to the source code see:

http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/Refactoring/

11.2 Refactoring Test Cases

http://sourceforge.net/apps/trac/jmlspecs/wiki/RefactorTestCases

	Specification	Visibility	Super	Sub	Refactoring	Notes
1	precondition	public	interface	interface	pull-up	see testcase/pullup/precond/iface/iface/package.html
2	precondition	public	interface	interface	push-down	*note 4*
3	precondition	public	interface	class	pull-up	see testcase/pullup/precond/iface/clazz/package.html
4	precondition	public	interface	class	push-down	*note 4*
5	precondition	public	class	interface	pull-up	*impossible 1*
6	precondition	public	class	interface	push-down	*impossible 1*
7	precondition	public	class	class	pull-up	
8	precondition	public	class	class	push-down	*note 4*
9	precondition	private	interface	interface	pull-up	*impossible 2, 5*
10	precondition	private	interface	interface	push-down	*impossible 2, 5*
11	precondition	private	interface	class	pull-up	*impossible 1, 2*
12	precondition	private	interface	class	push-down	*impossible 2, 5*
13	precondition	private	class	interface	pull-up	*impossible 1*
14	precondition	private	class	interface	push-down	*impossible 1, 2*
15	precondition	private	class	class	pull-up	*impossible 2*
16	precondition	private	class	class	push-down	*impossible 2*
17	precondition	protected	interface	interface	pull-up	*impossible 5*
18	precondition	protected	interface	interface	push-down	*impossible 5*
19	precondition	protected	interface	class	pull-up	
20	precondition	protected	interface	class	push-down	*impossible 5*
21	precondition	protected	class	interface	pull-up	*impossible 1*
22	precondition	protected	class	interface	push-down	*impossible 1*
23	precondition	protected	class	class	pull-up	

24	precondition	protected	class	class	push-down	*note 4*
25	precondition	package	interface	interface	pull-up	
26	precondition	package	interface	interface	push-down	*impossible 5*
27	precondition	package	interface	class	pull-up	
28	precondition	package	interface	class	push-down	*impossible 5*
29	precondition	package	class	interface	pull-up	*impossible 1*
30	precondition	package	class	interface	push-down	*impossible 1*
31	precondition	package	class	class	pull-up	
32	precondition	package	class	class	push-down	*note 4*
33	postcondition	public	interface	interface	pull-up	see testcase/pullup/postcond/iface/iface/package.html
34	postcondition	public	interface	interface	push-down	see testcase/pushdown/postcond/iface/iface/package.html
35	postcondition	public	interface	class	pull-up	see testcase/pullup/postcond/iface/clazz/package.html
36	postcondition	public	interface	class	push-down	see testcase/pushdown/postcond/iface/clazz/package.html |
37	postcondition	public	class	interface	pull-up	*impossible 1*
38	postcondition	public	class	interface	push-down	*impossible 1*
39	postcondition	public	class	class	pull-up	
40	postcondition	public	class	class	push-down	
41	postcondition	private	interface	interface	pull-up	*impossible 2, 5*
42	postcondition	private	interface	interface	push-down	*impossible 2, 5*
43	postcondition	private	interface	class	pull-up	*impossible 2*
44	postcondition	private	interface	class	push-down	*impossible 2, 5*
45	postcondition	private	class	interface	pull-up	*impossible 1, 2*
46	postcondition	private	class	interface	push-down	*impossible 1, 2*
47	postcondition	private	class	class	pull-up	*impossible 2*
48	postcondition	private	class	class	push-down	*impossible 2*
49	postcondition	protected	interface	interface	pull-up	
50	postcondition	protected	interface	interface	push-down	
51	postcondition	protected	interface	class	pull-up	
52	postcondition	protected	interface	class	push-down	*impossible 5*
53	postcondition	protected	class	interface	pull-up	*impossible 1*
54	postcondition	protected	class	interface	push-down	*impossible 1*
55	postcondition	protected	class	class	pull-up	
56	postcondition	protected	class	class	push-down	
57	postcondition	package	interface	interface	pull-up	
58	postcondition	package	interface	interface	push-down	
59	postcondition	package	interface	class	pull-up	
60	postcondition	package	interface	class	push-down	*impossible 5*
61	postcondition	package	class	interface	pull-up	*impossible 1*
62	postcondition	package	class	interface	push-down	*impossible 1*
63	postcondition	package	class	class	pull-up	
64	postcondition	package	class	class	push-down	
65	invariant	public	interface	interface	pull-up	see testcase/pullup/invariant/iface/ifac

						e/package.html
66	invariant	public	interface	interface	push-down	see testcase/pushdown/invariant/iface/iface/package.html
67	invariant	public	interface	class	pull-up	see testcase/pullup/invariant/iface/clazz/package.html
68	invariant	public	interface	class	push-down	see testcase/pushdown/invariant/iface/clazz/package.html
69	invariant	public	class	interface	pull-up	*impossible 1*
70	invariant	public	class	interface	push-down	*impossible 1*
71	invariant	public	class	class	pull-up	
72	invariant	public	class	class	push-down	
73	invariant	private	interface	interface	pull-up	*impossible 5*
74	invariant	private	interface	interface	push-down	*impossible 5*
75	invariant	private	interface	class	pull-up	
76	invariant	private	interface	class	push-down	*impossible 5*
77	invariant	private	class	interface	pull-up	*impossible 1*
78	invariant	private	class	interface	push-down	*impossible 1*
79	invariant	private	class	class	pull-up	
80	invariant	private	class	class	push-down	
81	invariant	protected	interface	interface	pull-up	
82	invariant	protected	interface	interface	push-down	
83	invariant	protected	interface	class	pull-up	
84	invariant	protected	interface	class	push-down	*impossible 5*
85	invariant	protected	class	interface	pull-up	*impossible 1*
86	invariant	protected	class	interface	push-down	*impossible 1*
87	invariant	protected	class	class	pull-up	
88	invariant	protected	class	class	push-down	
89	invariant	package	interface	interface	pull-up	
90	invariant	package	interface	interface	push-down	
91	invariant	package	interface	class	pull-up	
92	invariant	package	interface	class	push-down	*impossible 5*
93	invariant	package	class	interface	pull-up	*impossible 1*
94	invariant	package	class	interface	push-down	*impossible 1*
95	invariant	package	class	class	pull-up	
96	invariant	package	class	class	push-down	
97	model method	public	interface	interface	pull-up	
98	model method	public	interface	interface	push-down	
99	model method	public	interface	class	pull-up	
100	model method	public	interface	class	push-down	*impossible 5*
101	model method	public	class	interface	pull-up	*impossible 1*
102	model method	public	class	interface	push-down	*impossible 1*
103	model method	public	class	class	pull-up	
104	model method	public	class	class	push-down	
105	model method	private	interface	interface	pull-up	*impossible 3, 5*
106	model method	private	interface	interface	push-down	*impossible 3, 5*
107	model method	private	interface	class	pull-up	*impossible 3*
108	model method	private	interface	class	push-down	*impossible 3, 5*

109	model method	private	class	interface	pull-up	*impossible 1, 3*
110	model method	private	class	interface	push-down	*impossible 1, 3*
111	model method	private	class	class	pull-up	*impossible 3*
112	model method	private	class	class	push-down	*impossible 3*
113	model method	protected	interface	interface	pull-up	
114	model method	protected	interface	interface	push-down	
115	model method	protected	interface	class	pull-up	
116	model method	protected	interface	class	push-down	*impossible 5*
117	model method	protected	class	interface	pull-up	*impossible 1*
118	model method	protected	class	interface	push-down	*impossible 1*
119	model method	protected	class	class	pull-up	
120	model method	protected	class	class	push-down	
121	model method	package	interface	interface	pull-up	
122	model method	package	interface	interface	push-down	
123	model method	package	interface	class	pull-up	
124	model method	package	interface	class	push-down	*impossible 5*
125	model method	package	class	interface	pull-up	*impossible 1*
126	model method	package	class	interface	push-down	*impossible 1*
127	model method	package	class	class	pull-up	
128	model method	package	class	class	push-down	

11.2.1 Notes

1. An interface cannot extend a class.
2. Private methods cannot be overridden, so cannot pull-up or push-down their preconditions or postconditions.
3. Private model methods make no sense.
4. Pushing down a precondition breaks LSP, performing this refactoring forces the introduction of non-behavioural inheritance.
5. Interfaces only support public methods and fields.