Universidade de Pernambuco
Escola Politécnica de Pernambuco
Departamento de Sistemas e Computação
Programa de Pós-Graduação em Engenharia da Computação

Gabriel Ramos Falconieri Freitas

Refactoring Annotated Java Programs: A Rule-Based Approach

Dissertação de Mestrado

Recife, July 2009

UNIVERSIDADE DE PERNAMBUCO
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

GABRIEL RAMOS FALCONIERI FREITAS

Refactoring Annotated Java Programs: A Rule-Based Approach

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Cornelio Márcio
Advisor

Recife, July 2009

CIP – CATALOGING-IN-PUBLICATION

Falconieri Freitas, Gabriel Ramos

Refactoring Annotated Java Programs: A Rule-Based Approach / Gabriel Ramos Falconieri Freitas. – Recife: PPGEC da UPE, 2009.

240 f.: il.

Thesis (Master) – Universidade de Pernambuco. Programa de Pós-Graduação em Engenharia da Computação, Recife, BR–PE, 2009. Advisor: Cornelio Márcio.

1. JML, design by contract, programming laws, refactoring, Java. I. Márcio, Cornelio. II. Título.

To my mother, Lúcia.
“Every passing minute,
is another chance to turn it all around.”
— Cameron Crowe

Acknowledgments

Primeiramente gostaria de agradecer a Deus e ao meu guia Sri Sathya Sai Baba que me acompanhou desde o colegial até aqui. Agradeço também a minha mãe Lucia – a quem dedico este trabalho – que sempre me aconselhou e me deu forças, atenção, incentivos, etc para estar aqui. A meu pai que com certeza está rezando por mim la de cima e muito feliz em me ver onde estou hoje. A minha vó que rezou por tantos dias para eu ter forças para alcançar meus objetivos. Ao meu irmão Bruno que sempre esteve mesmo de longe, lado a lado comigo me ajudando e me dando forças. A Alexandrina por ter incrivelmente aguentado a minha "ausência" durante todo este último ano.

Gostaria de agradecer muito a Alessandro, Keity, Douglas e Paulinho por terem me entendido e me concedido dias preciosos para que eu pudesse estudar, realizar minhas pesquisas e concluir este trabalho.

Agradeço enormemente meus colegas de time, em especial, Zanini, Pacheco, Campinho, Helen e Anderson por terem sempre me incentivado e dizendo que mesmo trabalhando árduamente eu conseguiria chegar onde cheguei.

Agradeço também a todos os meus professores do Departamento de Sistemas Computacionais em especial ao professor Carlos Alexandre por ter sido fundamental durante toda minha vida acadêmica. E principalmente ao meu orientador Márcio Cornélio que é uma das pessoas mais excepcionais e atensiosas que já conheci na ida, sempre atencioso e presente como um companheiro. Também agradeço ao professor Tiago Massoni e Rohit Gheyi pelas sempre muito importantes ajudas.

Por fim, agradeço aos meus amigos da faculdade que sempre estiverem comigo me acompanhando principalmente nas madrugadas de estudo, em especial o meu amigo Flávio Oliveira, fiel companheiro dessa longa caminhada.

Contents

List of Abbreviations and Acronyms

JML	Java Modeling Language
RAC	Run Time Assertion Check
MES	Manufacturing Execution System
DbC	Design by Contract
BISL	Behavior Interface Specification Language

List of Figures

List of Tables

Abstract

Formal specification languages has an important role in the development of software whose reliability we can argue with a sound basis. One methodology that goes in the direction of practical application of formal specification languages is known as Design by Contract. In this methodology, a contract is established between classes of a system. However, changes are inherent to software due to corrective needs or evolution. The strong dependence between source-code and additional formal specifications may introduce a number of evolution-related difficulties. In order to accommodate new requirements or improve the software structure, code may be modified and specifications may become outdated. On the other hand, changes in specifications are sometimes needed. In the context of refactoring, changes must be behavior-preserving, maintaining source-code in conformance with its specification. In this work, we propose a systematic approach to deal with changes in source-code so that they aware of specifications. Also, we illustrate how specifications can be modified without affecting specifications already described. Primitive transformations are described by means of programming laws. We introduce a set of programming laws for object-oriented programming languages like Java combined with the Java Modeling Language (JML). The set of laws deals with object-oriented features taking into account specifications. Other laws deal only with features of the specification language. These laws constitute a set of small transformations for the development of more elaborate ones. An application is presented to show how a JML-specified version of a core module from a Manufacturing Execution System is refactored from successive applications of primitive transformations expressed by means of our laws. We have also investigated the impact on the reduction of Java programs to a normal form when specifications written in JML are present.

Keywords: JML, design by contract, programming laws, refactoring, Java.

Resumo

Linguagens de especificação formal possuem um papel importante no desenvolvimento de softwares cuja confiabilidade é um requisito forte. Uma metodologia que vai na direção da aplicação prática de linguagens de especificação formal é conhecida como *Design by Contract*. Nesta metodologia, um contrato é estabelecido entre classes do sistema. Contudo, mudanças são inerentes aos softwares devido a necessidades de correções ou evolução. A forte dependência entre código-fonte e especificações formais adicionais pode acarretar em várias dificuldades relacionadas e evolução. Para acomodar novos requisitos ou melhorar a estrutura do software, o código-fonte pode ser modificado tornando as especificações desatualizadas. Por outro lado, mudanças nas espcificações também podem ser necessárias. No contexto de refatoração de programas, as mudanças necessitam preservar o comportamento do programa, mantendo o código-fonte em conformidade com as suas especificações. Neste trabalho nós propomos uma abordagem sistemática para lidar com mudanças em código-fonte estando elas cientes que o código-fonte também possui especificações. Adicionalmente, nós ilustramos como especificações podem ser modificadas sem afetar especificações já descritas. Transformações primitivas são descritas como leis de programação. Nós introduzimos um conjunto de leis de programação para linguagens de orientação a objetos como Java combinadas com a linguagem de especificação formal JML (*Java Modeling Language*). O conjunto de leis lida com características da orientação a objetos levando em consideração especificações. Outras leis lidam apenas com características da linguagem de especificação formal. Essas leis constituem um pequeno conjunto de transformações que servem como base para o desenvolvimento de transformações mais elaboradas. Um aplicação é apresentada para mostrar como uma versão especificada com anotações JML de um módulo central de um Sistema de Execução da Manufatura é refatorado a partir de aplicações sucessivas de transformações primitivas expressas pelas nossas leis. Nós também investigamos o impacto da redução de programas Java para uma forma normal quando especificações escritas em JML estão presentes.

Palavras-chave: JML, design by contract, leis de programação, refactoring, Java.

Chapter 1

Introduction

Software changes constantly due to maintenance that leads to correction of fails or just to improve functionalities. However, some changes can take place to achieve quality factors like reuse and legibility. In these cases, changes should not alter the software behavior but only its internal structure. Improving the internal software structure is an activity known as refactoring [31]. To avoid errors due to modifications, every change has to be done following a discipline. Also, programming laws are a means to change software in a systematic and rigorous way. For instance, we can use compilation and tests after every modification.

Programming laws serve as guidelines to informal programming practices and establish a basis for formal and rigorous program development. They are largely known for imperative programming [38, 53]. Also, functional programming and logic programming have a set of laws described by Bird and de Moor [6] and Seres [61], respectively. Laws of object-oriented programming have also been addressed in [7, 23, 26].

Object-oriented programming laws were initially proposed by Borba, Sampaio and Cornélio [9] for an object-oriented language called ROOL [13], which was designed to allow reasoning about object-oriented programs and specification, mixing both constructs in the sytle of Morgan's refinement calculus [53]. They propose laws for classes and commands of ROOL and they define a normal form for object-oriented programs written in ROOL along with a reduction strategy. Also, they demonstrate that the set of laws is complete with respect to this normal form. Cornélio [23] proves the laws with respect the copy semantics of ROOL [13]. Silva, Sampaio, and Liu considers object-oriented programming laws in a language with a reference semantics [62], applying such laws to code refactoring. Duarte [26] adapts the programming laws initially written for ROOL for the Java programming and proposes other laws for language features that are not present in ROOL.

Programming laws are a good alternative to apply refactorings in a systematic and rigorous way [7, 23]. The application of programming laws can be seen as an activity accomplished in two stages. In the first stage, the conditions for the law application must be verified in order to determine if the law can be applied. The second stage consists of the transformation of the program as described in the law. For example, to eliminate a public method one needs to guarantee that the method is not called anywhere in the program.

Design by Contract (DbC) [51] is a development methodology that aims at the construction of reliable object-oriented systems. Its basic idea is that a contract is established among classes of a system. In this way, software developers should formally specify what

is required and ensured by methods and types. The use of specification languages, such as the Java Modeling Language (JML) [10, 42, 28], encourages implementations to follow pre-defined specifications, in order to control complexity, improve verification tool support and encourage Design by Contract [51].

In this context, software evolution brings additional challenges. When evolution tasks are carried out, either to fulfill new requirements or improve source-code quality, dependence between program code and specifications must be carefully considered. This dependence occurs in both directions. Changes in specifications usually must be accompanied with program code updates, in order to maintain conformance. On the other hand, changes in the program code require changes in specifications, where the original specification can no longer have the same meaning for the new behavior. For instance, moving a redefined method to its superclass can be illegal if this transformation weakens pre-conditions and strengthens post-conditions.

1.1 Objectives

The set of programming laws for object-oriented programming we have nowadays is designed for program transformation with no relation to specifications languages useful for DbC. Our objective is to define laws of object-oriented programming for Java that are aware of specifications written in JML. Our proposed hybrid laws were created by extending object-oriented programming laws from other works [7, 23, 26, 49]. Additionally, we introduce laws for specifications written in JML. The laws precisely indicate the modifications that can be done to a program, stating their corresponding proof obligations that are discharged for application. To our knowledge, there is not a comprehensive set of laws to deal with formally specified Java programs. In Java and JML context, we need to guarantee that source-code continues meeting its specifications written in JML, taking into account the semantics of JML specifications along with the notion of specification inheritance [40].

To demonstrate the applicability of our set of laws, we reduce a Java program with JML specification to the normal form presented by Duarte [26], which follows the main steps of the normal form reduction strategy of ROOL. The existence of specifications impose restrictions leading to a normal form slightly different from the one of Duarte. In this work, we discuss the main differences between them.

Also, we propose a rigorous and systematic approach to apply some of the refactorings proposed by Fowler [31] and evolve code through successive applications of primitive transformations expressed by means of our laws using as study case a JML-specified version of a core module of a Manufacturing Execution System (MES) [64].

1.2 Motivating Example

In order to show the relevance of the problem we deal with in this work, we present a small example of two JML-specified Java classes. The class shown in **Figure 1.1** represents a positive integer. In line 2 we find a example of an invariant. The **requires** clauses in the specifications of the methods `registerValue` and `format` specify two pre-conditions. For example, the pre-condition of the method `registerValue` demands that the value to be registered has to be not null and also has to be at least equals to zero. In the lines 8, 12

```
1  public class PositiveInteger {
2    //@ private invariant value.intValue() > -1;
3    private Integer value;
4
5    public PositiveInteger() { value = new Integer(0); }
6
7    /*@ requires newValue != null && newValue.intValue() > -1;
8      @ ensures getValue().intValue() == newValue.intValue();
9      @*/
10   public void registerValue(Integer newValue) { /* ... */ }
11
12   //@ ensures \result != null;
13   public /*@ pure @*/ Integer getValue() { /* ... */  }
14
15   /*@ requires getValue() != null;
16     @ ensures !(\result).equals(""); @*/
17   public String format() { /* ... */ }
18 }
```

Figure 1.1: JML specification of the class `PositiveInteger`. In **ensures** clauses, **\result** stands for the result that is returned by the method.

and 16 we have examples of postconditions. Postconditions start with the **ensures** clause. In addition, there is another class `EvenInteger` (Fig. 1.2) utilized to express even integers.

To characterize a positive integer, constraints (in the form of JML specifications) were written in the `PositiveInteger` class. The invariant of line 2 (Fig. 1.1) establishes that the integer value of `value` field should be at least equal to zero. Also, the pre-condition of line 7 (Fig. 1.1), obligates that only positive `Integer` values can be registered.

The `EvenInteger` class (Fig. 1.2) can only hold even positive integers because of the invariant written in line 2, assuring that the integer value of the class needs to be module of 2. And, to reinforce the invariant, pre-conditions of method `registerValue` guarantee that only even and positive values are allowed.

It may be assumed that a new type of integer might be implemented, for instance, odd integers. To accomplish this new requirement, it is important to prepare our source-code to receive the new code. This situation shows an example of a refactoring: there is a new class to be implemented and there are two classes that share several features. The refactoring Extract Superclass is frequently applied to create an abstraction, concentrating in it all shared or duplicated features of two or more subclasses.

Fowler [31] presents a mechanics of how to extract a superclass from some Java classes using a suite of tests, which is executed in each step of the mechanics to ensure that the code continues to meet its original observable behavior. His approach is based on having suitable test cases to execute and leads only with Java code. The application of this refactoring considering also formal specifications creates several new issues, such that:

- Pulling up the field `value` must be followed by the invariants that it refers to. Thus, we need to move only invariants that are shared between the subclasses.

- Moving up an invariant to a new generalization class will cause all subclasses to inherit it, making them potentially more restrictive?

```
1  public class EvenInteger {
2    //@ private invariant value.intValue() % 2 == 0;
3    //@ private invariant value.intValue() > -1;
4    private Integer value;
5
6    public EvenInteger() { value = new Integer(0); }
7
8    /*@ requires newValue != null;
9      @ requires newValue.intValue() % 2 == 0 && newValue.intValue() >
           -1;
10     @ ensures getValue().intValue() == newValue.intValue();
11     @*/
12   public void registerValue(Integer newValue) { /* ... */ }
13
14   //@ ensures \result != null;
15   public /*@ pure @*/ Integer getValue() { /* ... */ }
16
17   /*@ requires getValue() != null;
18     @ ensures !(\result).equals(""); @*/
19   public String format() { /* ... */ }
20 }
```

Figure 1.2: JML specification of the class `EvenInteger`.

- It might be useful to pull up methods (and its specifications) having the same meaning.

Regarding the third issue, if we pull up `registerValue` from `EvenIntegerValue` to a new common superclass, the subclass `PositiveInteger` will inherit its pre- and postconditions. And hence will generate a new constraint on the `PositiveInteger` class: objects of type `PositiveInteger` would only be able to register even integers.

Consider we want to introduce new features in a class. If we want to introduce a new redefined implementation of the method `registerValue` in a hypothetic newly created `OddIntegerData`. As a redefined method has the identity specification (pre-condition: true / postcondition: false) [40], we can make this specification explicit and thus weaken it to `getValue()`.`intValue()`% 2 == 1, for example. Then we can strengthen the postcondition to ensure that the candidate value is assigned to the `value` field.

Our approach investigates situations like these, proposing a rigorous behavior-preserving way to execute specification and code transformations. Our approach is based on laws (primitive transformations) including side-conditions that define when a transformation may be applied.

1.3 Contributions

The summary of our contributions is presented as follows:

- **Creation of programming laws to deal with JML specifications** – a set laws to deal with JML specifications and JML constructs like invariants, pre- and postconditions and privacy modifiers were defined.

- **Adaptation of programming laws for JML-specified Java programs** – we present programming laws for JML-specified Java programs created by reviewing and extending previously defined laws for ROOL and Java.
- **Proposition of new laws for Java** – we propose some new laws for Java and after it we extended them to consider JML specifications.
- **Proposition of new for Commands and Expressions** – we propose some new laws to deal with commands of Java.
- **Normal form reduction strategy for JML-specified Java programs** – a reduction strategy was proposed to reduce JML-specified Java programs to a normal form which follows the mains steps of the normal form reduction strategy of Java. We present an example of the application of our strategy.
- **A step by step case study showing how refactorings can be applied using programming laws** – to demonstrate the applicability of our set of laws we show step by step how a JML-specified version of a core module from a real Manufacturing Execution System, get refactored from successive applications of primitive transformations expressed by means of our laws.

1.4 Organization

This dissertation is structured as follows:

Chapter 2: we provide a brief introduction to the Java Modeling Language (JML), focusing mainly on its fundamentals and in the concepts we use in this work. We also detail the features and concepts of the language that are necessary to understand the subsequent chapters.

Chapter 3: we introduce our set of laws to deal with JML-specified Java programs discussing in depth each law.

Chapter 4: we present our normal form reduction strategy showing how the existence of specifications impose restrictions to reach the normal form previously defined for Java and ROOL. A practice example is also used to provide evidences.

Chapter 5: we show a rigorous and systematic approach to apply some refactorings proposed by Fowler [31]. A case study is used to present the laws application in action. We use a step-by-step approach in order to provide as much details as possible.

Chapter 6: we present our conclusions and directions for future work. We also discusses some related works.

Chapter 2

The Java Modeling Language

In this chapter, we provide a brief introduction to the Java Modeling Language (JML), focusing mainly on its fundamentals and in the concepts were used in this work. First we present an overview of JML using a short example. Then we detail some of the features initially introduced in the overview. At the end of the chapter we discuss about some JML tools like, ESC/Java2 [21], Krakatoa [11] and the JML compiler (*jmlc*) [10]. A complete description about JML can be found in innumerable other publications available at [41] and can be referred in the JML Reference Manual [28].

2.1 JML in a nutshell

The Java Modeling Language (JML) is a behavioral interface specification language [42, 28, 43] tailored to Java. Thus, JML serves to describe contracts with static information that appear in Java declarations and how they act. JML specifications are written in the form of *special annotation* comments that are inserted directly in source code of programs. These comments must begin with an at-sign (`@`) and can be written in two ways: by using `//@ ..` . or `/*@ ..` . `@*/`. In Figure 2.1, we present the class `Person`, with contracts written in JML.

The **model** modifier (lines 2 and 3) introduces specification-only fields, also called *model fields*. A model field should be thought of as an abstraction of a set of concrete fields used in the implementation of this type and its subtypes [28]. In the class `Person`, we have two model fields, i.e. `name` and `weight`, representing (via **represents** clause) the concrete attributes `_name` and `_weight`, respectively.

The **invariant** clause introduces predicates that are true in all visible states of objects of a class (see Section 2.5.1 for a full explanation). The invariant in the example has public visibility and establishes that the value of attribute `_name` is different from an empty string and the value of `_weight` is greater than or equal to zero.

The **requires** clause specifies the obligations of the caller of a method, what must be true to call a method. For instance, the precondition of the method `addKgs` insists on the added value to be greater than zero. A postcondition specifies the implementor's obligation, what must be true at the end of a method, just before it returns to the caller. In JML, the **ensures** clause introduces a postcondition. In the example, the post-condition introduce in line 21 asserts the value of the attribute `_weight` at the end of the method `addKgs` is equal to the value of the expression **\old**(`weight + kgs`). By using the **\old**, operator we can refer to the value of an expression in the pre-state of a method.

The **assignable** clause gives a frame axiom for a specification. Only locations named

```
1  public class Person {
2    //@ public model int weight;
3    //@ public model String name;
4
5    private String _name;
6    private int _weight;
7
8    //@ private represents name <- _name;
9    //@ private represents weight <- _weight;
10
11   //@ public invariant !name.equals("") &&
12   //@                  weight >= 0;
13
14   public Person(String pname, pweight) { ... }
15
16   //@ ensures \result == weight;
17   public /*@ pure @*/ int getWeight() { ... }
18
19   /*@ requires kgs > 0;
20     @ assignable weight;
21     @ ensures weight == \old(weight + kgs);
22     @*/
23    public void addKgs(int kgs) { ... }
24 }
```

Figure 2.1: `Person` class source-code.

and their associations can be assigned during method execution. In method `addKgs`, we state that only `weight` is changeable. The JML modifier **pure** indicates that the method doesn't have any side effects and hence can appear in specifications.

2.2 Assertions and Expressions

The JML specifications, i.e. expressions and assertions, are written in the syntax of Java [35]. These specifications are added as annotations (in the form of comments) within the source code of the program, which can be compiled by any Java compiler which facilitates the use of JML by Java developers.

JML expressions and assertions cannot have side-effects, in other words they must be *pure*. Hence, operators like =, +=, −= and other operators related to assignments (eg. ++ and −−) cannot appear in expressions or assertions because they have side-effects. As we said in Section 2.1 a *pure* is the one that does not modify any state, that is, does not cause any side effects to the program. It is important to say that expressions can throw exceptions even they are *pure*. Thus, an exceptional expression like `person.getWeight()` when `person` refers to a *null* instance, is permitted, although it does not terminates normally.

JML is a superset of Java. Hence, it provides special constructs that are used in expressions in addition to all the other Java expressions that are free of side-effects. In the sequel we present some of these constructs, the complete list of JML-constructs is presented in [28].

- **\result**(`E`), refers to the value returned by a method. Its type is the return type of the method.

- **old**(E) refers to the value of an expression immediately before a method is called. **old**(E) can also be used in assertions. In these cases, it refers to the value of the expression just before control reaches the statement in which it appears.

- \not_modified(v1, v2..vn) is used to verify if named fields are not modified. For example, \not_modified(v1, v2) verifies if the fields v1 and v2 are the same in pre- and post-states.

- \typeof(E) returns the most-specific dynamic type of an expression's value.

- $T_1 <: T_2$ compares two reference types returning true if T_1 is a subtype of T_2.

- \ **forall** and \exists are the universal and existential quantifiers, respectively.

- \sum, \product, \max and \min are constructs used to return the sum, product, maximum and minimum values of given expressions, respectively. For example, the follow equations is true: (**max int** i; 0 <= i && i < 5; i) == 4.

2.3 Attributes

2.3.1 Specification Visibility

Java defines four types of access modifiers to an attribute: *private*, *default* or *package*, *protected* or *public*. These access modifiers establishes when one can access (or not) an attribute, i.e., controls the visibility of attributes. Java modifiers are also used by the JML compiler (*jmlc*). However, JML introduces additional rules to deal with visibility control. A JML-specification cannot refer to elements (eg. an attribute) that have a more restrictive visibility than the specification itself. For example, a public invariant can only refer to public attributes, protected invariants can refer protected and public attributes (see that public elements are less restrictive than protected ones) and so on.

JML provides a way to alter the visibility of attributes only with respect to specifications. A private or default attribute may have its specification visibility modified to protected using the keyword **spec_protected**. In addition, a non-public attribute may have its specification visibility changed to public using the keyword **spec_public**. See in the examples below. The attribute _weight can be also used in public invariants for example, because for specifications it is public. As well as name can also be used in protected invariants.

```
private /*@ spec_public @*/ String _name;
private /*@ spec_protected @*/ int _weight;
```

2.3.2 Non Null References

In JML, null is not the default [28, 43]. Any declaration (that is not a local variable) whose type is a reference type is implicitly declared to be not null, except when one adorn the declaration with the keyword **nullable**. Thus, by default, JML always checks if an (not nullable) attribute is null in all visible states of the class that declares it. In fact, the JML compiler creates an invariant (eg. //@ **invariant** _name= null;!) for all attributes that are declared with a reference type, asserting that these attributes are not null. The

same behavior is achieved declaring an attribute with the **non_null** modifier (eg. **private** /* @ **spec_protected non_null** @*/ **int** `_weight`;). The **nullable** keyword does exactly the opposite of **non_null**, that is, it permits an attribute (or other non local variable declarations) to be null without throwing an exception.

2.4 Methods Specifications

JML contains the essential notations used in the Design by Contract (DbC) methodology as well as extends and improves the Hoare-style of using pre- and postconditions, including heavyweight and lightweight specifications, privacy of specifications, normal and exceptional postconditions and frame axioms.

Design By Contract [51] establishes a method of building software by explicitly specifying what each function in a module requires in order to operate correctly, and what it provides to the caller (contracts). They constitute a collection of assertions - mainly invariants, pre- and postconditions for methods - that precisely describe what methods require and ensure with respect to client classes.

In this section we focus on the JML notations related to DbC methodology.

2.4.1 Specification Clauses

2.4.1.1 Pre- and Postconditions

A pre-condition of a method is a predicate that should be satisfied at the beginning of a call to the method, in other words pre-conditions specifies the obligations of the caller of a method, what must be true to call a method. JML uses the **requires** clause to introduce pre-conditions. In Figure 2.1 the pre-condition of the method `addKgs` states that the value for the argument for the formal parameter `kgs` needs to be greater than zero. This pre-condition ensures that one can not add a negative value of kilograms.

A postcondition is a predicate that should hold at the end of the method call in the case that the method call ends without throwing an exception. That is, a postcondition specifies the implementor's obligation, what must be true at the end of a method, just before it returns to the caller. Postconditions start with the **ensures** clause.

In Figure 2.1 we have two post-conditions, in lines 16 and 21. The first one just assures that the value the method `addKgs` returns is equals to the value of the attribute `_weight`. The value a method returns is denoted by the expression **result**. The second one asserts the value of the attribute `_weight` at the end of the method `addKgs` is equals to the value of the weight (immediately before the method call) summed with the increment provided by `kgs` parameter. The value of an expression right before a method call is obtained via **old** expression (see Section 2.2).

2.4.1.2 Frame axioms

A frame axiom defines which variables can change in the execution of a statement. In JML, we use the **assignable** clause to define a list of locations that can be modified in the execution of a method. Only locations named and their associations can be assigned during the method execution. Locations can be attributes, model fields representing concrete attributes, and so forth. Local variables of a method are excluded from the **assignable** rules. When we want to state that a method cannot change anything, we use the keyword

\nothing. In the opposite case we use \everything to state that all locations in the program are changeable.

If one does not declare a **assignable** clause in a method using lightweight specification, the JML compiler assumes \not_specified as the default. In fact, for lightweight specifications the JML compiler considers the keyword \not_specified an equivalent keyword to \everything. In method addKgs of Figure 2.1, only the attribute weight can be modified.

2.4.1.3 The keyword *also*

Method specifications contain one or more specification cases. A JML specification case is formed by many clauses, including **requires**, **assignable** and **ensures** clauses [28]. Each specification case has a pre-condition (when it is omitted it assumes the value **true**). Two or more specifications are joined using the keyword **also**. The postcondition of a specification case needs to be true when its corresponding pre-condition holds. Normally a specification of a method consists of one or more specification cases that have to hold when the method is called.

Specification cases define more than one scenario of execution of a method. JML uses the keyword **also** to distinguish these scenarios. Figure 2.2 shows a modified version of method addKgs. The new specification case (lines 5 to 7) contemplates a scenario when a zero or negative value is passed as a argument and weight remains intact.

```
1   /*@ requires kgs > 0;
2     @ assignable weight;
3     @ ensures weight == \old(weight + kgs);
4    @ also
5    @ requires kgs <= 0;
6    @ assignable \nothing;
7    @ ensures \old(weight) == weight;
8     @*/
9    public void addKgs(int kgs) { ... }
```

Figure 2.2: addWeight method of Person class - a version with two specification cases separated using addWeight.

Leavens [40] introduces the semantics of specification inheritance and discuss how specification inheritance forces behavioral subtyping. In JML, a subclass inherits not only attributes and methods from its superclass, it also inherits specifications. According to Leavens' definition, the extended specification of a type is given by the extended specification of methods, invariants, history constraints and initially predicates (see more details in Section 2.7.

The extended specification of an instance method is given by joining the specifications added by the method itself and the inherited ones. In fact, the semantics of specification inheritance is the same of the joining of specification cases – via **also** – of a method. A specification of an overriden method must begin with the keyword **also**. Using the keyword **also** in these cases ensures that the specification cases of the overriden method are joined to the ones declared in the original method (of the superclass).

Joining specifications of a method leads to a pre-condition that is given by disjunction of the predicates of all pre-conditions (of all specification cases and the inherited ones), and a postcondition that is given by the conjuction of implications in which the antecedent

is the pre-condition of the corresponding specification case in the pre-state (the **\old**() operator is used for the precondition), and the consequent is the postcondition of the corresponding specification case. That is, the postconditions are conjoined in the form $\wedge$(**old**$(p_i) \Rightarrow q_j$), where p_i is the pre-condition for the corresponding postcondition q_j [43]. The join of **assignable** clauses is the union of the declared locations.

As an example, in Figure 2.3 we present a joined version of the specifications cases of Figure 2.2.

```
1  /*@ requires kgs > 0 || kgs <= 0;
2    @ assignable weight;
3    @ ensures (\old(kgs > 0) ==> weight == \old(weight + kgs)) &&
4      (\old(kgs <= 0) ==>  kgs > 0);
5    @*/
6   public void addKgs(int kgs) { ... }
```

Figure 2.3: `addWeight` method of `Person` class - a version with a unique specification case.

2.4.1.4 *signals* Clause

One can specify details about the program state – inside specification cases – when exceptions are thrown by a method. The `signal` clause is used to specify a predicate that holds at the end of a method or constructor invocation when this method, or constructor, ends abnormally by throwing the written exception. A **signals** clause has the form **signals** `(E e) R;` where `E` is a class of type `Exception` or a subclass of it, `e` is the instance of the exception in the moment it is thrown and `R` is a predicate (or `\not_specified`).

2.4.2 Heavyweight and Lightweight Specifications

In JML, we have two types of method specifications: lightweight and heavyweight. In lightweight specifications cases the user does not have to specify the complete behavior of the method, in this case it is up to the user to specify exactly what he really wants. In contrast, JML provides a style of method specification, called heavyweight, that waits the user to specify a complete specification case and omits only the parts he knows the default rules fit.

In fact there are two syntaxes to each of one these two types of method specifications what helps the user to distinguish when a method uses one type or the other one. In essence a heavyweight specification case can have three types of behaviors represented by the keywords, **behavior**, **normal_behavior** and **exceptional_behavior**. A specification case that does not define one of these three keywords is characterized as a lightweight specification case. A lightweight specification case is similar to a behavior specification case, but with different defaults [42]. It is important to say that is possible to desugar all type of specification cases into behavior specification cases [57].

The different defaults applied by lightweight and heavyweight specification cases may vary. We highlight in the sequel some of them, along [28] one can discover a complete list.

- **requires** clause: Lightweight specification case uses `\not_specified`. Heavyweight uses **true**.

- **ensures** clause: Lightweight specification case uses `\not_specified`. Heavyweight specification case uses **true**.
- **assignable** clause: Lightweight specification uses `\not_specified`. For a lightweight specification case, the default is `\everything`.
- For lightweight specifications the specification visibility for methods (as well of its specification cases) is the same of the Java visibility of the method itself. For heavyweight specifications one can define the specification visibility for each specification case. Also, one can change the specification visibility for the method itself using the JML modifiers **spec_protected** and **spec_public** as we explained in Section 2.3.1 for attributes.

The behavior of `\not_specified` may vary depending on the tool implementation [28]. In our work we consider the implementation of the JML Official Tools [1]. Our focus in this work is concerned in the foundations of lightweight specification cases since this is simpler and closer to the DbC techniques.

2.4.3 Syntactic Sugars

There are many syntactic sugars for JML, most of them described in [57]. Theses syntactic sugars inspired us in the development of some laws described in Section 3.3. JML syntactic sugars are used in special to write method specifications. For example, the specifications cases of `addWeight` of Figure 2.2 is desugared of the one of Figure 2.3. This example show how multiple specification cases can be collapsed in only one.

Another simple example is shown in Figure 2.4. As can be seen, a single specification case that uses more than one **requires** clause can be simplified to an unique **requires** clause separating each predicated by a && (and) operator. The same reasoning is applied to **ensures** clauses. In Figure 2.4 the left side is desugared to the right side.

```
1  /*@ requires P1;
2    @ requires P2;
3    @ assignable W;          /*@ requires P1 && P2;
4    @ ensures P3;              @ assignable W;
5    @ ensures P4;              @ ensures P3 && P4;
6    @*/                        @*/
7   public void m() { ... }     public void m() { ... }
```

Figure 2.4: Desugaring multiple **requires** and **ensures** clauses.

The use of **non_null** clauses as arguments is a short-hand for a `argument!=`**null** pre-condition predicate when the method does not provide any specification. Suppose a method does not have an explicit specification and has this signature: **public** `/*@` **non_null** `@*/ Boolean m(` `/*@` **non_null** `@*/` **int** `x)`, we can eliminate the second occurrence of the **non_null** clause and use the pre-condition **requires** `x !=` **null**`;`. For the same signature we can delete the **non_null** clause – the one used on the left side of the return type `Boolean` – and insert the postcondition **ensures** `\result !=` **null**.

Considering the use of **pure** clause in methods, the use of this clause adds the following clauses to each specification case for the method. And if the method has no specifications the following clauses (again) are added as a lightweight specification.

diverges false ;
assignable \nothing;

2.4.4 Privacy of Specifications and Visibility

The Java language is built on top of a rigorous set of access control rules for attributes, methods and constructors. The rules are directly related to the declared visibility of the cited elements. We can have public, protected, package (default) and private elements. Public elements may be accessed everywhere, protected may be accessed by subclasses and by classes of the same package (including the class declares them), package (default) elements may be accessed by classes declared in the same package (including the class declares them) and private elements may only be accessed inside the class declares them.

Besides those rules, JML adds the concept of specification visibility. An annotation context cannot refer to elements that are more hidden than the context's own visibility [28]. Thus, for a reference to an attribute `x` (for example) be legal, the specification visibility of the specification that does the reference to `x` must be at least as permissive as the visibility of `x` itself. It is important to say that these rule is an addition to the previous Java visibility rules, i.e., first a reference to an attribute must be valid considering the Java visibility.

Figure 2.5 presents a great example (from [28]) that shows how Java visibility interacts with specification visibility. In the example we used invariant specifications, but the same reasoning is applied to history constraints, methods specifications, initially specifications, and so forth. In short these are the considerations about the example:

- Specifications with public specification visibility can only refer public elements.
- Specifications with protected specification visibility can refer protected elements and public elements because public elements are more permissive than protected ones. Remember that these elements must also visible taking into account the Java visibility rules.
- Specifications with package (default) specification visibility can refer non-private elements if these elements are visible considering Java visibility.
- And, specifications with private specification visibility can refer elements with any declared visibility since they are visible in accord to Java visibility.

2.4.5 Not Null References in Methods

As we said in Section 2.4.5 the **non_null** clause may appear in method declarations. When it is used together with a method return type, it indicates that the method must return a **non_null** value. As well as when **non_null** clause is used together with a method formal parameter which works as a shorthand for a pre-condition stating that the attached formal parameter may not be null. As **non_null** acts in the two situations as pre- and postconditions, thus the clause is inherited in the same way as the equivalent pre- and postconditions would be. Hence one does not need to redeclare this clause in overriden methods in subtypes.

The opposite behavior can be achieved using the **nullable** clause in the two situations cited above. The **nullable** modifier is inherited from original methods in supertypes.

```
1  public class PrivacyDemoLegalAndIllegal {
2    public int pub;
3    protected int prot;
4    int def;
5    private int priv;
6
7    //@ public invariant pub > 0; // legal
8    //@ public invariant prot > 0; // illegal!
9    //@ public invariant def > 0; // illegal!
10   //@ public invariant priv < 0; // illegal!
11
12   //@ protected invariant pub > 1; // legal
13   //@ protected invariant prot > 1; // legal
14   //@ protected invariant def > 1; // illegal!
15   //@ protected invariant priv < 1; // illegal!
16
17   //@ invariant pub > 1; // legal
18   //@ invariant prot > 1; // legal
19   //@ invariant def > 1; // legal
20   //@ invariant priv < 1; // illegal!
21
22   //@ private invariant pub > 1; // legal
23   //@ private invariant prot > 1; // legal
24   //@ private invariant def > 1; // legal
25   //@ private invariant priv < 1; // legal
26 }
```

Figure 2.5: What you can and can not do when taking care about specification visibility and Java visibility rules.

2.5 Type Specifications

Type specifications refer to the set of specifications related to classes and interfaces and not to their members. This set is composed majorly by invariants predicates, history constraints, initially clauses and specification-only member declarations.

2.5.1 Invariants

An invariant (i.e., an *instance* invariant) is a predicate that is true in all visible states of objects of a class. JML has two types of invariants, *instance* invariants and *static* invariants. A *static* invariant may refer only *static* attributes and methods. On the other hand, *instance* invariants can refer to both *static* and *instance* methods and attributes. Only *instance* invariants are inherited by subtypes.

Understanding what the expression "visible state" means is of extreme importance to realize the semantics of invariants. A state is considered visible for an object *o* if this state occurs at one of the following moments in a program's execution [28]:

- at end of a non-helper [1] constructor invocation that is initializing *o*,
- at the beginning of a non-helper finalizer invocation that is finalizing *o*,

[1]The **helper** keyword is used on private methods or constructors when one wants to ignore invariants and history constraints that are relevant to the method. A non-helper method is the one that is not adorned with a **helper** keyword.

- at the beginning or end of a non-helper non-static non-finalizer method invocation with *o* as the receiver,
- at the beginning or end of a non-helper static method invocation for a method in *o*'s class or some superclass of *o*'s class, or
- when no constructor, destructor, non-static method invocation with *o* as receiver, or static method invocation for a method in *o*'s class or some superclass of *o*'s class is in progress.

In Figure 2.1 we have an *instance* invariant. It states that the value of the attribute `_name` has to be always different from an empty string. It also obliges a object of class `Person` to have a weight (attribute `_weight`) greater than zero. Thus, when (for example) an object of class `Person` is instantiated the constructor has to guarantee that a name, different from an empty string is set to the attribute `_name` and that a weight is set to the attribute `_weight` in order to meet the `Person` invariant.

2.5.2 History Constraints

History constraints [28] are introduced in JML by the **constraint** clause. They are used when one needs to restrict the possible states of an object. History constraints restrict the way attribute values can be changed during the program execution.

The history constraints work like postconditions for the methods (or for a specific list of methods determined by the user) of a class. History constraints do not work for constructors since objects do not have a previous state before the constructor call.

As with invariants, we have two kinds of history constraints: *static* constraints and *instance* ones. An instance constraint must be true only after the execution of instance methods, a static history constraint must be true after the execution of both instance and static methods. A constraint must be respect by a method only in the situations when the pre-conditions of the method are also satisfied.

Figure 2.6 presents a simple example that uses the **constraint** clause. The class named `InfiniteList` represents a list that only grows and has a method to read these elements. The constraint assures that each method of the list can increase its size or read an element keeping its size unchanged. In fact, the methods can not delete elements what would break the constraint.

```
1  public class InfiniteList {
2    private /*@ spec_public @*/ List list = new ArrayList();
3    //@ public constraint list.size() >= \old(list.size());
4    //@ public initially list != null && list.size() >= 0;
5
6    public Object getElementAt(int position) {
7      return list.get(position);
8    }
9    public void addElement(Object element) {
10     list.add(element);
11   }
12 }
```

Figure 2.6: Example of history constraints and iniatially predicates.

2.5.3 *iniatially* Clause

The *iniatially* clause defines a predicate that have to be satisfied by all object of a class after its instantiation. An *initially* predicate works as we write this predicate as a postcondition in all non-helper constructors of a class.

In Figure 2.6, the initially predicate guarantees that the list is not null after instantiation and enforces a non-negative size for the list also after instantiations.

2.5.4 Abstract Specifications: model fields and methods and ghost fields

JML allows us to define model elements (model fields, model methods, and model classes). All these model elements are introduced by the **model** clause. A **model** element is a specification-purpose element, i.e., is an element that exists only to be used in specifications and is not considered as part of the Java source-code itself. These elements serve to support the specification of certain properties that are not visible outside the specification context.

Treating specifically fields, JML also provides **ghost** fields. Ghost and model fields differ from each other because a ghost field does not have its value determined by a concrete field, i.e. by a **represents** clause. Ghost fields have its value initialized directly by its own initialization or by a *set-statement* [28]. The value of a model field is resolved by the concrete fields it abstracts from.

In the class `Person` of Figure 2.1, we have two model fields, i.e. `name` and `weight`, representing (via **represents** clause) the concrete attributes `_name` and `_weight`, respectively. We use model fields here in the place of **spec_public** modifiers. It is noticeable that attributes `_name` and `_weight` are private and we have occurrences of them in public methods. We would use **spec_public** to make these attributes public for specification purposes or create model fields (as we did) to represent and use them in the specifications.

In fact **spec_public** is a modifier which changes the visibility of a field. When we use **spec_public** in an attribute declaration, the JML compiler rename the attribute and create a model field to represent it. Suppose we have the following declaration:

```
private /*@ spec_public @*/ int weight;
```

For the JML compiler this is a shorthand for the declaration

```
//@ public model int weight;
private int _weight; //@ in weight;
//@ private represents weight <- _weight;
```

We consider that the desugared version (strictly above) is more friendly and helps in maintenance in the sense that one can change the Java field without affecting the readers of the specification.

2.6 Language Levels

JML is a large and rich language composed for a huge number of features. JML is not a complete language in the sense some features are not completely implemented and

other features are being implemented along the time. There are many tools (i.e. JML tools) already developed and other tools in the process of development that use different features of the language. Thus, it may be difficult to manage JML evolution since some modifications or evolution in the language may affect tools that are (or not) dependent on the features that are being changed or evolved.

To tackle this situation, research groups working on JML divided the language in several language levels. As a result, JML become a modular language avoiding part of the dependence-related problems. This modularity turns JML a language easier to be used, studied and understood. Another advantage of this modularity is that JML tools need not to be aware of the whole language focusing their implementation only in specific language levels of interests.

JML has the following language levels, a more elaborated explanation about JML's levels can be found in [28]:

- Level 0 is the most used and fundamental level and constitutes the core of the language. Users must be familiar with this level. It contains language constructs needed to use JML as documentation, as a formal specification language and as a DbC language. In addition, all JML tools should implement Level's 0 features.
- Level 1 adds three categories of features to level 0: redundancy features, syntactic sugars, and features to support static verification [21] and run time assertion check (RAC) [15].
- Level 2 incorporates some features considered more specialized to certain uses of JML. Some Level's 2 features are used by JML tools and are important to describe the JML's semantics.
- Level 3 contains not well-understood features and features that are not implemented by several tools.
- Level C incorporates features used to verify and specify concurrent programs.
- Level X has experimental features and some of these features can be moved to other levels, eventually.

2.7 How JML Deals with Specification Inheritance

In JML, specifications present in a type are inherited by its subtypes, provided they are not private. This leads us to two concepts: join of specifications and specification inheritance. In this section these two concepts are described in details.

2.7.1 Join of specifications

In a program written in Java and annotated with JML, classes inherit not only attributes and methods from superclasses, they also inherit specifications of invariants, methods, history constraints, and initialisation predicates [40, 44]. Concerning methods, a method specification may consist of several specifications cases, which are introduced by the use of clauses such as **requires**, **assignable**, **ensures** [28]. Each specification case has a precondition that states when the corresponding specification case applies to a call. The

keyword **also** joins specification cases. When a precondition of a specification case holds, the corresponding postcondition must hold also. The definitions we present here are taken from [44]. The notation $T \triangleright (pre, post)$ is related to a specification case of an instance method that type checks when its receiver (**this**) has static type T. It also type checks in contexts where **this** has some subtype of T. In what follows, we introduce the definition of the join of JML method specifications [44].

Definition 1. ***(Join of JML method specifications)*** *Let $T' \triangleright (pre', post')$ and $T \triangleright (pre, post)$ be specifications of an instance method m. Let U be a subtype of both T' and T. Then the join of $(pre', post')$ and $(pre, post)$ for U, written $(pre', post') \sqcup^U (pre, post)$, is the specification $U \triangleright (p, q)$ with precondition p:*

$$pre \parallel pre'$$

and postcondition q:

$$(\backslash old(pre') ==> post') \;\&\&\; (\backslash old(pre) ==> post)$$

□

In Definition 1, the precondition of the join of two method specifications is their disjunction. The postcondition of the join is a conjunction of implications (written ==> in JML's notation), stating that when a precondition holds (in the pre-state), the corresponding postcondition must hold.

2.7.2 Specification Inheritance

Specifications of subtypes in JML inherit specifications, besides attributes and methods. First, we introduce some notation for type specification. For a type T, the invariant predicate declared in the specification of T (without inheritance) is denoted by $added\_inv^T$. Also for a type T, the history constraint predicate declared in the specification of T (without inheritance) is denoted by $added\_hc^T$ and the iniatially predicate in the specification of T (without inheritance) is denoted by $added\_init^T$. For a method m declared in a type T, the notation $added\_spec_m^T = (added\_pre_m^T, added\_post_m^T)$ is the join of the specification cases in type T for m. If m is declared in T with no specification and is not overriding any method, then $added\_spec_m^T = (true, true)$, which is the default specification in JML. We use $supers(T)$ to denote the set of all supertypes of T (including T) and $methods(\mathcal{T})$ to denote the set of all instance method names declared in the specifications of the types in a set $\mathcal{T}$.

Definition 2. ***(Extended specification)*** *Suppose T has supertypes $supers(T)$, which includes T itself. Then the extended specification of T is a specification such that:*
methods: *for all methods $m \in methods(supers(T))$, the extended specification of m is the join of all added specifications for m in T and all its proper supertypes*

$$ext\_spec_m^T = \bigsqcup^T \{added\_spec_m^U \mid U \in supers(T)\}$$

invariant: *the extended invariant of T is the conjunction of all added invariants in T and its proper supertypes*

$$ext\_int^T = \bigwedge^T \{added\_inv^U \mid U \in supers(T)\}$$

history constraint: *the extended history constraint of T is the conjunction of all added history constraints in T and its proper supertypes*

$$ext\_hc^T = \bigwedge^T \{added\_hc^U \mid U \in supers(T)\}$$

initially predicate: *the extended initially predicate of T is the conjunction of all added initially predicates in T and its proper supertypes*

$$ext\_hc^T = \bigwedge^T \{added\_hc^U \mid U \in supers(T)\}$$

□

The definitions we present here were introduced in [44] and are the ones we use in our work to build our laws.

2.8 Behavioral Subtyping

In JML, each type is a behavioral subtype [46] of each one of its supertypes [40, 44]. This characteristic is achieved using specification inheritance and methodological restrictions on invariants, etc. [40, 44]. In this section we briefly explain the notion of behavioral subtype JML enforces. For more details about this theme refer to [40, 44].

2.8.1 Refinement of Methods Specifications

The next definition, also extracted from [40], enforces the refinement of method specifications. Since $T' \triangleright spec$ is a specification of method that type checks with a receiver of static type T we have:

Definition 3. ***(refinement w.r.t.)*** *Let $T' \triangleright spec$ and $T \triangleright spec$ be specifications of an instance method m, such that T' is a subtype of T. Then $spec'$ refines spec with respect to T', written $spec' \sqsupseteq^T spec$, if and only if for all calls of m where the receiver's dynamic type is a subtype of T', every correct implementation of $spec'$ satisfies spec.*

One can notice that the refining specification, $spec'$ is stronger than $spec$, since, to satisfy $spec'$, an implementation has to be more restrict than it would be to satisfy $spec$. In other words, the set of implementations that satisfies $spec$ is bigger than the one that satisfies $spec'$.

2.8.2 A Definition of Behavioral Subtyping for JML

The current JML implementations relies on the notion of behavioral subtyping based on the Liskov and Wing's constraint-based definition [46]. In the sequel we show the definition extracted from [40].

Definition 4. ***(strong behavioral subtype)*** *Let T' be a type specification and let T be a specification for a supertype of T'. Then T' is a strong behavioral subtype of T if and only if:*
methods: *for all instances methods m in T, the method specification form m in T' refines that of m with respect to T',*
invariant: *the instance invariant of T' implies the instance invariant of T' for objects of type T',*

history constraint: *the instance history constraint of T' implies the instance history constraint of T for objects of type T',and*
initially predicate: *the instance initially predicate of T' implies the initially predicated of T for objects of type T'.* □

2.9 JML Tools

Many research groups and independent contributors have collaborated on JML, developing tools to cover several kinds of necessitates such as writing, and verifying JML specifications. The most basic tools of JML executes type checking and parsing. Additionally, there are tools to deal with static analysis (e.g. ESC/Java2 [21]), formal verifications (LOOP tool [39] and Krakatoa [11]), run time assertion checking (RAC) (for example the *jmlrac* tool [14]), unit test generation (*jmlunit* [10]), automated testing (JET [15]), and documentation generation (see the *jmldoc* [10] tool).

In this section we present a brief overview of four important JML tools: *jmlc*, JET, ESC/Java2 and Krakatoa. The first three were used in our work helping us to elaborate and validate our laws and checking the conformance of pre- and post-states of our case study programs.

2.9.1 The *jmlc*

The JML compiler (*jmlc*) is part of the official suite of JML tools [1] developed by the creators of the language. *jmlc* was developed at Iowa State University as an extension to the MultiJava compiler [20]. The goal of *jmlc* is to translate specifications into run time assertion checks under the form of bytecode. This bytecode is then inserted in the Java code to handle specifications violations, i.e. to execute the run time assertion checking (RAC) of the code. This checking is transparent in the sense that if the program execution violates no assertions, its behavior (i.e. the behavior of the program before compiled by *jmlc*) remains unchanged except for performance measures (time and space).

The use of *jmlc* follows three steps:

- Parser checking of Java code and its specifications;
- Program compilation with specification translation in run time assertion checking bytecode;
- Insertion of previous generated bytecode in the Java bytecode;
- And in the end, the execution of the compiled RAC-modified Java code.

The bytecode generated as output from *jmlc* functions like a regular Java bytecode, except for the fact that JML's runtime library is needed to execute the JML-compiled bytecode.

An important feature of *jmlc* is the mechanism of isolating and presenting the problems occurred in the RAC activity. It provides static information, stating exactly the specifications violated and the right place in the code where the violation was detected. Also, the *jmlc* provides dynamic information about the current values of variables at the moment the violation occurred and what method calls led to the violation.

2.9.2 JET

JET [15] is a tool for automated test of formally JML-specified Java classes. It tests each method of a class separately. Tests on JET are completely automated since each step of the test is performed automatically, including selection and generation of test data, execution of test and measurement results [17]. JET generates tests that check whether the execution of each method meets its specifications and the specifications of the class that declares it. Summarizing, these are the steps JET follows to test a method:

- First, a test case is created. Basically a test case for a specific method is formed by a receiver object and real parameters. Taking as an example our `addKgs` method of `Person` class (Figure 2.1), JET creates a `Person` object that acts as the receiver and generates a random integer to pass as argument to the expected formal parameter `kgs` . JET always generates or selects test data randomly. In the cases where the formal parameter are declared as a reference type, JET executes an algorithm that executes a series of method invocations preceded by constructor invocations to create random objects. Random test data generation is described in details in [17].

- Second, the test case is executed. The target method is called by the generated test case. At this point the Java class that declares the method is compiled by a JML compiler with run time assertion checks enabled. The JML-compiled version of the class is tested, in fact only the chosen method is invoked and the specifications of the class and of the method are tested during the test execution.

- Finally, a test pass or fail according to the occurrences of JML assertion exceptions. A method is executed only when its pre-condition is satisfied. In general when a pre-condition is not satisfied the test case is considered incompatible to test the method. Furthermore, when a postcondition is not satisfied the test fails meaning that the source-code does not meet the specification for that test case [16].

The most attractive feature of JET is the full automation of unit testing, from test data generation to test execution and test result determination. Using JET, we can verify whether Java programs are in conformance with their formal specifications with only one click.

2.9.3 ESC/Java2

ESC/Java2 [21] is an extension of the ESC/Java [29] tool, a pioneer tool in program static analysis and formal verification of formal annotated Java programs. ESC/Java2 accepts as input complete Java programs.

The major function of ESC/Java2 is to find common run-time errors in JML-specified Java programs by static analysis of the program source-code and its formal specification. ESC/Java2 has a built-in prover, called Simplify [25] that operates automatically to execute the static analysis. The amount of source-code that has to be checked and the types of checking routines are controlled by the users by annotating classes and methods they want with JML specifications.

ESC/Java2 consists of three macro phases,

- a parsing phase when also occurs syntax checking. In this phase parser errors and cautions are generated;

- a typechecking phase to validate types and execute usage checks;
- and the static checking phase that finds potential bugs, executing as a background process the Simplify prover. This phase produces warnings reporting the result of the static analysis.

The main warnings (i.e. the report generated after a program execution) are categorized as follows:

- Possible runtime exceptions, like cast, null pointer, division by zero and negative array index exceptions.
- Possible method specification violations: pre- and postconditions and modifies clauses written by users in program methods. For example, regarding our class `Person` of Figure 2.1. If we write in a method body a code like `person.addKgs(−1)` where `person` is an instance of `Person`, ESC/Java2 generates a warning like: *Warning: Precondition possibly not established (Pre) addKgs(-1);*
- Non null violations. These violations are generated by checks against **non_null** modifiers in specifications of fields and formal parameters.
- loop and flow specifications like **assert** specifications.
- possible class specification violations: invariants, history constraints and **initially** clauses.

ESC/Java2 does not always report real source-code violations or bugs. In fact, ESC/Java2 may produce false positives. However, this is not a functional bug in the tool, actually, this was a design decision. Eliminating this characteristic of the tool could make it not automatic requiring user interaction on the static checking execution. Despite this fact, ESC/Java2 is being used for many people and also in study cases [21, 18].

2.9.4 Krakatoa

Krakatoa [11] is a tool designed to verify Java programs annotated with JML specifications. The main focus of the Krakatoa development team is to address JavaCard programs [63], short programs which require high levels of formality and confidentiality. Besides these programs, Krakatoa also supports Java programs with certain restrictions.

The general purpose of Krakatoa is to verify whether Java programs or JavaCard are in conformance with their formal specifications. However, its activity is restricted to verifying the conformity of pre- and postconditions (contained in the specifications), invariants of classes, as well as behavioral exceptions. The verification is made proving that preconditions and invariants are true at the beginning of a method call and, therefore, that invariants and post-conditions are valid at the end of the method execution. In this dissertation the versions we use for tests were 0.6x and 1.11 (the latest known version up to the writing of this text).

The environment of Krakatoa supports only the following JML constructs for methods: invariant, requires, assignable, ensures and signal clauses, as well as loop-invariants and decreases clauses for while-loops and for-loops. For recursive methods, the clause `\measure_by` is not supported, and the proof of the correctness of such methods is only

partial since Krakatoa does not prove their termination. In assertions (inside annotations), Krakatoa supports a specific subset of constructs, these are: **old**, **result**, **forall**, \exists, \fresh, \not_modified, and specific constructs related to the **assignable** clause, like \nothing and \everything (for a complete list see [11]). Model fields are also supported and are interpreted as new class attributes, however the use of model fields with **represents** clause is not allowed yet.

The Krakatoa's approach excels for the originality of its methodology. To certify Java annotated programs, the tool translates the program into a input language for Why [11], a stand-alone tool that produces proof obligations for programs written in its own language, which was created especially to perform certification of programs. Why uses a methodology based on a functional interpretation that utilizes static analysis of effects and monads and a weakest pre-conditions calculus. The Why input language is a ML-like minimal language with limited imperative characteristics. Why has the capacity to generate output for several theorem provers as Coq [5], Simplify [25] and ergo [22].

Chapter 3

Laws

3.1 Introduction

The refactoring activity consists in changing a program structure, to accommodate new requirements or to improve code structure, without changing its observable behaviour [31, 55].

Nowadays, the use of refactoring is a common activity among developers, and recommended by Extreme Programming (XP) [4] practitioners. Integrated Development Environments (IDEs) like Eclipse [30] and Visual Studio [56] give automatic support to apply refactorings for Java [35] and C# [24], respectively. However, such support do not work perfectly and present erroneous behavior in certain situations [60]. The reason is that these IDEs do not build its refactoring implementations on any kind of rigorous or systematic activity.

Programming laws [38] are a good alternative to transform programs in a systematic and rigorous way. In the context of object-oriented programming, Borba *et al* [7] and Cornélio [23], developed a set of programming laws for a language, named ROOL [12], that is a subset of Java, buth with a copy semantics. They focused efforts on ROOL's object-oriented features and presented how that set of laws is sufficient to transform a program into on in a normal-form expressed in a small set of constructs of the language.

Cornélio uses programming laws [23] to prove refactorings proposed by Fowler [31]. Each little change in the program is accomplished by the application of a law. To apply a law some conditions must be satisfied. This approach does not require tests because there are proofs to ensure that the programming laws are behavior-preserving, provided that the conditions for application are met. Cornélio proved the ROOL laws using a formal semantics of the language, ROOL [12].

Duarte [26] adapts programming laws initially proposed for ROOL to the programming language Java. As Java presents more constructs than ROOL, Duarte introduces laws for dealing with *contructors* and *static methods*, for instance.

We characterize the systems where our laws can be applied as limited open systems [26] in which classes of our systems can only depend on external libraries and no external elements depend on them. We consider that these systems are codified in only one package, the *default* package. We also assume that the identifiers of our classes are distinct from those of external libraries.

In this chapter, we introduce programming laws to deal with Java programs annotated with the Java Modeling Language (JML). Some laws cope with Java elements of a program, but it is necessary to take into consideration the existence of JML annotations.

Besides these laws, we present laws that only handle the transformation of annotations written in JML.

3.2 General conventions

The laws are written in an equational style. Each side of the equation corresponds to a template of a well-formed program. Programming laws relate the left-hand and the right-hand sides by equality, along with side conditions. These laws precisely indicate the modifications that can be done to a program, stating their corresponding proof obligations. In fact, to apply a law, it is necessary to check (syntactic or semantic) side-conditions that ensure that the transformation is behavior-preserving and also maintains the program well-formedness. In our approach we consider that we are dealing with only one package and working in a limited open system [26], in which classes of our system can depend on external libraries, but external classes do not depend on classes of our system.

The laws may have two sets of proof obligations (provisos). The one started with "JML" denotes the set of JML provisos. Regarding the "Java" set, it involves only Java elements for stating conditions.

A JML-annotated Java program has the format *cds Main*, where *cds* is the set of all classes of the program and *Main* corresponds to the unique class in the program that has a *main* method. It is important to emphasize the notion of equivalence used to compare equations. We use $cds_1\ Main_1 = cds_2\ Main_2$ to denote the equivalence of sets of classes declarations cds_1 and cds_2, i.e. to denote that the observable behavior of both sets is the same. We need to stress that this definition take into account only sequential programs.

The notation $cd_1 =_{cds,Main} cd_2$ is an abbreviation for $cds\ cd_1\ Main = cds\ cd_2\ Main$, meaning that the class declarations cd_1 e cd_2 are equivalent and *cds* refers to the set of all other classes of a program except for the *Main* class.

In some laws, we write $cd_1 \sqsubseteq_{cds,Main} cd_2$. This term is an abbreviation of $cds\ cd_1\ Main \sqsubseteq cds\ cd_2\ Main$, and means that the class declaration cd_1 is refined by cd_2.

The expressions *cnds, ads* and *mds* that appear inside a class represents the class constructors, attributes and methods, respectively. We have to emphasize that these expressions also contain the respective specifications of each constructor or method. It is not only Java code, we can also have the corresponding JML specifications.

We write *rt m (pds) { mbody }* to represent a method declaration where *m* is the method name, *rt* is its return type, *pds* is the list of formal parameters and *mbody* is the method body. We write $\alpha(pds)$ to denote the identifiers of a of formal parameters *pds*. We use the function $vardecs(pds, e)$ that introduces a list of variables which have the same names and types of the formal parameters pds, and are initialized exactly with the values of the arguments (e) used to call the method.

We use $B.a$ when we want to refer the access of an attribute named a by means of expressions of static type B, strictly. The notation $B.m$ refers to a call to a method named m by means of an expression of static type B, strictly. The subclass relationship is denoted by the symbol '$\leq$', thus '$B \leq C$' denotes that B is a subclass of C. The T symbol is used to represent an attribute type.

Predicates are described by the Greek letter ψ. Frame axioms are described by the Greek letter ω and represents a list of store-references [28] (see Section 2.4.1.2). A store-reference denotes a set of memory locations in general.

We write $@invs$, $@cons$ and $@inis$, in laws to denote the set of invariants, history constraints and initially clauses of a class, respectively. We use $@spec\_cases$ to rep-

resent a set of specification cases of a method. The notation @*spec_cases* can denote either one specification case, many specification cases or none. In addition, this notation may be used in conjunction with other specification cases. In situations like that, the set represented by @*spec_cases* starts with the **also** keyword, in order to guarantee the specification cases well-formedness. The same reasoning is applied to the previous at-sign-started expressions.

The use of the expressions @*invs*, @*cons*, @*inis*, and @*spec_cases* in the description of the laws is not mandatory. To simplify the laws descriptions, we write explicitly only the expressions used in the side-conditions of the laws. This fact does not mean that the classes described in the laws have no invariants or history constraints, for example. Every law of our set of laws was created considering the JML specifications inside the program.

We introduce here, various functions that are used throughout the text. The Functions *fspec*, *fpre* and *fpos*, when applied to a method, return its specification cases, pre- and post-conditions, respectively. Particularly the functions *fpre* and *fpos* may accept as input a set of specification cases (e.g. @*spec_cases*). In this case, these functions return the join of the pre- and postconditions of the set of specification cases, respectively. The functions *finv*, *fcons* and *finit* return the invariants, history constraints and initially clauses of a class, respectively. All these functions do not consider inherited specifications. To consider inherited specifications we use *fext_spec fext_pre*, *fext_pos*, *fext_inv*, *fext_cons* and *fext_init*. We use the convention *C.m[pds]*, to refer a method *m*, with formal parameters *pds* of some class *C*. Therefore, *fpre(C.m[pds])* returns the pre-conditions of method *m* of class *C*. Table 3.1 summarize all the cited functions.

Besides the functions mentioned above, we use the function *fassign(specc)* that accepts as argument a specification case or a set of specification cases and returns the set of locations attached to the **assignable** clause of *specc* or the union of the set of locations attached to all **assignable** clauses pertaining to each one of specification cases of the set.

Function name	**Accepted Inputs**	**Return inherited specifications?**
fspec	A method, in the style *C.m[pds]*.	No
fpre	A method in the style *C.m[pds]*, or a set of specification cases like @*spec_cases*.	No
fpos	A method in the style *C.m[pds]*, or a set of specification cases like @*spec_cases*.	No
fext_spec	A method in the style *C.m[pds]*.	Yes
fext_pre	A method in the style *C.m[pds]*, or a set of specification cases like @*spec_cases*.	Yes
fext_pos	A method in the style *C.m[pds]*, or a set of specification cases like @*spec_cases*.	Yes
finv	A class name	No
fcons	A method in the style *C.m[pds]* [1].	No
finit	A class name.	No
fext_inv	A class name.	Yes
fext_cons	A method in the style *C.m[pds]*.	Yes
fext_init	A class name.	Yes

Table 3.1: Auxiliary functions used in the laws

We write '→' to indicate the conditions that must to be satisfied to apply a law from

left to right. Likewise, we use '←' to indicate what have to be satisfied to allow the application of a law in the opposite direction. Conditions that must hold in both directions are indicated by '↔'.

In Section 3.3 we present a subset of the laws developed in this work. This subset consists in the most significant and interesting laws we created. The complete set of laws is presented in Appendix A.

3.3 Laws

Some of the laws described here are inspired on the laws previously described by Borba [7], Cornélio [23] and Duarte [26]. There are laws completely new since we are not aware about other works in the same direction. We described the "Java parts" of our laws extracting the main concepts of the laws, defined in [7, 23, 26], that deal with object-oriented code. The laws we describe are "JML-aware". We also have laws that only deal with JML annotations. However, all laws defined in this work take into consideration Java and JML elements of the program. With respect to JML elements, we mainly focused on a sub-set of JML's Level 0 constructs, specially the ones used in lightweight specifications. Some specific and common constructs of Level 1 (e.g. the **pure** modifier) are considered too.

Our laws follow the general conventions adopted in Section 3.2. We categorize our laws in seven sections: Classes, Invariants, Attributes, Methods, Constructors, Commands and Expressions, and Predicates.

3.3.1 Classes

Classes that are no longer used in a program can be eliminated. In the case of introducing a new class, we need to check whether the new class name is already present in the program and if the superclass of the new class is valid [2].

Law. ⟨*class elimination*⟩
$cds\ \ cd_1\ \ Main\ =\ cds\ \ Main$

provided

JML:

(→) The class declared in cd_1 is not referred in any specification declared in cds or $Main$.

Java:

(→) The class declared in cd_1 is not referred in cds or $Main$.

(←) (1) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; (2) The superclass appearing in cd_1 is either **Object** or declared in cds.

□

It is possible to make a concrete class abstract if this class is not instantiated in any place in the program. In contrast, we can make a abstract class concrete if its all methods are concrete.

[2] Remember that *Object* is also considered a valid class in Java.

Law. ⟨*make class abstract*⟩

class C **extends** D {
ads
$cnds$
mds
}
	$=_{cds,Main}$	**abstract class** C
		extends D {
		ads
		mds
		}

provided

JML:

(→) '*new C*' does not occur inside specifications of cds, $Main$, $cnds$ nor mds.

Java:

(→) '*new C*' does not occur in cds, $Main$, $cnds$ nor mds.

(←) Every method m of mds is concrete.

□

One can change the superclass of a class from `Object` to any other class if they do not share attributes with same names. In addition, it is necessary to guarantee that the invariant, history constraint and initially clauses of the superclass are weaker than the corresponding invariant, history constraint and initially clauses of the target class. And more, if the target class or one of its subclasses has any method with the same signature to any method of the superclass, the specification of the superclass' method must be the stronger pre-condition and the weaker postcondition regarding the specification of the method with same signature declared in the target class or in one of its subclasses.

Law. ⟨*change superclass: from **Object** to another class*⟩

class C **extends Object** {
$@invs$
$@cons$
$@inis$
 	
ads
$cnds$
mds
}
$cds,\ Main$
	=	**class** C **extends** D {
		$@invs$
		$@cons$
		$@inis$
		
		ads
		$cnds$
		mds
		}
		$cds',\ Main$

□

where

$cds' \mathrel{\widehat{=}} cds$[//@ **also** $fspec(m)/fspec(m)$], for every method m in mds that is a redefinition of a method introduced in some class E such that $D \leq E$.

provided

JML:

(→) (1) $@invs \Rightarrow finv(D)$; (2) $@cons \Rightarrow fcons(D)$; (3) $@inis \Rightarrow finit(D)$; (4) For any method m in mds that redefines a method m declared in D or in any class E

such that D $\leq E$, $fpre(E.m[pds]) \Rightarrow fpre(C.m[pds])$ e **old**$(fpre(E.m[pds])) \Rightarrow (fpos(C.m[pds]) \Rightarrow fpos(E.m[pds]))$.

(←) (1) C or any of its subclasses in cds is not used in type casts or tests involving any expression of type D or of any supertype of D in specifications; (2) **this**.a does not appear in specifications of C, nor in specifications of C's subclasses, for any attribute a of D or of any superclass of it with specification visibility default, protected or public; (3) $le.a$, for any $le : C$, does not occur in specifications in cds or $Main$, for any a of D or of any superclass of it with specification visibility default, protected or public; (4) There is no method call $E.m$ inside specifications of cds, for any pure method m, such that $E \leq C$ and m is declared in D or in any of its superclasses, but is not redefined in mds; (5) **super** does not appear in any specification of C.

Java:

(→) All attributes in ads and in subclasses of C are distinct from those declared in D and in superclasses of D.

(←) (1) C or any of its subclasses in cds is not used in type casts or tests involving any expression of type D or of any supertype of D;

(2) There are no assignments of the form $le = exp$, for any le whose declared type is D or any superclass of D and any exp whose type is C or any subclass of C;

(3) Expressions of type C or of any subclass of C are not used as value arguments in method/constructor calls with a corresponding formal parameter whose type is D or any superclass of D;

(4) Expressions whose declared type is D or any of its superclasses are not returned as a method result in calls with an expected result whose declared type is C or any subclass of C;

(5) **this**.a does not appear in C, nor in any subclass of C, for any public or protected attribute a of D or of any of its superclasses;

(6) $le.a$, for any $le : C$, does not appear in cds or c for public or protected attribute a of D or of any of its superclasses;

(7) There is no $E.m$, for any method m such that, $E \leq C$ and m is declared in D or in any of its superclasses, but is not redefined in mds.

(8) **super** does not appear in any method in mds.

Concerning changing the superclass of a class to `Object` much more conditions need to be satisfied, both Java and JML conditions.

3.3.2 Invariants

Law ⟨*move invariant to superclass*⟩ allows us to move an invariant ψ_2 from a subclass to its superclass. To apply this law in any direction, we require that calls to **super** do not occur in ψ_2, since after law application (in both directions) these calls may refer different elements. To apply this law from left to right, model fields cannot appear in ψ_2 and occurrences of **this** must be cast otherwise the elements they refer may not be visible.

Law. ⟨*move invariant to superclass*⟩

```
class B extends A {                    class B extends A {
  //@ private invariant ψ1;                //@ private invariant ψ1
  @invs                                        && ψ'2;
                                           @invs
  ads                                    ads
  cnds                                   cnds
  mds                                    mds
}                                      }
class C extends B {        =cds,Main   class C extends B {
  //@ private invariant ψ2;              @invs'
  @invs'
                                       ads'
  ads'                                 cnds'
  cnds'                                mds'
  mds'                                 }
}
```

where

$\psi'_2 \mathrel{\widehat{=}}$ **this instanceof** C ==> ψ_2

provided

JML:

(↔) **super** does not appear in ψ_2.

(→) ψ_2 does not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

□

Concerning the soundness of this law, we take in account the inheritance of specifications in JML (Section 2.7), in which inherited invariants are conjoined with locally added invariants. On the left-hand side, the invariant ψ_2, which is present in class C, is inherited by the subclasses of C and holds for all subclasses. On the right-hand side of the law, the invariant ψ'_2 (notice that ψ'_2 is actually ψ_2 with an antecedent condition) is inherited by all subclasses of B besides those that are not subclasses of C. For those classes that are subclasses of B, but not subclasses of C, the invariant holds because for objects of these classes the antecedent **instanceof** C fails and the whole implication is true, not changing the meaning of any original local invariant that inherits ψ'_2.

The following three laws allow us to change the specification visibility of an invariant from default to public, from public to private and from protected to private, as well as in the inverse directions.

Law. ⟨*change invariant visibility: from default to private*⟩

class C **extends** D {
//@ **invariant** ψ_1;
@$invs$
 	
ads
$cnds$
mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		//@ **private invariant** ψ_1;
		@$invs$
		
		ads
		$cnds$
		mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has non-private specification visibility;

□

Law. ⟨*change invariant visibility: from public to private*⟩

class C **extends** D {
//@ **public invariant** ψ_1;
@$invs$
 	
ads
$cnds$
mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		//@ **private invariant** ψ_1;
		@$invs$
		
		ads
		$cnds$
		mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has public specification visibility;

□

Law. ⟨*change invariant visibility: from protected to private*⟩

class C **extends** D {
  //@ **protected invariant** ψ_1;
  @*invs*
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  //@ **private invariant** ψ_1;
		  @*invs*
		
		  ads
		  cnds
		  mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has non-public specification visibility;

□

We can apply directly anyone of the laws ⟨*change invariant visibility: from default to private*⟩, ⟨*change invariant visibility: from public to private*⟩ and ⟨*change invariant visibility: from protected to private*⟩ from left to right, i.e. from any visibility to private since a private invariant can refer elements of any visibility. Regarding the application of the laws ⟨*change invariant visibility: from default to private*⟩, ⟨*change invariant visibility: from public to private*⟩ and ⟨*change invariant visibility: from protected to private*⟩ in the opposite direction it is necessary to check if the new visibility of the invariant is at least as permissive as the visibility of all referred attributes, pure methods and model fields.

The **Law** ⟨*collapse invariants*⟩ represents a JML syntactic sugar (see Section 2.4.5). One invariant written in more than one **invariant** clause (left-side of the law) can be simplified to an unique invariant clause separating each predicate by a '&&' (and) operator.

Law. ⟨*collapse invariants*⟩

class C **extends** D {
  //@ **private** *invariant* ψ_1;
  //@ **private** *invariant* ψ_2;
  ...
  //@ **private** *invariant* ψ_n;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  //@ **private** *invariant* ψ_1
		  && ψ_2 && ... && ψ_n;
		  ads
		  cnds
		  mds
		}

□

3.3.3 Attributes

In this subsection we present laws that deal with attributes. It is important to emphasize at this point that we are considering that programs are coded in just one package, the default

package [3]. Thus, all attributes (except, the private ones) can be accessed in any part of the program.

In Java, we have only three laws to alter the visibility of an attribute: from default to public, protected to public and private to public. But now, due to JML, we need to take into account the concept of specification visibility (refer to Section 2.4.4 for details). In fact, we have that attributes with less restricted visibilities can be accessed in more restricted specification visibilities contexts, i.e. a public attribute can be accessed in specifications with any more restricted specification visibility (protected, default or private), and public as well. Thus, we now have fourteen laws to cover all possible visibility changes situations, including Java visibility and specification visibility modifications. We show three of them in the sequel while the other laws can be found in Appendix A.3.

Making an attribute public since it is currently private is straightforward. Nevertheless, the opposite has to respect some conditions.

Law. ⟨*change attribute visibility: from private to public*⟩

class C **extends** D {
  private T a;
 	
  ads
  $cnds$
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  public T a;
		
		  ads
		  $cnds$
		  mds
		}

provided

JML:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in any specification of cds or $Main$; (2) $C.a$, occurs only inside specifications of C with private specification visibility.

Java:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.a$ occurs only in C's body.

□

Considering Java, we have to assure that accesses to the attribute occur only through instances (including **this**) of static type equal to the class declares it and inside the class. Taking into account JML, we have two provisos due to specification conformance maintenance, specially specification visibility, all accesses to the attribute should appear in specifications with private visibility. In fact, the specification visibility can not restrict the current Java visibility. Hence, an access to private attributes can only occur inside private specification visibility specifications.

JML provides alternatives to modify the specification visibility of attributes, methods and model fields. An attribute can have its specification visibility modified using the special JML modifiers **spec_protected** and **spec_public**. We find **spec_public** modifier in **Law** ⟨*change spec public attribute visibility: from private to public*⟩, a law similar to **Law**

[3] Extending our scope to consider also programs with different packages could discharge modifications in our laws that deal with visibility.

18, but that works for private (but specification public) attributes. Regarding Java, private attributes adorned with the **spec_public** modifier are still private, but regarding JML they are public. Thus, it can appear in specifications with public specification visibility. Hence, the **Law** ⟨*change spec public attribute visibility: from private to public*⟩ is equals to **Law 18** except for a JML condition that is no longer necessary.

Law. ⟨*change spec public attribute visibility: from private to public*⟩

```
class C extends D {
  private
    /*@ spec_public @*/ T a;

  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  public T a;

  ads
  cnds
  mds
}
```

provided

Java:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.a$ occurs only in C's body.

□

Law ⟨*change specification visibility of private attribute: from private to public*⟩ shows how it is possible to modify the specification visibility of an attribute. Applying this law from the left to right is straightforward, whereas, from the right to left it is necessary to guarantee that the attribute is referenced only inside specifications with private specification visibility.

Law. ⟨*change specification visibility of private attribute: from private to public*⟩

```
class C extends D {
  private T a;

  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  private /*@ spec_public
          @*/ T a;

  ads
  cnds
  mds
}
```

provided

JML:

(←) a, occurs only inside specifications with private specification visibility.

□

By using **Law** ⟨*move reference type attribute to superclass*⟩, we can move an attribute to a superclass if it is not already declared in the superclass and if it does not cause name conflicts. The application of **Law** ⟨*move reference type attribute to superclass*⟩, from right to left, allows us to move an attribute to a subclass. In this case, we allow only accesses

to a by C or subclasses of C, including accesses that appear in specifications.

Law. ⟨*move reference type attribute to superclass*⟩

class B **extends** A {
ads
$cnds$
mds
}
class C **extends** B {
public /*@ **nullable** @*/ T a;
ads'
$cnds'$
mds'
}
	$=_{cds,Main}$	**class** B **extends** A {
		public /*@ **nullable** @*/ T a;
		ads
		$cnds$
		mds
		}
		class C **extends** B {
		ads'
		$cnds'$
		mds'
		}

provided

JML:

(←) $D.a$ does not occur inside specifications in cds, $Main$, $cnds$, $cnds'$, mds nor mds', for any $D \leq B$ and $D \not\leq C$.

Java:

(↔) T is not a primitive type.

(→) (1) a is not declared in ads; (2) The attribute name a is not declared by the subclasses of B in cds.

(←) $D.a$ does not occur in cds, $Main$, $cnds$, $cnds'$, mds nor mds', for any $D \leq B$ e $D \not\leq C$.

□

In **Law** ⟨*move reference type attribute to superclass*⟩, we consider only attributes whose type is a reference type. There is another law (**Law 28**) for moving an attribute of primitive type. The reason for having two disctinct laws for dealing with attributes of primitive and reference types comes from the **nullable** keyword in **Law** ⟨*move reference type attribute to superclass*⟩. In JML, any declaration (except for local variables) whose type is a reference type is implicitly declared to be not null, except when in the declaration appears the a **nullable** modifier. Thus, by default, JML always checks if a attribute is null in all visible states of the class that declares it. When we move an attribute to a superclass, this is not aware about the newly moved attribute and, therefore, this action can cause a undesirable behavior. In fact, if one instantiates the superclass, JML will raise an invariant exception reporting that the new attribute is null. To avoid this, we force attribute nullability to move it up. If we want to move a non-null a attribute, it is necessary to introduce the **nullable** modifier before moving it.

We introduce the modifier **nullable** by applying **Law 26**. Remember that, in Java, only reference types can be null.

The type of an attribute may be modified to a superclass type, if every occurrence of the attribute inside specifications and in source-code is cast with the current attribute type or subtype. These conditions have to hold also when one changes the attribute type to any

type corresponding to a subclass of it. However, in this case it is also a requirement to check if the expressions assigned to the attribute are of the same type or of any subtype of it. The **Law** ⟨*change attribute type*⟩ allows us to change an attribute type.

Law. ⟨*change attribute type*⟩

class B **extends** A {
  public T a;
  ads
  $cnds$
  mds
}
	$=_{cds,Main}$	**class** B **extends** A {
		  public T' a;
		  ads
		  $cnds$
		  mds
		}

provided

JML:

(↔) $T \leq T'$ and every occurrence of a inside specifications of B, cds and $Main$, is cast with T or any subtype of T in cds.

Java:

(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of mds, cds e $Main$, is cast with T or any subtype of T in cds.

(←) Every expression assigned to a, in mds, cds e C, is of type T any subtype of T.

□

3.3.4 Methods

In this section we show laws that deal with methods as well as laws that treat methods specifications. Although history constraints are type specifications they impose constraints to methods. We do not define laws to deal with history constraints, but some laws take into consideration the existence of them to define side-conditions.

Law ⟨*weaken pre-condition*⟩ is an adaptation of the law *weaken precondition* defined by Morgan [52]. A predicate ψ'_1, is weaker than another predicate ψ_1, if $\psi_1 \Rightarrow \psi'_1$. This law and the next one are refinement laws.

Law. ⟨*weaken pre-condition*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$\sqsubseteq_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ'1;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(1) $\psi_1 \Rightarrow \psi'_1$; (2) $\psi'_1 \Rightarrow fpre(B.m[pds])$, for every class B such that $B \leq C$.

□

Given a method m with pre-condition ψ_1 declared in a class C, it is possible to apply the **Law** ⟨*weaken pre-condition*⟩, if the new pre-condition i.e. ψ'_1 is weaker than ψ_1. Furthermore, we must ensure that ψ'_1 implies each pre-condition of redefined methods m in subclasses of C. The previous proviso guarantees that the new pre-condition does not weaken the contract of redefinitions of methods m in subclasses. In other words, we can weaken a pre-condition ψ_1, if the new pre-condition is stronger than the pre-conditions of the redefinitions of m.

Another law adapted of Morgan's work is **Law** ⟨*strengthen post-condition*⟩. If a predicate ψ'_2, is stronger than another predicated ψ_2, if $\psi'_2 \Rightarrow \psi_2$. Given a method m with pre-condition ψ_1 and postcondition ψ_2 declared in a class C, it is possible to apply the **Law** ⟨*strengthen post-condition*⟩, if the new postcondition i.e. ψ'_2 is stronger than ψ_2. In addition, we must guarantee each postcondition of redefined methods m in subclasses of C implies ψ'_2. The previous proviso guarantees that the new postcondition does not strengthen the contract of the redefined methods m in subclasses. In other words, the condition (2) assures that whenever a call to m – by an object of type C – satisfy ψ_1, and ψ_2 is true, then ψ'_2 will also hold. Notice that simplifying the condition (2) by omitting its dependence on the pre-condition makes this condition more restrictive than it should be [40].

Law. ⟨*strengthen post-condition*⟩

class C **extends** D {
ads
$cnds$
 	
//@ **requires** ψ_1;
//@ **assignable** ω;
//@ **ensures** ψ_2;
@$spec\_cases$
rt $m(pds)$ {
$mbody$
}
 	
mds
}
	$\sqsubseteq_{cds,Main}$	**class** C **extends** D {
		ads
		$cnds$
		
		//@ **requires** ψ_1;
		//@ **assignable** ω;
		//@ **ensures** ψ'_2;
		@$spec\_cases$
		rt $m(pds)$ {
		$mbody$
		}
		
		mds
		}

provided

JML:

(1) $\psi'_2 \Rightarrow \psi_2$; (2) **old**$(\psi_1) \Rightarrow (fpos(B.m[pds]) \Rightarrow \psi'_2)$, for every class B such that $B \leq C$.

□

In some situations, to write a specification case for a method we need exactly the same pre-condition as that specified in the other non- **same** specification cases of a method or in the case of an override method we want, for instance, to write another post-condition (a stronger one) using the same pre-condition of the supertypes. In such cases, we should use the **same** keyword that stands for the disjunction of the pre-conditions in all non- **same** specification cases of the method in question together with all pre-conditions inherited from the methods specifications of its supertypes. We can insert or remove a specification case with **same** and postcondition default (true) if the method is an override or if the method has other specification cases and these specification cases must be non- **same**.

Law. ⟨*insert* **same** *specification case*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires \same;
  //@ assignable
                \not_specified;
  //@ ensures true;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(←) (1) @*spec_cases* has at least one specification case or *rt m*(*pds*) is an override; (2) @*spec_cases* does not have a specification case with pre-condition equals to **same**.

□

One can change a **assignable** clause of a specification case from **not_specified** to **everything** directly regarding the method specification is a lightweight specification.

Law. ⟨*change* **assignable** *from* **not_specified** *to* **everything** ⟩

assignable **not_specified**; = **assignable** **everything**;

□

Given that all specification cases of a method have a **nothing assignable** clause, we can make it pure. Recall that in Section 2.4.5 we showed that pure methods uses **assignable** **nothing** clause as default. In contrast, to transform a pure method in a non-pure one, we need to assure that this method is not called in any specification of the program. The **Law** ⟨*make method* **pure**⟩ allows us to make a method, pure.

Law. ⟨*make method* **pure**⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(↔) For all specification case *specc* such that $specc \in @spec\_cases$, $fassign(specc)$ is equivalent to **nothing**.

(←) $B.m(e)$ does not appear in specifications of cds, $Main$ nor in specifications of C, for any B such that $B \leq C$ and B does not redefine m.

□

The next three laws (**Law** ⟨*collapse pre-conditions*⟩, **Law** ⟨*collapse post-conditions*⟩ and **Law** ⟨*collapse* **also** *combinations*⟩), represent JML syntactic sugars (see Section 2.4.5). **Law** ⟨*collapse pre-conditions*⟩ and **Law** ⟨*collapse post-conditions*⟩ are rather similar to the **Law** ⟨*collapse invariants*⟩. The **Law** ⟨*collapse* **also** *combinations*⟩ executes the join (recall Section 2.7.1 for more details) of all specifications cases of a method in only one specification case. It is important to emphasize the result obtained when various **assignable** clauses are joined: the join of the locations of two or more **assignable** clauses is the union of these locations as we can see in the *where* clause of **Law** ⟨*collapse* **also** *combinations*⟩.

Law. ⟨*collapse pre-conditions*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ11;
  //@ requires ψ12;
  ...
  //@ requires ψ1n;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ11 && ψ12
       && ... && ψ1n;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law. ⟨*collapse post-conditions*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ21;
  //@ ensures ψ22;
  ...
  //@ ensures ψ2n;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ21 && ψ22
       && ... && ψ2n;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law. ⟨*collapse* ***also*** *combinations*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ11;
  //@ assignable ω1;
  //@ ensures ψ21;
  //@ also ...
  //@ also
  //@ requires ψ1n;
  //@ assignable ωn;
  //@ ensures ψ2n;
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ11
       || ... || ψ1n;
  //@ assignable ω;
  //@ ensures
    (\old(ψ11) ==> ψ21) &&
    ...
    && (\old(ψ1n) ==> ψ2n)
  rt m(pds) {
      mbody
  }

  mds
}
```

where

$\omega \mathrel{\widehat{=}} \omega_1 \cup \omega_2 \ldots \cup \omega_n$

□

As we described in Section 3.3.3 for attributes, changing visibility imposes constraints on how Java treats visibility modifiers and how JML deals with specification visibility. To deal with methods we also have to write laws to deal with pure methods. We define several laws to address all possible visibility modifications. Here we present **Law** ⟨*change pure method visibility from: private to public*⟩ and **Law** ⟨*change specification visibility of pure private method: from private to public*⟩. The complete set of method visibility laws, including laws to deal with non-pure methods, is found in Appendix A.4.

By applying **Law** ⟨*change pure method visibility from: private to public*⟩, we can change the visibility of a pure method from private to public or from public to private. In lightweight specifications, the specification visibility of the specifications cases of a method is the same as the method [4]. Hence, to change the visibility of a method from private to public we need only to check if the attributes, pure methods and model fields that appear in the specification cases of the method are public. Recall the fact that public specifications can refer only public elements.

[4] Recall that our laws were described considering only lightweight specifications for methods

Law. ⟨*change pure method visibility from: private to public*⟩

class C **extends** D {
ads
$cnds$
 	
$@spec\_cases$
private /*@ **pure** @*/
$rt\ m(pds)$ {
$mbody$
}
mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		ads
		$cnds$
		
		$@spec\_cases$
		public /*@ **pure** @*/
		$rt\ m(pds)$ {
		$mbody$
		}
		mds
		}

provided

JML:

(→) Every attribute, pure method and model field that occurs in $@spec\_cases$ has public specification visibility.

(←) (1) $B.m(e)$, for any $B \leq C$ except of strict type C, does not occur in any specification of cds or $Main$; (2) $C.m(e)$ occurs only inside specifications – that appears only in C's body – with private specification visibility.

Java:

(←) (1) $B.m(e)$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.m(e)$ occurs only in C's body.

□

To change the visibility from public to private it is necessary to satisfy several conditions. In the context of JML, we need to assure that calls to the method appear only inside specifications with private visibility of class C. Regarding Java we, need to ensure that calls to the method appear only inside C.

Law ⟨*change specification visibility of pure private method: from private to public*⟩ changes the specification visibility a the method imposing restrictions only in JML elements. One can modify the specification visibility of a method from private to public if the attributes, pure methods and model fields that appear in the specification cases of the method are public.

Law. ⟨*change specification visibility of pure private method: from private to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ pure @*/ rt
  m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_public
  pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in @*spec_cases* has public specification visibility.

(←) *m*(*e*) occurs only inside specifications with private specification visibility.

□

In Java, introducing a method redefinition is possible if the original method is not abstract and the target class does not declare a method with the same name. However, the presence of JML creates imposes some restrictions. Invariants and history constraints of subclasses can not restrict attributes and model fields of their superclasses. Ruby [59] refers to this constraint as a rule called *Super-call authorization rule*. He says that "a superclass method may only be called by subclass methods, if it has not been invalidated by that subclass". This is the condition (from the left to right) addressed in **Law** ⟨*introduce void method redefinition*⟩. To exemplify, suppose the invariant @*invs* of class *B* restricts a integer attribute x of B with the following predicated, $x > 10$, and also suppose that *mbody* is $x = 12$. Consider that @*invs*′ has a predicated like $x > 12$. If a call to m via **super** occurs in the body of a redefinition of m in class C, the invariant $x > 12$ of C will break.

Law. ⟨*introduce void method redefinition*⟩

```
class B extends A {
  @invs
  @cons

  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  void m(pds) {
      mbody
  }
  mds
}
class C extends B {
  @invs′
  @cons′

  ads′
  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  @invs
  @cons

  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  void m(pds) {
      mbody
  }
  mds
}
class C extends B {
  @invs′
  @cons′

  ads′
  cnds′

  @spec_cases
  void m(pds) {
      super.m(α(pds));
  }
  mds′
}
```

provided

JML:

(↔) (1) @*invs′* and @*cons′* does not restrict attributes in *ads*, model fields of *B* or any attribute or model field inherited by *B*.

Java:

(→) *m*(*pds*) is not abstract and is not declared in *mds′*.

□

Law ⟨*move original method to superclass*⟩ allows us to move an original method from a class to its superclass. The proviso concerning **super** is needed because its semantics may be affected when we move it from a subclass to a superclass, or vice versa. We can only move the specification of a method if it does not refer to model fields, attributes and pure methods of the class in which the method is originally declared. Also, the precondition ψ_1 must be stronger than the precondition of any method with signature *rt m*(*pds*) declared in subclasses of *B*. On the other hand, the postcondition must be weaker than that of methods declared in subclasses of *B* in *cds*. The *where* clause of the law ensures that

if we have a method with signature *rt m*(*pds*) with a specification declared in subclasses, the specifications are modified to start with an **also** keyword.

Law. ⟨*move original method to superclass*⟩

class B **extends** A {
ads
$cnds$
mds
}
class C **extends** B {
ads'
$cnds'$
 	
//@ **requires** ψ_1;
//@ **assignable** ω;
//@ **ensures** ψ_2;
$rt\ m(pds)$ { $mbody$ }
mds'
}
$cds,\ Main$
	=	**class** B **extends** A {
		ads
		$cnds$
		
		//@ **requires** ψ_1;
		//@ **assignable** ω;
		//@ **ensures** ψ_2;
		$rt\ m(pds)$ { $mbody$ }
		mds
		}
		class C **extends** B {
		ads'
		$cnds'$
		mds'
		}
		$cds',\ Main$

where

$cds' \mathrel{\widehat{=}} cds$[//@ **also** $fspec(m)/fspec(m)$], for every method m (with signature $rt\ m(pds)$ and that is not a redefinition) of any class E such that $E \leq B$ and $E \not\leq C$.

provided

JML:

(↔) (1) **super** does not appear in ψ_1 nor in ψ_2; (2) $\psi_1 \Rightarrow fpre(E[rt\ m(pds)])$ for every class E, such that $E \leq B$ but $E \not\leq C$, and E introduces a method $rt\ m(pds)$. (3) For any specification case for every method $rt\ m(pds)$, declared in any class E such that $E \leq B$ but $E \not\leq C$, with pre-condition PRE and postcondition $POST$, $\backslash\mathbf{old}(\psi_1) \Rightarrow ((\backslash\mathbf{old}(PRE) \Rightarrow POST) \Rightarrow (\backslash\mathbf{old}(\psi_1) \Rightarrow \psi_2)$.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes dos not appear in $mbody$; (2) $m(pds)$ is not declared in any superclass of B in cds.

(→) (1) $m(pds)$ is not declared in mds; (2) $mbody$ does not contain uncast occurrences of **this** nor expressions in the form $((C)\mathbf{this}).a$ and of the form $((C)\mathbf{this}).m(e)$ for any attribute a nor method m, in ads' and mds', respectively, with private visibility.

(←) (1) $m(pds)$ is not declared in mds'; (2) $D.m(e)$, for any $D \leq B$ and $D \not\leq C$, does not appear in cds, $Main$, mds or mds'.

□

By applying **Law** ⟨*move original method to superclass*⟩, from left to right, we move

the method *m* up only if the specification does not refer to elements of the class where it is declared through uncast occurrences **this**. Moreover, cast references to attributes or methods of class *C* cannot mention attributes whose specification visibility is protected.

By using **Law** ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩, we move a redefined method from a class to its superclass. The proviso concerning **super** is needed because its semantics may be affected when we move it from a subclass to a superclass, or vice-versa. We can only move the specification of a method if it does not refer to model fields of the class in which the method is originally declared. Furthermore, **this** expressions may occur in the target method specifications only if they are cast. In fact, as in the law the method has default visibility, only non-private elements can be referenced in its pre- and postconditions. This is similar to Java: the **this** expression may appear in *mbody'* if it has a cast and mention only non-private attributes or methods of class *C*. The right-side of **Law** ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩ introduces **instanceof** tests in each one of the specifications. In this way we assure that the original pre- and postconditions of the redefined method of *C* will only be applied to callers that are instances of *C* or instances of any of any subclass of *C*.

Law. ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩

```
class B extends A {
  ads
  cnds

  //@ requires ψ1;
  //@ ensures ψ2;
  rt m(pds) { mbody }
  mds
}
class C extends B {
  ads′
  cnds′

  //@ also
  //@ requires ψ′1;
  //@ ensures ψ′2;
  rt m(pds) { mbody′ }
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads
  cnds

  //@ requires (!(this
       instanceof C) && ψ1);
  //@ ensures (!(this
        instanceof C) && ψ2);
  //@ also
  //@ requires (this
       instanceof C && ψ′1);
  //@ ensures (this
        instanceof C && ψ′2);
  //@ also
  //@ requires (this
       instanceof C && ψ1);
  //@ ensures (this
        instanceof C && ψ2);
  rt m(pds) {
      if (!(this instanceof C))
      { mbody } else { mbody′ }
  }
  mds
}
class C extends B {
  ads′
  cnds′
  mds′
}
```

provided

JML:

(↔) **super** does not appear in ψ'_1 nor in ψ'_2.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes do not appear in *mbody′*; (2) **super**.*m* does not appear in *mds'*

(→) *mbody′* does not contain uncast occurrences of **this** nor expressions of the form ((*C*)**this**).*a* and of the form ((*C*)**this**).*m*(*e*) for any attribute *a* nor method *m*, in *ads′* and *mds′*, respectively, with private visibility.

(←) *m*(*pds*) is not declared in *mds′*.

□

The type of a method formal parameter may be modified to a superclass type, if every occurrence of the parameter inside specifications and in the program (only non-assignable ones) is cast with the current attribute type or subtype. These conditions have to hold also when one modifies the parameter type (T', for example) to any type corresponding to a subclass of it (for instance, T). However, in this case we also need to assure that every actual parameter corresponding to the formal parameter we are changing, is of type T or of a subtype of T, as well as the occurrences of the formal parameter itself. **Law** ⟨*change parameter type of pure method*⟩ allows us to change the type of a parameter; **Law** ⟨*change return type of pure method*⟩ changes the type of the value a method returns.

Law ⟨*change parameter type of pure method*⟩ and **Law** ⟨*change return type of pure method*⟩ deals with pure methods, there are similar laws that address non-pure methods. They can be found in Appendix A.4.

Law. ⟨*change parameter type of pure method*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt
  m(T x, pds) { mbody }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt
  m(T' x, pds) { mbody }
  mds
}
```

provided

JML:

(↔) every occurrence of x in expressions of $@spec\_cases$ are cast with T or with any subtype of T.

(←) every actual parameter associated with x found in specifications of C, cds e $Main$ is of type T or of any subtype of T.

Java:

(↔) $T \leq T'$ and every non-assignable occurrence of x in expressions of *mbody* are cast with T or any subtype of T.

(←) (1) every actual parameter associated with x in mds, cds and $Main$ is of type T or any subtype of T; (2) every expression assigned to x in *mbody* is of type T or any subtype of T; (3) every use of x as the method return in *mbody* is for a corresponding declared return of type T or any supertype of T.

□

Law. ⟨*change return type of pure method*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt' m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) (1) every call to $m(pds)$ that occurs in specifications in C, cds and $Main$ is cast with rt; (2) every occurrence of **result** in postconditions of $@spec\_cases$ are cast with rt or any subtype of rt.

Java:

(↔) $rt \leq rt'$.

(→) every call to $m(pds)$ used as a expression is cast to rt.

(←) every expression used in the **return** return clause in *mbody* is of type rt or of any subtype of rt.

□

To delete a method from a class is not a straightforward action. In the **Law** ⟨*method elimination: pure, redefined, non-default pre-existent specification*⟩ we deal with a specific situation in which the method has explicit specifications and it has some redefinition. To insert or remove a method in this situation, the precondition ψ_1 must be stronger than the precondition of any redefinition introduced in subclasses of C. On the other hand, the postcondition must be weaker than that of redefinitions declared in subclasses of C in cds. Consider now a situation of method elimination. Only calls to the method we want to eliminate, via objects of a subclass of C that does not have a redefinition of m, and that has direct superclass different of C, are allowed . To insert a method we need to choose a name that is not used in the target class nor in subclasses and superclasses.

Law. ⟨*method elimination: pure, redefined, non-default pre-existent specification*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  /*@ pure @*/ rt m(pds) {
    mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

JML:

(↔) (1) $\psi_1 \Rightarrow fpre(E[rt\ m(pds)])$ for every class E such that $E \leq B$, $E \not\leq C$ and E has a already defined method $rt\ m(pds)$. (2) For every class E such that $E \not\leq C$ and E has a already defined method $rt\ m(pds)$, there is a *specification case* for m with pre-condition PRE, post-condition $POST$, and frame W where $\backslash\mathbf{old}(\psi_1) \Rightarrow ((\backslash\mathbf{old}(PRE) \Rightarrow POST) \Rightarrow (\backslash\mathbf{old}(\psi_1) \Rightarrow \psi_2)$ and $\omega \subseteq W$.

(→) $B.m(e)$ does not occur inside specifications of C, cds and $Main$ for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

Java:

(↔) $rt\ m(pds)$ is already declared in any class E pertaining to cds such that $E \leq C$.

(→) $B.m(e)$ does not occur in cds, $Main$ nor in $cnds$, mds for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

(←) $rt\ m(pds)$ is not declared in mds nor in any superclass or subclass of C in cds.

□

Calls to methods via **super** can be eliminated using **Law** ⟨*eliminate calls to void methods via super*⟩. In fact, **Law** ⟨*eliminate calls to void methods via super*⟩ replace a method call using **super** by a copy of the body of the method declared in the superclass provided that **super** and **return** calls does not appear in the body as well as private attributes or methods. However, just copying the method body is not sufficient. We have to be aware about JML. When a method call via **super** is executed, invariants and history constraints (of the superclass) must be established as well as pre- and postconditions of the method. And also when the call is finished postconditions of the method must be satisfied and again the superclass' invariants. Notice that all these specifications also consider the inherited specifications from their superclasses.

The strategy we use here – and that is used in all of our laws that involving copy of method bodies – is based on creating JML-**assert** expressions to represent all JML speci-

fications that need to be satisfied at the beginning and at the end of the **super** method call execution. Notice that **assert** expressions can be created only if model fields representing private elements or private model fields do not appear in the predicates of the pre- and postconditions. Also, private attributes, private pure methods and private model fields can not appear in the invariants and history constraints of the superclass. This condition assures that all these elements will be visible after the **super** method call elimination. **Law** ⟨*eliminate calls to void methods via super*⟩ deals with *void* methods; another law deals with non-void methods.

The **assert** expressions can be seen at the right-side of the **Law** ⟨*eliminate calls to void methods via super*⟩. We chose JML-**assert** expressions instead of Java assert expressions because the JML ones are native and they were built to testing specifications.

Law. ⟨*eliminate calls to void methods via super*⟩
CDS is a set of two class declarations as follows.

class B **extends** A {
  ads
  $cnds$

  $@spec\_cases$
  $void\ m(pds)$ { $mbody$ }
  mds
}

class C **extends** B {
  ads'
  $cnds'$
  mds'
}

Thus, we have that:

$cds\ CDS,\ C \triangleright$ **super**.$m(e)$	=	$vardecs(pds,\ e);$ /*@ **assert** $fext\_inv(B)$ && $fext\_pre(B[m(pds)])$; @*/ $mbody$ /*@ **assert** $fext\_pos(B[m(pds)])$ && $fext\_inv(B)$ && $fext\_const(B[m(pds)])$ @*/

provided

JML:

(→) (1) **super** does not occur in $fext\_pre(B[m(pds)])$, $fext\_pos(B[m(pds)])$ nor in $fext\_inv(B)$ and $fext\_const(B[m(pds)])$; (2) Model fields that represent private attributes do not occur in $fext\_pre(B[m(pds)])$ or in $fext\_pos(B[m(pds)])$; (3) Private attributes, private pure methods, and model fields that represent private attributes or private model fields, declared in D, for any D such that $B \leq D$, do not occur in $finv(D)$ and $fconst(D[m(pds)])$.

Java:

(→) (1) **super**, private attributes and private methods declared in ads and mds, respectively, do not occur in $mbody$.

(2) $mbody$ does not contain **return** clauses.

□

Method calls can be eliminated by applying **Law** ⟨*void method call elimination*⟩ provided that the method is not redefined, the method body does not refer to **super**, all methods and attributes referred inside the method body are non-private and the body does not contain recursive calls. Notice that in the Java condition (4) we guarantee that the names of the real parameters are different from the formal ones, and in the Java condition (5) we force accesses to attributes and method calls to be made via the keyword **this** [5].

Complementing the previous considerations (reasoning now about JML) we have to ensure that private attributes, private pure methods, and model fields representing private elements or private model fields do not appear in specifications (i.e. invariants and history constraints) of the class that declares the method. We also force the use of **this** in the specifications as we did in the body of the method called. See that we use the same strategy (to deal with specifications) we use in **Law** ⟨*eliminate calls to void methods via super*⟩ previously explained. Besides the **Law** ⟨*void method call elimination*⟩, we have other two laws (**Law 84** and **Law 85**) to deal with the elimination of non-void methods calls. **Law 84** deals with a method call when it is used as an expression and **Law 85** deal with a method call then it is used as a statement.

[5] See the right-side of **Law** ⟨*void method call elimination*⟩ when we replace **this** by *le*. Not forcing the use of **this** we could not replace method calls and accesses to attributes. However this is not a problem, since we have a law (**Law 100**) to insert **this** in expressions and in specifications trivially.

Law. ⟨*void method call elimination*⟩
Consider that the following class declaration

class C **extends** D {
  ads
  $cnds$

  $@spec\_cases$
  $void\ m(pds)\ \{\ mbody\ \}$
  mds
}

is included in cds and that $cds,\ A \triangleright le : C$, meaning that le has static type C in the class A. Then

$$cds,\ A \triangleright le.m(e) \quad = \quad \begin{array}{l} \texttt{//@ } \textbf{assert}\ le\ \texttt{!=}\ \textbf{null}; \\ \texttt{/*@} \\ \ \textbf{assert}\ fext\_inv(C)[le/\textbf{this}] \\ \ \ \&\&\ fext\_pre(C[m(pds)])[le/\textbf{this}]; \\ \texttt{@*/} \\ vardecs(pds,\ e); \\ mbody[le/\textbf{this}] \\ \texttt{/*@} \\ \ \textbf{assert}\ fext\_pos(C[m(pds)])[le/\textbf{this}] \\ \ \ \&\&\ fext\_inv(C)[le/\textbf{this}] \\ \ \ \&\&\ fext\_const(C[m(pds)])[le/\textbf{this}]; \\ \texttt{@*/} \end{array}$$

provided

JML:

(→) (1) **super** does not occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$ nor in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$; (2) All attributes, pure methods and model fields that occur in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ are non-private. (3) All non-private model fields that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ represent only non-private attributes; (4) All accesses to non-private attributes and all calls to non-private pure methods that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$, are in the form **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

Java:

(→) (1) $m(pds)$ is not redefined in cds and $mbody$ does not cointain references to **super**; (2) all attributes and methods that occur in $mbody$ arenon-private. (3) $mbody$ does not contain recursive calls; (4) pds does not occur in e; (5) $mbody$ does not contain **return** clauses; (5) all accesses to non-private attributes and all calls to non-private methods that occur in $mbody$, are of type **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

□

We can make a method abstract since all subclasses (of the class that declares the method) provide a redefinition for the method, otherwise the program well-formedness will not be preserved.

Law. ⟨*make method abstract*⟩

abstract class *C*
  extends *D* {
  ads
  cnds
 	
  @*spec_cases*
  rt m(*pds*) { *mbody* }
  mds
}
	$=_{cds,Main}$	**abstract class** *C*
		  extends *D* {
		  ads
		  cnds
		
		  @*spec_cases*
		  abstract *rt m*(*pds*);
		  mds
		}

provided

Java:

(→) *rt m*(*pds*) is already declared in any class *E* pertaining to *cds* such that $E \leq C$.

□

In the next Section we will discuss about some constructors laws.

3.3.5 Constructors

This sections is dedicated to the explanation of some constructor laws. All the laws we defined to deal with constructors can be found in Appendix A.5. Constructors need to satisfy pre- and postconditions as well as in invariants and initially clauses. We do not provide laws to deal with initially clauses yet. However this kind of clause is taken into consideration in laws.

Custom **super**-constructors may be eliminated using **Law** ⟨*eliminate calls to* **super**(α(*pds*))⟩. The strategy used to deal with specifications is the same to that one we explained in Section 3.3.4, when we discussed the **Law** ⟨*void method call elimination*⟩. By applying **Law** ⟨*eliminate calls to* **super**(α(*pds*))⟩, from the left to right, the super constructor body is copied to the point of the call. Current pre- and postconditions, invariants and initially clauses that the super constructor need to satisfy are also copied (see in the right-side of the law). Private elements cannot appear in *cbody* as well as in the invariants and initially clauses that are copied. Model fields that represent private elements can not appear in the specification cases of the **super**-constructor. We also force that the superclass have an empty-body default constructor, with no explicit declared specifications, because when we call a super constructor it calls its default constructor.

Law. ⟨*eliminate calls to* **super**($\alpha(pds)$)⟩

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads'

  @spec_cases'
  C(pds) {
    super(α(pds));
    cbody'
  }

  cnds'
  mds'
}
```

$=_{cds,Main}$

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads'

  @spec_cases'
  C(pds) {
    /*@ assert
      fpre(@spec_cases);
    @*/
    cbody
    /*@ assert
      fpos(@spec_cases)
      && fext_inv(B)
      && fext_init(B);
    @*/
    cbody'
  }

  cnds'
  mds'
}
```

provided

JML:

(→) (1) Private attributes, private pure methods, and model fields that represent private attributes or private model fields declared in D, for any D such that $B \leq D$, do not occur in $finv(D)$ or in $finit(D)$; (2) Model fields that represent private attributes or private model fields, declared in B do not occur in $fpre$(@spec_cases) or in $fpos$(@spec_cases); (3) B's default constructor does not have explicit specification cases.

Java:

(↔) B's default constructor has a empty body.

(→) (1) *cbody* does not contain calls to **super**; (2) B has a default constructor; (3) private attributes and private methods declared in *ads* and *mds*, respectively, do not occur in *cbody*.

(←) B has a non-private constructor $B(pds)$, whose body is *cbody*.

□

Law ⟨*eliminate calls to* **this**(*e*)⟩ uses the same strategy to treat specifications as **Law** ⟨*void method call elimination*⟩. The Java conditions (2) and (3) are necessary to keep the well-formedness of the program.

Law. ⟨*eliminate calls to* **this**(*e*)⟩

```
class C extends D {
  ads′

  @spec_cases
  C(pds) { cbody }

  @spec_cases'
  C(pds′) {
    this(e);
    cbody′
  }

  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class C extends D {
  ads′

  @spec_cases
  C(pds) { cbody }

  @spec_cases'
  C(pds′) {
    vardecs(pds, e)
    /*@ assert
      fpre(@spec_cases);
    @*/
    cbody
    /*@ assert
      fpos(@spec_cases)
      && fext_inv(B)
      && fext_init(B);
    @*/
    cbody′
  }

  cnds′
  mds′
}
```

provided

Java:

(↔) *e* matches *pds*.

(→) (1) *cbody* does not contain calls to **super**; (2) *cbody′* does not contain calls to **super**.

□

Three laws are used to eliminate custom constructor calls: **Law** ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩ is used when the constructor's body have to call a superconstructor explicitly; **Law 91** is used when a call to a super constructor is not necessary; and **Law 92** is used when the target class has no superclass (except for `Object`). It is needed to have three distinct laws because Java obligates constructors to call another constructor explicitly or not.

For example, in **Law** ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩ we describe the situation where a superclass does not have an explicit default constructor and have only a custom constructor. In this situation when we apply **Law** ⟨*eliminate non-default constructors: when constructor's*

body have to call a superconstructor explicitly⟩, from the right to left, we need to guarantee that inside *cbody′* we have a call to the custom constructor, otherwise we could insert a Java compiler error. In **Law 91** we consider that the superclass does not have any declared constructor or has an explicit default constructor. In this way *cbody′* does not need to call a constructor explicitly, as Java inserts a default constructor call automatically.

Law. ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads′

  @spec_cases'
  C(pds′) { cbody′ }

  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads′
  cnds′
  mds′
}
```

provided

JML:

(→) **new** $C(\alpha(pds'))$ does not occur inside specifications of B, C, cds and $Main$.

Java:

(↔) (1) *cnds* does not have an explicit default constructor; (2) *cbody′* has a **super** call like **super**$(\alpha(pds_{\texttt{cnds}}))$ where $pds_{\texttt{cnds}}$ is the formal parameters list of some constructor that pertains to *cnds*.

(→) There are no calls to $C(pds')$ (including calls via **super** or **this**)

(←) $C(pds')$ is not declared in C

□

In order to apply **Law** ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩ from the left to right, we need to guarantee that the constructor is not called anywhere (including the program specifications) and that there is a custom constructor call inside its body. To apply **Law** ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩ in the opposite direction, we need to assure that the constructor that is being inserted is not

declared and that its body has a custom constructor call.

Law ⟨*eliminate calls to non-default constructors*⟩ eliminates a call to a non-default constructor. The strategy to deal with specifications is the same as the one we explained in Section 3.3.4, when we discussed the **Law** ⟨*void method call elimination*⟩. When applying **Law** ⟨*eliminate calls to non-default constructors*⟩, from the left to right, we need (among other things) to ensure that C has a default constructor (explicit or not) because even eliminating the constructor call we have to instantiate the class calling its default constructor. Another important safeguard is about the specifications of the default constructor. The default constructor of C cannot have specification cases, because as we continue to call a constructor (in this case the default constructor) we need to satisfy its specification cases.

Law. ⟨*eliminate calls to non-default constructors*⟩
Consider that the following class declaration

class *C* **extends** *D* {
  ads

  @spec_cases
  C(*pds*) { *cbody* }

  cnds
  mds
}

is included in *cds* and that *cds*, *A* ▷ *le* : *C*, meaning that *le* has static type *C* in the class *A*. Then

C le = **new** *C*(*e*); $=_{cds,Main}$

C le = **new** *C*();
vardecs(*pds*, *e*);
/*@ **assert**
  fpre(@spec_cases);
  [*le*/**this**]
@*/
cbody[*le*/**this**]
/*@ **assert**
 fpos(@spec_cases)
 [*le*/**this**]
 && *fext_inv*(*C*)
 [*le*/**this**]
 && *fext_init*(*C*);
 [*le*/**this**]
@*/

provided

JML:

(→) (1) **super** does not occur in *fpre*(@spec_cases), *fpos*(@spec_cases) nor in *fext_inv*(*C*) and *fext_init*(*C*); (2) Private attributes, private pure methods, and model fields that represent private attributes or private model fields declared in *B*, for every *B* such that *C* ≤ *B*, do not occur in *finv*(*B*) or *finit*(*B*); (3) All accesses to non-private attributes and all calls to non-private pure methods that occur in *fpre*(@spec_cases), *fpos*(@spec_cases), *fext_inv*(*C*) and *fext_init*(*C*), are in the **this**.*a* and **this**.*m*(*e*), respectively, where *a* is a non-private attribute and *m* is a non-private method; (4) *C*'s default constructor does not have explicit specification cases.

Java:

(→) (1) *C* has a default constructor; (2) there are no calls to **super** or **this**() in *cbody*; (3) all attributes and methods that occur in *cbody* are non-private. (4) *pds* does not occur in *e*;

□

3.3.6 Commands and Expressions

In this section we show two laws that apply to commands and expressions of Java, which we regard as small grain constructs. The whole set of laws of commands and expressions can be found in Appendix A.6.

Casts can be eliminated in expressions since the type of the expression is of the type of the cast. Notice that inserting a JML-**assert** expression we guarantee that e is really of type C.

Law. ⟨*eliminate cast of expressions*⟩
If $cds, A \rhd le : B$ and $cds, A \rhd le : B'$, with

$$cds,\ A\ \rhd\ le\ :=\ (C)\ e\ =\ \texttt{/*@}\ \textbf{assert}\ (e\ \textbf{instanceof}\ C);\ \texttt{@*/}\ le\ :=\ e$$ □

Variables can have their type changed in a similar way attributes can do as it is stated in **Law 29**.

Law. ⟨*change variable type*⟩

$$cds,\ A\ \rhd\ T\ x;\ c\ =\ T'\ x;\ c$$

provided

JML:

(↔) Every occurrence of x inside specifications of c, is cast with T or any subtype of T.

Java:

(↔) $T \leq T'$.

(←) (1) Every expression assigned to x in c is of type T or any subtype of T; (2) every use of x as the return expression in c is for a corresponding declared return of type T or any subtype of T.

□

3.3.7 Predicates

As well as we did in the previous Section we show here another category laws to deal with small grained constructs: predicates laws. All laws that deal with predicates are found in Appendix A.7.

If we have a predicate where we have an implication stating that an expression is of a certain type and this expression is cast to this type, we delete the cast and maintain only the expression. **Law** ⟨*delete trivial cast in* **instanceof** *implications inside predicates*⟩ and **Law** ⟨*eliminate cast of pure method call in predicates*⟩ are used to eliminate cast of expressions in the presence of a type test using an implication.

Law. ⟨*delete trivial cast in* **instanceof** *implications inside predicates*⟩
If $cds, A \rhd e : C$, then

$$e \ instanceof \ C \ ==> \ (C) \ e \quad = \quad e \ instanceof \ C \ ==> \ e \qquad \square$$

Law. ⟨*eliminate cast of pure method call in predicates*⟩
If $cds, A \rhd e : B$, $C \leq B$, m is pure and is declared in B or in any of its superclasses in cds and $((C)e).m(e')$ is written in a valid JML predicate, then

$$cds, \ A \ \rhd \ ((C)e).m(e') \quad = \quad e \ \textbf{instanceof} \ C \ ==> e.m(e') \qquad \square$$

3.4 Summary of Laws

In this section we present a summary of all laws discussed in this chapter. We categorized the laws with respect to the type of provisos they need to satisfy (JML and Java provisos) and if they affect Java code or JML specifications:

Need to satisfy JML provisos ((JML)) – laws which require to satisfy JML provisos.

Need to satisfy Java provisos ((J)) – laws which require to satisfy Java provisos.

Affects JML specifications ($[JML]$) – laws that insert, delete or modify JML specifications.

Affects Java code ($[J]$) – laws that insert, delete or change Java code.

Table 3.2 and Table 3.3 depict the summary of the laws presented in this chapter. The complete set of our laws can be found in Appendix A.

attributes	change attribute visibility: from private to public	$[J](JML)(J)$
	change spec public attribute visibility: from private to public	$[JML][J](J)$
	change specification visibility of private attribute: from private to public	$[JML](JML)$
	move reference type attribute to superclass	$[J](JML)(J)$
	change attribute type	$[J](JML)(J)$
methods	weaken pre-condition	$[JML](JML)$
	strengthen post-condition	$[JML](JML)$
	insert **same** specification case	$[JML](JML)$
	change assignable from **not_specified** to **everything**	$[JML](J)$
	make method **pure**	$[JML](JML)$
	collapse pre-conditions	$[JML]$
	collapse post-conditions	$[JML]$
	collapse also combinations	$[JML]$
	change pure method visibility from: private to public	$[J](JML)(J)$
	change specification visibility of pure private method: from private to public	$[JML](JML)$
	introduce void method redefinition	$[J](JML)(J)$
	move original method to superclass	$[JML][J](JML)(J)$
	move redefined method to superclass: overriden method with non-default specification case	$[JML][J](JML)(J)$
	change parameter type of pure method	$[J](JML)(J)$
	change return type of pure method	$[J](JML)(J)$
	method elimination: pure, redefined, non-default pre-existent specification	$[J](JML)(J)$
	eliminate calls to void methods via super	$[JML][J](JML)(J)$
	void method call elimination	$[JML][J](JML)(J)$
	make method abstract	$[J](J)$

Table 3.2: Summary of the laws about attributes and methods described in Chapter 3.

classes	class elimination/introducion	$[J](JML)(J)$
	make class abstract	$[J](JML)(J)$
	change superclass: from **Object** to another class	$[JML][J](JML)(J)$
invariants	move invariant to superclass	$[JML](JML)$
	change invariant visibility: from default to private	$[JML](JML)$
	change invariant visibility: from public to private	$[JML](JML)$
	change invariant visibility: from protected to private	$[JML](JML)$
	collapse invariants	$[JML]$
constructors	eliminate calls to **super**($\alpha(pds)$)	$[JML][J](JML)(J)$
	eliminate calls to **this**(e)	$[JML][J](J)$
	eliminate non-default constructors: when constructor's body have to call a supercon-structor explicitly	$[J](JML)(J)$
	eliminate calls to non-default constructors	$[JML][J](JML)(J)$
commands	eliminate cast of expressions	$[JML][J]$
	change variable type	$[J](JML)(J)$
predicates	delete trivial cast in **instanceof** implications inside predicates	$[JML]$
	eliminate cast of pure method call in predi-cates	$[JML]$

Table 3.3: Summary of the laws about classes, invariants, constructors, commands and predicates described in Chapter 3.

Chapter 4

A Specification-Aware Normal Form

4.1 Introduction

Borba [8] uses a normal form to show that a set of laws for the language ROOL is compreheensive. Duarte [26] follows the aproach proposed for ROOL with adaptations for Java because ROOL is limited to a sequential subset of Java. As a subset of our set of laws adapts laws originally proposed for object-oriented programming with no specifications (in the sense of Design by Contract), we follow the same strategy for reducing a program to a normal form as proposed by Duarte. However, the existence of specifications written in JML impose restrictions to the application of programming laws. For instance, when moving an attribute from a class to its superclass, we have to notice how JML deals with nullity. We could introduce a null reference, but null is not the default in JML [28].

4.2 Normal Form

The normal form that we have as target has the following characteristics:

- There is a `Main` class with a `main` method that is supposed to be the program start point;
- Classes (of our unique package) other than `_Object`[1] contains no attributes and methods;
- Methods can appear only in the class `_Object`[2];
- All local declarations in the `main` method are declared with a primitive type, or `_Object`;
- No type cast is allowed in the main method;
- Custom constructors are not allowed anywhere;
- Invariants and history constraints can occur only in `_Object`.

[1] Since every class in a Java program extends `Object`, changing this class affects all classes hierarchies and, in fact, it is part of the Java library. We use the class `_Object`, which extends `Object`, as the topmost class in the class hierarchy we reduce to the normal form.

[2] Only methods that can not be eliminated by our laws, i.e. recursive methods, and methods with no mutually exclusive return points.

This normal form preserves more constructs of object-oriented programming that those described for Java [26]. In particular, we cannot obtain a static method in the class `Main`, because turning an instance method into a static one requires changing invariants, referred attributes and pure methods in the method specification in a similar way, which may be not possible. As we said in Section 2.5.1, static invariants may refer only static attributes and methods.

Unlike, instance invariants can refer to both static and instance methods and attributes. Only instance invariants are inherited by subtype as we showed in the section about specification inheritance (Section 2.7).

Recall that we consider that we are dealing with only one package and working in a limited open system, in which classes of our system can depend on external libraries. We also assume that the identifiers of our classes are distinct from those of external libraries. Remember that in our approach a program has the format *cds Main*, where *cds* is the set of all classes of the program and *Main* corresponds to the unique class in the program that has a *main* method. Also, we make some other assumptions:

- Distinct classes in *cds* are not allowed to declare attributes with the same name;
- Invariants and history constraints of subclasses can not restrict attributes and model fields of their superclasses;
- All classes in *cds* must declare a default constructor;
- Pure methods must be accessor methods[3];
- All original methods and constructors obey their respective specifications (including invariants, **initially** clauses and constraints).

The first condition avoids name clashes when moving attributes up in the hierarchy. The second allows us to insert trivial methods redefinitions, which we will discuss in next sections. The third avoids breaks of invariant and iniatially specifications, as will be explained in the sequel. Finally, the fourth assumption is needed because it is not possible to inline a pure method inside a specification if this method contains any command that is not a getter-like expression.

4.3 Reduction Strategy

We follow the normal form reduction strategy proposed by Duarte [26] as a guideline. However, the presence of JML specifications impedes us from obtaining the same normal form as Duarte. As an example of such differences we point out:

- We end with the class `_Object` with attributes and methods that are not made static, whereas in Duarte's normal form the class `_Object` only contains attributes;
- In the normal form of Duarte, recursive methods and other non-inlined methods (i.e. methods with not mutually exclusive return points) are translated into new behavior-equivalent static methods in the class `Main` whereas in our normal form these kind of methods are placed in the `_Object` class.

[3] Accessor methods are methods that are usually small, simple and provides the means for the state of an object to be accessed from other parts of a program.

Another important difference between the normal form we present here and the one presented in [26] is the maintenance of explicit default constructors as it is not possible to eliminate them in the presence of invariants and initially clauses. Recall that Java creates a default constructor in any class that does not declare any constructor. Thus, since inlining and eliminating all custom constructors of all classes is a step of our reduction strategy, if we also eliminate explicit default constructors, Java will create a default constructor that will possibly not meet the invariant and initially specifications provoking contracts break.

In order to give some extra guarantee to our strategy and provide soundness to our approach, we applied state of the art JML tools in the program source code after each law application. In addition, we also ran the `Main` class to ratify that the program output was equals to the original one. More specifically, we used two classes of JML tools (see Section 2.9): the static program checker ESC/Java2 [21] and the test-based run time assertion checker tool, JET [15]. Moreover, after each step we compiled the source-code with the official JML compiler, *jmlc* (see Section 2.9.1), and ran the program with the official JML-RAC tool [10] to confer the output.

Here we present the steps for reducing a program written in Java and specified with JML to the normal form we introduced previously. The reduction strategy includes the five following major steps.

- Create a new root class (`_Object`), make all classes inherit it and move all the attribute declarations to it;
- Eliminate custom constructors;
- Move methods up and change types to `_Object`;
- Eliminate casts;
- Eliminate methods calls and the corresponding declarations.

In the next section we detail each one of these steps to give a more comprehensive explanation. We describe the process as a sequence of simple and incremental steps.

4.4 Reduction Strategy in Action

To demonstrate our strategy and show the exact differences between the non-specification-aware normal form by Duarte [26] and our specification-aware normal form we use the same example of his work. In this way, we can present and exploit each restriction imposed by JML-specifications in the pure Java normal form.

In Figure 4.1 we present the starting class diagram of our example extended with JML-specifications. This class diagram models a simple interpreter for a little expression language named *Exp1* that accepts Integer (`Integer` class) values that can be just summed (`Sum` class). The class `Expression` is the topmost expression class. Every expression of the language inherit directly or indirectly `Expression`. Values are subclasses of the `Value` class. In our example we have only Integers as values. Binary expressions are subclasses of `BinaryExpression`, in particular `Sum`. The *Exp1* interpreter is implemented in the `Interpreter` class that basically stores an expression and evaluates it via the method `run`. Finally, `Main` represents (via the method `main`) the program starting point.

The complete source code of the initial program (before the application of our normal form reduction) can be found in Appendix B.1. The method `main` in the class `Main` has

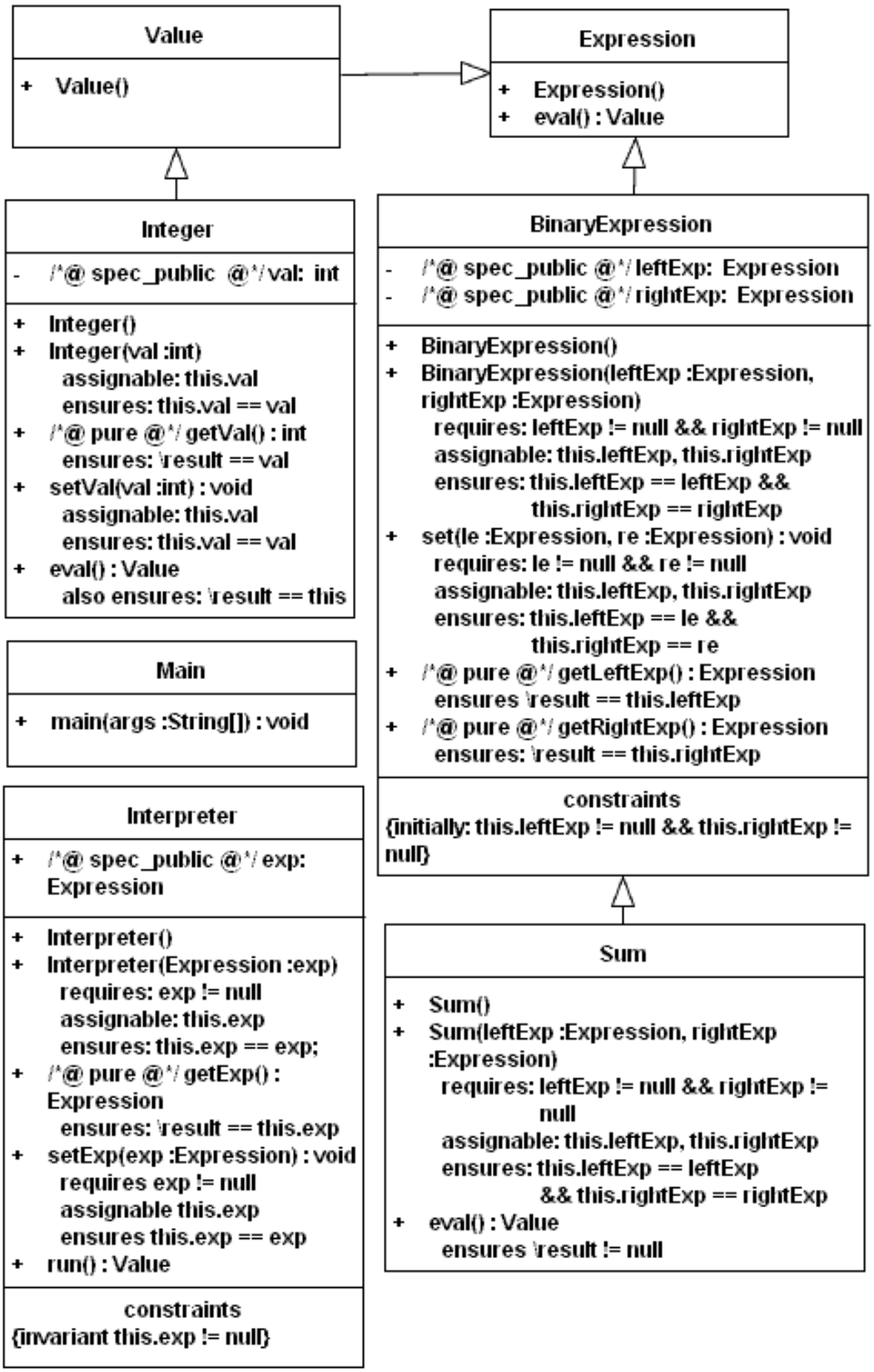


Figure 4.1: Extended class diagram of our JML-specified example program

two integers, 5 and 8. These values are set in a `Sum` object that is passed as argument to an instance of `Interpreter`. Then the `Sum` object is evaluated and the corresponding values are assigned to a `Value` object. After it the program execution ends. The reduction strategy is detailed in the sequel.

4.4.1 Create a new root class and make all classes inherit it

The first step is to introduce the new root superclass called `_Object` applying **Law 1** from the right to left. Creating a new empty class is straightforward since we need only to check if there is no class with the same name and if its superclass is a valid class.

After that, we have to make the classes (`Expression` and `Interpreter`) inherit `_Object`. To achieve it, we apply **Law 3**, from the left to right.

4.4.2 Make attributes public

In this step of the reduction strategy, we make all attributes public. Attributes with this visibility are inherited and considered valid by subclasses. Recall that we follow a strategy similar to those used by Borba [7] and Duarte [26].

To make an attribute public, provided it is currently private is straightforward, even it has public specification visibility. We have eight laws (**Law 19**, **Law 20**, **Law 18**, **Law 21**, **Law 22**, **Law 23**, **Law 24**, **Law 25**) to cover all possible visibility changes situations. By the successive application of these laws we can make all attributes of all classes public. Particularly, in our example we apply **Law 24** in the attributes `val` (`Integer`), `leftExp` and `rightExp` (`BinaryExpression`) and `exp` (`Interpreter`).

4.4.3 Move Attributes Upwards Towards `_Object`

This step consists in moving all attributes from subclasses up to superclasses until they reach `_Object`. We can move a public attribute to a superclass by applying **Law 27** from the left to right, if it is not already declared in the superclass and if it is nullable. Before move them, we have to apply **Law 26** from the left to right, in those in order to avoid non null exception checks (see Section 3.3.3 for more details about this question). Also, we need to apply **Law 28** to primitive attributes. At this point we exhaustively apply **Law 27** from the left to right to the reference-typed attributes of `Interpreter` and `BinaryExpression` to move them up to `_Object`. Then, we apply **Law 28** from the left to right to the attribute `val` of `Integer` to move it until it reaches `_Object`.

The result until here is sketched in Figure 4.2. As can be seen, all attributes are now nullable, public and are placed only in the class `_Object`.

4.4.4 Eliminate Custom Constructors Calls

Before eliminating custom constructors, we need to prepare the program – in order to satisfy the pre-conditions needed to execute the eliminations – executing the following steps: we apply Laws 101 and 102 in the custom constructors bodies and Laws 101 and 103 in constructors pre- and postconditions, all of them from the left to right, in order to facilitate inlining. Then, we eliminate calls to **super**() , to **super**($\alpha(e)$) and calls to **this**(e) inside the custom constructors bodies applying **Law 89**, **Law 88** and **Law 90**, respectively.

```
1  public class  _Object {
2    public int val;
3    public /*@ nullable @*/ Expression exp;
4    public /*@ nullable @*/ Expression rightExp;
5    public /*@ nullable @*/ Expression leftExp;
6  }
7
8  public class Expression extends _Object {
9    public Expression () {}
10   public Value eval() { return null; }
11 }
12
13 public class Value extends Expression {
14   public Value() {}
15 }
16
17 public class BinaryExpression extends Expression {
18   //@ initially this.leftExp != null && this.rightExp != null;
19
20   public BinaryExpression() {
21     this.leftExp = new Integer();
22     this.rightExp = new Integer();
23   }
24   /* ... */
25 }
26
27 public class Sum extends BinaryExpression {
28   public Sum() {
29     super();
30   }
31   /* ... */
32 }
33 public class Integer extends Value {
34   public Integer() {
35     super();
36     this.val = 0;
37   }
38   /* ... */
39 }
40
41 public class Interpreter extends _Object {
42   //@ public invariant this.exp != null;
43
44   public Interpreter() {
45     super();
46     this.exp = new Integer();
47   }
48   /* ... */
49 }
```

Figure 4.2: Example program source-code - attributes up

In our example we do not need to insert **this** in methods calls or attributes access because all calls and access already have the **this** keyword. Hence, we apply **Law 89**to the two `Integer` constructors, to the default constructor of `Sum`, to the default custom constructor of `BinaryExpression` and to both constructors of `Interpreter`. In our example, there are no calls to constructors via **this** . After eliminating every call to **super**, we eliminate the custom constructor call that occurs inside the custom constructor of `Sum` class. By applying **Law 88** eliminating the **super**(`leftExp`, `rightExp`) call in `Sum`.

With no calls to **super** and **this** inside constructors we can finally eliminate calls to custom constructors. We use **Law 94** to do this task. In the situations in which a new object instance is not assigned to a expression, we apply **Law 107** to create assignments. So every custom constructor call fits exactly the law template. Applying **Law 107** and **Law 94** exhaustively, we eliminate all custom constructor calls in the method `eval` of the class `Sum` and in the method `main` of `Main`.

4.4.5 Eliminate Custom Constructors

Now, with no calls to custom constructors anywhere in the program, we eliminate all custom constructor declarations. We have three laws: **Law 92**, **Law 91** and **Law 93**. Each law deals with a specific situation as explained in Section 3.3.5. As all provisos are equals in these laws and all provisos are satisfied at this point because of the execution of previous steps, we eliminate all custom constructors declarations of `Interpreter`, `BinaryExpression` , `Sum` and `Integer`.

The reason we do not delete default constructors is not so obvious. Actually, it is not possible to inline default constructors, because we have to maintain their calls. We can not create an object without calling at least the class default constructor. Duarte [26] created a law called *inline default constructor calls and eliminate its body* in order to inline default constructors and eliminate their bodies. But, if one deletes a default constructor body, this constructor will potentially no longer meet class invariants and `initally` clauses. Hence, we decided to keep the default constructors in classes.

At this point, we complete the second major step of our reduction strategy. A snapshot of the source code we have at this point is presented in Figure 4.3.

4.4.6 (Trivial) Cast Introduction

In order to facilitate the next steps, we introduce a trivial cast in every expression that access attributes or that is a method call target. By introducing casts we can move methods to its superclasses. Without the introduction of casts, we cannot move the method `eval` of `Integer` to the class `Expression` because in `Expression`, the type of **this** is `Expression`, not `Integer`, and, hence, the return point **return this** is ill-typed. The same situation may occur when we are treating specifications. The postcondition of `eval` says that the result of the method has to be equals to the object **this** itself. If, for example, we first move `eval` to `Value`, **this** will refer to instances of `Value` in opposite of `Integer`, resulting a different behavior.

Laws 100, 105, and 109 are applied to all attributes accesses and methods calls. If an attribute access or a method call is not referred using **this** either in methods and constructors bodies or in specifications, we need to apply one of these laws **Law 101**, **Law 102** and **Law 103** to introduce the trival casts afterwords.

```
1  /* ... */
2  public class BinaryExpression extends Expression {
3    //@ initially this.leftExp != null && this.rightExp != null;
4    public BinaryExpression() {
5      this.leftExp = new Integer();
6      this.rightExp = new Integer();
7    }
8    /* ... */
9  }
10 public class Sum extends BinaryExpression {
11   public Sum() {  }
12
13   /*@ also
14     @ ensures \result != null;
15     @*/
16   public Value eval() {
17     Expression le = this.getLeftExp();
18     Expression re = this.getRightExp();
19     Integer lint = new Integer();
20     Integer rint = new Integer();
21     lint.setVal(((Integer)le.eval()).getVal());
22     rint.setVal(((Integer)re.eval()).getVal());
23     Integer tmp = new Integer();
24     int val = lint.getVal() + rint.getVal();
25     tmp.val = val;
26     /*@ assert tmp.val == val; @*/
27     return tmp;
28   }
29 }
30 public class Integer extends Value {
31   public Integer() {
32     this.val = 0;
33   }
34   /* ... */
35 }
36  /* ... */
37 public class Main {
38   public static void main(String[] args) {
39     Interpreter in; Integer n1,n2; Sum s; Value v;
40     n1 = new Integer();
41     int val = 5;
42     n1.val = val;
43     /*@ assert n1.val == val; @*/
44     n2 = new Integer();
45     val = 3;
46     n2.val = val;
47     /*@ assert n2.val == val; @*/
48     /* ... */
49   }
50 }
```

Figure 4.3: Example program source-code - reduced constructors

4.4.7 Introduce (Trivial) Method Redefinitions

Following the strategy described in 4.4.2, we make all methods public. We exhaustively apply laws to modify the visibility of methods to public: **Law 57**, **Law 58**, **Law 59**, **Law 60**, **Law 61**, **Law 62**, **Law 63** and **Law 64** for pure methods; **Law 54**, **Law 55** , and **Law 56** for non-pure methods.

From now, all the methods are public. Our laws that deal with methods consider that the methods are *default*. Thus, from now on we consider that before applying the laws of methods listed throughout this section (except the laws that deal with visibility changes) we apply a law (for instance, **Law 54**) to change the visibility of the method to default. We also consider that at the end of the application of the laws of methods we change the visibility of the method back to public. This simplification is possible because the methods were originally public.

Introducing trivial method redefinitions are needed when we move methods up. Classes have its own methods and also have the methods they inherit. So, before moving up methods it is necessary to explicit the inherited methods by introducing trivial methods redefinitions via **super**. In this way we make the program text uniform and simplify methods movements. We apply **Law 67** to the classes `BinaryExpression` and `Value` creating redefinitions of the method `eval` of `Expression`. We also apply **Law 67** to the class `Sum` creating redefinitions of the methods `getLeftExp` and `getRightExp` of `BinaryExpression`. Additionally we apply **Law 66** to the class `Sum` creating the redefinition of the method **set** of `BinaryExpression`.

4.4.8 Eliminate Methods Calls via super

The provisos of Laws 68, 69, and 70 (laws we use in the next step to move up the methods) requires that there are no calls to **super** in method bodies or specifications, so that we can move a method (redefined or not) to the superclass of the class that introduces the method to be moved. The problem that can arise without those provisos are easy to understand. If you move a method m of a class C to a superclass B and there is a call to $m1$ of B like **super**.$m1()$ in m's body or in a specification case of m, the call **super**.$m1()$ will no more refer to $m1$ of B, leading to the execution of other method our causing a compiler error. Hence we eliminate method calls via **super**.

Two laws are used to eliminate methods calls that have **super** as target, **Law 81** and **Law 82**, that fit void and non-void methods, respectively. In our example, we apply **Law 81** from the left to right to the method **set** of `Sum` and **Law 82** (preceded of **Law 107**) (also from left to right) to the methods: `eval` of `BinaryExpression`; `getLeftExp` and `getRightExp` of `Sum`; and `eval` of `Value`. We start applying the laws to the immediate subclasses of `_Object`. Since all attributes of `_Object` are public and `_Object` is the topmost class in our example hierarchy, all the provisos of **Law 81** and **Law 82** are satisfied.

4.4.9 Move Methods Towards *_Object*

We safely can move all methods up to `_Object`. We have three laws with this purpose: **Law 68** allows us to move up a method to a superclass when the target method does not exists in its superclass; **Law 69** let us to move up a redefined method to a superclass since the super method has explicit specification cases; and **Law 70** allows us to move up a redefined method when the super method does not have explicit specification cases. The side conditions of these laws were previously satisfied by the application of laws in the

previous steps. All attributes are public, every call to **super** has already been eliminated, and every occurrence of **this** is casted.

We begin by applying **Law 68** and **Law 69** from the bottommost classes moving their methods until they reach `_Object`. Before applying one of these two laws it is necessary to collapse method's specification cases in an unique specification case. This is achieved applying firstly Laws 34 and 35 in methods with incomplete lightweight specification cases declarations. Then we apply Laws 41, 42 and 43, in this order.

At this point, we can move all methods of `Interpreter` directly to `_Object` applying **Law 68** from the left to right as many times as needed. We also move the methods `setVal` and `getVal` of `Integer` to `_Object`.

The methods **set**, `getLeftExp` and `getRightExp` originally declared only in the class `BinaryExpression` are combined with their respective trivially redefined methods of `Sum` via application of **Law 69** and after that moved upwards towards `_Object`.

The method `eval` of `Integer` is combined with `eval` (recently created by trivial redefinition) of `Value` and subsequent combined with `eval` of `Expression`. We do the same with `eval` of `Sum` that is combined with `eval` of `BinaryExpression`. Now, we combine it with the "mixed" `eval` of `Expression`. After that, we finally move it to `_Object`. The result after all methods movement, is a method that tests for all the possible dynamic types of `Expression`.

Method bodies that results from the previous steps can be simplified using laws like **Law 107**, **Law 96**, **Law 97** and **Law 98** combined with others like **Law 99**. In the case of specifications, predicates could be reduced using propositional calculus [54] and some laws, like for instance **Law 41**, **Law 42**, and **Law 43**. For instance, the method `getRightExp` presented in the Figure 4.4 can be simplified using the cited laws (the result is presented in Figure 4.5). The previous simplification-tasks are not part of our reduction strategy and their execution is not mandatory, however, our intention is only to show that these kind of simplifications may be executed.

4.4.10 Change Type to *_Object*

At this point, all methods and attributes are in `_Object`. We can change all attributes, method parameters and return, and local variable to `_Object`. Then, after that, eliminate trivial casts introduced before. To apply the laws 29, 72, 71, 74, 73, and 106 in the current scenario is semantic preserving because of the introduced casts and because the application of previous steps. The exhaustive application of these laws, allows the replacement of the types of all identifiers by `_Object`. Obviously, attributes, parameters, returns and variables of primitive types are not affected.

4.4.11 Cast elimination

Laws to deal with casts elimination are detailed in [9]. We eliminate all casts by applying laws 100, 105, and 109. The side conditions of these laws are satisfied by the fact that now, all attributes and methods of the program are in `_Object`. Notice that eliminate all casts in not only in the code but also in specifications. Figure 4.4 shows an excerpt of `_Object` class until here. Figure 4.6 shows the program.

```
1  public class  _Object {
2    /* ... */
3
4    //@ ensures \result == this.exp;
5    public /*@ pure @*/ _Object getExp() {
6      return this.exp;
7    }
8
9    /*@ requires exp != null;
10     @ assignable this.exp;
11     @ ensures this.exp == exp;
12     @*/
13   public void setExp(_Object exp) {
14     this.exp = exp;
15   }
16
17   public _Object run() {
18     return this.exp.eval();
19   }
20
21   //@ requires (!(this instanceof Sum) && true);
22   //@ assignable \not_specified;
23   //@ ensures (!(this instanceof Sum) && (\result == this.rightExp));
24   //@ also
25   //@ requires ((this instanceof Sum) && true);
26   //@ assignable \not_specified;
27   //@ ensures ((this instanceof Sum) && (\result == this.rightExp));
28   //@ also
29   //@ requires ((this instanceof Sum) && true);
30   //@ assignable \not_specified;
31   //@ ensures ((this instanceof Sum) && (\result == this.rightExp));
32   public /*@ pure @*/ _Object getRightExp() {
33
34     if (!(this instanceof Sum)) {
35       _Object tmp;
36       tmp = this.rightExp;
37       /*@ assert
38           tmp == this.rightExp;
39         @*/
40
41       return tmp;
42     } else {
43       return this.rightExp;
44     }
45   }
46     /* ... */
47 }
```

Figure 4.4: Example program source-code - Excerpt of _Object class with all methods declarations

```
1 //@ requires true;
2 //@ assignable \not_specified;
3 //@ ensures \result == this.rightExp;
4 public /*@ pure @*/ _Object getRightExp() {
5     return this.rightExp;
6 }
```

Figure 4.5: Example program source-code - proposed reduced getRightExp method.

4.4.12 Move Invariants Upwards Towards *_Object*

Before eliminating methods, we need to inline methods calls by using our laws (**Law 83**, **Law 84** and **Law 85**). However, as these laws use invariants and all methods are now in `_Object`, we need to move invariants up too using **Law 5**. By applying this law to the invariant of `Interpreter`, we move it to `_Object`.

```
1  public class Expression extends _Object {
2    public Expression () {}
3  }
4
5  public class Value extends Expression {
6    public Value() {}
7  }
8
9  public class Integer extends Value {
10   public Integer() {
11     this.val = 0;
12   }
13 }
14
15 public class BinaryExpression extends Expression {
16   public BinaryExpression() {
17     this.leftExp = new Integer();
18         this.rightExp = new Integer();
19   }
20 }
21
22 public class Sum extends BinaryExpression {
23   public Sum() {}
24 }
25
26 public class Interpreter extends _Object {
27   public Interpreter() {
28     this.exp = new Integer();
29   }
30 }
```

Figure 4.6: Example program source-code - classes (excepts for _Object) without methods

4.4.13 Methods Elimination

As we explained before (refer to Section 4.2), only non-recursive and methods that have no mutually exclusive return points can be eliminated. Unlike Duarte's approach, we cannot make these methods static because instance invariants are not applied to static methods. A law to deal with this situation would be too restrictive and its use would be quite reduced. This is why in our strategy, we impose that these methods remain in `_Object`. All other instance methods that can be inlined are eliminated.

We apply laws 83, 84 and 85 to remove all methods calls. After all methods calls are replaced with the bodies of the corresponding methods, the methods definitions can be eliminated using laws 75, 76, 77, for non-pure methods and laws 78, 79 and 80 for pure methods.

With this step we finish the reduction process. An excerpt of `_Object` is showed in Figure 4.7. An excerpt of the resultant `Main` class can be seen in Figure 4.8. In the Appendix B.2 we present the final source-code of the program.

```
1  public class  _Object {
2
3    //@ invariant this instanceof Interpreter ==> this.exp != null;
4
5    public int val;
6    public /*@ nullable @*/ _Object exp;
7    public /*@ nullable @*/ _Object rightExp;
8    public /*@ nullable @*/ _Object leftExp;
9
10   //@ requires ((!(this instanceof BinaryExpression)) &&  ((!(this
       instanceof Value)) || ((this instanceof Value) && ((!(this
       instanceof Integer)) || (this instanceof Integer && true)))));
11   //@ assignable \not_specified;
12   //@ ensures ((!(this instanceof BinaryExpression)) && ((\old((!(
       this instanceof Value))) ==> (!(this instanceof Value)))
13   //@ && (\old(((this instanceof Value) && ((!(this instanceof
       Integer)) || (this instanceof Integer && true)))) ==> ((this
       instanceof Value) && ((\old(!(this instanceof Integer)) ==> (!(
       this instanceof Integer))) && (\old((this instanceof Integer &&
        true)) ==> (this instanceof Integer && \result == this)))))));
14   /*@ also
15     @ requires ((this instanceof BinaryExpression) && ((!(this
         instanceof Sum)) || ((this instanceof Sum) && true)));
16     @ assignable \not_specified;
17     @ ensures ((this instanceof BinaryExpression) && ((\old((!(this
         instanceof Sum))) ==> (!(this instanceof Sum))) && (\old(((
         this instanceof Sum) && true)) ==> (this instanceof Sum && \
         result != null))));
18     @*/
19   public _Object eval() {
20   /* ... */
21   }
22 }
```

Figure 4.7: Example program source-code - Excerpt of _Object at the end.

```
1  public class Main {
2
3    public static void main(String[] args) {
4      _Object in;
5      _Object n1,n2;
6      _Object s;
7      _Object v;
8
9      n1 = new Integer();
10     int val = 5;
11     n1.val = val;
12     /*@ assert n1.val == val @*/
13
14     n2 = new Integer();
15     val = 3;
16
17     n2.val = val;
18     /*@ assert n2.val == val @*/
19
20     s = new Sum();
21     _Object leftExp = n1;
22     _Object rightExp = n2;
23     /*@ assert
24        leftExp != null && rightExp != null;
25       @*/
26     /*@ assert
27       leftExp != null && rightExp != null;
28       @*/
29     s.leftExp = leftExp;
30     s.rightExp = rightExp;
31     /*@ assert
32        s.leftExp == leftExp && s.rightExp == rightExp
33       &&  s.leftExp != null && s.rightExp != null;
34       @*/
35
36     /*@ assert
37       s.leftExp == leftExp && s.rightExp == rightExp
38        &&  s.leftExp != null && s.rightExp != null;
39       @*/
40     in = new Interpreter();
41     _Object exp = s;
42     /*@ assert
43       exp != null;
44       @*/
45     in.exp = exp;
46     /* ... */
47 }
```

Figure 4.8: Example program source-code - Excerpt of Main class at the end.

4.5 Reduction Strategy Considerations

The normal form reduction strategy we proposed here, is a result of the review and adaptation of the strategies used for Java and ROOL [9] programs. We can enhance our reduction strategy elaborating new laws to eliminate invariants, history constraints and initially clauses, for example. We plan to create laws to distribute invariants and history constraints in the methods of the classes they are declared. In this direction it is possible to create laws to eliminate initially clauses by copying their predicates in the constructors they affect. As a result we can eliminate methods and transform their specifications in JML assertions.

Another important point, additional to the one we discussed in the paragraph above, is the possibility to transform all JML specifications of a program in Java code representing run time assertion checks (RAC) code. We could do it in the way Krakatoa [11] and other JML tools like, JAJML [36] and *jmlc* [1] do. We can transform (after applying our strategy) all remaining specifications in Java code representing the behavior of the specifications. Thus, we can obtain a reduction strategy that transforms a JML-specified Java program into a normal form containing only Java code.

Chapter 5

Application: Code and Specification Refactoring

Software changes constantly due to maintenance that leads to correction of fails or evolution. However, some changes can take place due to other quality related factors such as code reuse or legibility. In this case, the changes may not alter the software behavior but only its internal structure, thus, making it better. This kind of change is an activity known as refactoring [31]. To avoid errors due to modifications, every change has to be done following a discipline which can be based on compilation and test cycles, for instance. Also, programming laws are a means to change software in a rigorous way.

The presence of specifications in source-code may cause a number of evolution-related difficulties. In special, when refactoring software, in order to either accommodate new requirements or improve its expressiveness, specifications may become outdated. Additionally, changes in specifications are sometimes needed as well. In the context of refactoring, these changes must be behavior-preserving, since the program must be kept in conformance with its specification.

In this chapter we show a systematic approach to apply some refactorings proposed by Fowler [31]. In the sequel, a JML-specified and adapted version of a core module from a huge Manufacturing Execution System [64] (MES) is refactored from successive applications of primitive transformations expressed by means of our programming laws (Chapter 3).

5.1 A Program to Refactor

The Manufacturing Execution Systems were created to fill the communication gap among manufacturing planning systems (MRP, MRPII, ERP, etc.) and control systems used to operate equipment in industries. MESA International [2] provides a definition of what really a MES system is: "Manufacturing Execution Systems (MES) manipulates information that allows the optimizing of the production activities, from the creation of the order to the finished product. Using updated and precise data, the MES guides, initiates, answers, and reports about the plant activities, as they occur. The immediate response to the conditions in constant alteration, joined to the goal of minimizing activities that do not aggregate value to the product, result in processes and operations effective of the plant. The MES increases the return on the operational assets, delivers in time, profits, and performance of the capital flow. The MES provides information of critical mission on the activities of production in all corporations and the supply chain."

A MES system just formalizes methods and procedures of production in an integrated system and presents data in more useful and systematic way. Hence, a MES system assembles all the activities that are not present in the planning layer nor in devices control layer.

5.1.1 The Meta Data API in Focus

Here, we describe an essential module (i.e., an Application Program Interface - API) of the target program we use as an example to the application of our laws. We have chosen this module because it is independent and needs to be implemented in a rigorous way.

In order to control and manipulate data dynamically and in a highly configurable way, our target program is built on top of a Meta Data API. This API provides capabilities to create, edit and delete user-defined data types at execution time as well as to instantiate, to save, to delete and to edit data values to those types. Basically, the Meta Data API has two kind of abstractions: meta data which defines the data type, and the rules the concrete data have to meet, and data, which is used to store concrete values.

When one defines a new meta data, it is necessary to chose the data type, name, default value, a read-only measure unit and validation rules for it. Thus, a data assigned to a meta data has to meet all the meta data characteristics. Figure 5.1 briefly presents a class diagram representing our API. In Appendix C is presented a reduced version of our API's source-code.

The possible types for a meta data are: Integer, Double, String, Boolean and Date. Each one of these types are constants of the class `DataType`, that is a enumeration-like class. Besides it, each data type is represented by a specific class; `IntegerData`, `DoubleData` , `StringData`, `BooleanData` and `DateData`. In addition, a data has a status that can be one of the following types: not registered, valid and invalid. These status compose the values of the enumeration-like class `DateStatus`.

We define validation rules for a meta data instantiating one or more rules of the classes showed in Table 5.1. Each one of the rules listed in Table 5.1 is a value of the enumeration-like class `ValidationType`. The use of validation rules by a meta data is not mandatory. A validation rule can be responsible to validate or invalidate a data or can have no effect. This behavior is defined by its purpose, defined by the enumeration-like class `ValidationPurpose`. If a validation rule has a purpose set to `VALIDATE`, it can be responsible to invalidate or validate a data, however if this purpose is set to `NONE`, this causes no effect.

Data can be registered via the methods `registerValue` and `registerValueFromText`. In addition, we validate using the method `validate`. This method call the method `validate` from all validation rules set in their corresponding meta data as can be seen in Appendix C.1.

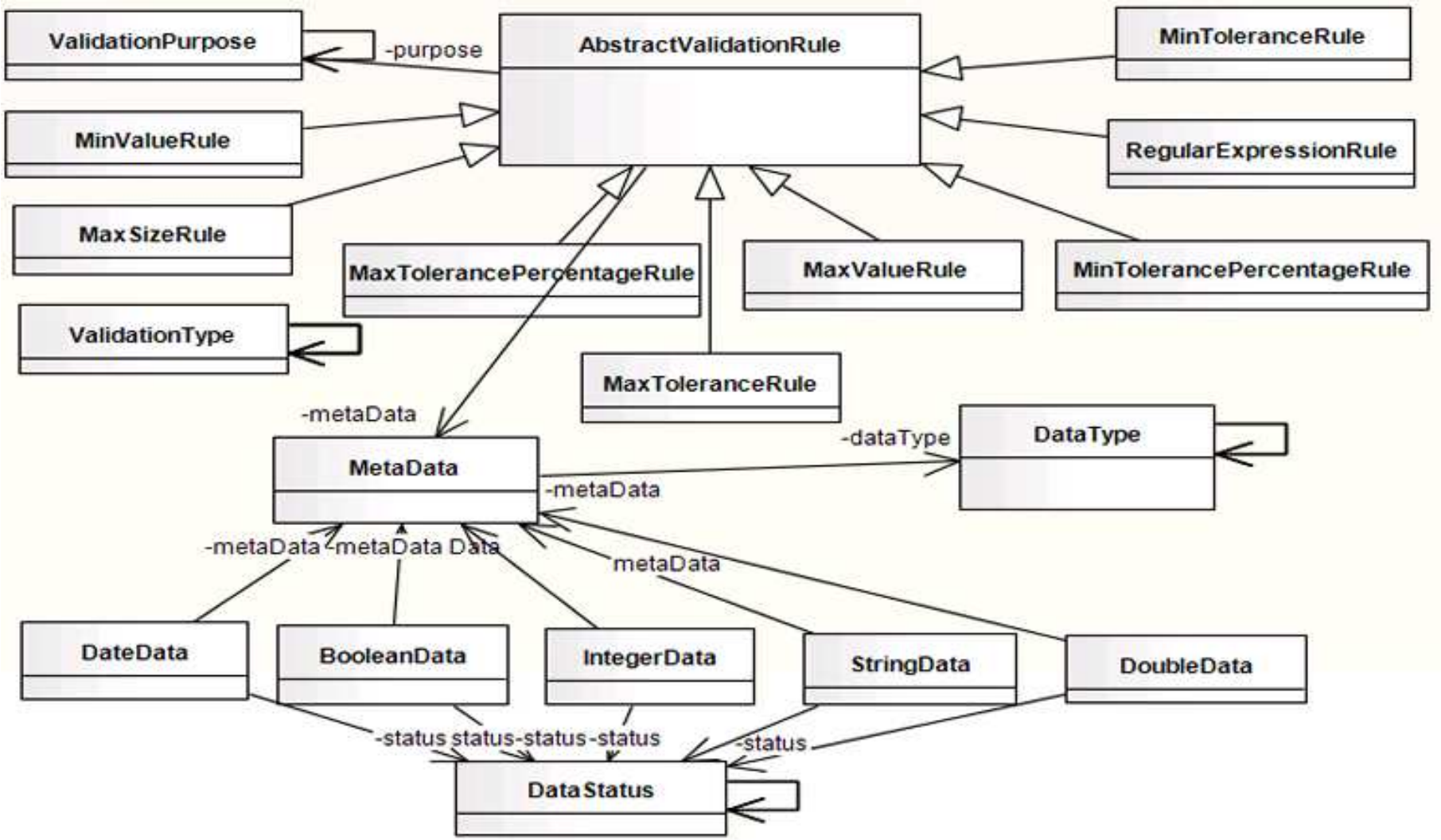


Figure 5.1: Original object diagram from Meta Data API

5.2 Laws Application in Action

The most common way to determine when to refactor is to identify code bad smells. Code smells are implementation structures that negatively affect system lifecycle properties, such as understandability, testability, extensibility, and reusability; that is, code smells ultimately result in maintainability problems [31, 50, 32].

Following these principles we show here how to refactor some code related to bad smells by applying two refactorings by applying our programming laws. We show the usefulness of a systematic approach to evolve source code. It is important to emphasize that our code is JML-specified and our laws are JML-aware what brings various difficulties and challenges to refactoring activities and code evolution. These issues are addressed during the rest of this chapter.

5.2.1 Eliminating Duplicate Code and Introducing Common Interface via Extract Superclass

One of the most common example of code bad smell is duplicate code [47, 32]. Analyzing the source-code of classes `IntegerData`, `StringData`, `DoubleData`, `DateData` and `BooleanData` of Appenidx C.1 is notorious the presence of duplicated code. In Figure 5.2 and Figure 5.3, we show two code excerpts of the classes `DateData` and `IntegerData`. These two figures show some of the common attributes and methods between these two classes and the others cited. Code and specifications are the same.

Besides those methods showed in the Figures 5.2 and 5.3 others methods are equally codified in the classes `*Data` (the notation `*Data` will be used to refer all data classes, `IntegerData`, `DateData`, and so forth): `getRegisteredDate`, `setRegisteredDate`, `getStatus`, `checkValidationRule`, `setEditedDate`, `setValue`, `validateRule`, and `getEditedDate`. See the Appendix C.1 to a better understanding. All these methods should be moved to a generic superclass as well as common attributes, i.e., all attributes listed in the Figures 5.2 and 5.3 and the other with the same name (and semantic) of the classes `StringData`, `DoubleData`, and `BooleanData`.

Other methods can not be moved, i.e., `convertToValue`, `isValid`, `registerValueFromText` and `getFormattedValue`, but we discuss why they cannot be moved later in this section.

Create a generic superclass called `Data`

The first step is to create a new class to serve as a generic superclass, to keep all common methods and attributes as well as common method interfaces (methods that need to be implemented by subclasses and need to be implemented in different manners). We create a new class called `Data` applying **Law 1** from the left to right. All conditions are satisfied since `Data` is fresh in *cds*.

Make all `*Data` classes inherit `Data`

Now we can make the classes that represent data (`IntegerData`, `StringData`, `DoubleData`, `DateData` and `BooleanData`) subclasses of the newly created `Data` class. This is achieved applying **Law 3** from the left to right. As the superclass is new, all the conditions are satisfied and the applications are executed with no problems.

Name	Description	Java Class
Minimum Value	the set value in a data connected to a Meta Data that defines this rule, needs to be equals or greater than a specific value defined in the rule.	`MinValueRule`
Maximum Value	the set value in a data connected to a Meta Data that defines this rule, needs to be equals or less than a specific value defined in the rule.	`MaxValueRule`
Maximum Size	the set value in a data connected to a Meta Data that defines this rule, needs to have length at maximum equals to a specific value defined in the rule.	`MaxSizeRule`
Inferior Tolerance Percentage	the set value in a data connected to a Meta Data that defines this rule, will be considered valid if it is in the range of values between the default value (defined in the Meta Data) subtracted from the percentage of inferior tolerance (a specific value defined in the rule) and the default value.	`MinTolerancePercentageRule`
Superior Tolerance Percentage	the set value in a data connected to a Meta Data that defines this rule, will be considered valid if it is in the range of values between the default value (defined in the Meta Data), and the default value added to the percentage of inferior tolerance (a specific value defined in the rule).	`MaxTolerancePercentageRule`
Inferior Tolerance	work as the inferior tolerance percentage however considers absolute value and not percentage.	`MaxToleranceRule`
Superior Tolerance	work as the superior tolerance percentage however considers absolute value and not percentage	`MinToleranceRule`
Regular Expression	the set value in a data connected to a Meta Data that defines this rule will be considered valid if it matches a specific regular expression (a specific value defined in the rule).	`RegularExpressionRule`

Table 5.1: Validation Rules

```
1  /* ... */
2  public class DateData {
3    //@ invariant this.getMetaData() != null;
4
5    private /*@ spec_public @*/ DataStatus status /* ... */;
6    private MetaData metaData;
7    private /*@ spec_public nullable @*/ Object value;
8    private /*@ spec_public @*/ Date registeredDate /* ... */;
9    private Date editedDate /* ... */;
10   /* ... */
11   public /*@ pure @*/ Object getValue() {
12     return this.value;
13   }
14
15   //@ assignable this.registeredDate, this.status;
16   private void doRegisterActions() {
17     this.setRegisteredDate(new Date());
18     this.validate();
19   }
20
21   /*@ requires value != null;
22     @ assignable this.value, this.registeredDate, this.status;
23     @ ensures this.getValue() == value;
24     @*/
25   public void registerValue(Object value) {
26     this.value = value;
27     this.doRegisterActions();
28   }
29
30   /*@ requires this.getValue() != null;
31     @ assignable this.status;
32     @ ensures true;
33     @*/
34   public void validate() {
35     if (this.metaData.getValidationRules() != null && !this.metaData.
         getValidationRules().isEmpty()) {
36       Iterator iter = this.metaData.getValidationRules().iterator();
37       while (iter.hasNext()) {
38         this.validateRule((AbstractValidationRule)iter.next());
39       }
40     }
41   }
42
43   public /*@ pure @*/ MetaData getMetaData() {
44     return this.metaData;
45   }
46
47   public void setMetaData(MetaData metaData) {
48     this.metaData = metaData;
49   }
50   /* ... */
51 }
```

Figure 5.2: Excerpt of `DateData` class source-code.

```
1  /* ... */
2  public class IntegerData {
3    //@ invariant this.getMetaData() != null;
4
5    private /*@ spec_public @*/ DataStatus status /* ... */;
6    private MetaData metaData;
7    private /*@ spec_public nullable @*/ Object value;
8    private /*@ spec_public @*/ Date registeredDate /* ... */;
9    private Date editedDate /* ... */;
10   /* ... */
11   public /*@ pure @*/ Object getValue() {
12     return this.value;
13   }
14
15   //@ assignable this.registeredDate, this.status;
16   private void doRegisterActions() {
17     this.setRegisteredDate(new Date());
18     this.validate();
19   }
20
21   /*@ requires value != null;
22     @ assignable this.value, this.registeredDate, this.status;
23     @ ensures this.getValue() == value;
24     @*/
25   public void registerValue(Object value) {
26     this.value = value;
27     this.doRegisterActions();
28   }
29
30   /*@ requires this.getValue() != null;
31     @ assignable this.status;
32     @ ensures true;
33     @*/
34   public void validate() {
35     if (this.metaData.getValidationRules() != null && !this.metaData.
         getValidationRules().isEmpty()) {
36       Iterator iter = this.metaData.getValidationRules().iterator();
37       while (iter.hasNext()) {
38         this.validateRule((AbstractValidationRule)iter.next());
39       }
40     }
41   }
42
43   public /*@ pure @*/ MetaData getMetaData() {
44     return this.metaData;
45   }
46
47   public void setMetaData(MetaData metaData) {
48     this.metaData = metaData;
49   }
50   /* ... */
51 }
```

Figure 5.3: Excerpt of `IntegerData` class source-code.

```
public class Data {
2 /*@ spec_public nullable @*/ DataStatus status /* ... */;
3 /*@ spec_public nullable @*/ MetaData metaData;
4 /*@ spec_public nullable @*/ Object value;
5 /*@ spec_public nullable @*/ Date registeredDate /* ... */;
6 /*@ spec_public nullable @*/ Date editedDate /* ... */;
}
```

Figure 5.4: `Data` class source-code with attributes up.

Move all common attributes to `Data`

As we already said in the introduction of this section, all attributes of `*Data` classes have the same meaning. Thus, we can put the attributes in the superclass and remove from the subclasses [1]. To do it, we have to execute the following tasks.

- Choose one of the subclasses, in our case, we chose `BooleanData`, to move the common attributes. First we need to make the attributes public. Public attributes can appear in specifications of private, default, protected and public methods. To make them public, we apply **Law 18** from the left to right to the attributes `metaData` and `editedDate` and **Law 24** from the left to right to the attributes `status`, `value` and `registeredDate`. See the difference between these two laws, a private spec public attribute works like a public attribute if one considers only specification visibility.

- Secondly, its necessary to make all attributes that are typed with a reference type, nullable. The attribute `value` is already nullable, but the others are not. Then we apply law **Law 26** from the left to right to the attributes `metaData`, `registeredDate`, `status`, and `editedDate`.

- We move each one of the attributes cited above using **Law 27**. As `Data` is a new class it has no attributes, and the unique law condition is satisfied.

- The previous step made all attributes in the other `Data` subclasses shadows. We have to delete them. In order to delete these attributes we apply **Law 30** from the left to right. Before applying **Law 30** we need to make the attributes (in subclasses of `Data`) public, applying the same laws as discussed before. The condition expressed in **Law 30** is trivially satisfied since `Data` is a new class and all its attributes are new too. All accesses to the now non-existent attributes of `IntegerData`, `StringData`, `DoubleData` and `DateData` are now redirected to the same attributes in the superclass `Data`.

- Normally we use protected attributes in a superclass (not public ones) when we want to make them visible to its subclasses. However, in our case we make all attributes *default* since we consider that there is a unique package context. But, we also need to consider the JML-specifications. Hence, we make the attributes of `Data` *default* and *spec public*. We achieve it by applying **Law 21** from the right to left.

Figure 5.4 shows the newly created superclass `Data`.

[1] We know that when removing attributes from the subclasses the accesses to these attributes will point to the attributes of the superclass what characterizes a situation of data refinement. We do not deal with data refinement in this work and we assume here that we can do this operation safely.

Move common methods to the `Data` class

As we discussed previously, the methods presented in Figures 5.2 and 5.3 and others methods equally codified in the classes *Data (`checkValidationRule`, `setValue`, `getStatus`, `getRegisteredDate`, `setRegisteredDate`, `validateRule`, `setEditedDate` and `getEditedDate`) need be moved to the class `Data`.

The methods cited above are completely equal. But the other methods `convertToValue` , `isValid`, `registerValueFromText` and `getFormattedValue` have behaviors that are specific to the classes in which they are declared.

We begin with the methods that have the same code. We choose one of the subclasses, in our case, we chose `BooleanData`, to move the common methods. First we choose to move getter and setter methods: `setMetaData`, `getMetaData`, `setValue`, `getValue` , `setEditedDate`, `getEditedDate`, `getStatus`, `getRegisteredDate`, and `setRegisteredDate`. The `Data` class has no methods, thus we use **Law 68** from the left to right, to move up these methods [2]. Almost all conditions are satisfied for these methods. **super** does not occur in the methods bodies nor in their specifications. As all methods specifications are equals which satisfy the JML conditions (2) and (3). At this point all attributes are default. However, there is one condition that is unsatisfied, all occurrences of **this** are uncast. Then, we apply law **Law 100** from the left to right to such occurrences. After these steps we can safely apply **Law 68** from the left to right.

Now we move up the others methods that have the same code: `doRegisterActions`, `validate`, `registerValue`, `validateRule` and `checkValidationRule`. All conditions satisfied by the getter and setter methods cited in the previous paragraph are also satisfied in these methods for the same reason. Again we have to apply **Law 100** from the left to right to occurrences of **this** in the bodies of these methods. Before we move the private methods `doRegisterActions`, `validateRule` and `checkValidationRule`, we need to make them public in all subclasses of `Data` (applying **Law 56** from the left to right) in order to fit **Law 68**. The condition of **Law 56** (from the left to right) is satisfied since all attributes are default and have public specification visibility and pure methods and model fields do not occur in the specifications. After these steps we apply **Law 68** from the left to right [3]. It is important to highlight that all specifications cases of the methods with the same name of the recently moved methods of `BooleanData` in the subclasses `StringData`, `DateData`, `DoubleData`, and `IntegerData` were changed and now they start with an **also** clause.

The methods `isValid`, `convertToValue`, `registerValueFromText` and `getFormattedValue` remain in their classes.

The method `isValid` has a different implementation and specification cases in the class `StringData`. Moving up the method `isValid` of classes `DoubleData`, `IntegerData` and `DateData` would break the conditions 2 and 3 of **Law 68** since the postcondition of `isValid` method of `DateData` does not imply the postcondition of `StringData`, for example (see Appendix C.1). The same reasoning is used to treat the methods `registerValueFromText` and `getFormattedValue`. Considering the method `convertToValue`, the conditions (2) and (3) of **Law 68** are satisfied, however, this method has different implementations in subclasses

[2] Our laws that deal with methods consider that the methods are *default*. Thus, from now on we consider that before applying the laws of methods listed throughout this section (except the laws that deal with visibility changes) we apply a law (for instance, **Law 54**) to change the visibility of the method to default. We also consider that at the end of the application of the laws of methods we change the visibility of the method back to public.

[3] Recall that we have to transform the methods to *default* before applying the law to the methods and that at the end of the law application we make the methods back to public

```
1  public class Data {
2
3     /* ... */
4
5    /*@ requires (!(this instanceof DateData)) && this.getValue() !=
         null;
6      @ assignable this.status;
7      @ ensures (!(this instanceof DateData)) && true;
8      @ also
9      @ requires (this instanceof DateData) && this.getValue() != null;
10     @ assignable this.status;
11     @ ensures (this instanceof DateData) && true;
12     @ also
13     @ requires (this instanceof DateData) && this.getValue() != null;
14     @ assignable this.status;
15     @ ensures (this instanceof DateData) && true;
16     @*/
17   public void validate() {
18     if (!(this instanceof DateData)) {
19       if (this.metaData.getValidationRules() != null && !this.
            metaData.getValidationRules().isEmpty()) {
20         Iterator iter = this.metaData.getValidationRules().iterator()
              ;
21         while (iter.hasNext()) {
22           this.validateRule((AbstractValidationRule)iter.next());
23         }
24       }
25     } else {
26       if (this.metaData.getValidationRules() != null && !this.
            metaData.getValidationRules().isEmpty()) {
27         Iterator iter = this.metaData.getValidationRules().iterator()
              ;
28         while (iter.hasNext()) {
29           this.validateRule((AbstractValidationRule)iter.next());
30         }
31       }
32     }
33   }
34
35  /* ... */
36
37 }
```

Figure 5.5: `validate` method immediately after moved up from `DateData` to `Data` class.

of Data, thus we keep the methods convertToValue in their classes. We will resume the discussion about these methods in a later section. After the methods movement, we can exclude the trivial casts inserted in previous steps using the same laws we used but in opposite direction.

After moving up setters, getters and the methods doRegisterActions, registerValue , validate, validateRule, checkValidationRule of BooleanData we need to eliminate the same methods in the others subclasses of Data. To accomplish it, we need to apply **Law 70**, from the left to right. In order to satisfy the conditions of this law, we apply **Law 100**, from the left to right to cast occurrences of **this** in methods bodies and to occurrences of **this** in specifications of pure methods.

Now we can move up the methods. Each one of the methods setValue, getValue , setMetaData, getMetaData, setEditedDate, getEditedDate, doRegisterActions, validate, getRegisteredDate, setRegisteredDate, checkValidationRule, registerValue, getStatus, and validateRule are moved up by application of **Law 70**, from the left to right. After each application we apply **Law 97**, from the left to right, to reduce if-else clauses to a single command, and **Law 100** from the right to left to uncast back **this** occurrences. We apply propositional calculus to reduce specification cases to the original ones. The last reduction is not mandatory and we applied it only to simplify the specifications (we did not created laws to execute this task).

In order to exemplify the previous steps we show in Figure 5.5 the method validate immediately after the application of **Law 70** from DateData to Data class. As can be seen, the disjunction of the conditionals (if-else, lines 18 and 25) is true, and the same command (lines 19 to 24 and 26 to 32) appears in both branches of the conditionals. We can replace the alternation by just the command. Considering the specification cases it is not difficult to realize that the specification cases presented in Figure 5.5 can be simplified to the one showed in Figure 5.6. In Figure 5.6 we present the version of the method validated in the class Data at the end of methods movements. Note that there are no casts in **this** expressions.

```
1  public class Data {
2    /* ... */
3
4    /*@ requires this.getValue() != null;
5      @ assignable this.status;
6      @ ensures true;
7      @*/
8    public void validate() {
9      if (this.metaData.getValidationRules() != null && !this.metaData.
           getValidationRules().isEmpty()) {
10       Iterator iter = this.metaData.getValidationRules().iterator();
11       while (iter.hasNext()) {
12         this.validateRule((AbstractValidationRule)iter.next());
13       }
14     }
15   }
16
17  /* ... */
18 }
```

Figure 5.6: The final version of validate method in Data class.

```
1  public class AbstractValidationRule {
2
3   /* ... */
4  public boolean validate(Object data) {
5    if (this.getType().equals(ValidationType.MAX_VALUE)) {
6      return this.validateMaxValue(data);
7    } else if (this.getType().equals(ValidationType.MIN_VALUE)) {
8      return this.validateMinValue(data);
9    } else if (this.getType().equals(ValidationType.MAX_TOLERANCE)) {
10     return this.validateMaxTolerance(data);
11   } else if (this.getType().equals(ValidationType.MIN_TOLERANCE)) {
12     return this.validateMinTolerance(data);
13   } else if (this.getType().equals(ValidationType.
       MAX_TOLERANCE_PERCENTAGE)) {
14     return this.validateMaxTolerancePercentage(data);
15   } else if (this.getType().equals(ValidationType.
       MIN_TOLERANCE_PERCENTAGE)) {
16     return this.validateMinTolerancePercentage(data);
17   } else if (this.getType().equals(ValidationType.REGULAR_EXPRESSION)
       ) {
18     return this.validateRegularExpression(data);
19   } else if (this.getType().equals(ValidationType.MAX_SIZE)) {
20     return this.validateMaxSize(data);
21   } else if (this.getType().equals(ValidationType.NONE)) {
22     return true;
23   } else {
24     return false;
25   }
26 }
27 /* ... */
28 }
```

Figure 5.7: `AbstractValidationRule` class with a zoom in the `validate` method.

Change parameter, return types and local variables types to `Data`

After moving up all the common elements, we need to check if clients of the subclasses use only the common interface, i.e., methods that were moved up and now are in `Data` class. If so, we can change the required type to `Data`. In what follows, we describe the places in which this action can be executed.

In the class `Data`, the parameter `data` of the method `validate` (see Figure 5.7) is of type `Object`. However, as now we have a generic superclass to represent data abstraction we can modify the type to `Data`. Our first intention is to change the type of `data` directly using **Law 72**. Nevertheless, the condition (from the left to right and vice-versa) is not satisfied because there are no casts in the body of method `validate`. To address this condition we have to execute the following tactic:

- Use **Law 107**, from the left to right introducing a temporary variable `tmp1`.
- Apply **Law 104** from the right to left, to introduce cast in the newly created assignment (line 7, Figure 5.8). Figure 5.8 presents the method `validate` after these two steps.
- There is only one occurrence of `data` inside the body of the method `validate` body

and this is casted. Hence, the condition (from the left to right and vice-versa) of **Law 72** is satisfied.

- We use **Law 107** from the right to left to eliminate `tmp1` and replace this occurrences by `(Data) data`. We also apply **Law 100** from the right to left to remove the trivial casts introduced.

The condition (1) (from the right to left) of **Law 72**, is not satisfied since there is an uncast occurrence of `validate` in the body of the method `checkValidationRule` of class `Data` (see the body of the method in Appendix C.2). Thus, we have to a introduce trivial cast in this occurrence applying **Law 100** from the left to right. There are no specification cases in the method `validate`, thus the JML conditions of **Law 72** are satisfied. Finally we can apply **Law 72**, from the left to right. Figure 5.9 presents the final version of the method `validate` of class `AbstractValidationRule`.

```
1  public class AbstractValidationRule {
2
3   /* ... */
4  public boolean validate(Object data) {
5    Object tmp1;
6    /*@ assert (data instanceof Data); @*/
7    tmp1 = (Data)data;
8    if (this.getType().equals(ValidationType.MAX_VALUE)) {
9      return this.validateMaxValue(tmp1);
10   } else if (this.getType().equals(ValidationType.MIN_VALUE)) {
11     return this.validateMinValue(tmp1);
12   } else if (this.getType().equals(ValidationType.MAX_TOLERANCE)) {
13     return this.validateMaxTolerance(tmp1);
14   } else if (this.getType().equals(ValidationType.MIN_TOLERANCE)) {
15     return this.validateMinTolerance(tmp1);
16   } else if (this.getType().equals(ValidationType.
        MAX_TOLERANCE_PERCENTAGE)) {
17     return this.validateMaxTolerancePercentage(tmp1);
18   } else if (this.getType().equals(ValidationType.
        MIN_TOLERANCE_PERCENTAGE)) {
19     return this.validateMinTolerancePercentage(tmp1);
20   } else if (this.getType().equals(ValidationType.REGULAR_EXPRESSION)
        ) {
21     return this.validateRegularExpression(tmp1);
22   } else if (this.getType().equals(ValidationType.MAX_SIZE)) {
23     return this.validateMaxSize(tmp1);
24   } else if (this.getType().equals(ValidationType.NONE)) {
25     return true;
26   } else{
27     return false;
28   }
29 }
30 /* ... */
31 }
```

Figure 5.8: `AbstractValidationRule` class with a zoom in the `validate` method after replace expression by variable `tmp1`

Continuing the process of changing types, we change the type of parameter `data` of the methods `validateMaxTolerance`, `validateMinTolerance`, `validateMaxTolerancePercentage`,

```
1  public class AbstractValidationRule {
2
3   /* ... */
4  public boolean validate(Data data) {
5    /*@ assert (data instanceof Data); @*/
6    if (this.getType().equals(ValidationType.MAX_VALUE)) {
7      return this.validateMaxValue(data);
8    } else if (this.getType().equals(ValidationType.MIN_VALUE)) {
9      return this.validateMinValue(data);
10   } else if (this.getType().equals(ValidationType.MAX_TOLERANCE)) {
11     return this.validateMaxTolerance(data);
12   } else if (this.getType().equals(ValidationType.MIN_TOLERANCE)) {
13     return this.validateMinTolerance(data);
14   } else if (this.getType().equals(ValidationType.
         MAX_TOLERANCE_PERCENTAGE)) {
15     return this.validateMaxTolerancePercentage(data);
16   } else if (this.getType().equals(ValidationType.
         MIN_TOLERANCE_PERCENTAGE)) {
17     return this.validateMinTolerancePercentage(data);
18   } else if (this.getType().equals(ValidationType.REGULAR_EXPRESSION)
         ) {
19     return this.validateRegularExpression(data);
20   } else if (this.getType().equals(ValidationType.MAX_SIZE)) {
21     return this.validateMaxSize(data);
22   } else if (this.getType().equals(ValidationType.NONE)) {
23     return true;
24   } else {
25     return false;
26   }
27 }
28 /* ... */
29 }
```

Figure 5.9: `AbstractValidationRule` class with a zoom in the `validate` method. Version with `data` parameter changed do `Data`.

`validateMinTolerancePercentage`, `validateMaxSize`, `validateMaxValue` and of the method `validateMinValue` from `Object` to `Data` by applying **Law 71** and of we change the type of the method `validateRegularExpression` by applying **Law 72** since it is not pure. Before applying these two laws we need to make all those methods default – using **Law 59** and **Law 57** for pure methods, and, **Law 56** and **Law 54** for the non-pure ones – in order to fit **Law 71** and **Law 72**. We can do it because all pure methods that appear in the specification cases of those methods are public.

The conditions of **Law 59** and **Law 56** are satisfied (considering **Law 57** and **Law 54** there are no conditions to be satisfied). The JML and Java conditions (1) (for right to left) are satisfied since the methods were private. The Java and JML (for application in both directions) conditions are satisfied because the parameter type of `data` is `Object` and all accesses to `data` attributes and methods are cast. And, the Java and JML conditions (2) (for left to right) are also satisfied. Finally, we modify the type of parameter `data` of the methods `validateMaxTolerance`, `validateMinTolerance`, `validateMaxTolerancePercentage`, `validateMinTolerancePercentage`, `validateMaxSize`, `validateMaxValue` and `validateMinValue` by applying **Law 71**, and of `validateRegularExpression`, by applying **Law 72**.

At last, we remove all trivial casts introduced in last steps. and make the methods

```
1  public class AbstractValidationRule {
2   /* ... */
3    /*@ requires (data instanceof StringData) && ((StringData)data).
         getValue() != null
4         && ((MaxSizeRule)this).getReferenceValue() != null;
5       @ assignable \nothing;
6       @ ensures \result == ((String)((StringData)data).getValue()).
           length() <= ((MaxSizeRule)this).getReferenceValue().intValue
           ();
7       @ also
8       @ requires !(data instanceof StringData);
9       @ assignable \nothing;
10      @ ensures \result == false;
11      @*/
12    private /*@ pure @*/ boolean validateMaxSize(Object data) {
13      if (data instanceof StringData) {
14        if (((StringData)data).getValue() != null) {
15          return ((String)((StringData)data).getValue()).length() <= ((
              MaxSizeRule)this).getReferenceValue().intValue();
16        }
17      }
18      return false;
19    }
20  /* ... */
21  }
```

Figure 5.10: `AbstractValidationRule` class with a zoom in the `validate` method.

public by applying **Law 57** and **Law 54**. Figure 5.10 shows the method `validateMaxSize` before laws application and Figure 5.11 shows the final version.

Move duplicated and weaker invariants to `Data` class

This section describes an important step added to the refactoring Extract Superclass. Common and weaker invariants should be moved up to the most abstract class in the hierarchy, in our case, the new superclass `Data`. The same action may be applied to history constraints however since our program does not use history constraints a detailed discussion is omitted.

As can be seen in Appendix C.1 we have similar invariants – **invariant this**.`getMetaData`()!= **null** – in each one of the subclasses of `Data`: `BooleanData`, `DateData`, `IntegerData`, `DoubleData` and `StringData`. To move each one of the invariants we need apply **Law 5** from the left to right.

Before apply **Law 5**, we need to prepare all invariants introducing trivial casts in **this** expressions in all invariants applying **Law 100**, from the left to right. Now it is safe to apply **Law 5** from the left to right in all invariants. We delete trivial casts using **Law 108**, from the left to right. An excerpt of `Data` class highlighting the recently moved invariants is presented in Figure 5.12. Propositional calculus is used to simplify the invariant predicates. The invariant of Data is the conjunction of predicates written in lines 3 to 7 of Figure 5.12. All **instanceof** expressions cover all possible subtypes of `Data` which means that the common predicate can be used in the place of all invariants of lines 3 to 7 of Figure 5.12. The final invariant of data is: **invariant this**.`getMetaData`()!= **null**.

```
1  public class AbstractValidationRule {
2   /* ... */
3    /*@ requires (data instanceof StringData) && data.getValue() !=
         null
4        && ((MaxSizeRule)this).getReferenceValue() != null;
5      @ assignable \nothing;
6      @ ensures \result == ((String)data.getValue()).length() <= ((
           MaxSizeRule)this).getReferenceValue().intValue();
7      @ also
8      @ requires !(data instanceof StringData);
9      @ assignable \nothing;
10     @ ensures \result == false;
11     @*/
12   public /*@ pure @*/ boolean validateMaxSize(Data data) {
13     if (data instanceof StringData) {
14       if (((StringData)data).getValue() != null) {
15         return ((String)data.getValue()).length() <= ((MaxSizeRule)
               this).getReferenceValue().intValue();
16       }
17     }
18     return false;
19   }
20 /* ... */
21 }
```

Figure 5.11: `AbstractValidationRule` class with a zoom in the `validateMaxSize` method after laws application.

Check common interface and create empty methods for it in the superclass

In this last step we create a common interface in `Data` class to the methods `convertToValue`, `isValid`, `registerValueFromText` and `getFormattedValue`. Following Fowler's instructions, we make `Data` abstract and create abstract methods to represent those methods and obligate new subclasses to implement the common interface. In addition, all calls to those methods may be made via `Data` instances and not only via subclasses instances. Hence, some casts need are eliminated and class `Data` is used in some places where `Object` is used.

First we make `Data` abstract applying **Law 2**, from the left to right. Second, we apply **Law 87** from the left to right to introduce the four methods `convertToValue`, `isValid` , `registerValueFromText` and `getFormattedValue`. All formal parameters, return types, names, and specification keywords (eg. pure, spec_public) are replicated in the new abstract methods.

Now we have a complete common interface in `Data`. We can find and change subtypes occurrences, like declarations, by `Data` declarations. In fact, in the method `main` of class `Main` we have calls to `isValid`, `registerValueFromText` and `getFormattedValue` with local variables, which are not declared as of `Data` class (see Figure 5.13 lines 14, 15, 16, 23, 24, 25, 32, 33 and 34).

We change the local variable types cited above (lines 13, 22, and 31 of Figure 5.13) and the others found in the method `main` of class `Main` applying **Law 106** from the left to right. In order to meet the conditions of that law we apply **Law 100** from the left to right to those occurrences and to the occurrences of the local variables in the **assert** clauses (see lines 19, 28 and 37 of Figure 5.13). After that we change local variables types to `Data`. **Law 105** and **Law 109** are used to eliminate trivial casts introduced previously. With this

```
1 public class Data {
2 /* ... */
3 //@ invariant (this instanceof DateData) ==> this.getMetaData() !=
      null;
4 //@ invariant (this instanceof DoubleData) ==> this.getMetaData() !=
      null;
5 //@ invariant (this instanceof IntegerData) ==> this.getMetaData() !=
       null;
6 //@ invariant (this instanceof StringData) ==> this.getMetaData() !=
      null;
7 //@ invariant (this instanceof BooleanData) ==> this.getMetaData() !=
       null;
8 /* ... */
9 }
```

Figure 5.12: Invariants moved up to `Data` class immediately before reduction.

step, we finish the Extract Superclass refactoring for class `Data`. Figure 5.14 shows the excerpt of the class `Main` after the application of laws. Appendix C.2 shows the whole program after this refactoring.

5.2.2 Introducing Replace Conditional With Polymorphism

One of the most important features in object-oriented development is polymorphism [31]. Using this feature, one can avoid writing specific conditionals to address specific behaviors in a superclass and leaving these responsibilities to subclasses. Even when there are no subclasses, creating some to implement the specific behavior methods may be a good choice. As a result the existence of switch statements to deal with type codes or if-then-else statements which makes selections based on type strings are much less common in object-oriented programs [31].

The situation explained above can be visualized in the method `validate` of the class `AbstractValidationRule`. Figure 5.9 gives details of that method. We have many *if* branches; each branch executes a type test (via the method `getType`) and, depending on the type, executes a specific code, i.e., calls a specific method. This situation is a common example when polymorphism can be used to clean code and to improve code design. The refactoring *Replace Conditional with Polymorphism* is commonly used in situations like that. In what follows we show how this refactoring is applied in the `validate` method of `AbstractValidationRule` class using our laws. We emphasize that our program is not a common Java program, but a formally specified Java program. The peculiarities of this fact are discussed along this section.

Introducing method redefinitions in subclasses and copying superclass method body

We must introduce the method `validate` in each subclass, `MinValueRule`, `MaxValueRule` , `MinValueRule`, `MaxSizeRule`, `MinTolerancePercentageRule`, `MaxTolerancePercentageRule`, `MaxToleranceRule`, `MinToleranceRule` and `RegularExpressionRule`. First, we apply **Law 67** from the left to right in all those subclasses. Invariants of those subclasses do not restrict elements of superclass [4], thus, the JML condition is satisfied. The Java condition

[4]Calls to super inside a subclasse when the subclasse's invariant restricts elements of its superclass is not a good practice in DbC, see [59].

```
1  public class Main {
2
3    public static void main (String[] args) {
4      //Simulating a set of properties to a Product
5      MetaData length = createLengthProperty();
6      MetaData weigth = createWeigthProperty();
7      MetaData code = createCodeProperty();
8      MetaData productionDate = createProductionDateProperty();
9      MetaData numberOfInternalParts =
           createNumberOfInternalPartsProperty();
10     MetaData needsPacking = createNeedsPackingProperty();
11
12     //Simulating data registering for Length
13     DoubleData dataLength = new DoubleData(length);
14     dataLength.registerValueFromText("10");
15     System.out.println("data value: " + dataLength.getFormattedValue
           () +
16         " is a " + dataLength.isValid() + " well-formed value as
               expected and " +
17         "its expected status is " +
18         "INVALID, the real status is: " + dataLength.getStatus());
19     //@ assert dataLength.getStatus().equals(DataStatus.INVALID);
20
21     //Simulating data registering for Weigth
22     DoubleData dataWeigth = new DoubleData(weigth);
23     dataWeigth.registerValueFromText("4");
24     System.out.println("data value: " + dataWeigth.getFormattedValue
           () +
25         " is a " + dataWeigth.isValid() + " well-formed value as
               expected and " +
26         "its expected status is " +
27         "VALID, the real status is: " + dataWeigth.getStatus());
28     //@ assert dataWeigth.getStatus().equals(DataStatus.VALID);
29
30     //Simulating data registering for Code
31     StringData dataCode = new StringData(code);
32     dataCode.registerValueFromText("XYZ001");
33     System.out.println("data value: " + dataCode.getFormattedValue()
           +
34         " is a " + dataCode.isValid() + " well-formed value as
               expected and " +
35         "its expected status is " +
36         "VALID, the real status is: " + dataCode.getStatus());
37     //@ assert dataCode.getStatus().equals(DataStatus.VALID);
38
39   /* ... */
40   }
41 }
```

Figure 5.13: Excerpt of the original version of Main class of our target program.

```
1  public class Main {
2    public static void main (String[] args) {
3      //Simulating a set of properties to a Product
4      MetaData length = createLengthProperty();
5      MetaData weigth = createWeigthProperty();
6      MetaData code = createCodeProperty();
7      MetaData productionDate = createProductionDateProperty();
8      MetaData numberOfInternalParts =
           createNumberOfInternalPartsProperty();
9      MetaData needsPacking = createNeedsPackingProperty();
10
11     //Simulating data registering for Length
12     Data dataLength = new DoubleData(length);
13     dataLength.registerValueFromText("10");
14     System.out.println("data value: " + dataLength.getFormattedValue
           () +
15         " is a " + dataLength.isValid() + " well-formed value as
               expected and " +
16         "its expected status is " +
17         "INVALID, the real status is: " + dataLength.getStatus());
18     //@ assert dataLength.getStatus().equals(DataStatus.INVALID);
19
20     //Simulating data registering for Weigth
21     Data dataWeigth = new DoubleData(weigth);
22     dataWeigth.registerValueFromText("4");
23     System.out.println("data value: " + dataWeigth.getFormattedValue
           () +
24         " is a " + dataWeigth.isValid() + " well-formed value as
               expected and " +
25         "its expected status is " +
26         "VALID, the real status is: " + dataWeigth.getStatus());
27     //@ assert dataWeigth.getStatus().equals(DataStatus.VALID);
28
29     //Simulating data registering for Code
30     Data dataCode = new StringData(code);
31     dataCode.registerValueFromText("XYZ001");
32     System.out.println("data value: " + dataCode.getFormattedValue()
           +
33         " is a " + dataCode.isValid() + " well-formed value as
               expected and " +
34         "its expected status is " +
35         "VALID, the real status is: " + dataCode.getStatus());
36     //@ assert dataCode.getStatus().equals(DataStatus.VALID);
37     /* ... */
38   }
39 }
```

Figure 5.14: Excerpt of the final version of Main class of our target program after execution of Extract Superclass `Data`.

is satisfied because abstract methods does not make sense in this refactoring. Therefore, we create methods with specific behavior.

Our goal in this step is to have a copy of method `validate` in subclasses of the class `AbstractValidationRule`. Hence, we have to copy the body of `validate` to the recently created `validate` methods in subclasses. To achieve it, we must apply **Law 82** from the left to right, in each method `validate` of the subclasses of `AbstractValidationRule`.

We need to satisfy the conditions (only the ones necessary to apply the law from the left to right) of the **Law 82**. We follow the micro steps that below:

- Eliminate the multiple return points in method `validate`, by using **Law 65** from the left to right. Condition (2) is satisfied since we have mutually exclusive conditionals.

- Before applying **Law 65**, it is necessary to make `validate`'s body fit the template of this law. Thus, we apply **Law 107** from the left to right.

The other conditions of **Law 82** already are satisfied. JML conditions are satisfied because we do not have **super** and non-private elements in specifications. Now, we can apply **Law 82** from the left to right safely to the methods `validate` of classes `MaxToleranceRule`, `MinToleranceRule`, `MinValueRule`, `MaxSizeRule`, `RegularExpressionRule`, `MinTolerancePercentageRule`, `MaxTolerancePercentageRule`, `MinValueRule`, `MaxValueRule` and Figure 5.15 shows an excerpt of class `MaxValueRule` showing the method `validate`. The others methods `validate` of the others subclasses are equals.

Eliminate non specific conditionals and use specific behavior command

As is exemplified in Figure 5.15, the bodies of the method `validate` of subclasses of `AbstractValidationRule` have many conditional branches. However, those branches call a specific `getType` method in each subclass. Each one of these methods returns a constant depending on the subclass. For example, in the method `getType` of `MaxValueRule` class of Figure 5.15, the constant `ValidationType.MAX_VALUE` is returned. We can reduce all the branches of `validate` of `MaxValueRule` to specific branch where the branch test matches the constant `ValidationType.MAX_VALUE`. The same reasoning can be applied to the others `validate` methods of the others subclasses of `AbstractValidationRule`.

Before these steps we apply **Law 107** from the left to right followed by **Law 65** from the left to right in each method `validate` of the subclasses of `AbstractValidationRule` to facilitate the application of our strategy. Then, we reduce the conditional branches of the methods `validate`. Figure 5.16 shows the `validate` method of `MaxValueRule` at this point. We eliminate the remaining *if* via **Law 96** from the left to right.

At this point we may finish the refactoring. However, to improve code design we choose to move down the methods `validateMaxTolerance`, `validateMinTolerance`, `valida-teMaxSize`, `validateMaxValue`, `validateMinValue`, `validateMaxTolerancePercentage`, `va-lidateMinTolerancePercentage`, and `validateRegularExpression` to the corresponding subclasses, `MaxToleranceRule`, `MinToleranceRule`, `MaxSizeRule`, `MaxValueRule`, `MinValueRule`, `MaxTolerancePercentageRule`, `MinTolerancePercentageRule`, and `RegularExpressionRule`, respectively. Hence, we apply **Law 68** from the right to left to each of these methods to their respective target subclasses. To apply this law we need to eliminate each call to the involved methods inside `AbstractValidationRule` class. Each call is eliminated by applying **Law 84** from the left to right. After that, all the conditions of **Law 68** (for applying

```
1  public class MaxValueRule extends AbstractValidationRule {
2    /* ... */
3    /*@ also
4      @ requires \same;
5      @ assignable \nothing;
6      @ ensures \result.equals(ValidationType.MAX_VALUE);
7      @*/
8    public /*@ pure @*/ ValidationType getType() {
9      return ValidationType.MAX_VALUE;
10   }
11
12   public boolean validate(Data data) {
13     boolean tmp;
14     /*@ assert this.getPurpose() != null; @*/
15     boolean result;
16     /*@ assert (data instanceof Data); @*/
17     if (this.getType().equals(ValidationType.MAX_VALUE)) {
18       result = this.validateMaxValue(data);
19     } else if (this.getType().equals(ValidationType.MIN_VALUE)) {
20       result = this.validateMinValue(data);
21     } else if (this.getType().equals(ValidationType.MAX_TOLERANCE)) {
22       result = this.validateMaxTolerance(data);
23     } else if (this.getType().equals(ValidationType.MIN_TOLERANCE)) {
24       result = this.validateMinTolerance(data);
25     } else if (this.getType().equals(ValidationType.
          MAX_TOLERANCE_PERCENTAGE)) {
26       result = this.validateMaxTolerancePercentage(data);
27     } else if (this.getType().equals(ValidationType.
          MIN_TOLERANCE_PERCENTAGE)) {
28       result = this.validateMinTolerancePercentage(data);
29     } else if (this.getType().equals(ValidationType.
          REGULAR_EXPRESSION)) {
30       result = this.validateRegularExpression(data);
31     } else if (this.getType().equals(ValidationType.MAX_SIZE)) {
32       result = this.validateMaxSize(data);
33     } else if (this.getType().equals(ValidationType.NONE)) {
34       result = true;
35     } else {
36       result = false;
37     }
38     tmp = result;
39     /*@ assert this.getPurpose() != null; @*/
40     return tmp;
41   }
42 }
```

Figure 5.15: Excerpt of class `MaxValueRule` after the first step of *Replace Conditional with Polymorphism*.

```
1 public class MaxValueRule extends AbstractValidationRule {
2   /* ... */
3   public boolean validate(Data data) {
4     /*@ assert (data instanceof Data); @*/
5     if (this.getType().equals(ValidationType.MAX_VALUE)) {
6       return this.validateMaxValue(data);
7     }
8   }
9 }
```

Figure 5.16: Excerpt of class `MaxValueRule` after the conditionals reducing in step 2 of *Replace Conditional with Polymorphism*.

the law from the right to left) are satisfied since there is no **super** calls and those methods are not redefined in their target classes (which satisfies JML conditions). Thus, we move down the cited methods. To finish this step we make the recently moved down methods private applying **Law 56** from the right to left or **Law 59** from the right to left.

Figure 5.17 shows an excerpt of the final version of `MaxValueRule` class. All the other subclasses of `AbstractValidationRule` seems like `MaxValueRule`, see Appendix C.3 for the whole source code of these classes.

5.2.3 Extracting a More Specialized Superclass to Number-Based Validation Rules

The subclasses of `AbstractValidationRule` (except for the class `RegularExpressionRule`) base they validation logic in a reference number, see for example the method `validateMaxValue` of `MaxValueRule` in Figure 5.17. These kind of classes, i.e., number-based validation rules, can be generalized. Thus, we apply the refactoring *Extract Superclass* again to create a superclass called `AbstractNumberValidationRule`.

We follow the same strategy we used in Section 5.2.1, however here, we simplify the steps and the discussion about them.

Create a generic superclass called `AbstractNumberValidationRule`

The first step is to create a new class to serve as a superclass, as we did in Section 5.2.1. We create a new class called `AbstractNumberValidationRule`, applying **Law 1** from the left to right and make it subclass of `AbstractValidationRule` by applying **Law 3** from the left to right. All conditions are satisfied since `AbstractNumberValidationRule` is fresh in *cds*.

Make all number-based validation rules classes inherit `AbstractNumberValidationRule`

Now we make the classes `MaxValueRule`, `MinValueRule`, `MaxToleranceRule` `MinToleranceRule` , `MaxSizeRule`, `MinTolerancePercentageRule`, and `MaxTolerancePercentageRule` subclasses of the newly created `AbstractNumberValidationRule` class by applying **Law 4** from the left to right. As the superclass is a new fresh class, all the conditions are satisfied and the applications are executed with no problems.

```
1  public class MaxValueRule extends AbstractValidationRule {
2    //@ invariant this.getReferenceValue() != null;
3    private Double referenceValue;
4    /* ... */
5    /*@ also
6      @ requires \same;
7      @ assignable \nothing;
8      @ ensures \result.equals(ValidationType.MAX_VALUE);
9      @*/
10   public /*@ pure @*/ ValidationType getType() {
11     return ValidationType.MAX_VALUE;
12   }
13   public void setReferenceValue(Double referenceValue) {
14     this.referenceValue = referenceValue;
15   }
16   public /*@ pure @*/ Double getReferenceValue() {
17     return referenceValue;
18   }
19   /*@ requires (data instanceof DoubleData) && data.getValue() !=
         null && ((MaxValueRule)this).getReferenceValue() != null;
20     @ assignable \nothing;
21     @ ensures \result == (((MaxValueRule)this).getReferenceValue().
         compareTo(data.getValue()) >= 0);
22     @ also
23     @ requires (data instanceof IntegerData) && data.getValue() !=
         null && ((MaxValueRule)this).getReferenceValue() != null;
24     @ assignable \nothing;
25     @ ensures \result == ((Integer)data.getValue()).intValue() >= ((
         MaxValueRule)this).getReferenceValue().intValue();
26     @ also
27     @ requires (!(data instanceof DoubleData) && !(data instanceof
         IntegerData));
28     @ assignable \nothing;
29     @ ensures \result == false;
30     @*/
31   private /*@ pure @*/ boolean validateMaxValue(Data data) {
32     if (data instanceof DoubleData) {
33       if (((MaxValueRule)this).getReferenceValue().compareTo(data.
           getValue()) >= 0) {
34         return true;
35       } else if (data instanceof IntegerData) {
36         return ((Integer)data.getValue()).intValue() >= ((
             MaxValueRule)this).getReferenceValue().intValue();
37       }
38     }
39     return false;
40   }
41   public boolean validate(Data data) {
42     /*@ assert (data instanceof Data); @*/
43     return this.validateMaxValue(data);
44   }
45 }
```

Figure 5.17: Excerpt of class `MaxValueRule` after the application *Replace Conditional with Polymorphism* refactoring.

```
1 public class AbstractNumberValidationRule extends
      AbstractValidationRule {
2
3   /*@ nullable @*/ Double referenceValue;
4
5 }
```

Figure 5.18: `AbstractNumberValidationRule` class source-code with attribute up.

Move attribute `referenceValue` to `AbstractNumberValidationRule`

The following steps allows us to move the attribute `referenceValue` to the class `AbstractNumberValidationRule`:

- We move up `referenceValue` from `MaxValidationRule` to the newly created superclass `AbstractNumberValidationRule`. First we apply **Law 18** from the left to right to turn it public.
- We apply law **Law 26** from the left to right to `referenceValue`.
- Finally, we move `referenceValue` applying **Law 27**, from the left to right.
- The previous step made the attributes named `referenceValue` in the others number-based validation rule classes shadows. We have to delete them applying **Law 30** (from the left to right). We omit details here as we said before.
- Apply **Law 21** from the right to left to `referenceValue` to make it default.

Figure 5.18 shows class `AbstractNumberValidationRule`.

Move common methods to `AbstractNumberValidationRule` class

Now we move the methods related to the attribute `referenceValue` (`setReferenceValue` and `getReferenceValue`) up to class `MaxValidationRule`. We use **Law 68** from the left to right to move up `setReferenceValue` and `getReferenceValue`. We cast references of **this** (that are uncast) by applying **Law 100** from the left to right. After this, we apply **Law 68** from the left to right to the methods `setReferenceValue` and `getReferenceValue`. Trivial casts are eliminated using **Law 100** from the right to left.

Now we move up the methods `setReferenceValue` and `getReferenceValue` of the subclasses `MinValueRule`, `MaxSizeRule`, `MinTolerancePercentageRule`, `MaxTolerancePercentageRule`, `MaxToleranceRule` and `MinToleranceRule`. These methods are moved up by application of **Law 70** from the left to right. After each application, we apply **Law 97** from the left to right to reduce if-else commands to a single command. By using **Law 100** from right to left we remove casts from **this** occurrences.

In order to exemplify the previous steps we show in Figure 5.19 the method `setReference` immediately after the application of **Law 70** from `MaxSizeRule` to `AbstractNumberValidationRule` class. Figure 5.20 shows the version of `setReferenceValue` method in `AbstractNumberValidationRule` class at the end of methods movements. See that there are no casts in **this** expressions.

```
1  public class AbstractNumberValidationRule {
2    /* ... */
3    /*@ requires (!(this instanceof MaxSizeRule));
4      @ assignable \not_specified;
5      @ ensures (!(this instanceof MaxSizeRule));
6      @ also
7      @ requires (this instanceof MaxSizeRule && true);
8      @ assignable \not_specified;
9      @ ensures (this instanceof MaxSizeRule && true);
10     @ also
11     @ requires ((this instanceof MaxSizeRule));
12     @ assignable \not_specified;
13     @ ensures ((this  instanceof MaxSizeRule));
14     @*/
15   public void setReferenceValue(Double referenceValue) {
16     if (!(this instanceof MaxSizeRule)) {
17       this.referenceValue = referenceValue;
18     } else {
19       this.referenceValue = referenceValue;
20     }
21   }
22   /* ... */
23 }
```

Figure 5.19: `setReferenceValue` method immediately after moved up from `MaxSizeRule` to `AbstractNumberValidationRule` class.

```
public class AbstractNumberValidationRule extends
    AbstractValidationRule {
2  /* ... */
3  public void setReferenceValue(Double referenceValue) {
4    this.referenceValue = referenceValue;
5  }
6  /* ... */
}
```

Figure 5.20: The final version of `setReferenceValue` method in `AbstractNumberValidationRule` class.

Move duplicated and weaker invariants to `AbstractNumberValidationRule` class

As we discussed in Section 5.2.1, common and weaker invariants should be moved up to the more abstract class in the hierarchy, in this case `AbstractNumberValidationRule` class.

As can be seen in Appendix C.1 we have similar invariants – **invariant this**.`getReferenceValue()`!= **null** – in each one of classes `MinValueRule`, `MaxSizeRule`, `MaxToleranceRule`, `MinToleranceRule`, `MinTolerancePercentageRule` and `MaxTolerancePercentageRule`. To move each one of the invariants we need apply **Law 5** (from the left to right).

Before apply this law we need to prepare all invariants to satisfy the law conditions. Hence, we introduce trivial casts in **this** expressions in all invariants applying **Law 100** from the left to right. Now it is safe to apply **Law 5** from the left to right to all invariants. We remove trivial casts using **Law 108** from the left to right. An excerpt of `Data` class highlighting the recently moved invariants is presented in Figure 5.21. Propositional calculus is used to simplify the invariant predicates.

The final invariant of data is **invariant this**.getReferenceValue()!= **null**. At this point we finish the *Extract Superclass* refactoring obtaining the new superclass `AbstractNumberValidationRule` . Final source code of the number-base validation rules is presented in Appendix C.4.

```
1  public class AbstractNumberValidationRule {
2  /* ... */
3  //@ invariant this instanceof MaxSizeRule ==> ((MaxSizeRule)this).
       getReferenceValue() != null;
4  //@ invariant this instanceof MaxTolerancePercentageRule ==> ((
       MaxTolerancePercentageRule)this).getReferenceValue() != null;
5  //@ invariant this instanceof MaxValueRule ==> ((MaxValueRule)this).
       getReferenceValue() != null;
6  //@ invariant this instanceof MinTolerancePercentageRule ==> ((
       MinTolerancePercentageRule)this).getReferenceValue() != null;
7  //@ invariant this instanceof MinToleranceRule ==> ((MinToleranceRule
       )this).getReferenceValue() != null;
8  //@ invariant this instanceof MaxToleranceRule ==> ((MaxToleranceRule
       )this).getReferenceValue() != null;
9  //@ invariant this instanceof MinValueRule ==> ((MinValueRule)this).
       getReferenceValue() != null;
10 /* ... */
11 }
```

Figure 5.21: Invariants moved up to `AbstractNumberValidationRules` class immediately before reduction.

5.2.4 Evolving Our Validation Rules API: Creating a Fresh Validation Rule Class

In this section we discuss software evolution. To attend a new requirement, it is necessary to create a new validation rule. In fact, we must create a new validation rule to check if a determined data connected to a meta data is not null, in other words to ensure that a data is a mandatory value. Thus in this section we show how our laws can help to evolve code safely. The following steps are necessary to accomplish the creation of the new validation rule.

- Create a new fresh class `NotNullRule` using **Law 1** (from the left to right) with no problems.
- Create a new method with no specifications with signature: **public boolean** `validate` (`Data data`) with our own implementation using **Law 75** from the right to left since `NotNullRule` is new and does not have any superclass or subclass.
- Make `NotNullRule` inherits `AbstractValidationRule` by applying **Law 3** from the left to right directly since there are no attributes in `NotNullRule` class, the recently created method `validate` does not have specification cases as well as the remaining (and also unused) `validate` method of `AbstractValidationRule`. Note that the default constructor of `NotNullRule` automatically introduced by the Java compiler calls the default constructor of `AbstractValidationRule`, what satisfies the inherited invariant from `AbstractValidationRule`.

- To complete the process we insert a specification case for the new `validate` method. To do so we apply **Law 33** from the left to right. Remember that the method is new and `NotNullRule` class too, thus the conditions (1) and (2) are satisfied. After it we apply **Law 31** from the left to right weakening the identity pre-condition from **false** to **true**. To finish we strengthen the identity postcondition **true** to `data` != **null** && `data.getValue()`!= **null** by applying **Law 32**. The conditions of these two last laws are satisfied since `NotNullRule` has no subclasses. And its over. We have a new validation rule class.

Figure 5.22 present the final version of `NotNullRule` class.

```
1  public class NotNullRule extends AbstractValidationRule {
2
3    /*@ also
4      @ requires true;
5      @ assignable \not_specified;
6      @ ensures data != null && data.getValue() != null;
7      @*/
8    public boolean validate(Data data) {
9      return data != null && data.getValue() != null;
10   }
11 }
```

Figure 5.22: `NotNullRule` class.

5.2.5 Final actions and considerations

Some actions can be executed to finish our macro transformations. The unused `validate` method of `AbstractValidationRule` should become abstract or even deleted. The classes `AbstractValidationRule` and `AbstractNumberValidationRule` may be become abstracts.

We apply **Law 2** from the left to right to the class `AbstractNumberValidationRule`. There is no instantiations of this class in our program satisfying the conditions. We also apply **Law 2** from the left to right to `AbstractValidationRule` class with no problems since only subclasses of this class is instantiated in the program.

And to finish we use **Law 86** from the left to right to make the method `validate` (of `AbstractValidationRule`) abstract. All subclasses of `AbstractValidationRule` implement the method `validate`. There are no instantiations of `AbstractValidationRule` so there are no dynamic calls to `validate` via objects of type `AbstractValidationRule` satisfying the conditions required to apply **Law 86**.

Appendix C present the main parts of the source-code of our program involved in each macro step detailed in this chapter as well as the original source-code. Figure 5.23 presents the diagram of the final version of our Meta Data API.

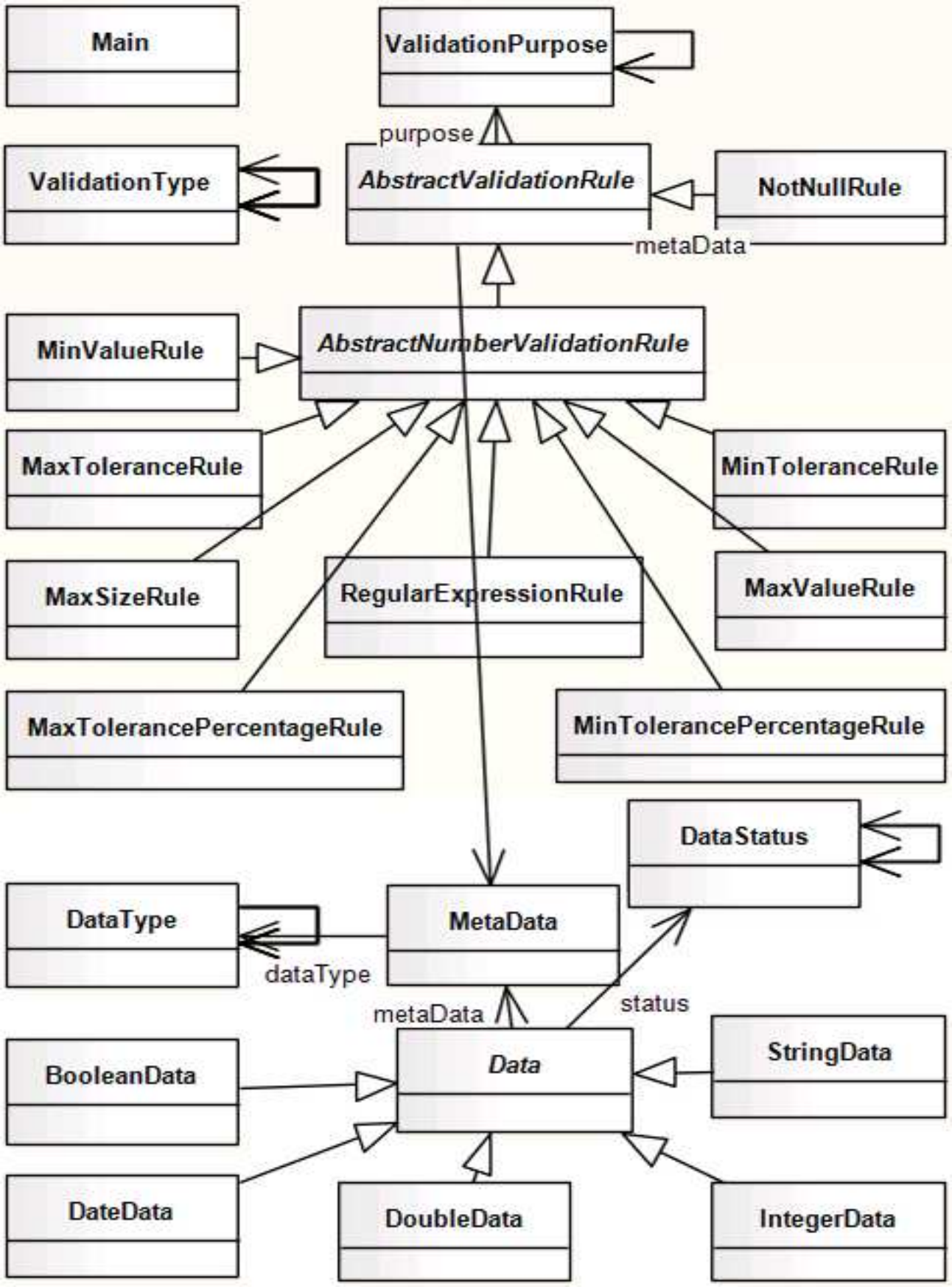


Figure 5.23: Final object diagram from Meta Data API

Chapter 6

Conclusions

Programming laws serve as guidelines to informal programming practices and establish a basis for formal and rigorous program development. Object-oriented programming laws were initially proposed by Borba, Sampaio and Cornélio [9] for ROOL [13]. They propose laws for classes and commands of ROOL and they define a normal form for object-oriented programs written in ROOL along with a reduction strategy. Duarte [26] reviewed and extended the programming laws written for ROOL for the Java programming and created other laws for language features that are not present in ROOL.

In view of the necessity to perform behavior-preserving changes in program source-code, it is fundamental to execute the changes in a disciplined way. Programming laws are a means to achieve such purpose.

In this work, we proposed a rigorous approach for refactoring annotated Java programs, based on successive applications of laws (primitive transformations) for object-oriented programming in the presence of a behavioral interface specification language. Our laws are behavior-preserving since they ensure that the program continues to fulfill its specification described by means of annotations written with the Java Modeling Language (JML). Our laws treat source-code transformation considering the impacts caused by its internal specifications. Some of our laws are inspired on programming laws from previous work [7, 23] (that were proved to be sound to the language ROOL [12]), and specially on the laws from Duarte's work [26]. Other laws of our catalog, specially those that transform JML specifications are completely new to our knowledge.

Differently from laws that deal only with constructs of object-oriented programming language, the presence of a behavioral interface specification language (BISL) requires that we be aware of many issues related to the semantics of a BISL language, like JML:

- The visibility of specifications imposes some drawback to change the Java visibility of attributes and methods. For example, in a lightweight method specification, the specification visibility is assumed to be the same as the method visibility. If we try to change the visibility of a method from private to public we need to check if the elements referred to in the specification of the method have public specification visibility, since the specification visibility of a specification must be at least as permissive as the visibility of the elements it refers to;

- We need to preserve invariant (history constraint and initially clauses) of a subclass when introducing calls to **super**. Introducing a method redefinition calling a **super** method is not trivial because the **super** method can break the invariant, history constraint and initially clauses predicates;

- Changing a parameter type (or a return type) to a supertype requires introducing casts in occurrences of the parameter in specifications of the method that contains the parameter in its signature;

- To eliminate a pure method we have to verify if it is used in any specification in a program;

- In JML, any declaration (except for local variables) whose type is a reference type is implicitly declared to be not null, except when one adorns the declaration with a nullable annotation. Thus, by default, JML always verifies if a not nullable attribute is null in all visible states of the class that declares it. When we move an attribute to a superclass, this is not aware about the newly moved attribute and, therefore, this action can cause an undesirable behavior. Hence, to move an attribute whose type is a reference type, we need before to adorn the attribute with the keyword **nullable**;

- Invariants, and initially clauses prevent us to eliminate default constructors. Recall that Java creates a default constructor in any class that does not declare any constructor. If we eliminate explicit default constructors of a class in this situation, Java will create a default constructor that will possibly not meet the invariant and initially clauses leading to exceptions.

We started our study on the relative completeness of our set of laws by a normal form reduction strategy. We have applied our set of laws reducing a JML-specified Java program to a normal form inspired in the one presented by Duarte [26], which follows the main steps of the normal form reduction strategy of ROOL. A program in the normal form we defined in Chapter 4, preserves the class hierarchy, but all attributes and methods that are non-recursive and with no mutually exclusive return points are located in the class `_Object`. Also, invariants, initially clauses and constraints are placed in the class Object. Specification cases of non-eliminated methods are written as JML assert statements. We still need to evolve our strategy to eliminate all JML elements that appear in the normal form we have now.

The laws of our catalog were also used to show how a JML-specified version of a core module from a Manufacturing Execution System, get refactored from successive applications of primitive transformations expressed by means of our laws. Although our work does not provide a way to transform programs automatically yet, it provides a reliable, systematic and extensible alternative to address refactorings.

The example presented in the Chapter 5 shows a real situation in which our laws can be applied to improve code structure and refactoring code to accommodate new implementations. Although our application example is very expressive, we do not provide guidelines to support generic situations where we can apply the same and additional refactorings. We intend to fulfill this gap using our catalog of programming laws to elaborate guidelines to execute the refactorings we applied in this work and other refactorings described by Fowler [31]. Also, we intend to create more elaborated examples to validate and help to extend our catalog of laws.

Finally, we can summarize the following contributions resulted from this work: creation of a catalog of programming laws to deal with JML specifications and JML-specified Java programs; proposition of a normal form reduction strategy for JML-specified Java programs; and the presentation (step by step) of a case study – using a real program – showing how refactorings can be applied using our programming laws.

6.1 Related Work

Object-oriented programming laws were initially proposed by Borba, Cornélio and Sampaio [9] for ROOL [13], which was designed to allow reasoning about object-oriented programs and specification, mixing both constructs in the sytle of Morgan's refinement calculus [53]. They propose laws for classes and commands of ROOL and they define a normal form for object-oriented programs written in ROOL along with a reduction strategy. Also, they demonstrate that the set of laws is complete with respect to this normal form. These laws do not consider specifications and were designed to ROOL while our laws uses a language used in the industry (Java) as target language.

Cornélio [23] proves the laws with respect the copy semantics of ROOL [13] and formally justifies, by using programming laws and data refinement, refactoring practices documented by Fowler [31]. Silva, Sampaio, and Liu considers object-oriented programming laws in a language with a reference semantics [62], applying such laws to code refactoring. Duarte [26] adapts the programming laws initially written for ROOL for the Java programming and proposes other laws for language features that are not present in ROOL.

Although Duarte [26] developed programming laws for Java, his work do not take in consideration JML specifications or any specification languages. The focus of his work was using programming laws to perform program parallelization.

Garrido et al. [33] uses Maude [19] to formalize Java and prove some transformations, i.e. refactorings. Their work do not use programming laws to perform the program transformations. Although their work guarantees that the transformations are behavior-preserving in respect to a Java semantic defined by them. It is possible to implement our programming laws as rules in Maude (as Lira implemented ROOL's programming laws in Maude [45]), therefore we could use this rewriting system to implement transformations.

Bannwart [3] proposes a technique to apply refactorings to a program that preserves the external behavior of the program if the transformed program fulfill the refactoring's conditions. Bannwart adds the refactoring conditions to the code in the form of assertions that can be verified using static verifications tools or used to generate unit tests or even can be used for runtime assertion checking. Bannwart uses a small sequential class-based programming language but argues that his technique can be used to handle realistic languages. Our work treats the refactoring activity as small transformations based on side-conditions for application and consider a program as the source-code (Java) itself and its specifications (JML), while Bannwart uses a simple language and do not consider specifications.

Goldstein developed an eclipse plugin [34] to manipulate DbC specifications and to perform run time assertion check. In addition, he implemented a set of refactorings that takes in consideration both Java code and specifications. In [34] is explained how the plugin acts in the refactorings *Extract Superclass* and *Add Inheritance* in the presence of DbC specifications. However, Goldstein does not provide any formalism or systematization, to their methodologies, and focus on its own specification languages in preference to JML. He presented some techniques used by a plugin to refactor specified Java code. The plugin uses a theorem prover to verify the relation between the assertions of the specifications. Our approach to execute refactorings in specified Java code treats refactorings as behavior-preserving transformations described by the application of programming laws.

6.2 Future Work

In this work, we have considered laws that address only a subset of the JML's Level 0 constructs as well as some Level 1 constructs, specially for lightweight specifications. Nevertheless, our preliminary focus is to cover most of the JML constructs that form the core notation used in the design by contract methodology. As future work, we intend to describe laws to support other JML clauses like **initially** , **constraint**, **represents**, and model fields.

Concerning Java, we can elaborate new laws to accept features not addressed by the laws defined in this work. There are some largely used Java features that we have to take into consideration to make our approach less restrict. For example, we can consider exceptions and interfaces and create laws to deal with these features.

In [27], we started to work in proofs for our laws. The JML semantics as well as the Java semantics are not completely defined restricting the proof work. However, we intend to prove the JML parts of our laws using the semantics of JML defined by Leavens [40]. Concerning the Java parts, we plan to work in the same direction of the works of Silva [62], and Massoni [48].

The normal form reduction strategy we proposed in Chapter 4, is a result of the review and adaptation of the strategies used for Java and ROOL [9] programs. As a future work, we can enhance our reduction strategy elaborating new laws to eliminate invariants, history constraints and initially clauses, for example. We plan to create laws to distribute invariants and history constraints in the methods of the classes they are declared. In this direction it is possible to create laws to eliminate initially clauses by copying their predicates in the constructors they affect. As a result we can eliminate methods and transform their specifications in JML assertions.

Another idea on enhance our normal form reduction strategy is to transform all JML specification of a program in Java code representing run time assertion checks (RAC) code. This is the way Krakatoa [11] and other JML tools like, JAJML [36] and *jmlc* [1] works. We can transform (after applying our strategy) all remaining specifications in Java code representing the behavior of the specifications.

Finally, a challenging and complementary work is to build a tool to support annotated Java program transformations based on our set of laws. We have many works in this direction like the work of Garrido [33] that uses the rewriting system Maude [19] to execute behavior-preserving transformations of Java programs. We can extend the Garrido's work adding the JML grammar and tokens and after write code to deal with transformations based on our laws.

Another work is the JAJML project [36] which uses the JastAdd [37] – a extensible Java compiler framework – to build an extensible runtime assertion checker for JML. With JAJML is possible to parser a JML-specified Java program creating and inserting JML specifications and Java code in a program. We can use the JAJML infrastructure to execute program transformations based on our laws.

The use of JML6 [58] is another possible future direction of work. JML6 is a clean Eclipse [30] plug-in that provides a JML intermediate representation and supporting infrastructure to unify JML front-ends and backends. As JML6 is an official work of the JML community, joining in the JML6 team to design and develop a tool to mechanize the application of our laws would be interesting since in this way we can unify efforts and work in a unique JML parser/compiler platform.

References

[1] Java modeling language (jml) projects website, http://sourceforge.net/projects/jmlspecs/., 2009.

[2] Mesa international web site., 2009.

[3] Fabian Bannwart and Peter Müller. Changing programs correctly: Refactoring with specifications. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 492–507. Springer, 2006.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1 edition, 1999.

[5] Yves Bertot. Coq in a hurry. Nov 2008.

[6] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[7] P. Borba et al. Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.*, 52(1-3):53–100, 2004.

[8] P. Borba et al. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, 2004.

[9] P. Borba, A. Sampaio, and M. Cornélio. A Refinement Algebra for Object-oriented Programming. In Luca Cardelli, editor, *European Conference on Object-oriented Programming, ECOOP'2003*, volume 2743 of *Lecture Notes in Computer Science*, pages 257–282, Darmstadt, Germany, July 2003. Springer-Verlag.

[10] L. Burdy, Y. Cheon, D., M. D. Ernst, J. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[11] C. Paulin-Mohring C. Marché and X. Urbain. The krakatoa tool for certificationof java/javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[12] A. Cavalcanti and D. A. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. Softw. Eng.*, 26(8):713–728, 2000.

[13] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.

[14] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science Iowa State University, April 2003.

[15] Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (jml). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas*, pages 322–328. CSREA Press, 2002.

[16] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002, volume 2374 of LNCS*, pages 231–255. Springer, 2002.

[17] Y. Cheon and C. E. Rubio-Medrano. Random test data generation for java classes annotated with jml specifications. In *Software Engineering Research and Practice*, pages 385–391, 2007.

[18] J. Chrzaszcz and A. Schubert. Esc/java2 as a tool to ensure security in the source code of java applications. In *Software Engineering Techniques: Design for Quality*, volume Volume 227/2007 of *IFIP International Federation for Information Processing*, pages 337–348. Springer Boston, 2007.

[19] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[20] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[21] D. R. Cok and J. R. Kiniry. Esc/java2: Uniting esc/java and jml - progress and issues in building and using esc/java2, including a case study involving the use of the tool to verify portions of an internet voting tally system. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, Lecture Notes in Computer Science, pages 108–128. SpringerVerlag, 2004.

[22] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[23] M. Cornélio. *Refactoring as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco (UFPE), 2004.

[24] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[25] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[26] R. Duarte. Parallelizing java programs using transformation laws. Master's thesis, Universidade Federal de Pernambuco (UFPE), 2008.

[27] G. Falconieri Freitas et al. Object-oriented programming laws for annotated java programs. *Electronic Proceedings in Theoretical Computer Science*, 2009. To appear in Proceedings of the Tenth International Workshop on Rule-Based Programming.

[28] G. T. Leavens et al. *JML Reference Manual*, July 2008.

[29] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[30] The Eclipse Foundation. Eclipse integrated development environment, 2008. http://www.eclipse.org.

[31] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[32] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In Andreas Winter, Rudolf Ferenc, and Jens Knodel, editors, *CSMR*, pages 255–258. IEEE, 2009.

[33] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 165–174, Washington, DC, USA, 2006. IEEE Computer Society.

[34] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz. Refactoring with contracts. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 53–64, Washington, DC, USA, 2006. IEEE Computer Society.

[35] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The*. Addison-Wesley Professional, third edition, 2005.

[36] Ghaith Haddad and Gary T. Leavens. Extensible dynamic analysis for jml: A case study with loop annotations. Technical Report CS-TR-08-05, School of Electrical Engineering and Computer Science - University of Central Florida, 2008.

[37] Görel Hedin and Eva Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[38] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[39] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, U. Hensel, and H. Tews. Reasoning about java classes: preliminary report. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 329–340, New York, NY, USA, 1998. ACM.

[40] G. T. Leavens. Jml's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34. Springer, 2006.

[41] G. T. Leavens. Several references to papers on jml can be found on the jml project website, http://www.cs.iastate.edu/ leavens/jml/papers.shtml., 2009.

[42] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[43] G. T. Leavens and Y. Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005.

[44] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University, July 2006.

[45] B. Lira. Automação de regras para a programação orientada a objetos. Master's thesis, Universidade Federal de Pernambuco (UFPE), 2002.

[46] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[47] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 31–34, New York, NY, USA, 2007. ACM.

[48] T. Massoni. *A Model-driven Approach to Formal Refactoring*. PhD thesis, Universidade Federal de Pernambuco (UFPE), 2008.

[49] T. Massoni, R. Gheyi, and P. Borba. Formal model-driven program refactoring. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2008.

[50] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[51] B. Meyer. Applying design by contract. *IEEE Computer*, 25:40–51, 1992.

[52] C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[53] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[54] P. H. Nidditch. *Propositional calculus, by P.H. Nidditch*. Routledge & K. Paul; Dover Publications London, New York,, 1965.

[55] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[56] L. Powers and M. Snell. *Microsoft Visual Studio 2005: Unleashed*. Sams, Indianapolis, IN, 2007.

[57] A. D. Raghavan and G. T. Leavens. Desugaring jml method specifications. Technical Report 00-03a, 2000.

[58] Robby and Patrice Chalin. Preliminary design of a unified jml representation and software infrastructure. Technical report, SAnToS Laboratory, Department of Computing and Information Sciences, Kansas State University, April 2009.

[59] Clyde D. Ruby. Safely creating correct subclasses without seeing superclass code. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 155–156, New York, NY, USA, 2000. ACM.

[60] Max S., T. Ekman, M. Verbaere, and R. Ettinger. Refactoring bugs, 2008. `http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports`. Last access in 04/04/2009.

[61] S. Seres. *The Algebra of Logic Programming*. PhD thesis, Oxfor University Computing Laboratory, 2001.

[62] L. Silva, A. Sampaio, and Z. Liu. Laws of object-orientation with reference semantics. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 217–226, Washington, DC, USA, 2008. IEEE Computer Society.

[63] Joachim van den Berg, Bart Jacobs, and Erik Poll. Formal specification and verification of javacard's application identifier class. In Isabelle Attali and Thomas P. Jensen, editors, *Java Card Workshop*, volume 2041 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2000.

[64] R. R. Zagidullin and E. B. Frolov. Control of manufacturing production by means of mes systems. *Russian Engineering Research*, 28(2):166–168, February 2008.

APPENDIX A

Laws

We present in this appendix our complete catalog of laws. The laws are divided in sections according to the elements they focus. The laws that start with an asterisk-sign are laws defined by Duarte [26].

A.1 Classes

Law 1. ⟨*class elimination*⟩
$cds\ \ cd_1\ \ Main\ =\ cds\ \ Main$

provided

JML:

(→) The class declared in cd_1 is not referred in any specification declared in cds or $Main$.

Java:

(→) The class declared in cd_1 is not referred in cds or $Main$.

(←) (1) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; (2) The superclass appearing in cd_1 is either **Object** or declared in cds.

□

Law 2. ⟨*make class abstract*⟩

class C **extends** D {
ads
$cnds$
mds
}
	$=_{cds,Main}$	**abstract class** C
		extends D {
		ads
		mds
		}

provided

JML:

(→) '*new C*' does not occur inside specifications of cds, $Main$, $cnds$ nor mds.

Java:

(→) '*new C*' does not occur in cds, $Main$, $cnds$ nor mds.

(←) Every method m of mds is concrete.

□

Law 3. ⟨*change superclass: from* ***Object*** *to another class*⟩

class C **extends Object** {
$@invs$
$@cons$
$@inis$
 	
ads
$cnds$
mds
}
$cds,\ Main$
	=	**class** C **extends** D {
		$@invs$
		$@cons$
		$@inis$
		
		ads
		$cnds$
		mds
		}
		$cds',\ Main$

□

where

$cds' \mathrel{\widehat{=}} cds[//@\ \textbf{also}\ fspec(m)/fspec(m)]$, for every method m in mds that is a redefinition of a method introduced in some class E such that $D \leq E$.

provided

JML:

(→) (1) $@invs \Rightarrow finv(D)$; (2) $@cons \Rightarrow fcons(D)$; (3) $@inis \Rightarrow finit(D)$; (4) For any method m in mds that redefines a method m declared in D or in any class E such that D $\leq E$, $fpre(E.m[pds]) \Rightarrow fpre(C.m[pds])$ e $\backslash\textbf{old}(fpre(E.m[pds])) \Rightarrow (fpos(C.m[pds]) \Rightarrow fpos(E.m[pds]))$.

(←) (1) C or any of its subclasses in cds is not used in type casts or tests involving any expression of type D or of any supertype of D in specifications; (2) **this**.a does not appear in specifications of C, nor in specifications of C's subclasses, for any attribute a of D or of any superclass of it with specification visibility default, protected or public; (3) $le.a$, for any $le : C$, does not occur in specifications in cds or $Main$, for any a of D or of any superclass of it with specification visibility

default, protected or public; (4) There is no method call $E.m$ inside specifications of cds, for any pure method m, such that $E \leq C$ and m is declared in D or in any of its superclasses, but is not redefined in mds; (5) **super** does not appear in any specification of C.

Java:

(→) All attributes in ads and in subclasses of C are distinct from those declared in D and in superclasses of D.

(←) (1) C or any of its subclasses in cds is not used in type casts or tests involving any expression of type D or of any supertype of D;

(2) There are no assignments of the form $le = exp$, for any le whose declared type is D or any superclass of D and any exp whose type is C or any subclass of C;

(3) Expressions of type C or of any subclass of C are not used as value arguments in method/constructor calls with a corresponding formal parameter whose type is D or any superclass of D;

(4) Expressions whose declared type is D or any of its superclasses are not returned as a method result in calls with an expected result whose declared type is C or any subclass of C;

(5) **this**.a does not appear in C, nor in any subclass of C, for any public or protected attribute a of D or of any of its superclasses;

(6) $le.a$, for any $le : C$, does not appear in cds or c for public or protected attribute a of D or of any of its superclasses;

(7) There is no $E.m$, for any method m such that, $E \leq C$ and m is declared in D or in any of its superclasses, but is not redefined in mds.

(8) **super** does not appear in any method in mds.

Law 4. ⟨*change superclass: from an empty class to immediate superclass*⟩

class *B* **extends** *A* {
}
class *C* **extends** *B* {
 	
  ads
  cnds
  mds
}
cds, *Main*
	=	**class** *B* **extends** *A* {
		}
		class *C* **extends** *A* {
		
		  ads
		  cnds
		  mds
		}
		cds′, *Main*

provided

Java:

(→) (1) *C* or any of its subclasses in *cds* is not used in type casts involving expressions of type *B*;

(2) There are no assignments of the form *le* = *exp*, for any *le* whose declared type is *B* or any of its superclasses and the type of *exp* is *C* or any subclass of *C*;

(3) Expressions of type *C* or of any subclass of *C* are not used as value arguments in calls with a corresponding formal value parameter whose type is *B*;

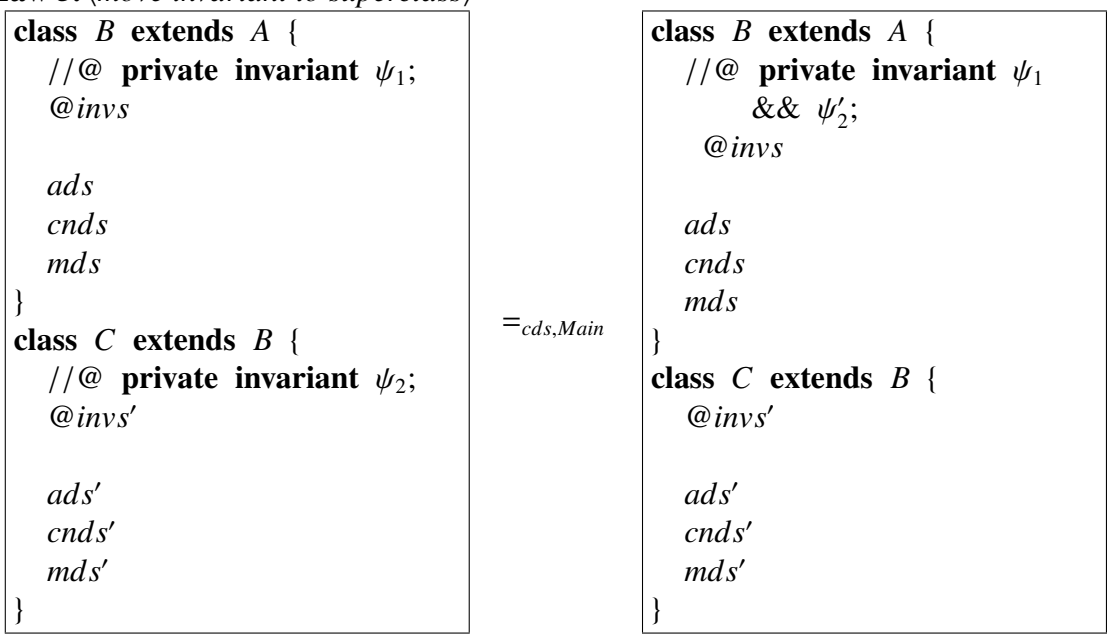
(4) Expressions whose declared type is *B* are not result arguments in calls with a corresponding formal result parameter whose declared type is *C* or any subclass of *C*;

(5) Casts to class *B* are not applied to attributes, variables or parameters of type *A* to which are assigned expressions of type *C*.

□

A.2 Invariants

Law 5. ⟨*move invariant to superclass*⟩



where

$\psi_2' \mathrel{\widehat{=}}$ **this instanceof** C ==> ψ_2

provided

JML:

(↔) **super** does not appear in ψ_2.

(→) ψ_2 does not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

□

Law 6. ⟨*change invariant visibility: from default to private*⟩

class C **extends** D {
//@ **invariant** ψ_1;
@$invs$
 	
ads
$cnds$
mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		//@ **private invariant** ψ_1;
		@$invs$
		
		ads
		$cnds$
		mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has non-private specification visibility;

□

Law 7. ⟨*change invariant visibility: from public to private*⟩

class C **extends** D {
//@ **public invariant** ψ_1;
@$invs$
 	
ads
$cnds$
mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		//@ **private invariant** ψ_1;
		@$invs$
		
		ads
		$cnds$
		mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has public specification visibility;

□

Law 8. ⟨*change invariant visibility: from protected to private*⟩

class C **extends** D {
  //@ **protected invariant** ψ_1;
  @*invs*
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  //@ **private invariant** ψ_1;
		  @*invs*
		
		  ads
		  cnds
		  mds
		}

provided

JML:

(←) Every attribute, pure method and model field that occurs in ψ_1 has non-public specification visibility;

□

Law 9. ⟨*collapse invariants*⟩

class C **extends** D {
  //@ **private** *invariant* ψ_1;
  //@ **private** *invariant* ψ_2;
  ...
  //@ **private** *invariant* ψ_n;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  //@ **private** *invariant* ψ_1
		  && ψ_2 && ... && ψ_n;
		  ads
		  cnds
		  mds
		}

□

Law 10. ⟨*delete duplicated invariant from subclass*⟩

```
class B extends A {
   //@ private invariant ψ;
   @invs

   ads
   cnds
   mds
}
class C extends B {
   //@ private invariant ψ;
   @invs′

   ads′
   cnds′
   mds′
}
```

$=_{cds,Main}$

```
class B extends A {
   //@ private invariant ψ;
   @invs

   ads
   cnds
   mds
}
class C extends B {
   @invs′

   ads′
   cnds′
   mds′
}
```

□

Law 11. ⟨*insert default invariant*⟩

```
class C extends D {

   ads
   cnds
   mds
}
```

$=_{cds,Main}$

```
class C extends D {
   //@ private invariant true;

   ads
   cnds
   mds
}
```

□

A.3 Attributes

Law 12. ⟨*change specification visibility of default attribute: from default to public*⟩

class *C* **extends** *D* {
  T a;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  /*@ **spec_public** @*/ *T a*;
		
		  ads
		  cnds
		  mds
		}

provided

JML:

(←) *B.a*, for any $B \leq C$, occurs only inside specifications with default or private specification visibility.

□

Law 13. ⟨*change specification visibility of default attribute: from protected to public*⟩

class *C* **extends** *D* {
  /*@ **spec_protected** @*/ *T a*;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  /*@ **spec_public** @*/ *T a*;
		
		  ads
		  cnds
		  mds
		}

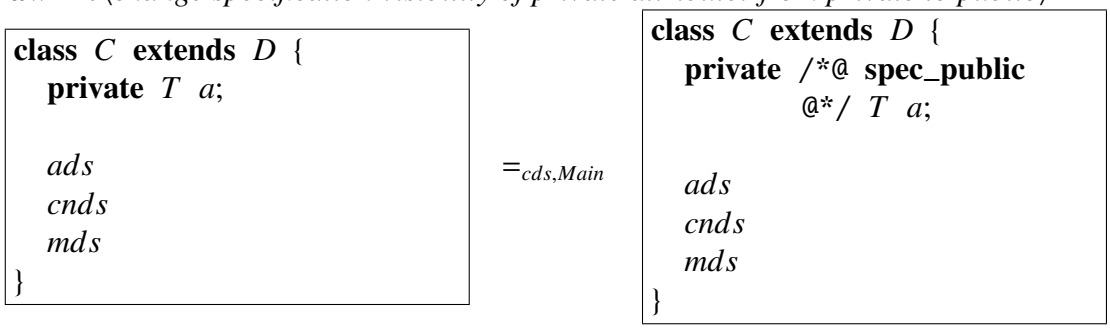
provided

JML:

(←) *B.a*, for any $B \leq C$, occurs only inside specifications with non-public specification visibility.

□

Law 14. ⟨*change specification visibility of private attribute: from private to public*⟩



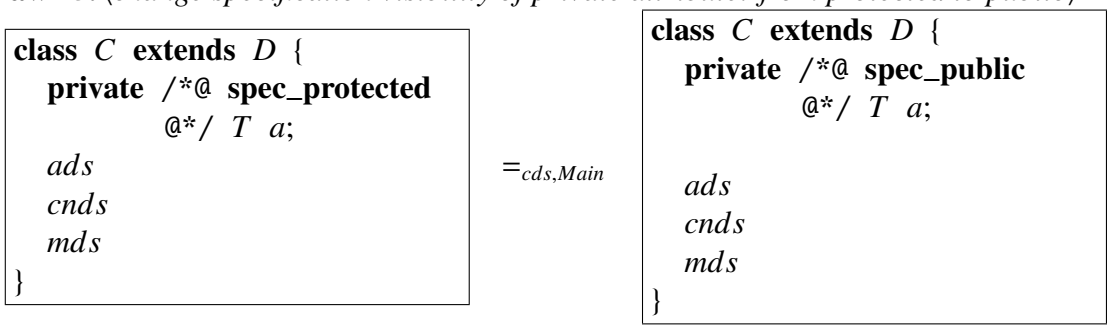
provided

JML:

(←) a, occurs only inside specifications with private specification visibility.

□

Law 15. ⟨*change specification visibility of private attribute: from protected to public*⟩



provided

JML:

(←) a, occurs only inside specifications with non-public specification visibility.

□

Law 16. ⟨*change specification visibility of protected attribute: from protected to public*⟩

```
class C extends D {
  protected T a;
  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  protected /*@ spec_public
            @*/ T a;

  ads
  cnds
  mds
}
```

provided

JML:

(←) $B.a$, for any $B \leq C$, occurs only inside specifications with non-public specification visibility.

□

Law 17. ⟨*change specification visibility of private attribute: from private to public*⟩

```
class C extends D {
  private T a;

  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  private /*@ spec_public
          @*/ T a;

  ads
  cnds
  mds
}
```

provided

JML:

(←) a, occurs only inside specifications with private specification visibility.

□

Law 18. ⟨*change attribute visibility: from private to public*⟩

class C **extends** D {
  private T a;
 	
  ads
  $cnds$
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  public T a;
		
		  ads
		  $cnds$
		  mds
		}

provided

JML:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in any specification of cds or $Main$; (2) $C.a$, occurs only inside specifications of C with private specification visibility.

Java:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.a$ occurs only in C's body.

□

Law 19. ⟨*change attribute visibility: from default to public*⟩

class C **extends** D {
  T a;
 	
  ads
  $cnds$
  mds
}
	$=_{cds,Main}$	**class** C **extends** D {
		  public T a;
		
		  ads
		  $cnds$
		  mds
		}

provided

JML:

(←) $C.a$ occurs only inside specifications of C and cds with private, or default specification visibility.

□

Law 20. ⟨*change attribute visibility: from protected to public*⟩

class *C* **extends** *D* {
  protected *T a*;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  public *T a*;
		
		  ads
		  cnds
		  mds
		}

provided

JML:

(←) *B.a*, for any $B \leq C$, occurs only inside specifications of *C* and *cds* with non-public specification visibility.

□

Law 21. ⟨*change spec public attribute visibility: from default to public*⟩

class *C* **extends** *D* {
  /*@ **spec_public** @*/ *T a*;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  public *T a*;
		
		  ads
		  cnds
		  mds
		}

□

Law 22. ⟨*change spec protected attribute visibility: from default to protected*⟩

class *C* **extends** *D* {
  /*@ **spec_protected** @*/ *T a*;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  protected *T a*;
		
		  ads
		  cnds
		  mds
		}

□

Law 23. ⟨*change spec public attribute visibility: from protected to public*⟩

class *C* **extends** *D* {
  protected
  /*@ **spec_public** @*/ *T a*;
 	
  ads
  cnds
  mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		  public *T a*;
		
		  ads
		  cnds
		  mds
		}

□

Law 24. ⟨*change spec public attribute visibility: from private to public*⟩

```
class C extends D {
  private
    /*@ spec_public @*/ T a;

  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  public T a;

  ads
  cnds
  mds
}
```

provided

Java:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.a$ occurs only in C's body.

□

Law 25. ⟨*change spec protected attribute visibility: from private to protected*⟩

```
class C extends D {
  private
  /*@ spec_protected @*/ T a;

  ads
  cnds
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  protected T a;

  ads
  cnds
  mds
}
```

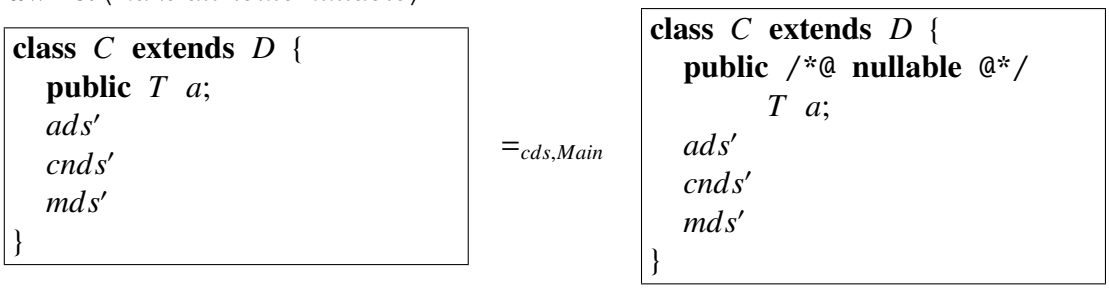
provided

Java:

(←) (1) $B.a$, for any $B \leq C$ excepts of strict type C, does not occur in cds or $Main$; (2) $C.a$ occurs only in C's body.

□

Law 26. ⟨*make attribute nullable*⟩



provided

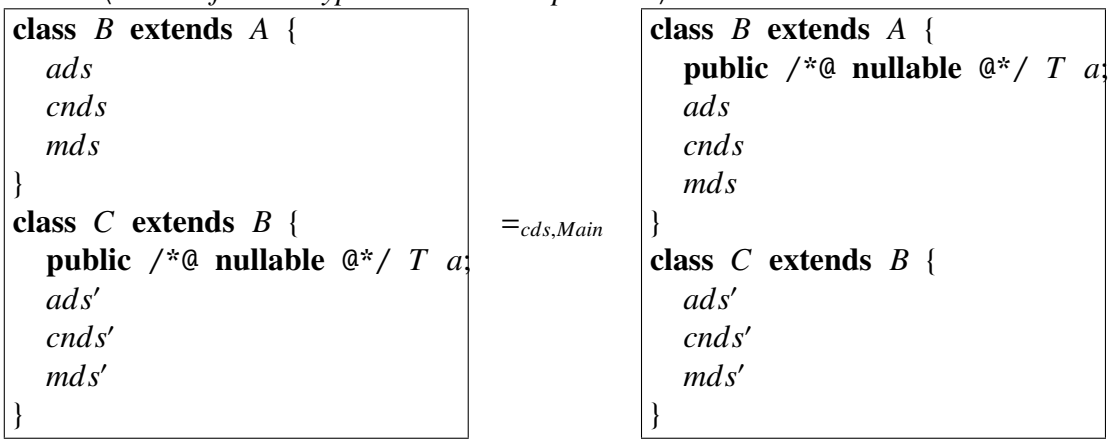
Java:

(↔) *T* is not a primitive type.

(←) (1) The initial value of *a* is different from **null**; (2) **null** is not assigned to *a* (directly or indirectly).

□

Law 27. ⟨*move reference type attribute to superclass*⟩



provided

JML:

(←) *D.a* does not occur inside specifications in *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*, for any $D \leq B$ and $D \not\leq C$.

Java:

(↔) *T* is not a primitive type.

(→) (1) *a* is not declared in *ads*; (2) The attribute name a is not declared by the subclasses of B in *cds*.

(←) *D.a* does not occur in *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*, for any $D \leq B$ e $D \not\leq C$.

□

Law 28. ⟨*move primitive type attribute to superclass*⟩

class B **extends** A {
ads
$cnds$
mds
}
class C **extends** B {
public T a;
ads'
$cnds'$
mds'
}
	$=_{cds,Main}$	**class** B **extends** A {
		public T a;
		ads
		$cnds$
		mds
		}
		class C **extends** B {
		ads'
		$cnds'$
		mds'
		}

provided

JML:

(←) $D.a$, for any $D \leq B$ and $D \not\leq C$ does not occur inside specifications of cds, $Main$, $cnds$, $cnds'$, mds nor mds'.

Java:

(↔) T a primitive type.

(→) (1) a is not declared in ads; (2) The attribute name a is not declared by the subclasses of B in cds.

(←) $D.a$, for any $D \leq B$ e $D \not\leq C$ does not occur in cds, $Main$, $cnds$, $cnds'$, mds nor mds'.

□

Law 29. ⟨*change attribute type*⟩

class B **extends** A {
  public T a;
  ads
  $cnds$
  mds
}
	$=_{cds,Main}$	**class** B **extends** A {
		  public T' a;
		  ads
		  $cnds$
		  mds
		}

provided

JML:

(↔) $T \leq T'$ and every occurrence of a inside specifications of B, cds and $Main$, is cast with T or any subtype of T in cds.

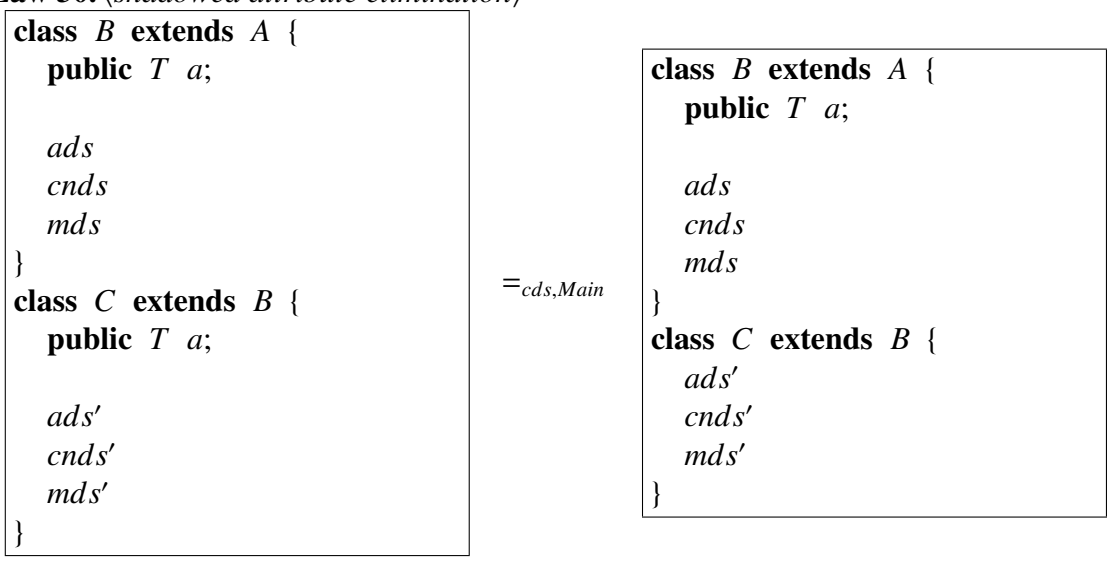
Java:

(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of mds, cds e $Main$, is cast with T or any subtype of T in cds.

(←) Every expression assigned to a, in mds, cds e C, is of type T any subtype of T.

□

Law 30. ⟨*shadowed attribute elimination*⟩



provided

JML:

(→) ((*B*)*e*).*a*, for any $e \leq C$, does not occur inside specifications of *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*.

(←) All accesses to *a* inside specifications of *cds*, *Main*, *cnds*, *cnds'*, *mds* or *mds'*, is of type ((*B*)*e*).*a*.

Java:

(→) ((*B*)*e*).*a*, for any $e \leq C$, does not occur in *cds*, *Main*, *cnds*, *cnds'*, *mds* nor *mds'*.

(←) All accesses to *a* in *cds*, *Main*, *cnds*, *cnds'*, *mds* or *mds'*, is of type ((*B*)*e*).*a*.

□

A.4 Methods

Law 31. ⟨*weaken pre-condition*⟩

class C **extends** D {
ads
$cnds$
 	
//@ **requires** ψ_1;
//@ **assignable** ω;
//@ **ensures** ψ_2;
@$spec\_cases$
rt $m(pds)$ {
$mbody$
}
 	
mds
}
	$\sqsubseteq_{cds,Main}$	**class** C **extends** D {
		ads
		$cnds$
		
		//@ **requires** ψ'_1;
		//@ **assignable** ω;
		//@ **ensures** ψ_2;
		@$spec\_cases$
		rt $m(pds)$ {
		$mbody$
		}
		
		mds
		}

provided

JML:

(1) $\psi_1 \Rightarrow \psi'_1$; (2) $\psi'_1 \Rightarrow fpre(B.m[pds])$, for every class B such that $B \leq C$.

□

Law 32. ⟨*strengthen post-condition*⟩

class C **extends** D {
ads
$cnds$
 	
//@ **requires** ψ_1;
//@ **assignable** ω;
//@ **ensures** ψ_2;
@$spec\_cases$
rt $m(pds)$ {
$mbody$
}
 	
mds
}
	$\sqsubseteq_{cds,Main}$	**class** C **extends** D {
		ads
		$cnds$
		
		//@ **requires** ψ_1;
		//@ **assignable** ω;
		//@ **ensures** ψ'_2;
		@$spec\_cases$
		rt $m(pds)$ {
		$mbody$
		}
		
		mds
		}

provided

JML:

(1) $\psi'_2 \Rightarrow \psi_2$; (2) $\backslash$**old**$(\psi_1) \Rightarrow (fpos(B.m[pds]) \Rightarrow \psi'_2)$, for every class B such that $B \leq C$.

□

Law 33. ⟨*insert identity specification case*⟩

```
class C extends D {
  ads
  cnds

  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires false;
  //@ assignable
                \not_specified;
  //@ ensures true;
  rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(→) For every redefined method $m(pds)$, of a class E, such that $E \leq C$, m has an explicit specification case.

Java:

(→) $C.m(e)$ does not appear in cds, $Main$, or mds.

□

Law 34. ⟨*insert default pre-condition*⟩

```
class C extends D {
  ads
  cnds

  //@ assignable ω;
  //@ ensures ψ;
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires true;
  //@ assignable ω;
  //@ ensures ψ;
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law 35. ⟨*insert default postcondition*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ;
  //@ assignable ω;
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ;
  //@ assignable ω;
  //@ ensures true;
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law 36. ⟨*insert default specification case in a method with no redefinitions*⟩

```
class C extends D {
  ads
  cnds

  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires true;
  //@ assignable
                \not_specified;
  //@ ensures true;
  rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(↔) For every method $m(pds)$, of a class B, such that $C \leq B$, m does not provide an explicit specification case.

Java:

(↔) $m(pds)$ is not declared in any class F such that $F \leq C$.

□

Law 37. ⟨*insert* **same** *specification case*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires \same;
  //@ assignable
                \not_specified;
  //@ ensures true;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

provided

JML:

(←) (1) @*spec_cases* has at least one specification case or *rt m*(*pds*) is an override; (2) @*spec_cases* does not have a specification case with pre-condition equals to **same**.

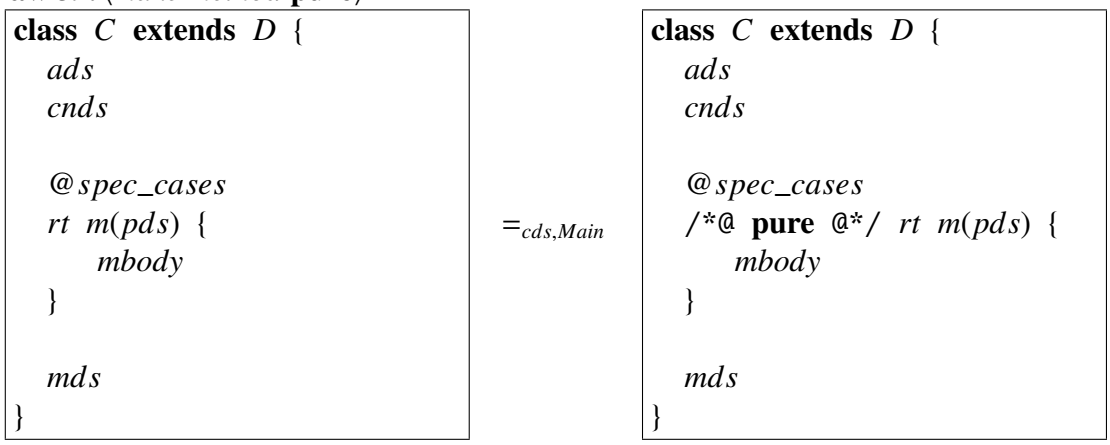
□

Law 38. ⟨*change* **assignable** *from* **not_specified** *to* **everything** ⟩

assignable **not_specified**; = **assignable** **everything**;

□

Law 39. ⟨*make method* **pure**⟩



provided

JML:

(↔) For all specification case *specc* such that $specc \in @spec\_cases$, $fassign(specc)$ is equivalent to **nothing**.

(←) $B.m(e)$ does not appear in specifications of cds, $Main$ nor in specifications of C, for any B such that $B \leq C$ and B does not redefine m.

□

Law 40. ⟨*delete duplicated spec case from redefined method*⟩

```
class B extends A {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ1;
  @spec_cases
  rt m(pds) {
      mbody
  }
  mds
}
class C extends B {
  ads′
  cnds′

  //@ also
  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ1;
  @spec_cases
  rt m(pds) {
      mbody
  }
  mds
}
class C extends B {
  ads′
  cnds′

  @spec_cases
  rt m(pds) {
      mbody
  }
  mds′
}
```

□

Law 41. ⟨*collapse pre-conditions*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ11;
  //@ requires ψ12;
  ...
  //@ requires ψ1n;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ11 && ψ12
       && ... && ψ1n;
  //@ assignable ω;
  //@ ensures ψ2;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law 42. ⟨*collapse post-conditions*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ21;
  //@ ensures ψ22;
  ...
  //@ ensures ψ2n;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ21 && ψ22
       && ... && ψ2n;
  @spec_cases
  rt m(pds) {
      mbody
  }

  mds
}
```

□

Law 43. ⟨*collapse **also** combinations*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ11;
  //@ assignable ω1;
  //@ ensures ψ21;
  //@ also ...
  //@ also
  //@ requires ψ1n;
  //@ assignable ωn;
  //@ ensures ψ2n;
  rt m(pds) {
      mbody
  }

  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  //@ requires ψ11
       || ... || ψ1n;
  //@ assignable ω;
  //@ ensures
    (\old(ψ11) ==> ψ21) &&
    ...
    && (\old(ψ1n) ==> ψ2n)
  rt m(pds) {
      mbody
  }

  mds
}
```

where

$\omega \mathrel{\widehat{=}} \omega_1 \cup \omega_2 \ldots \cup \omega_n$

□

Law 44. ⟨*change specification visibility of default method: from default to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ spec_public @*/ rt
  m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in @*spec_cases* has public specification visibility.

□

Law 45. ⟨*change specification visibility of a default method: from protected to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ spec_protected @*/ rt
  m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ spec_public @*/ rt
  m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 46. ⟨*change specification visibility of a private method: from private to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_public @*/
  rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 47. ⟨*change specification visibility of a private method: from protected to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_protected
  @*/ rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_public @*/
  rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 48. ⟨*change specification visibility of a protected method: from protected to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  protected rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  protected /*@ spec_public
  @*/ rt m(pds) {
      mbody
  }
  mds
}
```

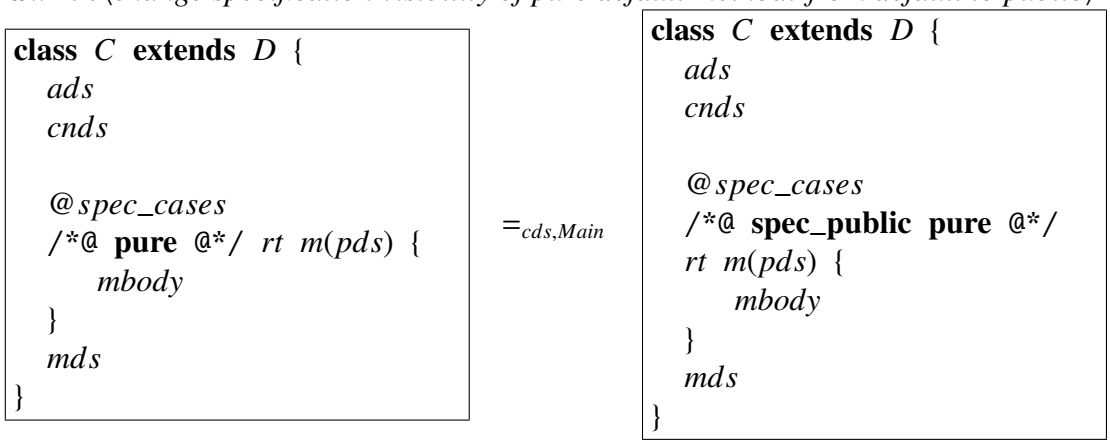
provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 49. ⟨*change specification visibility of pure default method: from default to public*⟩



provided

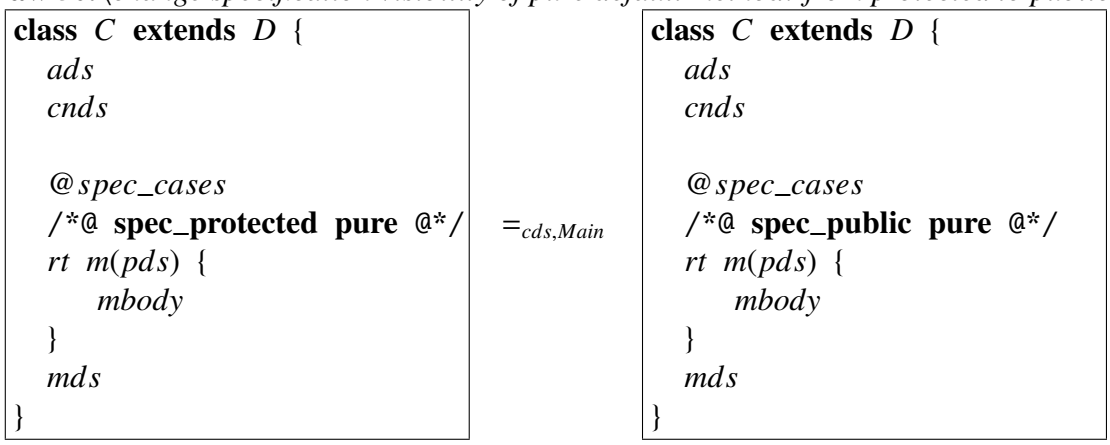
JML:

(→) Every attribute, pure method and model field that occurs in @*spec_cases* has public specification visibility.

(←) *B.m*(*e*), for any $B \leq C$, occurs only inside specifications with default or private specification visibility.

□

Law 50. ⟨*change specification visibility of pure default method: from protected to public*⟩



provided

JML:

(→) Every attribute, pure method and model field that occurs in @*spec_cases* has public specification visibility.

(←) *B.m*(*e*), for any $B \leq C$, occurs only inside specifications with non-public specification visibility.

□

Law 51. ⟨*change specification visibility of pure private method: from private to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ pure @*/ rt
  m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_public
  pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

(←) *m*(*e*) occurs only inside specifications with private specification visibility.

□

Law 52. ⟨*change specification visibility of pure private method: from protected to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_protected
  pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  private /*@ spec_public
  pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

provided

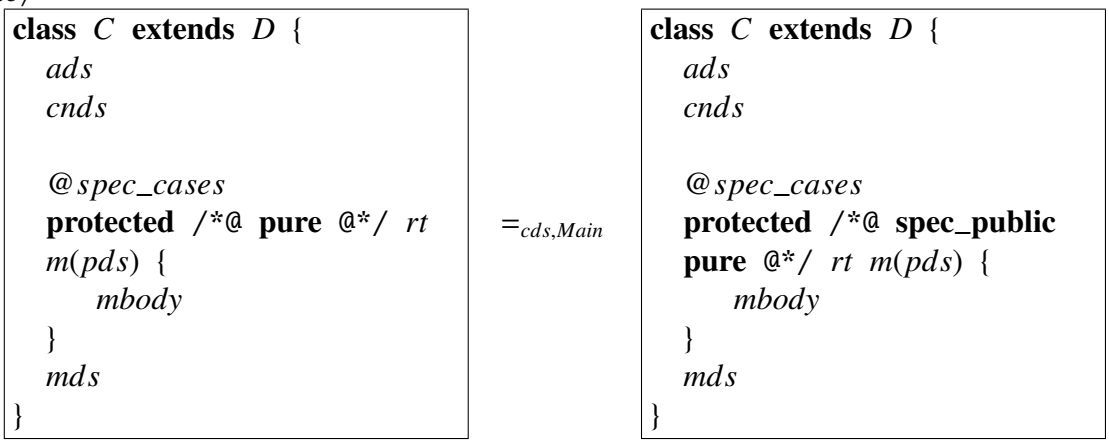
JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

(←) *m*(*e*), occurs only inside specifications with non-public specification visibility.

□

Law 53. ⟨*change specification visibility of pure protected method: from protected to public*⟩



provided

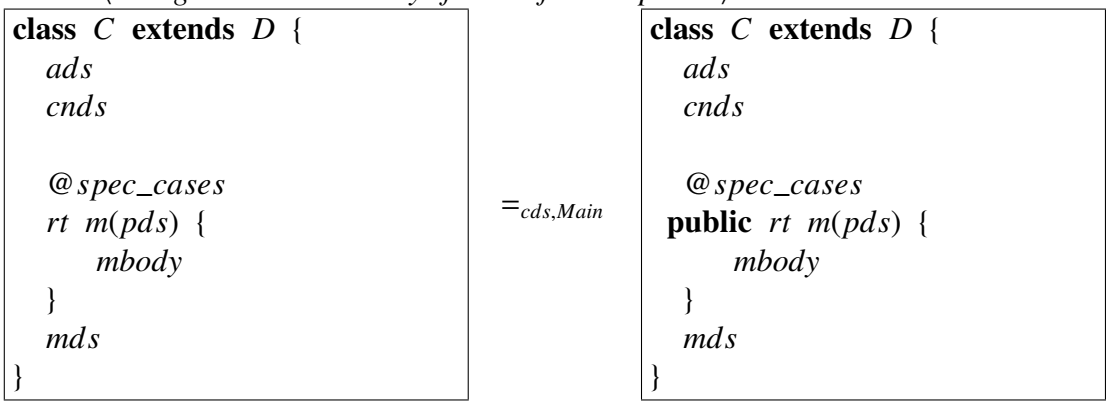
JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

(←) $B.m(e)$, for any $B \leq C$, occurs only inside specifications with non-public specification visibility.

□

Law 54. ⟨*change method visibility: from default to public*⟩



provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 55. ⟨*change method visibility: from protected to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  protected rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  public rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

□

Law 56. ⟨*change method visibility: from private to public*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  private rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  public rt m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

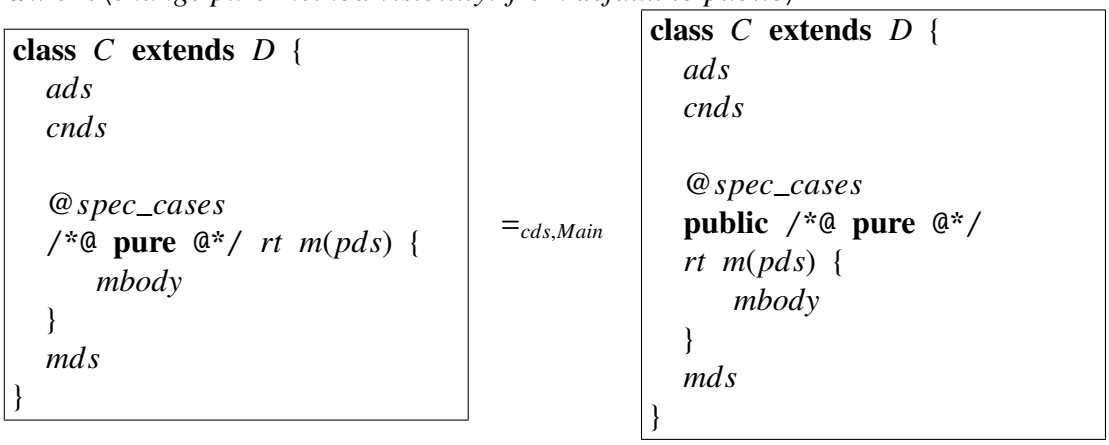
(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

Java:

(←) (1) $B.m(e)$, for any $B \leq C$ except of strict type C, does not occur in *cds* or *Main*; (2) $C.m(e)$ occurs only in C's body.

□

Law 57. ⟨*change pure method visibility: from default to public*⟩



provided

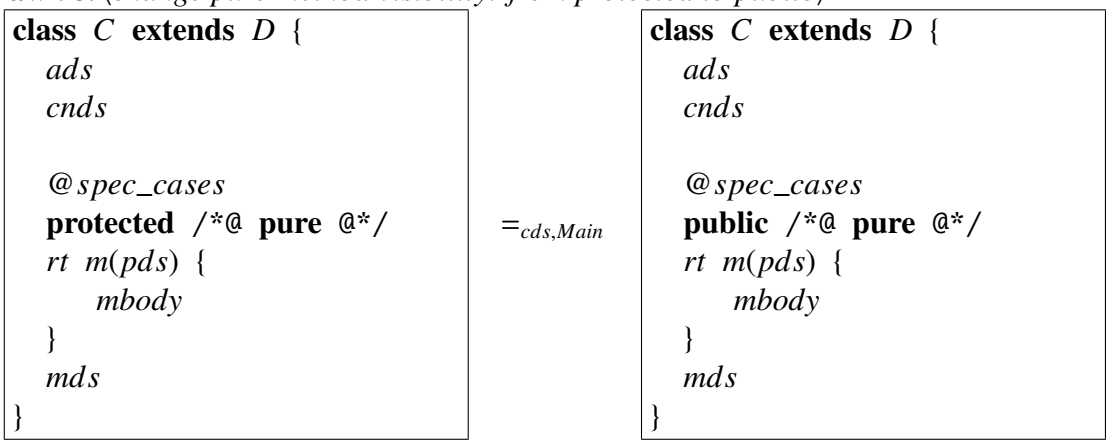
JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

(←) *B.m(e)*, for any $B \leq C$, occurs only inside specifications with default or private specification visibility.

□

Law 58. ⟨*change pure method visibility: from protected to public*⟩



provided

JML:

(→) Every attribute, pure method and model field that occurs in *@spec_cases* has public specification visibility.

(←) *B.m(e)*, for any $B \leq C$, occurs only inside specifications with non-public specification visibility.

□

Law 59. ⟨*change pure method visibility from: private to public*⟩

class *C* **extends** *D* {
ads
cnds
 	
@*spec_cases*
private /*@ **pure** @*/
rt m(*pds*) {
mbody
}
mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		ads
		cnds
		
		@*spec_cases*
		public /*@ **pure** @*/
		rt m(*pds*) {
		mbody
		}
		mds
		}

provided

JML:

(→) Every attribute, pure method and model field that occurs in @*spec_cases* has public specification visibility.

(←) (1) $B.m(e)$, for any $B \leq C$ except of strict type C, does not occur in any specification of *cds* or *Main*; (2) $C.m(e)$ occurs only inside specifications – that appears only in C's body – with private specification visibility.

Java:

(←) (1) $B.m(e)$, for any $B \leq C$ excepts of strict type C, does not occur in *cds* or *Main*; (2) $C.m(e)$ occurs only in C's body.

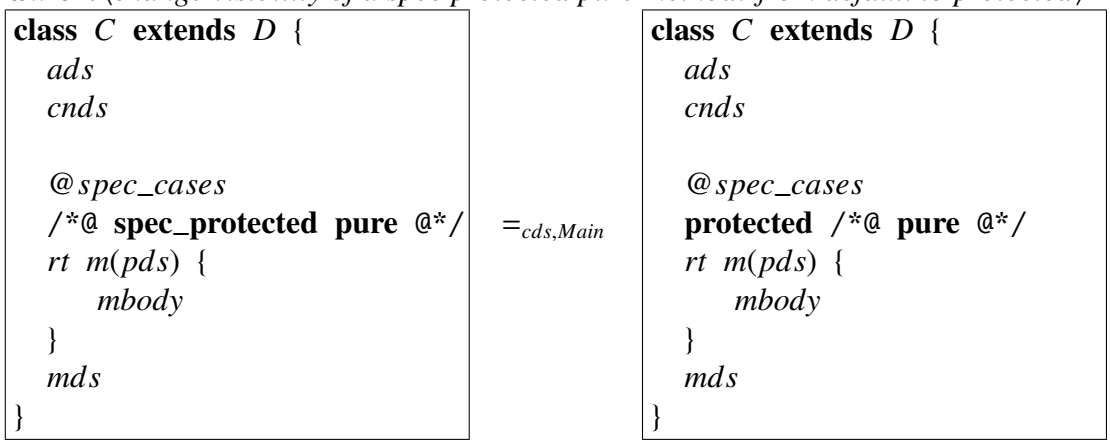
□

Law 60. ⟨*change visibility of a spec public pure method: from default to public*⟩

class *C* **extends** *D* {
ads
cnds
 	
@*spec_cases*
/*@ **spec_public** **pure** @*/
rt m(*pds*) {
mbody
}
mds
}
	$=_{cds,Main}$	**class** *C* **extends** *D* {
		ads
		cnds
		
		@*spec_cases*
		public /*@ **pure** @*/
		rt m(*pds*) {
		mbody
		}
		mds
		}

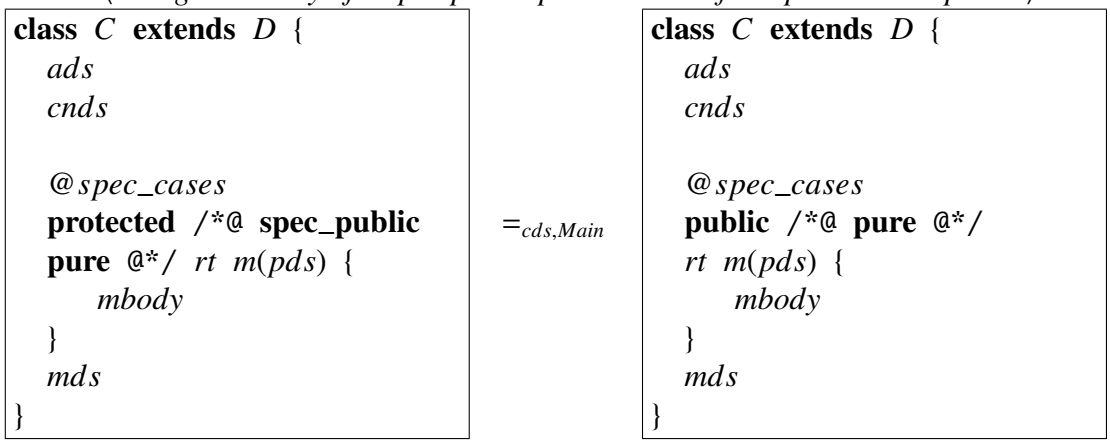
□

Law 61. ⟨*change visibility of a spec protected pure method: from default to protected*⟩



□

Law 62. ⟨*change visibility of a spec public pure method: from protected to public*⟩



□

Law 63. ⟨*change visibility of a spec public pure method: from private to public*⟩

```
class C extends D {                    =cds,Main    class C extends D {
  ads                                                   ads
  cnds                                                  cnds

  @spec_cases                                           @spec_cases
  private /*@ spec_public                               public /*@ pure @*/
  pure @*/ rt m(pds) {                                  rt m(pds) {
      mbody                                                 mbody
  }                                                     }
  mds                                                   mds
}                                                     }
```

provided

JML:

(←) (1) $B.m(e)$, for any $B \leq C$ excepts of strict type C, does not occur in any specification of *cds* or *Main*; (2) $C.m(e)$ occurs only inside specifications of C's body.

Java:

(←) (1) $B.m(e)$, for any $B \leq C$ excepts of strict type C, does not occur in *cds* or *Main*; (2) $C.m(e)$ occurs only in C's body.

□

Law 64. ⟨*change visibility of a spec protected pure method: from private to protected*⟩

```
class C extends D {                    =cds,Main    class C extends D {
  ads                                                   ads
  cnds                                                  cnds

  @spec_cases                                           @spec_cases
  private /*@ spec_protected                            protected /*@ pure @*/
  pure @*/ rt m(pds) {                                  rt m(pds) {
      mbody                                                 mbody
  }                                                     }
  mds                                                   mds
}                                                     }
```

provided

JML:

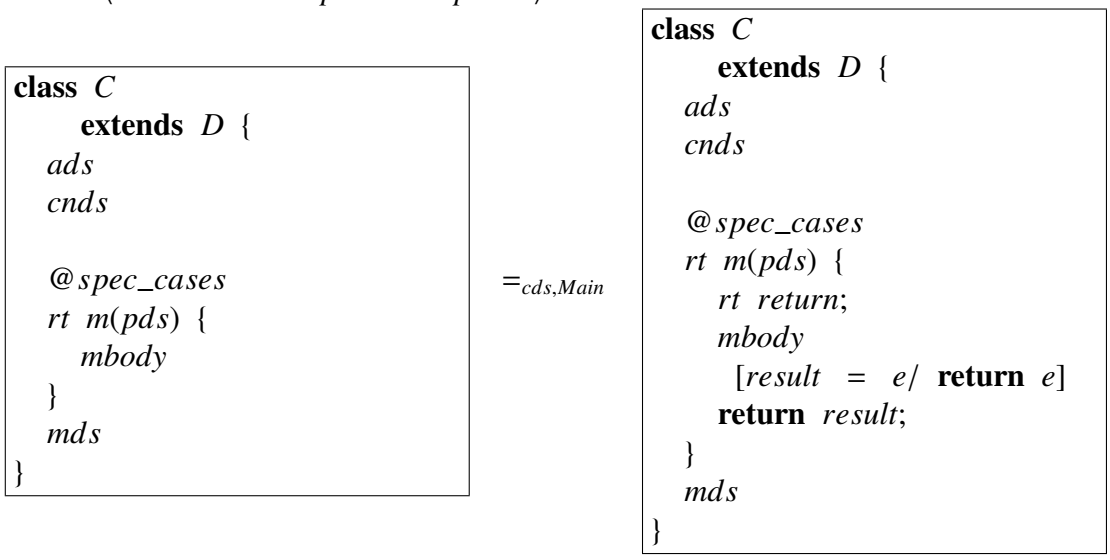
(←) (1) $B.m(e)$, for any $B \leq C$ except of strict type C, does not occur in any specification of *cds* or *Main*; (2) $C.m(e)$ occurs only inside specifications of C's body.

Java:

(←) (1) $B.m(e)$, for any $B \leq C$ excepts of strict type C, does not occur in *cds* or *Main*; (2) $C.m(e)$ occurs only in C's body.

□

Law 65. ⟨**eliminate multiple return points*⟩



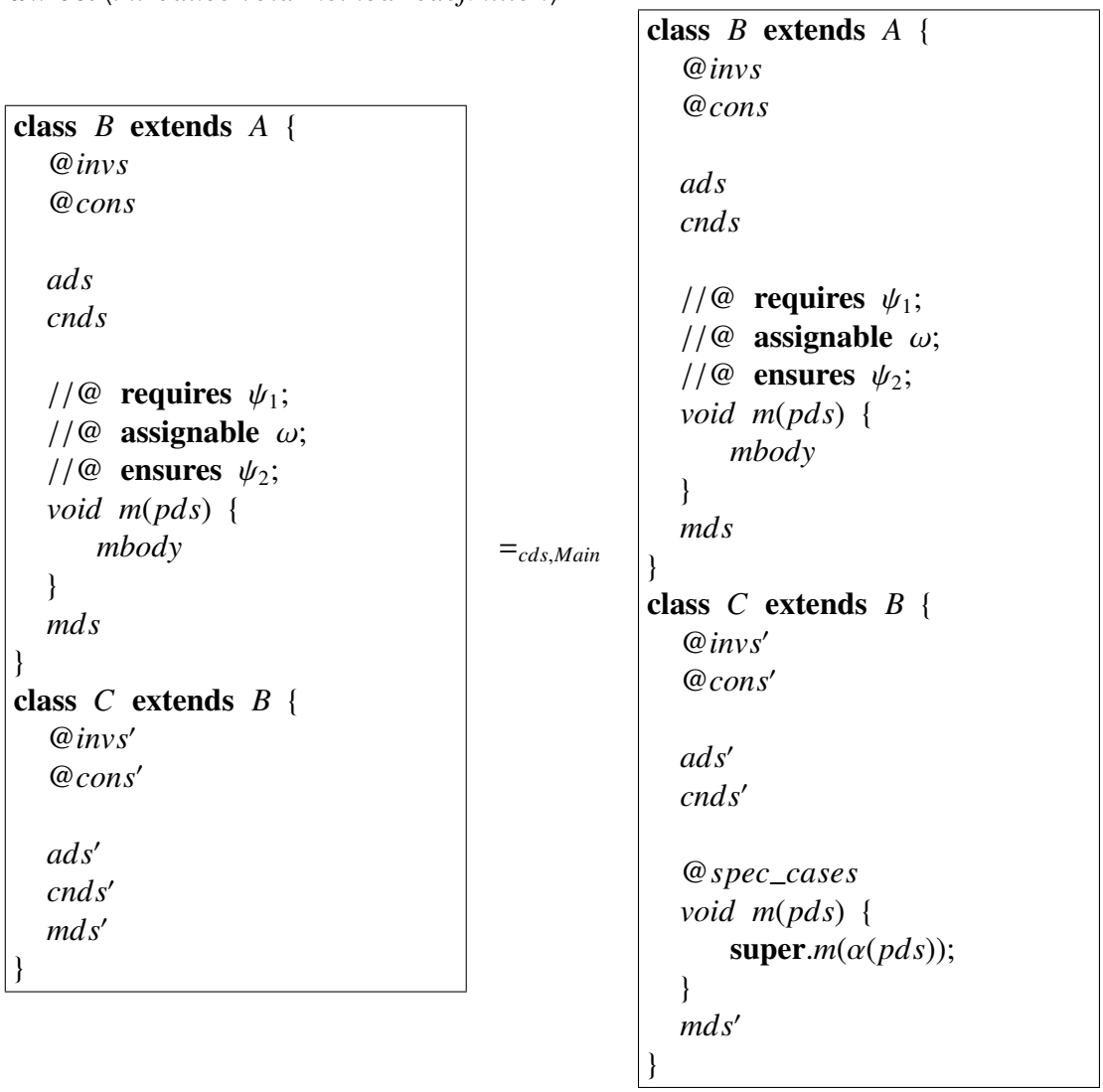
provided

Java:

(→) (1) the variable result is not already declared in *mbody*; (2) **return** clauses are present only inside mutually exclusive conditionals.

□

Law 66. ⟨*introduce void method redefinition*⟩



provided

JML:

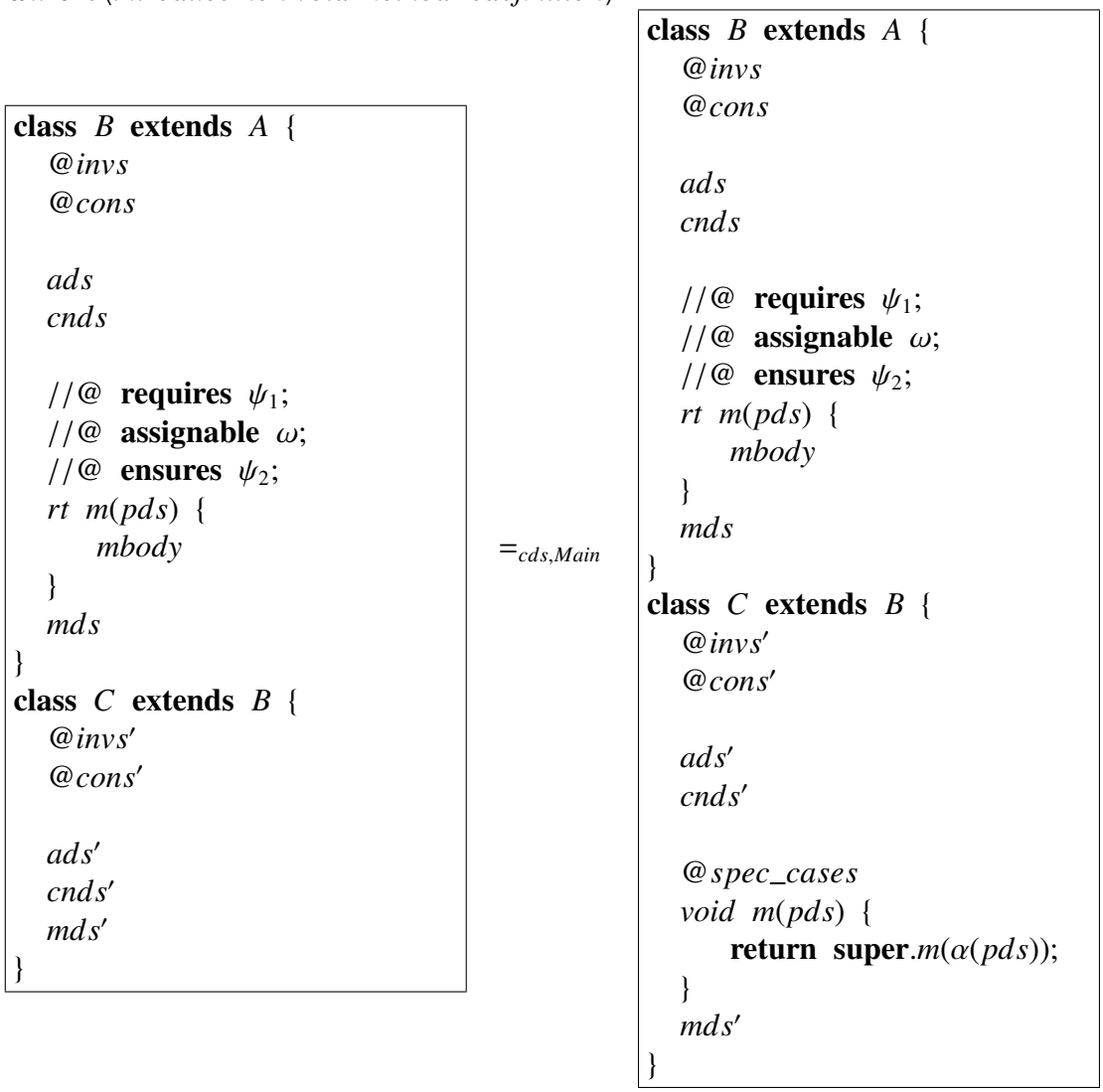
(↔) (1) @*invs′* and @*cons′* does not restrict attributes in *ads*, model fields of *B* or any attribute or model field inherited by *B*.

Java:

(→) *m*(*pds*) is not abstract and is not declared in *mds′*.

□

Law 67. ⟨*introduce non void method redefinition*⟩



provided

JML:

(↔) (1) @*invs′* and @*cons′* does not restrict attributes in *ads*, model fields of *B* or any attribute or model field inherited by *B*.

Java:

(→) *m*(*pds*) is not abstract and is not declared in *mds′*.

□

Law 68. ⟨*move original method to superclass*⟩

class B **extends** A {
ads
$cnds$
mds
}
class C **extends** B {
ads'
$cnds'$
 	
//@ **requires** ψ_1;
//@ **assignable** ω;
//@ **ensures** ψ_2;
$rt\ m(pds)$ { $mbody$ }
mds'
}
$cds,\ Main$
	=	**class** B **extends** A {
		ads
		$cnds$
		
		//@ **requires** ψ_1;
		//@ **assignable** ω;
		//@ **ensures** ψ_2;
		$rt\ m(pds)$ { $mbody$ }
		mds
		}
		class C **extends** B {
		ads'
		$cnds'$
		mds'
		}
		$cds',\ Main$

where

$cds' \mathrel{\widehat{=}} cds$[//@ **also** $fspec(m)/fspec(m)$], for every method m (with signature $rt\ m(pds)$ and that is not a redefinition) of any class E such that $E \leq B$ and $E \not\leq C$.

provided

JML:

(↔) (1) **super** does not appear in ψ_1 nor in ψ_2; (2) $\psi_1 \Rightarrow fpre(E[rt\ m(pds)])$ for every class E, such that $E \leq B$ but $E \not\leq C$, and E introduces a method $rt\ m(pds)$. (3) For any specification case for every method $rt\ m(pds)$, declared in any class E such that $E \leq B$ but $E \not\leq C$, with pre-condition PRE and postcondition $POST$, $\backslash\mathbf{old}(\psi_1) \Rightarrow ((\backslash\mathbf{old}(PRE) \Rightarrow POST) \Rightarrow (\backslash\mathbf{old}(\psi_1) \Rightarrow \psi_2)$.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes dos not appear in $mbody$; (2) $m(pds)$ is not declared in any superclass of B in cds.

(→) (1) $m(pds)$ is not declared in mds; (2) $mbody$ does not contain uncast occurrences of **this** nor expressions in the form $((C)\mathbf{this}).a$ and of the form $((C)\mathbf{this}).m(e)$ for any attribute a nor method m, in ads' and mds', respectively, with private visibility.

(←) (1) $m(pds)$ is not declared in mds'; (2) $D.m(e)$, for any $D \leq B$ and $D \not\leq C$, does not appear in cds, $Main$, mds or mds'.

□

Law 69. ⟨*move redefined method to superclass: overriden method with non-default specification case*⟩

```
class B extends A {
  ads
  cnds

  //@ requires ψ1;
  //@ ensures ψ2;
  rt m(pds) { mbody }
  mds
}
class C extends B {
  ads′
  cnds′

  //@ also
  //@ requires ψ′1;
  //@ ensures ψ′2;
  rt m(pds) { mbody′ }
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads
  cnds

  //@ requires (!(this
       instanceof C) && ψ1);
  //@ ensures (!(this
       instanceof C) && ψ2);
  //@ also
  //@ requires (this
       instanceof C && ψ′1);
  //@ ensures (this
       instanceof C && ψ′2);
  //@ also
  //@ requires (this
       instanceof C && ψ1);
  //@ ensures (this
       instanceof C && ψ2);
  rt m(pds) {
     if (!(this instanceof C))
     { mbody } else { mbody′ }
  }
  mds
}
class C extends B {
  ads′
  cnds′
  mds′
}
```

provided

JML:

(↔) **super** does not appear in ψ'_1 nor in ψ'_2.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes do not appear in *mbody′*; (2) **super**.*m* does not appear in *mds'*

(→) *mbody′* does not contain uncast occurrences of **this** nor expressions of the form ((*C*)**this**).*a* and of the form ((*C*)**this**).*m*(*e*) for any attribute *a* nor method *m*, in *ads′* and *mds′*, respectively, with private visibility.

(←) *m*(*pds*) is not declared in *mds′*.

□

Law 70. ⟨*move redefined method to superclass: overriden method with no specification cases*⟩

```
class B extends A {
  ads
  cnds

  rt m(pds) {
      mbody
  }
  mds
}
class C extends B {
  ads′
  cnds′

  //@ requires ψ′1;
  //@ assignable ω′;
  //@ ensures ψ′2;
  rt m(pds) { mbody′ }
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads
  cnds

  //@ requires (!(this
       instanceof C);
  //@ assignable
          \not_specified;
  //@ ensures (!(this
        instanceof C);
  //@ also
  //@ requires (this
       instanceof C && ψ′1);
  //@ assignable ω′;
  //@ ensures (this
        instanceof C && ψ′2);
  rt m(pds) {
      if (!(this instanceof C))
      { mbody } else {
          mbody′
      }
  }
  mds
}
class C extends B {
  ads′
  cnds′
  mds′
}
```

provided

JML:

(↔) (1) **super**, private attributes and private pure methods does not appear in ψ'_1 nor in ψ'_2.

(→) Both ψ_1 and ψ_2 do not contain occurrences of model fields declared in C, nor uncast occurrences of **this**.

Java:

(↔) (1) **super** and private attributes do not appear in *mbody′*; (2) **super**.*m* does not appear in *mds'*

(→) *mbody′* does not contain uncast occurrences of **this** nor expressions of the form ((*C*)**this**).*a* and of the form ((*C*)**this**).*m*(*e*) for any attribute *a* nor method *m*, in *ads′* and *mds′*, respectively, with private visibility.

(←) *m*(*pds*) is not declared in *mds′*.

□

Law 71. ⟨*change parameter type of pure method*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt
  m(T x, pds) { mbody }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt
  m(T′ x, pds) { mbody }
  mds
}
```

provided

JML:

(↔) every occurrence of x in expressions of $@spec\_cases$ are cast with T or with any subtype of T.

(←) every actual parameter associated with x found in specifications of C, cds e $Main$ is of type T or of any subtype of T.

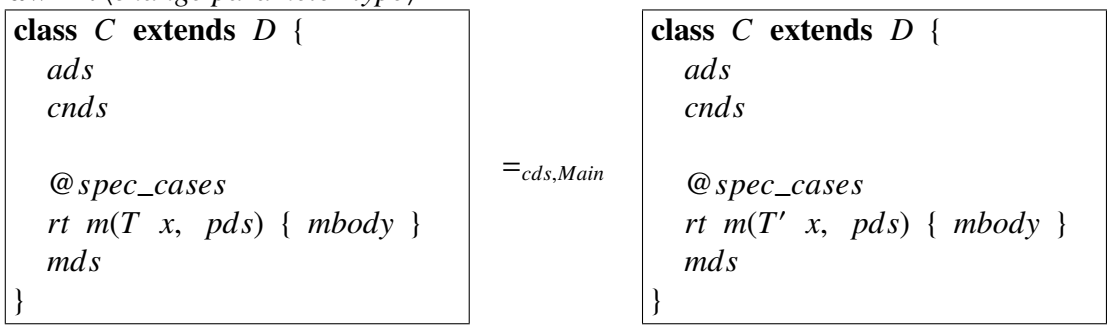
Java:

(↔) $T \leq T'$ and every non-assignable occurrence of x in expressions of *mbody* are cast with T or any subtype of T.

(←) (1) every actual parameter associated with x in mds, cds and $Main$ is of type T or any subtype of T; (2) every expression assigned to x in *mbody* is of type T or any subtype of T; (3) every use of x as the method return in *mbody* is for a corresponding declared return of type T or any supertype of T.

□

Law 72. ⟨*change parameter type*⟩



provided

JML:

(↔) every occurrence of x in expressions of $@spec\_cases$ are cast with T or with any subtype of T.

Java:

(↔) $T \leq T'$ and every non-assignable occurrence of x in expressions of *mbody* are cast with T or any subtype of T.

(←) (1) every actual parameter associated with x in mds, cds and $Main$ is of type T or any subtype of T; (2) every expression assigned to x in *mbody* is of type T or any subtype of T; (3) every use of x as the method return in *mbody* is for a corresponding declared return of type T or any supertype of T.

□

Law 73. ⟨*change return type of pure method*⟩

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt m(pds) {
      mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds

  @spec_cases
  /*@ pure @*/ rt′ m(pds) {
      mbody
  }
  mds
}
```

provided

JML:

(→) (1) every call to $m(pds)$ that occurs in specifications in C, cds and $Main$ is cast with rt; (2) every occurrence of **result** in postconditions of $@spec\_cases$ are cast with rt or any subtype of rt.

Java:

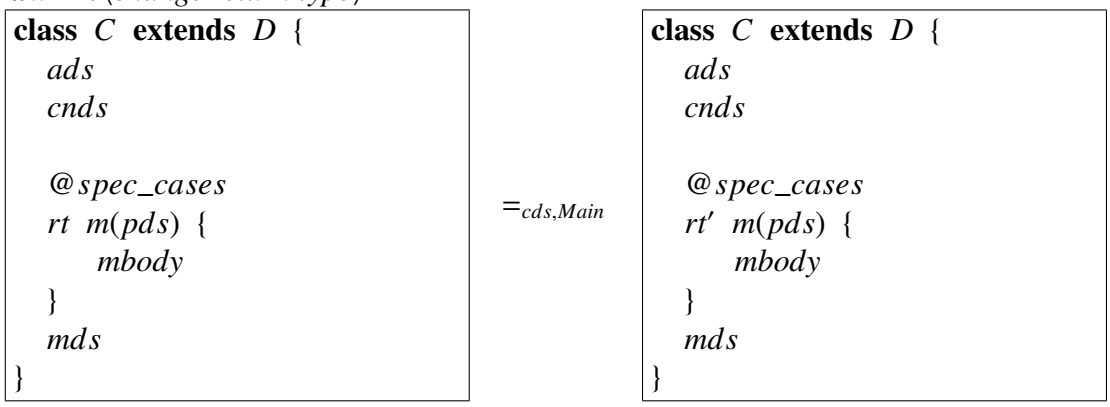
(↔) $rt \leq rt'$.

(→) every call to $m(pds)$ used as a expression is cast to rt.

(←) every expression used in the **return** return clause in $mbody$ is of type rt or of any subtype of rt.

□

Law 74. ⟨*change return type*⟩



provided

JML:

(→) every occurrence of **result** in postconditions of *@spec_cases* are cast with *rt* or any subtype of *rt*.

Java:

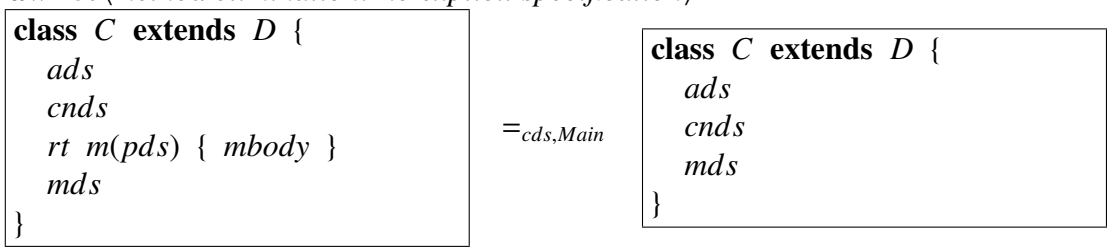
(↔) $rt \leq rt'$

(→) every call to $m(pds)$ used as a expression is cast to *rt*.

(←) every expression used in the **return** return clause in *mbody* is of type *rt* or of any subtype of *rt*.

□

Law 75. ⟨*method elimination: no explicit specification*⟩



provided

Java:

(→) $B.m(e)$ does not occur in *cds*, *Main* nor in *cnds*, *mds* for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

(←) $m(pds)$ is not declared in *mds* nor in any superclass or subclass of C in *cds*.

□

Law 76. ⟨*method elimination: some redefinition and non-default specification elimination*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  rt m(pds) { mbody }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

JML:

(↔) (1) $\psi_1 \Rightarrow fpre(E[rt\ m(pds)])$ for every class E such that $E \leq B$, $E \not\leq C$ and E has a already defined method $rt\ m(pds)$. (2) For every class E such that $E \not\leq C$ and E has a already defined method $rt\ m(pds)$, there is a *specification case* for m with pre-condition PRE, post-condition $POST$, and frame W where $\backslash\mathbf{old}(\psi_1) \Rightarrow ((\backslash\mathbf{old}(PRE) \Rightarrow POST) \Rightarrow (\backslash\mathbf{old}(\psi_1) \Rightarrow \psi_2)$ and $\omega \subseteq W$.

Java:

(↔) $rt\ m(pds)$ is already declared in any class E pertaining to cds such that $E \leq C$.

(→) $B.m(e)$ does not occur in cds, $Main$ nor in $cnds$, mds for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

(←) $rt\ m(pds)$ is not declared in mds nor in any superclass or subclass of C in cds.

□

Law 77. ⟨*method elimination: no redefinition and non-default pre-existent specification*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  rt m(pds) { mbody }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

Java:

(↔) *rt m*(*pds*) is not declared in any class *E* pertaining to *cds* such that $E \leq C$.

(→) *B.m*(*e*) does not occur in *cds*, *Main* nor in *cnds*, *mds* for any *B* such that $B \leq$ C.

(←) *rt m*(*pds*) is not declared in *mds* nor in any superclass or subclass of *C* in *cds*.

□

Law 78. ⟨*method elimination: pure, redefined, non-default pre-existent specification*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  /*@ pure @*/ rt m(pds) {
    mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

JML:

(↔) (1) $\psi_1 \Rightarrow fpre(E[rt\ m(pds)])$ for every class E such that $E \leq B$, $E \not\leq C$ and E has a already defined method $rt\ m(pds)$. (2) For every class E such that $E \not\leq C$ and E has a already defined method $rt\ m(pds)$, there is a *specification case* for m with pre-condition PRE, post-condition $POST$, and frame W where $\backslash\mathbf{old}(\psi_1) \Rightarrow ((\backslash\mathbf{old}(PRE) \Rightarrow POST) \Rightarrow (\backslash\mathbf{old}(\psi_1) \Rightarrow \psi_2)$ and $\omega \subseteq W$.

(→) $B.m(e)$ does not occur inside specifications of C, cds and $Main$ for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

Java:

(↔) $rt\ m(pds)$ is already declared in any class E pertaining to cds such that $E \leq C$.

(→) $B.m(e)$ does not occur in cds, $Main$ nor in $cnds$, mds for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

(←) $rt\ m(pds)$ is not declared in mds nor in any superclass or subclass of C in cds.

□

Law 79. ⟨*method elimination: pure, not redefined, no default pre-existent specification*⟩

```
class C extends D {
  ads
  cnds

  //@ requires ψ1;
  //@ assignable ω;
  //@ ensures ψ2;
  /*@ pure @*/ rt m(pds) {
    mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

JML:

(↔) *rt m*(*pds*) is not declared in any class *E* pertaining to *cds* such that $E \leq C$.

(→) *B.m*(*e*) does not occur inside specifications of *C*, *cds* and *Main* for any *B* such that $B \leq$ C.

Java:

(→) *B.m*(*e*) does not occur in *cds*, *Main* nor in *cnds*, *mds* for any *B* such that $B \leq$ C.

(←) *rt m*(*pds*) is not declared in *mds* nor in any superclass or subclass of *C* in *cds*.

□

Law 80. ⟨*method elimination: pure, no explicit specification*⟩

```
class C extends D {
  ads
  cnds

  /*@ pure @*/ rt m(pds) {
    mbody
  }
  mds
}
```

$=_{cds,Main}$

```
class C extends D {
  ads
  cnds
  mds
}
```

provided

JML:

(→) $B.m(e)$ does not occur inside specifications of C, cds and $Main$ for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

Java:

(→) $B.m(e)$ does not occur in cds, $Main$ nor in $cnds$, mds for any B such that $B \leq C$, B does not redefine m and the first superclass in its hierarchy that declares m is C or B is strictly C.

(←) $m(pds)$ is not declared in mds nor in any superclass or subclass of C in cds.

□

Law 81. ⟨*eliminate calls to void methods via super*⟩
CDS is a set of two class declarations as follows.

class *B* **extends** *A* {
  ads
  cnds

  @spec_cases
  void m(*pds*) { *mbody* }
  mds
}

class *C* **extends** *B* {
  ads′
  cnds′
  mds′
}

Thus, we have that:

cds CDS, *C* ▷ **super**.*m*(*e*)	=	*vardecs*(*pds*, *e*); /*@ **assert** *fext_inv*(*B*) && *fext_pre*(*B*[*m*(*pds*)]); @*/ *mbody* /*@ **assert** *fext_pos*(*B*[*m*(*pds*)]) && *fext_inv*(*B*) && *fext_const*(*B*[*m*(*pds*)]) @*/

provided

JML:

(→) (1) **super** does not occur in $fext\_pre(B[m(pds)])$, $fext\_pos(B[m(pds)])$ nor in $fext\_inv(B)$ and $fext\_const(B[m(pds)])$; (2) Model fields that represent private attributes do not occur in $fext\_pre(B[m(pds)])$ or in $fext\_pos(B[m(pds)])$; (3) Private attributes, private pure methods, and model fields that represent private attributes or private model fields, declared in D, for any D such that $B \leq D$, do not occur in $finv(D)$ and $fconst(D[m(pds)])$.

Java:

(→) (1) **super**, private attributes and private methods declared in *ads* and *mds*, respectively, do not occur in *mbody*.

(2) *mbody* does not contain **return** clauses.

□

Law 82. ⟨*eliminate calls to non void methods via super*⟩
CDS is a set of two class declarations as follows.

class *B* **extends** *A* {
  ads
  cnds

  @*spec_cases*
  void m(*pds*) { *mbody* }
  mds
}

class *C* **extends** *B* {
  ads′
  cnds′
  mds′
}

Thus, *cds CDS* , *C* ▷

rt a = **super**.*m*(*e*)	=	*rt a*; *vardecs*(*pds*, *e*); /*@ **assert** *fext_inv*(*B*) && *fext_pre*(*B*[*m*(*pds*)]); @*/ *mbody* [*a* = *result* / **return** *result*] /*@ **assert** *fext_pos*(*B*[*m*(*pds*)]) [*a*/**result*] && *fext_inv*(*B*) && *fext_const*(*B*[*m*(*pds*)]) @*/

provided

JML:

(→) **super** does not occur in *fext_pre*(*B*[*m*(*pds*)]), *fext_pos*(*B*[*m*(*pds*)]) nor in *fext_inv*(*B*) and *fext_const*(*B*[*m*(*pds*)]); (2) Model fields that represent private attributes do not occur in *fext_pre*(@spec_cases) or in *fext_pos*(@spec_cases); (3) Private attributes, private pure methods, and model fields that represent private attributes or private model fields do not occur in *fext_inv*(*B*) and *fext_const*(*B*[*m*(*pds*)]).

Java:

(→) (1) **super**, private attributes and private methods declared in *ads* and *mds*, respectively, do not occur in *mbody*. (2) *mbody* does not contain multiple return points.

□

Law 83. ⟨*void method call elimination*⟩
Consider that the following class declaration

class C **extends** D {
  ads
  $cnds$

  $@spec\_cases$
  $void\ m(pds)\ \{\ mbody\ \}$
  mds
}

is included in cds and that $cds,\ A \triangleright le : C$, meaning that le has static type C in the class A. Then

$$cds,\ A \triangleright le.m(e) \quad = \quad \begin{array}{l} \texttt{//@ } \textbf{assert}\ le\ \texttt{!=}\ \textbf{null}; \\ \texttt{/*@} \\ \ \textbf{assert}\ fext\_inv(C)[le/\textbf{this}] \\ \ \ \&\&\ fext\_pre(C[m(pds)])[le/\textbf{this}]; \\ \texttt{@*/} \\ vardecs(pds,\ e); \\ mbody[le/\textbf{this}] \\ \texttt{/*@} \\ \ \textbf{assert}\ fext\_pos(C[m(pds)])[le/\textbf{this}] \\ \ \ \&\&\ fext\_inv(C)[le/\textbf{this}] \\ \ \ \&\&\ fext\_const(C[m(pds)])[le/\textbf{this}]; \\ \texttt{@*/} \end{array}$$

provided

JML:

(→) (1) **super** does not occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$ nor in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$; (2) All attributes, pure methods and model fields that occur in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ are non-private. (3) All non-private model fields that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ represent only non-private attributes; (4) All accesses to non-private attributes and all calls to non-private pure methods that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$, are in the form **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

Java:

(→) (1) $m(pds)$ is not redefined in cds and $mbody$ does not cointain references to **super**; (2) all attributes and methods that occur in $mbody$ arenon-private. (3) $mbody$ does not contain recursive calls; (4) pds does not occur in e; (5) $mbody$ does not contain **return** clauses; (5) all accesses to non-private attributes and all calls to non-private methods that occur in $mbody$, are of type **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

□

Law 84. ⟨*non void method call elimination - when used as expression*⟩
Consider that the following class declaration

class C **extends** D {
  ads
  $cnds$

  $@spec\_cases$
  $rt\ m(pds)$ { $mbody$ }
  mds
}
is included in cds and that cds, $A \triangleright le : C$, meaning that le has static type C in the class A. Then

$$
\begin{array}{lll}
 & & rt\ a\mathbf{;} \\
 & & \texttt{//@}\ \mathbf{assert}\ le\ \mathbf{!=}\ \mathbf{null}; \\
 & & \texttt{/*@} \\
 & & \ \mathbf{assert}\ fext\_inv(C)[le/\mathbf{this}] \\
 & & \ \ \&\&\ fext\_pre(C[m(pds)])[le/\mathbf{this}]\mathbf{;} \\
 & & \texttt{@*/} \\
 & & vardecs(pds,\ e); \\
cds,\ A \triangleright rt\ a\ =\ le.m(e) & = & mbody[le/\mathbf{this}] \\
 & & [a\ =\ result/\mathbf{return}\ result] \\
 & & \texttt{/*@} \\
 & & \ \mathbf{assert}\ fext\_pos(C[m(pds)])[le/\mathbf{this}] \\
 & & \ \ [a/\backslash\mathbf{result}] \\
 & & \ \ \&\&\ fext\_inv(C)[le/\mathbf{this}] \\
 & & \ \ \&\&\ fext\_const(C[m(pds)])[le/\mathbf{this}]\mathbf{;} \\
 & & \texttt{@*/}
\end{array}
$$

provided

JML:

(→) (1) **super** does not occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$ nor in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$; (2) All attributes, pure methods and model fields that occur in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ are non-private. (3) all non-private model fields that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ represent only non-private attributes; (4) all accesses to non-private attributes and all calls to non-private pure methods that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$, are in the form **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is a non-private method.

Java:

(→) (1) $m(pds)$ is not redefined in cds and $mbody$ does not cointain references to **super**; (2) all attributes and methods that occur in $mbody$ arenon-private. (3) $mbody$ does not contain recursive calls; (4) pds does not occur in e; (5) $mbody$ does not contain multiple return points. (6) all accesses to non-private attributes and all calls to non-private methods that occur in $mbody$, are of type **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

□

Law 85. ⟨*non void method call elimination - when used as a statement*⟩
Consider that the following class declaration

class C **extends** D {
  ads
  $cnds$

  $@spec\_cases$
  $rt\ m(pds)\ \{\ mbody\ \}$
  mds
}

is included in cds and that $cds,\ A \triangleright le : C$, meaning that le has static type C in the class A. And assume that $\_a$ is fresh, i.e., not used elsewhere. Then

$$
\begin{array}{lcl}
cds,\ A \triangleright le.m(e) & = & \begin{array}{l}
rt\ \_a; \\
\texttt{//@}\ \textbf{assert}\ le\ !=\ \textbf{null}; \\
\texttt{/*@} \\
\ \textbf{assert}\ fext\_pre(C[m(pds)])[le/\textbf{this}]; \\
\texttt{@*/} \\
vardecs(pds,\ e); \\
mbody[le/\textbf{this}] \\
[\_a\ =\ result/\textbf{return}\ result] \\
\texttt{/*@} \\
\ \textbf{assert}\ fext\_pos(C[m(pds)])[le/\textbf{this}] \\
\quad [\_a/\backslash\textbf{result}] \\
\quad \&\&\ fext\_inv(C)[le/\textbf{this}] \\
\quad \&\&\ fext\_const(C[m(pds)])[le/\textbf{this}]; \\
\texttt{@*/}
\end{array}
\end{array}
$$

provided

JML:

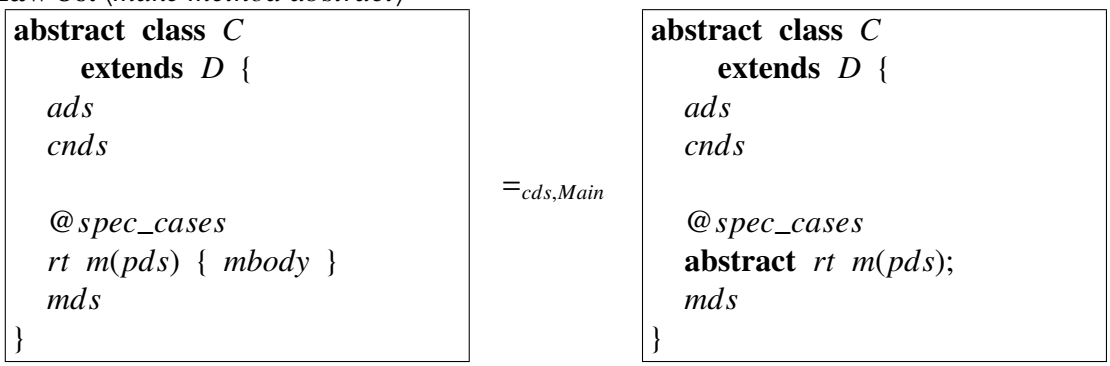
(→) (1) **super** does not occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$ nor in $fext\_inv(C)$ and $fext\_const(C[m(pds)])$; (2) All non-private model fields that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$ represent only non-private attributes; (3) All accesses to non-private attributes and all calls to non-private pure methods that occur in $fext\_pre(C[m(pds)])$, $fext\_pos(C[m(pds)])$, $fext\_inv(C)$ and $fext\_const(C[m(pds)])$, are in the form **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is a non-private method.

Java:

(→) (1) $m(pds)$ is not redefined in cds and $mbody$ does not cointain references to **super**; (2) all attributes and methods that occur in $mbody$ arenon-private. (3) $mbody$ does not contain recursive calls; (4) pds does not occur in e; (5) all accesses to non-private attributes and all calls to non-private methods that occur in $mbody$, are of type **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is anon-private method.

□

Law 86. ⟨*make method abstract*⟩



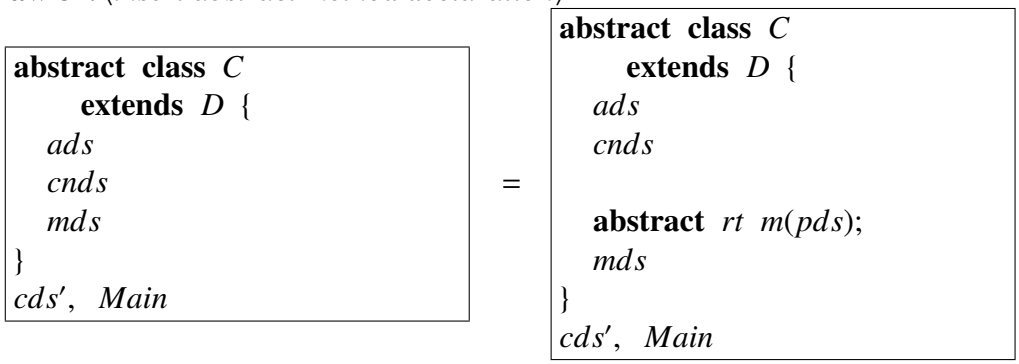
provided

Java:

(→) *rt m*(*pds*) is already declared in any class *E* pertaining to *cds* such that $E \leq C$.

□

Law 87. ⟨*insert abstract method declaration*⟩



where

$cds' \mathrel{\widehat{=}} cds[//@\ \textbf{also}\ fspec(m)/fspec(m)]$, for every method *m*(*pds*) of any class *E* such that $E \leq C$.

provided

Java:

(→) *rt m*(*pds*) is already declared in any class *E* pertaining to *cds* such that $E \leq C$.

□

A.5 Constructors

Law 88. ⟨*eliminate calls to* **super**($\alpha(pds)$)⟩

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads′

  @spec_cases'
  C(pds) {
    super(α(pds));
    cbody′
  }

  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads

  @spec_cases
  B(pds) { cbody }

  cnds
  mds
}
class C extends B {
  ads′

  @spec_cases'
  C(pds) {
    /*@ assert
      fpre(@spec_cases);
    @*/
    cbody
    /*@ assert
      fpos(@spec_cases)
      && fext_inv(B)
      && fext_init(B);
    @*/
    cbody′
  }

  cnds′
  mds′
}
```

provided

JML:

(→) (1) Private attributes, private pure methods, and model fields that represent private attributes or private model fields declared in D, for any D such that $B \leq D$, do not occur in $finv(D)$ or in $finit(D)$; (2) Model fields that represent private attributes or private model fields, declared in B do not occur in $fpre$(@spec_cases) or in $fpos$(@spec_cases); (3) B's default constructor does not have explicit specification cases.

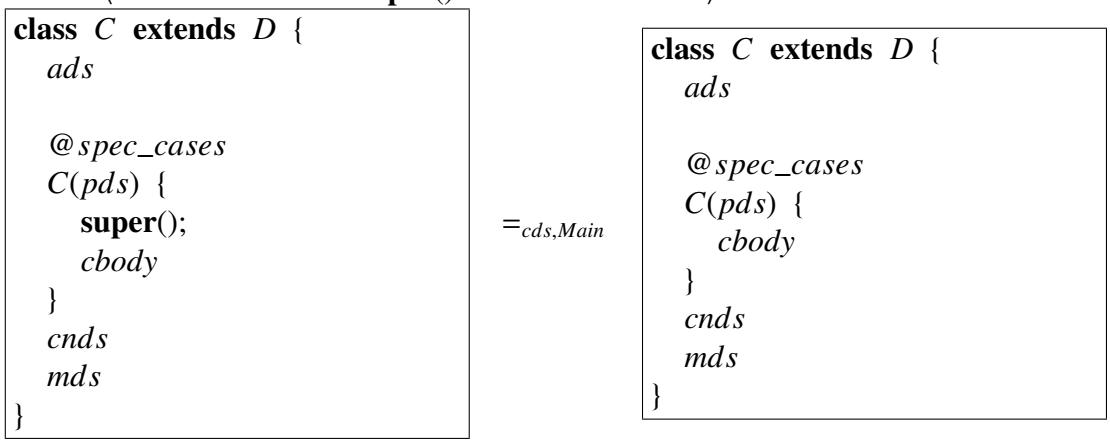
Java:

(↔) B's default constructor has a empty body.

(→) (1) *cbody* does not contain calls to **super**; (2) B has a default constructor; (3) private attributes and private methods declared in *ads* and *mds*, respectively, do not occur in *cbody*.

(←) B has a non-private constructor $B(pds)$, whose body is *cbody*.

Law 89. ⟨**eliminate calls to* **super**() *inside constructors*⟩



provided

Java:

(←) *cbody* does not contain any call to a super constructor.

□

Law 90. ⟨*eliminate calls to* **this**(*e*)⟩

```
class C extends D {
  ads′

  @spec_cases
  C(pds) { cbody }

  @spec_cases'
  C(pds′) {
    this(e);
    cbody′
  }

  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class C extends D {
  ads′

  @spec_cases
  C(pds) { cbody }

  @spec_cases'
  C(pds′) {
    vardecs(pds, e)
    /*@ assert
      fpre(@spec_cases);
    @*/
    cbody
    /*@ assert
      fpos(@spec_cases)
      && fext_inv(B)
      && fext_init(B);
    @*/
    cbody′
  }

  cnds′
  mds′
}
```

provided

Java:

(↔) *e* matches *pds*.

(→) (1) *cbody* does not contain calls to **super**; (2) *cbody′* does not contain calls to **super**.

□

Law 91. ⟨*eliminate non-default constructors: when constructor's body does not have to call a superconstructor explicitly*⟩

```
class B extends A {
  ads
  cnds
  mds
}
class C extends B {
  ads′

  @spec_cases
  C(pds′) { cbody′ }

  cnds′
  mds′
}
```

$=_{cds,Main}$

```
class B extends A {
  ads
  cnds
  mds
}
class C extends B {
  ads′
  cnds′
  mds′
}
```

provided

JML:

(→) **new** $C(\alpha(pds'))$ does not occur inside specifications of B, C, cds and $Main$.

Java:

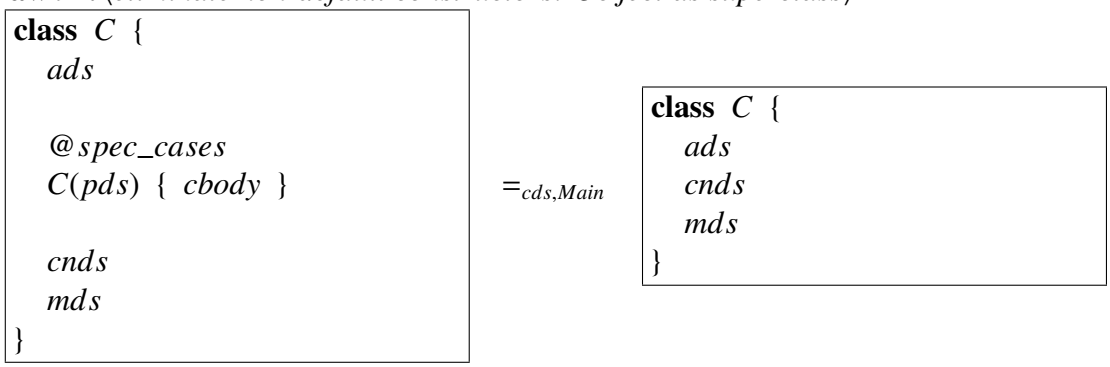
(↔) $cnds$ is empty or $cnds$ has an explicit default constructor.

(→) There are no calls to $C(pds')$ (including calls via **super** or **this**)

(←) $C(pds')$ is not declared in C

□

Law 92. ⟨*eliminate non-default constructors: Object as superclass*⟩



provided

JML:

(→) **new** $C(\alpha(pds)$ does not occur inside specifications of C, cds and $Main$.

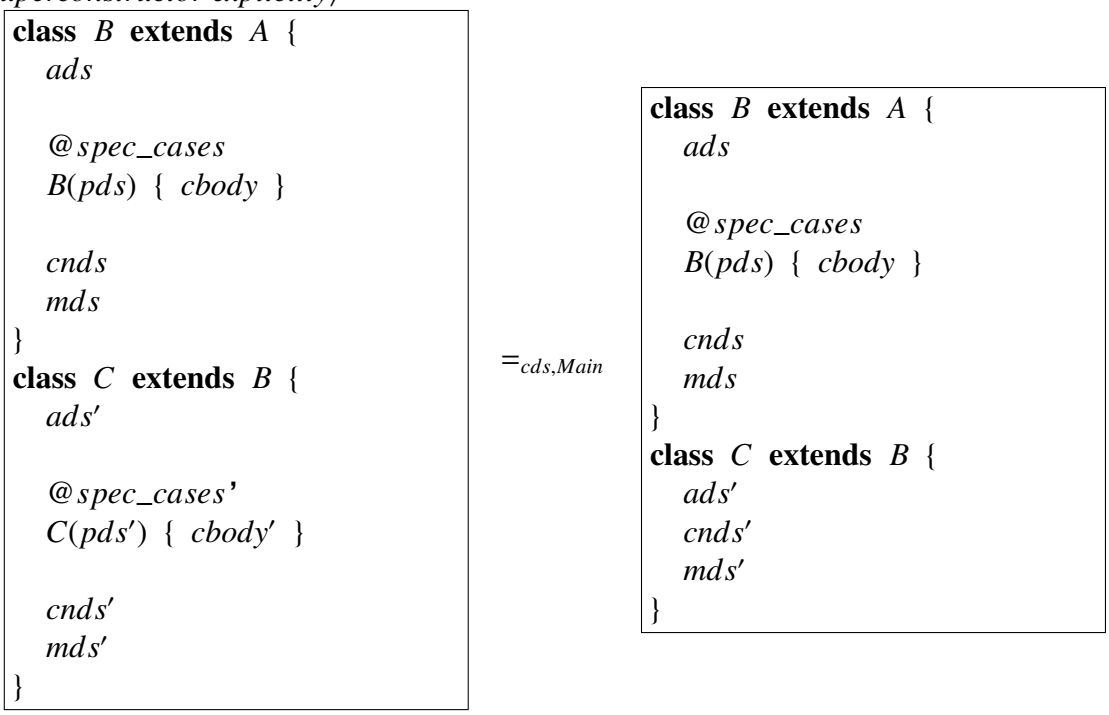
Java:

(→) There are no calls to $C(pds)$ (including calls via **super** or **this**)

(←) $C(pds)$ is no already declared in C

□

Law 93. ⟨*eliminate non-default constructors: when constructor's body have to call a superconstructor explicitly*⟩



provided

JML:

(→) **new** $C(\alpha(pds'))$ does not occur inside specifications of B, C, cds and $Main$.

Java:

(↔) (1) $cnds$ does not have an explicit default constructor; (2) $cbody'$ has a **super** call like **super**$(\alpha(pds_{\texttt{cnds}}))$ where $pds_{\texttt{cnds}}$ is the formal parameters list of some constructor that pertains to $cnds$.

(→) There are no calls to $C(pds')$ (including calls via **super** or **this**)

(←) $C(pds')$ is not declared in C

□

Law 94. ⟨*eliminate calls to non-default constructors*⟩
Consider that the following class declaration

class C **extends** D {
  ads

  $@spec\_cases$
  $C(pds)$ { $cbody$ }

  $cnds$
  mds
}
is included in cds and that cds, $A \triangleright le : C$, meaning that le has static type C in the class A. Then

$C\ le\ =\ \mathbf{new}\ C(e);$ $\quad =_{cds,Main}$

$C\ le\ =\ \mathbf{new}\ C();$
$vardecs(pds,\ e);$
/*@ **assert**
  $fpre$(@spec_cases);
  [le/**this**]
@*/
$cbody$[le/**this**]
/*@ **assert**
  $fpos$(@spec_cases)
  [le/**this**]
  && $fext\_inv(C)$
  [le/**this**]
  && $fext\_init(C)$;
  [le/**this**]
@*/

provided

JML:

(→) (1) **super** does not occur in $fpre$(@spec_cases), $fpos$(@spec_cases) nor in $fext\_inv(C)$ and $fext\_init(C)$; (2) Private attributes, private pure methods, and model fields that represent private attributes or private model fields declared in B, for every B such that $C \leq B$, do not occur in $finv(B)$ or $finit(B)$; (3) All accesses to non-private attributes and all calls to non-private pure methods that occur in $fpre$(@spec_cases), $fpos$(@spec_cases), $fext\_inv(C)$ and $fext\_init(C)$, are in the **this**.a and **this**.$m(e)$, respectively, where a is a non-private attribute and m is a non-private method; (4) C's default constructor does not have explicit specification cases.

Java:

(→) (1) C has a default constructor; (2) there are no calls to **super** or **this**() in $cbody$; (3) all attributes and methods that occur in $cbody$ are non-private. (4) pds does not occur in e;

□

A.6 Commands and Expressions

Law 95. ⟨*replace switch by* **if**-**else** *clauses*⟩
If i ranges over $1..n$, then

```
switch (e) {                          if (e == e1) { c1; }
  case e1: c1; break;                 else if (e == ei) { ci; }
  case ei: ci; break;          =      else { cdef; }                    □
  default : cdef ;
}
```

Law 96. ⟨*if* **true** *evaluation*⟩
Since $e_1 == e_2$ is evaluated to **true** in any evaluation, then

```
if (e1 == e2) { c; }          =       c                                 □
```

Law 97. ⟨**if**-**else** *identical commands*⟩

```
if (e) { c; }
else { c; }                   =       c                                 □
```

Law 98. ⟨**if** *identical commands*⟩
If $\bigvee i : 1..n \bullet e_i$ = **true**, then

```
if (e1) { c; }
else if (ei) { c; }           =       c                                 □
else { c; }
```

Law 99. ⟨*introduce a trivial JML-assert expression after assignment*⟩

```
le = e;  =  /*@ assert (le == e); @*/  le = e;                          □
```

Law 100. ⟨*introduce trivial cast in expressions*⟩
If $cds, A \rhd e : C$, then

```
cds, A ▷ e = (C) e                                                      □
```

Law 101. ⟨**eliminate/introduce* **this** *in attribute access*⟩
Consider the following class declaration

class *C* **extends** *D* {
  public *T att*;
  ads
  cnds
  mds
}
then *cds*, *C* ▷ *att* = *this.att* □

Law 102. ⟨**eliminate/introduce* **this** *in method calls*⟩
Consider the following class declaration

class *C* **extends** *D* {
  ads
  cnds
  rt m(*pds*) { *mbody* }
  mds
}
then *cds*, *C* ▷ *m*(*e*) = *this.m*(*e*) □

Law 103. ⟨*introduce* **this** *in pure method calls in predicates*⟩
Consider the following class declaration and that *m*(*e*) is written in a valid JML predicate

class *C* **extends** *D* {
  ads
  cnds
  /*@ **pure** @*/ *rt m*(*pds*) { *mbody* }
  mds
}
then *cds*, *C* ▷ *m*(*e*) = *this.m*(*e*) □

Law 104. ⟨*eliminate cast of expressions*⟩
If *cds*, *A* ▷ *le* : *B* and *cds*, *A* ▷ *le* : *B*′, with

cds, *A* ▷ *le* := (*C*) *e* = /*@ **assert** (*e* **instanceof** *C*); @*/ *le* := *e* □

Law 105. ⟨*eliminate cast of method call*⟩
If *cds*, *A* ▷ *e* : *B*, *C* ≤ *B* and *m* is declared in *B* or in any of its superclasses in *cds*, then

cds, *A* ▷ ((*C*)*e*).*m*(*e*′) = /*@ **assert** (*e* **instanceof** *C*); @*/ *e*.*m*(*e*′) □

Law 106. ⟨*change variable type*⟩

$$cds,\ A \ \triangleright\ T\ x;\ c\ =\ T'\ x;\ c$$

provided

JML:

(↔) Every occurrence of x inside specifications of c, is cast with T or any subtype of T.

Java:

(↔) $T \leq T'$.

(←) (1) Every expression assigned to x in c is of type T or any subtype of T; (2) every use of x as the return expression in c is for a corresponding declared return of type T or any subtype of T.

□

Ct stands for the scope where the expression exp is being used that is usually a method or constructor body, or conditionals and loops delimited with braces.

Law 107. ⟨**replace expression by variable*⟩
Consider that $cds, N \triangleright T\ exp$, then $cds, N \triangleright$

$$Ct[exp] \quad = \quad \begin{array}{l} ExpType\ tmp\ =\ exp; \\ Ct[tmp = exp] \end{array}$$

provided

Java:

(→) (1) tmp is not already declared in Ct; (2) variables used in exp are not assigned in Ct.

□

A.7 Predicates

Law 108. ⟨*delete trivial cast in* **instanceof** *implications inside predicates*⟩
If $cds, A \triangleright e : C$, then

$$e\ instanceof\ C\ ==>\ (C)\ e \quad = \quad e\ instanceof\ C\ ==>\ e$$

□

Law 109. ⟨*eliminate cast of pure method call in predicates*⟩
If cds, $A \rhd e : B$, $C \leq B$, m is pure and is declared in B or in any of its superclasses in cds and $((C)e).m(e')$ is written in a valid JML predicate, then

$$cds,\ A \rhd ((C)e).m(e') \quad = \quad e\ \textbf{instanceof}\ C\ ==>\ e.m(e') \qquad \square$$

APPENDIX B

Initial and Final Source-Code of the *Exp*1 Interpreter

This appendix shows the initial and final source-code of the *Exp*1 Interpreter used in Chapter 4.

B.1 Initial Source-Code of the *Exp*1 Interpreter

```
public class Expression {

  public Expression () {}
  public Value eval() { return null; }
}

public class BinaryExpression extends Expression {

  //@ initially this.leftExp != null && this.rightExp != null;

  private /*@ spec_public @*/ Expression leftExp;
  private /*@ spec_public @*/ Expression rightExp;

  public BinaryExpression() {
    this.leftExp = new Integer();
        this.rightExp = new Integer();
  }

  /*@ requires leftExp != null && rightExp != null;
    @ assignable this.leftExp, this.rightExp;
    @ ensures this.leftExp == leftExp && this.rightExp == rightExp;
    @*/
  public BinaryExpression(Expression leftExp, Expression rightExp) {
    super();
    this.leftExp = leftExp;
    this.rightExp = rightExp;
  }

  /*@ requires le != null && re != null;
    @ assignable this.leftExp, this.rightExp;
    @ ensures this.leftExp == le && this.rightExp == re;
    @*/
  public void set(Expression le, Expression re) {
```

```
    this.leftExp = le;
    this.rightExp = re;
  }

  //@ ensures \result == this.leftExp;
  public /*@ pure @*/ Expression getLeftExp() {
    return this.leftExp;
  }

  //@ ensures \result == this.rightExp;
  public /*@ pure @*/ Expression getRightExp() {
    return this.rightExp;
  }
}

public class Value extends Expression {
  public Value() {}
}

public class Sum extends BinaryExpression {
  public Sum() {
    super();
  }

  /*@ requires leftExp != null && rightExp != null;
    @ assignable this.leftExp, this.rightExp;
    @ ensures this.leftExp == leftExp && this.rightExp == rightExp;
    @*/
  public Sum(Expression leftExp, Expression rightExp) {
    super(leftExp, rightExp);
  }

  /*@ also
    @ ensures \result != null;
    @*/
  public Value eval() {
    Expression le = this.getLeftExp();
    Expression re = this.getRightExp();
    Integer lint = new Integer();
    Integer rint = new Integer();
    lint.setVal(((Integer)le.eval()).getVal());
    rint.setVal(((Integer)re.eval()).getVal());
    return new Integer(lint.getVal() + rint.getVal());
  }
}

public class Integer extends Value {

  private /*@ spec_public @*/ int val;

  public Integer() {
    super();
    this.val = 0;
  }

  /*@ assignable this.val;
    @ ensures this.val == val;
    @*/
```

```
  public Integer(int val) {
    super();
    this.val = val;
  }

    //@ ensures \result == val;
  public /*@ pure @*/ int getVal() {
    return this.val;
  }

  /*@ assignable this.val;
    @ ensures this.val == val;
    @*/
  public void setVal(int val) {
    this.val = val;
  }
  /*@ also
      @ ensures \result == this;
      @*/
  public Value eval() {
    return this;
  }
}

public class Interpreter {

  //@ public invariant this.exp != null;
    private /*@ spec_public @*/ Expression exp;

    public Interpreter() {
      super();
      this.exp = new Integer();
    }

  /*@ requires exp != null;
    @ assignable this.exp;
    @ ensures this.exp == exp;
    @*/
  public Interpreter(Expression exp) {
    super();
    this.exp = exp;
  }

  //@ ensures \result == this.exp;
  public /*@ pure @*/ Expression getExp() {
    return this.exp;
  }

  /*@ requires exp != null;
    @ assignable this.exp;
    @ ensures this.exp == exp;
    @*/
  public void setExp(Expression exp) {
    this.exp = exp;
  }

    public Value run() {
    return this.exp.eval();
```

```
  }
}

public class Main {

  public static void main(String[] args) {
    Interpreter in;
    Integer n1,n2;
    Sum s;
    Value v;
    n1 = new Integer(5);
    n2 = new Integer(3);
    s = new Sum(n1,n2);
    in = new Interpreter(s);
    v = in.run();
  }
}
```

B.2 Final Source-Code of the $Exp1$ Interpreter

```
public class  _Object {

  //@ invariant this instanceof Interpreter ==> this.exp != null;

  public int val;

  public /*@ nullable @*/ _Object exp;

  public /*@ nullable @*/ _Object rightExp;
  public /*@ nullable @*/ _Object leftExp;

    //@ requires ((!(this instanceof BinaryExpression)) &&  ((!(this
       instanceof Value)) || ((this instanceof Value) && ((!(this
       instanceof Integer)) || (this instanceof Integer && true)))));
  //@ assignable \not_specified;
  //@ ensures ((!(this instanceof BinaryExpression)) && ((\old((!(this
     instanceof Value))) ==> (!(this instanceof Value)))
  //@ && (\old(((this instanceof Value) && ((!(this instanceof Integer)
     ) || (this instanceof Integer && true)))) ==> ((this instanceof
     Value) && ((\old(!(this instanceof Integer)) ==> (!(this
     instanceof Integer))) && (\old((this instanceof Integer && true))
      ==> (this instanceof Integer && \result == this)))))));
  /*@ also
    @ requires ((this instanceof BinaryExpression) && ((!(this
       instanceof Sum)) || ((this instanceof Sum) && true)));
    @ assignable \not_specified;
    @ ensures ((this instanceof BinaryExpression) && ((\old((!(this
       instanceof Sum))) ==> (!(this instanceof Sum))) && (\old(((this
        instanceof Sum) && true)) ==> (this instanceof Sum && \result
       != null))));
    @*/
  public _Object eval() {

    if (!(this instanceof BinaryExpression)) {
      if (!(this instanceof Value)) {
        _Object tmp;
        /*@ assert
                  true;
```

```
          @*/
      tmp = null;
      /*@ assert
            true;
          @*/

      return tmp;
    } else {

      if (!(this instanceof Integer)) {
        _Object tmp;
        /*@ assert
          true;
        @*/
        tmp = null;
        /*@ assert
            true;
         @*/

        return tmp;
      } else {
        return this;
      }

    }
  } else {
    if (!(this instanceof Sum)) {

      _Object tmp;
      /*@ assert
           true;
        @*/
      tmp = null;
      /*@ assert
          true;
        @*/

      return tmp;
    } else {

      _Object le;
      //@ assert this != null;
      /*@ assert
          this instanceof Interpreter ==> this.exp != null
          && ((!(this instanceof Sum) && true) || ((this instanceof
              Sum) && true) || ((this instanceof Sum) && true));
       @*/
        le = this.leftExp;
      /*@ assert
          ((\old((!(this instanceof Sum) && true)) ==> (!(this
              instanceof Sum) && (le == this.leftExp)))
        && (\old(((this instanceof Sum) && true)) ==> ((this
            instanceof Sum) && (le == this.leftExp)))
        && (\old(((this instanceof Sum) && true)) ==> ((this
            instanceof Sum) && (le == this.leftExp))))
        &&  this instanceof Interpreter ==> this.exp != null;
        @*/
```

```
_Object re; //GETRIGHTEXP
//@ assert this != null;
/*@ assert
      this instanceof Interpreter ==> this.exp != null
      && ((!(this instanceof Sum) && true) || ((this instanceof
          Sum) && true) || ((this instanceof Sum) && true));
  @*/
  re = this.rightExp;
/*@ assert
     (\old((!(this instanceof Sum) && true)) ==> (!(this
        instanceof Sum) && (re == this.rightExp)))
     && (\old(((this instanceof Sum) && true)) ==> ((this
        instanceof Sum) && (re == this.rightExp)))
    && (\old(((this instanceof Sum) && true)) ==> ((this
       instanceof Sum) && (re == this.rightExp)))
     && ((!(this instanceof Sum) && true) || ((this instanceof
        Sum) && true));
  @*/

_Object lint = new Integer();
_Object rint = new Integer();

_Object tmp2;
int tmp1;
tmp2 = le.eval();

//@ assert tmp2 != null;
/*@ assert
    tmp2 instanceof Interpreter ==> tmp2.exp != null
    && true;
 @*/
tmp1 = tmp2.val;
/*@ assert
            tmp1 == tmp2.val
      && tmp2 instanceof Interpreter ==> tmp2.exp != null;
    @*/

int val1 = tmp1;
//@ assert lint != null;
/*@ assert
  lint instanceof Interpreter ==> lint.exp != null
  && true;
 @*/
lint.val = val1;
/*@ assert
  lint.val == val1
  && lint instanceof Interpreter ==> lint.exp != null;
 @*/

_Object tmp4;
int tmp3;
tmp4 = re.eval();

//@ assert tmp4 != null;
/*@ assert
    tmp4 instanceof Interpreter ==> tmp4.exp != null
    && true;
  @*/
```

```
tmp3 = tmp4.val;

int val2 = tmp3;
//@ assert rint != null;
/*@ assert
  rint instanceof Interpreter ==> rint.exp != null
  && true;
 @*/
rint.val = val2;
/*@ assert
  rint.val == val2
  && rint instanceof Interpreter ==> rint.exp != null;
 @*/

/*@ assert
          tmp3 == tmp4.val
      && tmp4 instanceof Interpreter ==> tmp4.exp != null;
    @*/

_Object tmp = new Integer();

int tmp5;

//@ assert lint != null;
/*@ assert
    lint instanceof Interpreter ==> lint.exp != null
    && true;
  @*/
tmp5 = lint.val;
/*@ assert
          tmp5 == lint.val
      && lint instanceof Interpreter ==> lint.exp != null;
    @*/

int tmp6;

//@ assert lint != null;
/*@ assert
    rint instanceof Interpreter ==> rint.exp != null
    && true;
  @*/
tmp6 = rint.val;
/*@ assert
          tmp6 == rint.val
      && rint instanceof Interpreter ==> rint.exp != null;
    @*/

int val = tmp5 + tmp6;

/*@ assert
   true;
  @*/
tmp.val = val;
/*@ assert
   tmp.val == val
   && true
   && true;
@*/
```

```
        return tmp;
      }
    }

  }

}

public class Expression extends _Object {
  public Expression () {}
}

public class BinaryExpression extends Expression {
  //@ initially this.leftExp != null && this.rightExp != null;

  public BinaryExpression() {
    this.leftExp = new Integer();
      this.rightExp = new Integer();
  }
}

public class Sum extends BinaryExpression {
  public Sum() {
  }
}

public class Value extends Expression {
  public Value() {}
}

public class Integer extends Value {
  public Integer() {
    this.val = 0;
  }
}

public class Main {

  public static void main(String[] args) {

    _Object in;
    _Object n1,n2;
    _Object s;
    _Object v;

    n1 = new Integer();
    int val = 5;
    /*@ assert
       true;
      @*/
    n1.val = val;
    /*@ assert
       n1.val == val
       && true
       && true;
    @*/

    n2 = new Integer();
```

```
    val = 3;
    /*@ assert
       true;
      @*/
    n2.val = val;
    /*@ assert
       n2.val == val
       && true
       && true;
    @*/

    s = new Sum();
    _Object leftExp = n1;
    _Object rightExp = n2;
    /*@ assert
       leftExp != null && rightExp != null;
      @*/
    /*@ assert
          leftExp != null && rightExp != null;
       @*/
    s.leftExp = leftExp;
    s.rightExp = rightExp;
    /*@ assert
       s.leftExp == leftExp && s.rightExp == rightExp
       &&  s.leftExp != null && s.rightExp != null;
       @*/

    /*@ assert
       s.leftExp == leftExp && s.rightExp == rightExp
       &&  s.leftExp != null && s.rightExp != null;
      @*/

    in = new Interpreter();
    _Object exp = s;
    /*@ assert
       exp != null;
      @*/
    in.exp = exp;
    /*@ assert
       in.exp == exp
       && in.exp != null;
       @*/

    //@ assert in != null;
    /*@ assert
       in instanceof Interpreter ==> in.exp != null;
     @*/
    v = in.exp.eval();
    /*@ assert
       in instanceof Interpreter ==> in.exp != null;
      @*/
  }
}
```

APPENDIX C

Source-code of the Meta Data API from the beginning to the end

This appendix show the source-code snapshots of the Meta Data API during the code and specification refactorings explained in Chapter 5.

C.1 Original source-code of Meta Data API

```
import java.util.HashSet;
import java.util.Set;

public class Main {

  public static void main (String[] args) {
    //Simulating a set of properties to a Product
    MetaData length = createLengthProperty();
    MetaData weigth = createWeigthProperty();
    MetaData code = createCodeProperty();
    MetaData productionDate = createProductionDateProperty();
    MetaData numberOfInternalParts =
        createNumberOfInternalPartsProperty();
    MetaData needsPacking = createNeedsPackingProperty();

    //Simulating data registering for Length
    DoubleData dataLength = new DoubleData(length);
    dataLength.registerValueFromText("10");
    System.out.println("data value: " + dataLength.getFormattedValue()
        +
        " is a " + dataLength.isValid() + " well-formed value as
            expected and " +
        "its expected status is " +
        "INVALID, the real status is: " + dataLength.getStatus());
    //@ assert dataLength.getStatus().equals(DataStatus.INVALID);

    //Simulating data registering for Weigth
    DoubleData dataWeigth = new DoubleData(weigth);
    dataWeigth.registerValueFromText("4");
    System.out.println("data value: " + dataWeigth.getFormattedValue()
        +
        " is a " + dataWeigth.isValid() + " well-formed value as
            expected and " +
```

```
        "its expected status is " +
        "VALID, the real status is: " + dataWeigth.getStatus());
    //@ assert dataWeigth.getStatus().equals(DataStatus.VALID);

    //Simulating data registering for Code
    StringData dataCode = new StringData(code);
    dataCode.registerValueFromText("XYZ001");
    System.out.println("data value: " + dataCode.getFormattedValue() +
        " is a " + dataCode.isValid() + " well-formed value as expected
            and " +
        "its expected status is " +
        "VALID, the real status is: " + dataCode.getStatus());
    //@ assert dataCode.getStatus().equals(DataStatus.VALID);

    DateData productionDateData = new DateData(productionDate);
    productionDateData.registerValueFromText("11-11-2008");
    System.out.println("data value: " + productionDateData.
       getFormattedValue() +
        " is a " + productionDateData.isValid() + " well-formed value
           as expected and " +
        "its expected status is " +
        "NOT_REGISTERED, the real status is: " + productionDateData.
           getStatus());
    //@ assert productionDateData.getStatus().equals(DataStatus.
       NOT_REGISTERED);

    IntegerData numberOfInternalPartsData = new IntegerData(
       numberOfInternalParts);
    numberOfInternalPartsData.registerValueFromText("5");
    System.out.println("data value: " + numberOfInternalPartsData.
       getFormattedValue() +
        " is a " + numberOfInternalPartsData.isValid() + " well-formed
           value as expected and " +
        "its expected status is " +
        "VALID, the real status is: " + numberOfInternalPartsData.
           getStatus());
    //@ assert numberOfInternalPartsData.getStatus().equals(DataStatus.
       VALID);

    BooleanData needsPackingData = new BooleanData(needsPacking);
    needsPackingData.registerValueFromText("true");
    System.out.println("data value: " + needsPackingData.
       getFormattedValue() +
        " is a " + needsPackingData.isValid() + " well-formed value as
           expected and " +
        "its expected status is " +
        "NOT_REGISTERED, the real status is: " + needsPackingData.
           getStatus());
    //@ assert needsPackingData.getStatus().equals(DataStatus.
       NOT_REGISTERED);
}

private static MetaData createLengthProperty() {
    MetaData length = new MetaData("length", DataType.DECIMAL);
    length.setDefaultValue("25");
    length.setUseDefaultValue(Boolean.valueOf(true));
    length.setMeasureUnit("cm");
    length.setDescription("Property to store product's length");
```

```
    Set rules = new HashSet();
    MaxValueRule maxValueRule = new MaxValueRule();
    maxValueRule.setPurpose(ValidationPurpose.VALIDATE);
    maxValueRule.setReferenceValue(new Double(30));
    rules.add(maxValueRule);
    MinValueRule minValueRule = new MinValueRule();
    minValueRule.setPurpose(ValidationPurpose.VALIDATE);
    minValueRule.setReferenceValue(new Double(20));
    rules.add(minValueRule);

    length.setValidationRules(rules);
    return length;
  }

  private static MetaData createWeigthProperty() {
    MetaData weigth = new MetaData("weight", DataType.DECIMAL);
    weigth.setMeasureUnit("kg");
    weigth.setUseDefaultValue(Boolean.valueOf(true));
    weigth.setDefaultValue("5");
    weigth.setDescription("Property to store product's weigth");

    Set rules = new HashSet();
    MinToleranceRule minToleranceRule = new MinToleranceRule();
    minToleranceRule.setPurpose(ValidationPurpose.VALIDATE);
    minToleranceRule.setReferenceValue(new Double(2));
    rules.add(minToleranceRule);

    weigth.setValidationRules(rules);
    return weigth;
  }

  private static MetaData createCodeProperty() {
    MetaData code = new MetaData ("code", DataType.TEXT);
    code.setDescription("Property to store product's code");

    Set rules = new HashSet();
    MaxSizeRule maxSizeRule = new MaxSizeRule();
    maxSizeRule.setPurpose(ValidationPurpose.VALIDATE);
    maxSizeRule.setReferenceValue(new Double(10));
    rules.add(maxSizeRule);
    code.setValidationRules(rules);
    return code;
  }

  private static MetaData createProductionDateProperty() {
    MetaData productionDate = new MetaData("Production date", DataType.
        DATE);
    productionDate.setDescription("Property to store product's
        production date");
    return productionDate;
  }

  private static MetaData createNumberOfInternalPartsProperty() {
    MetaData numberOfInternalParts = new MetaData("Number of internal
        parts", DataType.INTEGER);
    numberOfInternalParts.setDefaultValue("4");
    numberOfInternalParts.setUseDefaultValue(Boolean.valueOf(true));
```

```
    numberOfInternalParts.setDescription("Property to store the number
        of internal parts of the product");
    Set rules = new HashSet();
    MaxTolerancePercentageRule maxTolerancePercentageRule = new
        MaxTolerancePercentageRule();
    maxTolerancePercentageRule.setPurpose(ValidationPurpose.VALIDATE);
    maxTolerancePercentageRule.setReferenceValue(new Double(50));
    rules.add(maxTolerancePercentageRule);
    numberOfInternalParts.setValidationRules(rules);
    return numberOfInternalParts;
  }

  private static MetaData createNeedsPackingProperty() {
    MetaData needsPacking = new MetaData("Needs to Pack", DataType.
        BOOLEAN);
    needsPacking.setDefaultValue("false");
    needsPacking.setUseDefaultValue(Boolean.valueOf(true));
    needsPacking.setDescription("Property to store whether the product
        needs to be packed");
    return needsPacking;
  }
}


public final class DataStatus {
  public static final DataStatus NOT_REGISTERED = new DataStatus("
      NOT_REGISTERED", 1);
  public static final DataStatus INVALID = new DataStatus("INVALID", 2)
      ;
  public static final DataStatus VALID = new DataStatus("VALID", 3);
  private String name;
  private int number;

  private DataStatus(String name, int number) {
    this.name = name;
    this.number = number;
  }
  public String toString() {
    return this.name;
  }
}

public class DataType {

  public static final DataType INTEGER = new DataType("INTEGER", 0);
  public static final DataType DECIMAL = new DataType("DECIMAL", 1);
  public static final DataType BOOLEAN = new DataType("BOOLEAN", 2);
  public static final DataType DATE = new DataType("DATE", 3);
  public static final DataType TEXT = new DataType("TEXT", 4);
  private String name;
  private int number;

  private DataType(String name, int number) {
    this.name = name;
    this.number = number;
  }
}

import java.util.HashSet;
```

```
import java.util.Set;
public class MetaData {

  private String name;
  private String description;
  private Boolean useDefaultValue;
  private String defaultValue;
  private String measureUnit;
  private DataType dataType;
  private Set validationRules = new HashSet();

  public MetaData() {
  }

  public MetaData(String name, DataType dataType){
    this.name = name;
    this.dataType = dataType;
    this.description = "";
    this.defaultValue = "";
    this.useDefaultValue = Boolean.valueOf(false);
    this.measureUnit = "";
  }

  public /*@ pure @*/ String getName() {
    return this.name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public /*@ pure @*/ String getDescription() {
    return this.description;
  }
  public void setDescription(String description) {
    this.description = description;
  }
  public /*@ pure @*/ Boolean getUseDefaultValue() {
    return this.useDefaultValue;
  }
  public void setUseDefaultValue(Boolean useDefaultValue) {
    this.useDefaultValue = useDefaultValue;
  }
  public /*@ pure @*/ String getDefaultValue() {
    return this.defaultValue;
  }
  public void setDefaultValue(String defaultValue) {
    this.defaultValue = defaultValue;
  }
  public /*@ pure @*/ DataType getDataType() {
    return this.dataType;
  }
  public void setDataType(DataType dataType) {
    this.dataType = dataType;
  }
  public /*@ pure @*/ Set getValidationRules() {
    return this.validationRules;
  }
  public void setValidationRules(Set validationRules) {
    this.validationRules = validationRules;
```

```
  }
  public /*@ pure @*/ String getMeasureUnit() {
    return this.measureUnit;
  }
  public void setMeasureUnit(String measureUnit) {
    this.measureUnit = measureUnit;
  }
}

import java.util.Date;
import java.util.Iterator;

public class BooleanData {
  //@ invariant this.getMetaData() != null;

  private /*@ spec_public @*/ DataStatus status = DataStatus.
     NOT_REGISTERED;
  private MetaData metaData;
  private /*@ spec_public nullable @*/ Object value;
  private /*@ spec_public @*/ Date registeredDate = new Date();
  private Date editedDate = new Date();

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public BooleanData(MetaData metaData) {
    this.metaData = metaData;
    this.value = new Boolean(true);
  }

  /*@ pure nullable @*/ Object convertToValue(String value) {
    if ("true".equals(value)) {
      return Boolean.valueOf(true);
    } else if ("false".equals(value)) {
      return Boolean.valueOf(false);
    } else {
      return null;
    }
  }

  //@ requires this.getValue() != null;
  //@ assignable \nothing;
  //@ ensures Boolean.valueOf(\result).equals(this.getValue());
  public /*@ pure @*/ String getFormattedValue() {
    return this.getValue().toString();
  }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public /*@ pure @*/ boolean isValid() {
    if (this.value == null) {
      return false;
```

```
    }
    return true;
  }

  public /*@ pure @*/ Object getValue() { /* ... */ }

  //@ assignable this.registeredDate, this.status;
  private void doRegisterActions() { /* ... */}

  /*@ requires value.equals("true");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue().equals(Boolean.valueOf(true));
    @ also
    @ requires value.equals("false");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue().equals(Boolean.valueOf(false));
    @ also
    @ requires !value.equals("true") && !value.equals("false");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ */
  public void registerValueFromText(String value) {
    this.value = this.convertToValue(value);
    this.doRegisterActions();
  }

  /*@ requires value != null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == value;
    @*/
  public void registerValue(Object value) {
    this.value = value;
    this.doRegisterActions();
  }

  /*@ requires this.getValue() != null;
    @ assignable this.status;
    @ ensures true;
    @*/
  public void validate() {
    if (this.metaData.getValidationRules() != null && !this.metaData.
        getValidationRules().isEmpty()) {
      Iterator iter = this.metaData.getValidationRules().iterator();
      while (iter.hasNext()) {
        this.validateRule((AbstractValidationRule)iter.next());
      }
    }
  }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }

  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }
```

```
  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  //@ assignable this.status;
  private void validateRule(AbstractValidationRule rule) {
    if (rule.getPurpose().equals(ValidationPurpose.NONE)) {
      this.status = DataStatus.VALID;
    } else {
      this.checkValidationRule(rule);
    }
  }

  //@ assignable this.status;
  private void checkValidationRule(AbstractValidationRule rule) {
    if (!this.status.equals(DataStatus.INVALID)) {
      if (rule.validate(this)) {
        this.status = DataStatus.VALID;
      } else {
        this.status = DataStatus.INVALID;
      }
    }
  }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;

public class DateData {

  //@ invariant this.getMetaData() != null;

  private /*@ spec_public @*/ DataStatus status = DataStatus.
     NOT_REGISTERED;
  private MetaData metaData;
  private /*@ spec_public nullable @*/ Object value;
  private /*@ spec_public @*/ Date registeredDate = new Date();
  private Date editedDate = new Date();

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public DateData(MetaData metaData) {
    this.metaData = metaData;
    this.value = new Date();
  }

  /*@ pure nullable @*/ Object convertToValue(String value) {
    if (value == null || value.equals("")) {
      return null;
    }
    try {
```

```
    SimpleDateFormat sformat = new SimpleDateFormat("MM-dd-yyyy");
    return sformat.parse(value);
  } catch (ParseException p) {
    return null;
  }
}

/*@ requires this.getValue() != null;
  @ assignable \nothing;
  @ ensures !\result.equals("");
  @*/
public /*@ pure @*/ String getFormattedValue() {
  SimpleDateFormat sformat = new SimpleDateFormat("MM-dd-yyyy");
  String result = sformat.format((Date)this.value);
  return result;
}

/*@ requires this.getValue() == null;
  @ assignable \nothing;
  @ ensures \result == false;
  @ also
  @ requires this.getValue() != null;
  @ assignable \nothing;
  @ ensures \result == true;
  @*/
public /*@ pure @*/ boolean isValid() {
  if (this.value == null) {
    return false;
  }
  return true;
}

public /*@ pure @*/ Object getValue() { /* ... */ }

//@ assignable this.registeredDate, this.status;
private void doRegisterActions() { /* ... */ }

/*@ requires this.getValue() == null || this.getValue().equals("");
  @ assignable this.value, this.registeredDate, this.status;
  @ ensures this.getValue() == null;
  @ also
  @ requires (this.getValue() != null && !this.getValue().equals(""))
     ;
  @ assignable this.value, this.registeredDate, this.status;
  @ ensures (this.getValue() instanceof Date) || this.getValue() ==
     null;
  @*/
public void registerValueFromText(String value) { /* ... */ }


/*@ requires value != null;
  @ assignable this.value, this.registeredDate, this.status;
  @ ensures this.getValue() == value;
  @*/
public void registerValue(Object value) { /* ... */ }

/*@ requires this.getValue() != null;
  @ assignable this.status;
```

```
    @ ensures true;
    @*/
  public void validate() { /* ... */ }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }

  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }

  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  //@ assignable this.status;
  private void validateRule(AbstractValidationRule rule) { /* ... */ }

  //@ assignable this.status;
  private void checkValidationRule(AbstractValidationRule rule) { /* ..
     . */ }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

import java.text.NumberFormat;
import java.util.Date;
import java.util.Iterator;

public class DoubleData {

  //@ invariant this.getMetaData() != null;

  private /*@ spec_public @*/ DataStatus status = DataStatus.
     NOT_REGISTERED;
  private MetaData metaData;
  private /*@ spec_public nullable @*/ Object value;
  private /*@ spec_public @*/ Date registeredDate = new Date();
  private Date editedDate = new Date();

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public DoubleData(MetaData metaData) {
    this.metaData = metaData;
    this.value = new Double(0);
  }

  /*@ pure nullable @*/ Object convertToValue(String value) {
    if (value != null) {
      try {
        return Double.valueOf(value);
      } catch (NumberFormatException e) {
        return null;
```

```
      }
    }
    return null;
  }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures !\result.equals("");
    @*/
  public /*@ pure @*/ String getFormattedValue() {
    NumberFormat formatter = NumberFormat.getInstance();
    return formatter.format(this.getValue());
  }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public boolean isValid() { /* ... */ }

  public /*@ pure @*/ Object getValue() { /* ... */ }

  //@ assignable this.registeredDate, this.status;
  private void doRegisterActions() { /* ... */ }

  /*@ requires this.getValue() == null || this.getValue().equals("");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires (this.getValue() != null && !this.getValue().equals(""))
       ;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null || (this.getValue() instanceof
       Double);
    @*/
  public void registerValueFromText(String value) { /* ... */ }

  /*@ requires value != null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == value;
    @*/
  public void registerValue(Object value) { /* ... */ }

  /*@ requires this.getValue() != null;
    @ assignable this.status;
    @ ensures true;
    @*/
  public void validate() { /* ... */ }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }
```

```
  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }

  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  //@ assignable this.status;
  private void validateRule(AbstractValidationRule rule) { /* ... */ }

  //@ assignable this.status;
  private void checkValidationRule(AbstractValidationRule rule) { /* ..
     . */ }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

import java.util.Date;
import java.util.Iterator;

public class IntegerData {

  //@ invariant this.getMetaData() != null;

  private /*@ spec_public @*/ DataStatus status = DataStatus.
     NOT_REGISTERED;
  private MetaData metaData;
  private /*@ spec_public nullable @*/ Object value;
  private /*@ spec_public @*/ Date registeredDate = new Date();
  private Date editedDate = new Date();

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public IntegerData(MetaData metaData) {
    this.metaData = metaData;
    this.value = new Integer(0);
  }

  /*@ pure nullable @*/ Object convertToValue(String value) {
    try {
      return Integer.valueOf(value);
    } catch (NumberFormatException e) {
      return null;
    }
  }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures Integer.valueOf(\result).equals(this.getValue());
    @*/
  public /*@ pure @*/ String getFormattedValue() {
    return ((Integer) this.getValue()).toString();
  }
```

```
  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public boolean isValid() { /* ... */ }

  public /*@ pure @*/ Object getValue() {{ /* ... */ }

  //@ assignable this.registeredDate, this.status;
  private void doRegisterActions() { /* ... */ }

  /*@ requires this.getValue() == null || this.getValue().equals("");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires (this.getValue() != null && !this.getValue().equals(""))
       ;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null || ((this.getValue() instanceof
       Integer) ==> this.getValue().equals(Integer.valueOf(value)));
    @*/
  public void registerValueFromText(String value) { /* ... */ }

  /*@ requires value != null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == value;
    @*/
  public void registerValue(Object value) { /* ... */ }

  /*@ requires this.getValue() != null;
    @ assignable this.status;
    @ ensures true;
    @*/
  public void validate() { /* ... */ }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }

  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }

  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  //@ assignable this.status;
  private void validateRule(AbstractValidationRule rule) { /* ... */ }

  //@ assignable this.status;
```

```
  private void checkValidationRule(AbstractValidationRule rule) { /* ..
     . */ }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

import java.util.Date;
import java.util.Iterator;

public class StringData {

  //@ invariant this.getMetaData() != null;

  private /*@ spec_public @*/ DataStatus status = DataStatus.
     NOT_REGISTERED;
  private MetaData metaData;
  private /*@ spec_public nullable @*/ Object value;
  private /*@ spec_public @*/ Date registeredDate = new Date();
  private Date editedDate = new Date();

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public StringData(MetaData metaData) {
    this.value = "";
    this.metaData = metaData;
  }

  /*@ pure nullable @*/ Object convertToValue(String value) {
    if (value == null) {
      return null;
    }
    return (String) value;
  }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result.equals(this.getValue());
    @*/
  public /*@ pure @*/ String getFormattedValue() {
    return (String) this.value;
  }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures ((String)this.getValue()).length() < 255;
    @*/
  public /*@ pure @*/ boolean isValid() {
    return ((String)this.value).length() < 255;
  }

  public /*@ pure @*/ Object getValue() { /* ... */ }

  //@ assignable this.registeredDate, this.status;
```

```
  private void doRegisterActions() { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires !(this.getValue() == null);
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures value.equals(this.getValue());
    @*/
  public void registerValueFromText(String value) { /* ... */ }

  /*@ requires value != null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == value;
    @*/
  public void registerValue(Object value) { /* ... */ }

  /*@ requires this.getValue() != null;
    @ assignable this.status;
    @ ensures true;
    @*/
  public void validate() { /* ... */ }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }

  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }

  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  //@ assignable this.status;
  private void validateRule(AbstractValidationRule rule) { /* ... */ }

  //@ assignable this.status;
  private void checkValidationRule(AbstractValidationRule rule) { /* ..
     . */ }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

public class ValidationPurpose {
  public static final ValidationPurpose NONE = new ValidationPurpose("
     NONE", 0);
  public static final ValidationPurpose VALIDATE = new
     ValidationPurpose("VALIDATE", 0);
  private String name;
  private int number;

  private ValidationPurpose(String name, int number) {
```

```
    this.name = name;
    this.number = number;
  }
}

public final class ValidationType {
  public static final ValidationType REGULAR_EXPRESSION = new
      ValidationType("REGULAR_EXPRESSION", 0);
  public static final ValidationType MAX_SIZE = new ValidationType("
      MAX_SIZE", 1);
  public static final ValidationType MIN_TOLERANCE = new ValidationType
      ("MIN_TOLERANCE", 2);
  public static final ValidationType MAX_TOLERANCE = new ValidationType
      ("MAX_TOLERANCE", 3);
  public static final ValidationType MIN_TOLERANCE_PERCENTAGE = new
      ValidationType("MIN_TOLERANCE_PERCENTAGE", 4);
  public static final ValidationType MAX_TOLERANCE_PERCENTAGE = new
      ValidationType("MAX_TOLERANCE_PERCENTAGE", 5);
  public static final ValidationType MIN_VALUE = new ValidationType("
      MIN_VALUE", 6);
  public static final ValidationType MAX_VALUE = new ValidationType("
      MAX_VALUE", 7);
  public static final ValidationType NONE = new ValidationType("NONE",
      8);
  private String name;
  private int number;

  private ValidationType(String name, int number) {
    this.name = name;
    this.number = number;
  }
}

import java.util.regex.Pattern;

public class AbstractValidationRule {
  //@ invariant this.getPurpose() != null;
  private ValidationPurpose purpose;

  public AbstractValidationRule() {
    this.purpose = ValidationPurpose.NONE;
  }

  public  /*@ pure @*/ ValidationPurpose getPurpose() { /* ... */ }

  //@ requires true;
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  //@ requires purpose != null;
  public void setPurpose(ValidationPurpose purpose) { /* ... */ }

  //PAREI AQUI
  public boolean validate(Object data) {
    if (this.getType().equals(ValidationType.MAX_VALUE)) {
      return this.validateMaxValue(data);
    } else if (this.getType().equals(ValidationType.MIN_VALUE)) {
      return this.validateMinValue(data);
    } else if (this.getType().equals(ValidationType.MAX_TOLERANCE)) {
```

```
    return this.validateMaxTolerance(data);
  } else if (this.getType().equals(ValidationType.MIN_TOLERANCE)) {
    return this.validateMinTolerance(data);
  } else if (this.getType().equals(ValidationType.
     MAX_TOLERANCE_PERCENTAGE)) {
    return this.validateMaxTolerancePercentage(data);
  } else if (this.getType().equals(ValidationType.
     MIN_TOLERANCE_PERCENTAGE)) {
    return this.validateMinTolerancePercentage(data);
  } else if (this.getType().equals(ValidationType.REGULAR_EXPRESSION)
     ) {
    return this.validateRegularExpression(data);
  } else if (this.getType().equals(ValidationType.MAX_SIZE)) {
    return this.validateMaxSize(data);
  } else if (this.getType().equals(ValidationType.NONE)) {
    return true;
  } else {
    return false;
  }
}

/*@ requires (data instanceof IntegerData) && ((IntegerData) data).
    getMetaData().getDefaultValue() != null
    && ((IntegerData)data).getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)((IntegerData)data).getValue()).
     intValue() <=
      Integer.parseInt(((IntegerData) data).getMetaData().
         getDefaultValue()) +
      ((MaxToleranceRule)this).getReferenceValue().intValue() )
      && ((Integer)((IntegerData)data).getValue()).intValue() >=
      Integer.parseInt(((IntegerData) data).getMetaData().
         getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && ((DoubleData) data).
     getMetaData().getDefaultValue() != null
    && ((DoubleData)data).getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Double)((DoubleData)data).getValue()).
     intValue() <=
      Double.valueOf(((DoubleData) data).getMetaData().
         getDefaultValue()).doubleValue() +
      ((MaxToleranceRule)this).getReferenceValue().doubleValue() )
      && ((Double)((DoubleData)data).getValue()).doubleValue() >=
      Double.valueOf(((DoubleData) data).getMetaData().
         getDefaultValue()).doubleValue();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
     DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
  @
 */
private /*@ pure @*/ boolean validateMaxTolerance(Object data) {
  if (data instanceof IntegerData) {
    int defaultValue = Integer.parseInt(((IntegerData) data).
       getMetaData().getDefaultValue());
```

```
    boolean result = ((Integer)((IntegerData)data).getValue()).
        intValue() <= defaultValue + ((MaxToleranceRule)this).
        getReferenceValue().intValue();
    return result && ((Integer)((IntegerData)data).getValue()).
        intValue() >= defaultValue;
  } else if (data instanceof DoubleData) {
    double defaultValue = Double.valueOf(((DoubleData) data).
        getMetaData().getDefaultValue()).doubleValue();
    boolean result = ((Double)((DoubleData)data).getValue()).
        doubleValue() <= defaultValue + (((MaxToleranceRule)this).
        getReferenceValue()).doubleValue();
    return result && ((Double)((DoubleData)data).getValue()).
        doubleValue() >= defaultValue;
  }
  return false;
}

/*@ requires (data instanceof IntegerData) && ((IntegerData) data).
   getMetaData().getDefaultValue() != null
    && ((IntegerData)data).getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)((IntegerData)data).getValue()).
      intValue() >=
       Integer.parseInt(((IntegerData) data).getMetaData().
          getDefaultValue()) -
       ((MinToleranceRule)this).getReferenceValue().intValue() )
       && ((Integer)((IntegerData)data).getValue()).intValue() <=
       Integer.parseInt(((IntegerData) data).getMetaData().
          getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && ((DoubleData) data).
      getMetaData().getDefaultValue() != null
    && ((DoubleData)data).getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Double)((DoubleData)data).getValue()).
      doubleValue() >=
       Double.valueOf(((DoubleData) data).getMetaData().
          getDefaultValue()).doubleValue() -
       (((MinToleranceRule)this).getReferenceValue()).doubleValue() )
       && ((Double)((DoubleData)data).getValue()).doubleValue() <=
       Double.valueOf(((DoubleData) data).getMetaData().
          getDefaultValue()).doubleValue();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
      DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
 */
private /*@ pure @*/ boolean validateMinTolerance(Object data) {
  if (data instanceof IntegerData) {
    int defaultValue = Integer.parseInt(((IntegerData) data).
        getMetaData().getDefaultValue());
    boolean result = ((Integer)((IntegerData)data).getValue()).
        intValue() >= defaultValue - ((MinToleranceRule)this).
        getReferenceValue().intValue();
    return result && ((Integer)((IntegerData)data).getValue()).
        intValue() <= defaultValue;
  } else if (data instanceof DoubleData) {
```

```
    double defaultValue = Double.valueOf(((DoubleData) data).
        getMetaData().getDefaultValue()).doubleValue();
    boolean result = ((Double)((DoubleData)data).getValue()).
        doubleValue() >= defaultValue - (((MinToleranceRule)this).
        getReferenceValue()).doubleValue();
    return result && ((Double)((DoubleData)data).getValue()).
        doubleValue() <= defaultValue;
  }
  return false;
}

/*@ requires (data instanceof IntegerData) && ((IntegerData) data).
    getMetaData().getDefaultValue() != null
    && ((MaxTolerancePercentageRule)this).getReferenceValue() != null
        ;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)((IntegerData)data).getValue()).
      intValue() <=
      Integer.parseInt(((IntegerData) data).getMetaData().
          getDefaultValue()) +
      Integer.parseInt(((IntegerData) data).getMetaData().
          getDefaultValue()) *
      ((MaxTolerancePercentageRule)this).getReferenceValue().
          doubleValue() / 100 )
      && ((Integer)((IntegerData)data).getValue()).intValue() >=
      Integer.parseInt(((IntegerData) data).getMetaData().
          getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && ((DoubleData) data).
      getMetaData().getDefaultValue() != null
    && ((MaxTolerancePercentageRule)this).getReferenceValue() != null
        ;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)((IntegerData)data).getValue()).
      intValue() <=
      Double.valueOf(((DoubleData) data).getMetaData().
          getDefaultValue()).doubleValue() +
      Double.valueOf(((DoubleData) data).getMetaData().
          getDefaultValue()).doubleValue() *
      (((MaxTolerancePercentageRule)this).getReferenceValue()).
          doubleValue() / 100 ) &&
      ((Double)((DoubleData)data).getValue()).doubleValue() >=
      Double.valueOf(((DoubleData) data).getMetaData().
          getDefaultValue()).doubleValue();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
      DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
  @*/
private /*@ pure @*/ boolean validateMaxTolerancePercentage(Object
    data) {
  if (data instanceof IntegerData) {
    int defaultValue = Integer.parseInt(((IntegerData) data).
        getMetaData().getDefaultValue());
    boolean result = ((Integer)((IntegerData)data).getValue()).
        intValue() <=
```

```
      defaultValue + defaultValue * ((MaxTolerancePercentageRule)this
         ).getReferenceValue().doubleValue() / 100;
    return result && ((Integer)((IntegerData)data).getValue()).
       intValue() >= defaultValue;
  } else if (data instanceof DoubleData) {
    double defaultValue = Double.valueOf(((DoubleData) data).
       getMetaData().getDefaultValue()).doubleValue();
    boolean result = ((Double)((DoubleData)data).getValue()).
       doubleValue() <=
      defaultValue + defaultValue * (((MaxTolerancePercentageRule)
         this).getReferenceValue()).doubleValue() / 100;
    return result && ((Double)((DoubleData)data).getValue()).
       doubleValue() >= defaultValue;
  }
  return false;
}

/*@ requires (data instanceof IntegerData) && ((IntegerData) data).
   getMetaData().getDefaultValue() != null
       && ((MinTolerancePercentageRule)this).getReferenceValue() !=
          null;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)((IntegerData)data).getValue()).
     intValue() >=
        Integer.parseInt(((IntegerData) data).getMetaData().
           getDefaultValue()) -
        Integer.parseInt(((IntegerData) data).getMetaData().
           getDefaultValue()) *
        ((MinTolerancePercentageRule)this).getReferenceValue().
           doubleValue() / 100 )
        && ((Integer)((IntegerData)data).getValue()).intValue()
        <= Integer.parseInt(((IntegerData) data).getMetaData().
           getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && ((DoubleData) data).
     getMetaData().getDefaultValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Double)((DoubleData)data).getValue()).
     doubleValue() >=
      Double.valueOf(((DoubleData) data).getMetaData().
         getDefaultValue()).doubleValue() -
      Double.valueOf(((DoubleData) data).getMetaData().
         getDefaultValue()).doubleValue() *
      (((MinTolerancePercentageRule)this).getReferenceValue()).
         doubleValue() / 100 )
      && ((Double)((DoubleData)data).getValue()).doubleValue()
      <= Double.valueOf(((DoubleData) data).getMetaData().
         getDefaultValue()).doubleValue();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
     DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
  @*/
private /*@ pure @*/ boolean validateMinTolerancePercentage(Object
   data) {
  if (data instanceof IntegerData) {
```

```
      int defaultValue = Integer.parseInt(((IntegerData) data).
         getMetaData().getDefaultValue());
      boolean result = ((Integer)((IntegerData)data).getValue()).
         intValue() >=
        defaultValue - defaultValue * ((MinTolerancePercentageRule)this
           ).getReferenceValue().doubleValue() / 100;
      return result && ((Integer)((IntegerData)data).getValue()).
         intValue() <= defaultValue;
    } else if (data instanceof DoubleData) {
      double defaultValue = Double.valueOf(((DoubleData) data).
         getMetaData().getDefaultValue()).doubleValue();
      boolean result = ((Double)((DoubleData)data).getValue()).
         doubleValue() >=
        defaultValue - defaultValue * (((MinTolerancePercentageRule)
           this).getReferenceValue()).doubleValue() / 100;
      return result && ((Double)((DoubleData)data).getValue()).
         doubleValue() <= defaultValue;
    }
    return false;
  }

  /*@ requires (data instanceof StringData);
    @ assignable \not_specified;
    @ ensures true || false;
    @ also
    @ requires !(data instanceof StringData);
    @ assignable \not_specified;
    @ ensures true;
   */
  private boolean validateRegularExpression(Object data) {
    boolean result = true;
    if (data instanceof StringData) {
      Pattern regexPattern;
      if (((RegularExpressionRule)this).getIgnoreCase().booleanValue())
          {
        regexPattern = Pattern.compile(((RegularExpressionRule)this).
           getValidValue(), Pattern.CASE_INSENSITIVE);
      } else {
        regexPattern = Pattern.compile(((RegularExpressionRule)this).
           getValidValue());
      }
      result = regexPattern.matcher(((String)((StringData)data).
         getValue())).matches();
    }
    return result;
  }

  /*@ requires (data instanceof StringData) && ((StringData)data).
     getValue() != null
     && ((MaxSizeRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @ ensures \result == ((String)((StringData)data).getValue()).length
       () <= ((MaxSizeRule)this).getReferenceValue().intValue();
    @ also
    @ requires !(data instanceof StringData);
    @ assignable \nothing;
    @ ensures \result == false;
    @*/
```

```
private /*@ pure @*/ boolean validateMaxSize(Object data) {
  if (data instanceof StringData) {
    if (((StringData)data).getValue() != null) {
      return ((String)((StringData)data).getValue()).length() <= ((
          MaxSizeRule)this).getReferenceValue().intValue();
    }
  }
  return false;
}

/*@ requires (data instanceof DoubleData) && ((DoubleData)data).
   getValue() != null && ((MaxValueRule)this).getReferenceValue() !=
    null;
  @ assignable \nothing;
  @ ensures \result == (((MaxValueRule)this).getReferenceValue().
     compareTo(((DoubleData)data).getValue()) >= 0);
  @ also
  @ requires (data instanceof IntegerData) && ((IntegerData)data).
     getValue() != null && ((MaxValueRule)this).getReferenceValue()
     != null;
  @ assignable \nothing;
  @ ensures \result == ((Integer)((IntegerData)data).getValue()).
     intValue() >= ((MaxValueRule)this).getReferenceValue().intValue
     ();
  @ also
  @ requires (!(data instanceof DoubleData) && !(data instanceof
     IntegerData));
  @ assignable \nothing;
  @ ensures \result == false;
  @*/
private /*@ pure @*/ boolean validateMaxValue(Object data) {
  if (data instanceof DoubleData) {
    if (((MaxValueRule)this).getReferenceValue().compareTo(((
        DoubleData)data).getValue()) >= 0) {
      return true;
    } else if (data instanceof IntegerData) {
      return ((Integer)((IntegerData)data).getValue()).intValue() >=
          ((MaxValueRule)this).getReferenceValue().intValue();
    }
  }
  return false;
}

/*@ requires (data instanceof DoubleData) && ((DoubleData)data).
   getValue() != null && ((MinValueRule)this).getReferenceValue() !=
    null;
  @ assignable \nothing;
  @ ensures \result == (((MinValueRule)this).getReferenceValue().
     compareTo(((DoubleData)data).getValue()) <= 0);
  @ also
  @ requires (data instanceof IntegerData) && ((IntegerData)data).
     getValue() != null && ((MinValueRule)this).getReferenceValue()
     != null;
  @ assignable \nothing;
  @ ensures \result == ((Integer)((IntegerData)data).getValue()).
     intValue() <= ((MinValueRule)this).getReferenceValue().intValue
     ();
  @ also
```

```
    @ requires (!(data instanceof DoubleData) && !(data instanceof
        IntegerData));
    @ assignable \nothing;
    @ ensures \result == false;
    @*/
  private /*@ pure @*/ boolean validateMinValue(Object data) {
    if (data instanceof DoubleData) {
      if (((MinValueRule)this).getReferenceValue().compareTo(((
          DoubleData)data).getValue()) <= 0) {
        return true;
      }
    } else if (data instanceof IntegerData) {
      return ((Integer)((IntegerData)data).getValue()).intValue() <= ((
          MinValueRule)this).getReferenceValue().intValue();
    }
    return false;
  }
}

public class MaxSizeRule  extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxSizeRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_SIZE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue)  { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue()  { /* ... */ }
}

public class MaxValueRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxValueRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_VALUE);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.MAX_SIZE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }
```

```
  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MinValueRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinValueRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MIN_VALUE);
    @*/
  public /*@ pure @*/ ValidationType getType()  {
    return ValidationType.MIN_VALUE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MaxToleranceRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxToleranceRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_TOLERANCE);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.MAX_TOLERANCE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MinToleranceRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinToleranceRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
```

```
    @ ensures \result.equals(ValidationType.MIN_TOLERANCE);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.MIN_TOLERANCE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MaxTolerancePercentageRule extends AbstractValidationRule
    {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxTolerancePercentageRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_TOLERANCE_PERCENTAGE);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.MAX_TOLERANCE_PERCENTAGE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MinTolerancePercentageRule extends AbstractValidationRule
    {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinTolerancePercentageRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MIN_TOLERANCE_PERCENTAGE);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.MIN_TOLERANCE_PERCENTAGE;
  }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class RegularExpressionRule extends AbstractValidationRule {
```

```
  private String validValue;
  private Boolean ignoreCase;

  //@ ensures this.getValidValue() != null && this.getIgnoreCase() !=
      null;
  public RegularExpressionRule() {
    this.validValue = "";
    this.ignoreCase = Boolean.valueOf(true);
  }

  public /*@ pure @*/ String getValidValue() { /* ... */ }

  public void setValidValue(String validValue) { /* ... */ }

  public /*@ pure @*/ Boolean getIgnoreCase() { /* ... */ }

  public void setIgnoreCase(Boolean ignoreCase) { /* ... */ }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.REGULAR_EXPRESSION);
    @*/
  public /*@ pure @*/ ValidationType getType() {
    return ValidationType.REGULAR_EXPRESSION;
  }
}
```

C.2 Classes after extracting superclass *Data*

```
import java.util.Date;
import java.util.Iterator;

public class Data {

  //@ invariant this.getMetaData() != null;

  /*@ spec_public nullable @*/ DataStatus status = DataStatus.
      NOT_REGISTERED;
  /*@ spec_public nullable @*/ MetaData metaData;
  /*@ spec_public nullable @*/ Object value;
  /*@ spec_public nullable @*/ Date registeredDate = new Date();
  /*@ spec_public nullable @*/ Date editedDate = new Date();

    /*@ requires this.getValue() != null;
    @ assignable this.status;
    @ ensures true;
    @*/
  public void validate() { /* ... */ }

  //@ assignable this.registeredDate, this.status;
  public void doRegisterActions() { /* ... */ }

  //@ assignable this.status;
  public void validateRule(AbstractValidationRule rule) { /* ... */ }

  //@ assignable this.status;
  public void checkValidationRule(AbstractValidationRule rule) {
```

```
    if (!this.status.equals(DataStatus.INVALID)) {
      if (rule.validate((Data)this)) {
        this.status = DataStatus.VALID;
      } else {
        this.status = DataStatus.INVALID;
      }
    }
  }

  /*@ requires value != null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == value;
    @*/
  public void registerValue(Object value) { /* ... */ }

  public void setMetaData(MetaData metaData) { /* ... */ }

  //@ assignable this.registeredDate;
  public void setRegisteredDate(Date registeredDate) { /* ... */ }

  public void setEditedDate(Date editedDate) { /* ... */ }

  public void setValue(Object value) { /* ... */ }

  public /*@ pure @*/ Object getValue() { /* ... */ }

  public /*@ pure @*/ MetaData getMetaData() { /* ... */ }

  public /*@ pure @*/ Date getRegisteredDate() { /* ... */ }

  public /*@ pure @*/ Date getEditedDate() { /* ... */ }

  public /*@ pure @*/ DataStatus getStatus() { /* ... */ }
}

public class BooleanData extends Data {

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public BooleanData(MetaData metaData) { /* ... */  }

  /*@ requires value.equals("true");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue().equals(Boolean.valueOf(true));
    @ also
    @ requires value.equals("false");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue().equals(Boolean.valueOf(false));
    @ also
    @ requires !value.equals("true") && !value.equals("false");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ */
  public void registerValueFromText(String value) { /* ... */ }

  /*@ pure nullable @*/ Object convertToValue(String value) { /* ... */
      }
```

```
  //@ requires this.getValue() != null;
  //@ assignable \nothing;
  //@ ensures Boolean.valueOf(\result).equals(this.getValue());
  public /*@ pure @*/ String getFormattedValue() { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public /*@ pure @*/ boolean isValid() { /* ... */ }
}

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateData extends Data {
  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public DateData(MetaData metaData) { /* ... */ }

  /*@ pure nullable @*/ Object convertToValue(String value)  { /* ...
     */ }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures !\result.equals("");
    @*/
  public /*@ pure @*/ String getFormattedValue()  { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public /*@ pure @*/ boolean isValid()  { /* ... */ }

  /*@ requires this.getValue() == null || this.getValue().equals("");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires (this.getValue() != null && !this.getValue().equals(""))
       ;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures (this.getValue() instanceof Date) || this.getValue() ==
       null;
    @*/
  public void registerValueFromText(String value)  { /* ... */ }
}
```

```
import java.text.NumberFormat;
public class DoubleData extends Data {

  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public DoubleData(MetaData metaData) { /* ... */ }

  /*@ pure nullable @*/ Object convertToValue(String value) { /* ... */
      }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures !\result.equals("");
    @*/
  public /*@ pure @*/ String getFormattedValue() { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public boolean isValid() { /* ... */ }

  /*@ requires this.getValue() == null || this.getValue().equals("");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires (this.getValue() != null && !this.getValue().equals(""))
      ;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null || (this.getValue() instanceof
      Double);
    @*/
  public void registerValueFromText(String value) { /* ... */ }
}

public class IntegerData extends Data {
  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public IntegerData(MetaData metaData) {
    this.metaData = metaData;
    this.value = new Integer(0);
  }

  /*@ pure nullable @*/ Object convertToValue(String value) { /* ... */
      }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures Integer.valueOf(\result).equals(this.getValue());
    @*/
  public /*@ pure @*/ String getFormattedValue() { /* ... */ }
```

```
  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == true;
    @*/
  public boolean isValid() { /* ... */ }

  /*@ requires this.getValue() == null || this.getValue().equals("");
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires (this.getValue() != null && !this.getValue().equals(""))
      ;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null || ((this.getValue() instanceof
      Integer) ==> this.getValue().equals(Integer.valueOf(value)));
    @*/
  public void registerValueFromText(String value) { /* ... */ }
}

public class StringData extends Data {
  /*@ requires metaData != null;
    @ ensures this.getMetaData() == metaData;
    @*/
  public StringData(MetaData metaData) { /* ... */ }

  /*@ pure nullable @*/ Object convertToValue(String value) { /* ... */
      }

  /*@ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures \result.equals(this.getValue());
    @*/
  public /*@ pure @*/ String getFormattedValue() { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable \nothing;
    @ ensures \result == false;
    @ also
    @ requires this.getValue() != null;
    @ assignable \nothing;
    @ ensures ((String)this.getValue()).length() < 255;
    @*/
  public /*@ pure @*/ boolean isValid() { /* ... */ }

  /*@ requires this.getValue() == null;
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures this.getValue() == null;
    @ also
    @ requires !(this.getValue() == null);
    @ assignable this.value, this.registeredDate, this.status;
    @ ensures value.equals(this.getValue());
    @*/
  public void registerValueFromText(String value) { /* ... */ }
```

```
}
```

C.3 Validation rules classes after *Replace Conditional with Polymorphism*

```
public class AbstractValidationRule {
  //@ invariant this.getPurpose() != null;
  private ValidationPurpose purpose;

  public AbstractValidationRule() { /* ... */ }

  public  /*@ pure @*/ ValidationPurpose getPurpose() { /* ... */ }

  //@ requires true;
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  //@ requires purpose != null;
  public void setPurpose(ValidationPurpose purpose) { /* ... */ }

  public boolean validate(Data data) { /* unused method */ }
}

public class MaxSizeRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxSizeRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_SIZE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }

    /*@ requires (data instanceof StringData) && ((StringData)data).
        getValue() != null
      && ((MaxSizeRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @  ensures \result == ((String)((StringData)data).getValue()).
        length() <= ((MaxSizeRule)this).getReferenceValue().intValue();
    @ also
    @ requires !(data instanceof StringData);
    @ assignable \nothing;
    @ ensures \result == false;
    @*/
  private /*@ pure @*/ boolean validateMaxSize(Data data) {
    if (data instanceof StringData) {
      if (((StringData)data).getValue() != null) {
        return ((String)((StringData)data).getValue()).length() <= ((
            MaxSizeRule)this).getReferenceValue().intValue();
```

```
      }
    }
    return false;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return  this.validateMaxSize(data);
  }
}

public class MaxValueRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxValueRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_VALUE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }


  /*@ requires (data instanceof DoubleData) && data.getValue() != null
     && ((MaxValueRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @ ensures \result == (((MaxValueRule)this).getReferenceValue().
       compareTo(data.getValue()) >= 0);
    @ also
    @ requires (data instanceof IntegerData) && data.getValue() != null
       && ((MaxValueRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @ ensures \result == ((Integer)data.getValue()).intValue() >= ((
       MaxValueRule)this).getReferenceValue().intValue();
    @ also
    @ requires (!(data instanceof DoubleData) && !(data instanceof
       IntegerData));
    @ assignable \nothing;
    @ ensures \result == false;
    @*/
  private /*@ pure @*/ boolean validateMaxValue(Data data) {
    if (data instanceof DoubleData) {
      if (((MaxValueRule)this).getReferenceValue().compareTo(data.
         getValue()) >= 0) {
        return true;
      } else if (data instanceof IntegerData) {
        return ((Integer)data.getValue()).intValue() >= ((MaxValueRule)
           this).getReferenceValue().intValue();
      }
    }
```

```
    return false;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateMaxValue(data);
  }
}

public class MinValueRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinValueRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MIN_VALUE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }

  /*@ requires (data instanceof DoubleData) && data.getValue() != null
     && ((MinValueRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @ ensures \result == (((MinValueRule)this).getReferenceValue().
       compareTo(data.getValue()) <= 0);
    @ also
    @ requires (data instanceof IntegerData) && data.getValue() != null
       && ((MinValueRule)this).getReferenceValue() != null;
    @ assignable \nothing;
    @ ensures \result == ((Integer)data.getValue()).intValue() <= ((
       MinValueRule)this).getReferenceValue().intValue();
    @ also
    @ requires (!(data instanceof DoubleData) && !(data instanceof
       IntegerData));
    @ assignable \nothing;
    @ ensures \result == false;
    @*/
  private /*@ pure @*/ boolean validateMinValue(Data data) {
    if (data instanceof DoubleData) {
      if (((MinValueRule)this).getReferenceValue().compareTo(data.
         getValue()) <= 0) {
        return true;
      }
    } else if (data instanceof IntegerData) {
      return ((Integer)data.getValue()).intValue() <= ((MinValueRule)
         this).getReferenceValue().intValue();
    }
    return false;
  }
```

```
  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateMinValue(data);
  }
}

public class MaxToleranceRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxToleranceRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_TOLERANCE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
  }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && data.getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)data.getValue()).intValue() <=
      Integer.parseInt(data.getMetaData().getDefaultValue()) +
      ((MaxToleranceRule)this).getReferenceValue().intValue() )
      && ((Integer)data.getValue()).intValue() >=
      Integer.parseInt(data.getMetaData().getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && data.getMetaData().
      getDefaultValue() != null
      && data.getValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Double)data.getValue()).intValue() <=
      Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
          () +
      ((MaxToleranceRule)this).getReferenceValue().doubleValue() )
      && ((Double)data.getValue()).doubleValue() >=
      Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
          ();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
      DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
  @
  @*/
  private /*@ pure @*/ boolean validateMaxTolerance(Data data) {
    if (data instanceof IntegerData) {
      int defaultValue = Integer.parseInt(data.getMetaData().
          getDefaultValue());
```

```
      boolean result = ((Integer)data.getValue()).intValue() <=
          defaultValue + ((MaxToleranceRule)this).getReferenceValue().
          intValue();
      return result && ((Integer)data.getValue()).intValue() >=
          defaultValue;
    } else if (data instanceof DoubleData) {
      double defaultValue = Double.valueOf(data.getMetaData().
          getDefaultValue()).doubleValue();
      boolean result = ((Double)data.getValue()).doubleValue() <=
          defaultValue + (((MaxToleranceRule)this).getReferenceValue())
          .doubleValue();
      return result && ((Double)data.getValue()).doubleValue() >=
          defaultValue;
    }
    return false;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateMaxTolerance(data);
  }
}

public class MinToleranceRule extends AbstractValidationRule {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinToleranceRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MIN_TOLERANCE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && data.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == ( ((Integer)data.getValue()).intValue() >=
        Integer.parseInt(data.getMetaData().getDefaultValue()) -
        ((MinToleranceRule)this).getReferenceValue().intValue() )
        && ((Integer)data.getValue()).intValue() <=
        Integer.parseInt(data.getMetaData().getDefaultValue());
    @ also
    @ requires (data instanceof DoubleData) && data.getMetaData().
       getDefaultValue() != null
      && data.getValue() != null;
    @ assignable \nothing;
    @ ensures \result == ( ((Double)data.getValue()).doubleValue() >=
```

```
        Double.valueOf(data.getMetaData().getDefaultValue()).
            doubleValue() -
        (((MinToleranceRule)this).getReferenceValue()).doubleValue() )
        && ((Double)data.getValue()).doubleValue() <=
        Double.valueOf(data.getMetaData().getDefaultValue()).
            doubleValue();
    @ also
    @ requires !(data instanceof IntegerData) && !(data instanceof
        DoubleData);
    @ assignable \nothing;
    @ ensures \result == false;
   */
  private /*@ pure @*/ boolean validateMinTolerance(Data data) {
    if (data instanceof IntegerData) {
      int defaultValue = Integer.parseInt(data.getMetaData().
          getDefaultValue());
      boolean result = ((Integer)data.getValue()).intValue() >=
          defaultValue - ((MinToleranceRule)this).getReferenceValue().
          intValue();
      return result && ((Integer)data.getValue()).intValue() <=
          defaultValue;
    } else if (data instanceof DoubleData) {
      double defaultValue = Double.valueOf(data.getMetaData().
          getDefaultValue()).doubleValue();
      boolean result = ((Double)data.getValue()).doubleValue() >=
          defaultValue - (((MinToleranceRule)this).getReferenceValue())
          .doubleValue();
      return result && ((Double)data.getValue()).doubleValue() <=
          defaultValue;
    }
    return false;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateMinTolerance(data);
  }
}

public class MaxTolerancePercentageRule extends AbstractValidationRule
    {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MaxTolerancePercentageRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MAX_TOLERANCE_PERCENTAGE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
```

```
/*@ requires (data instanceof IntegerData) && data.getMetaData().
    getDefaultValue() != null
    && ((MaxTolerancePercentageRule)this).getReferenceValue() != null
        ;
@ assignable \nothing;
@ ensures \result == ( ((Integer)data.getValue()).intValue() <=
    Integer.parseInt(data.getMetaData().getDefaultValue()) +
    Integer.parseInt(data.getMetaData().getDefaultValue()) *
    ((MaxTolerancePercentageRule)this).getReferenceValue().
        doubleValue() / 100 )
    && ((Integer)data.getValue()).intValue() >=
    Integer.parseInt(data.getMetaData().getDefaultValue());
@ also
@ requires (data instanceof DoubleData) && data.getMetaData().
    getDefaultValue() != null
    && ((MaxTolerancePercentageRule)this).getReferenceValue() != null
        ;
@ assignable \nothing;
@ ensures \result == ( ((Integer)data.getValue()).intValue() <=
    Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
        () +
    Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
        () *
    (((MaxTolerancePercentageRule)this).getReferenceValue()).
        doubleValue() / 100 ) &&
    ((Double)data.getValue()).doubleValue() >=
    Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
        ();
@ also
@ requires !(data instanceof IntegerData) && !(data instanceof
    DoubleData);
@ assignable \nothing;
@ ensures \result == false;
@*/
private /*@ pure @*/ boolean validateMaxTolerancePercentage(Data data
    ) {
  if (data instanceof IntegerData) {
    int defaultValue = Integer.parseInt(data.getMetaData().
        getDefaultValue());
    boolean result = ((Integer)data.getValue()).intValue() <=
      defaultValue + defaultValue * ((MaxTolerancePercentageRule)this
          ).getReferenceValue().doubleValue() / 100;
    return result && ((Integer)data.getValue()).intValue() >=
        defaultValue;
  } else if (data instanceof DoubleData) {
    double defaultValue = Double.valueOf(data.getMetaData().
        getDefaultValue()).doubleValue();
    boolean result = ((Double)data.getValue()).doubleValue() <=
      defaultValue + defaultValue * (((MaxTolerancePercentageRule)
          this).getReferenceValue()).doubleValue() / 100;
    return result && ((Double)data.getValue()).doubleValue() >=
        defaultValue;
  }
  return false;
}

public boolean validate(Data data) {
```

```
    /*@ assert (data instanceof Data); @*/
    return this.validateMaxTolerancePercentage(data);
  }
}

public class MinTolerancePercentageRule extends AbstractValidationRule
    {
  //@ invariant this.getReferenceValue() != null;
  private Double referenceValue;

  public MinTolerancePercentageRule() {
    this.referenceValue = new Double(0);
  }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.MIN_TOLERANCE_PERCENTAGE);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }


  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && ((MinTolerancePercentageRule)this).getReferenceValue() != null
          ;
  @ assignable \nothing;
  @ ensures \result == ( ((Integer)data.getValue()).intValue() >=
        Integer.parseInt(data.getMetaData().getDefaultValue()) -
        Integer.parseInt(data.getMetaData().getDefaultValue()) *
        ((MinTolerancePercentageRule)this).getReferenceValue().
           doubleValue() / 100 )
       && ((Integer)data.getValue()).intValue()
       <= Integer.parseInt(data.getMetaData().getDefaultValue());
  @ also
  @ requires (data instanceof DoubleData) && data.getMetaData().
     getDefaultValue() != null;
  @ assignable \nothing;
  @ ensures \result == ( ((Double)data.getValue()).doubleValue() >=
      Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
          () -
      Double.valueOf(data.getMetaData().getDefaultValue()).doubleValue
          () *
      (((MinTolerancePercentageRule)this).getReferenceValue()).
          doubleValue() / 100 )
      && ((Double)data.getValue()).doubleValue()
      <= Double.valueOf(data.getMetaData().getDefaultValue()).
          doubleValue();
  @ also
  @ requires !(data instanceof IntegerData) && !(data instanceof
     DoubleData);
  @ assignable \nothing;
  @ ensures \result == false;
  @*/
```

```
  private /*@ pure @*/ boolean validateMinTolerancePercentage(Data data
      ) {
    if (data instanceof IntegerData) {
      int defaultValue = Integer.parseInt(data.getMetaData().
          getDefaultValue());
      boolean result = ((Integer)data.getValue()).intValue() >=
        defaultValue - defaultValue * ((MinTolerancePercentageRule)this
            ).getReferenceValue().doubleValue() / 100;
      return result && ((Integer)data.getValue()).intValue() <=
          defaultValue;
    } else if (data instanceof DoubleData) {
      double defaultValue = Double.valueOf(data.getMetaData().
          getDefaultValue()).doubleValue();
      boolean result = ((Double)data.getValue()).doubleValue() >=
        defaultValue - defaultValue * (((MinTolerancePercentageRule)
            this).getReferenceValue()).doubleValue() / 100;
      return result && ((Double)data.getValue()).doubleValue() <=
          defaultValue;
    }
    return false;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateMinTolerancePercentage(data);
  }
}

import java.util.regex.Pattern;

public class RegularExpressionRule extends AbstractValidationRule {
  private String validValue;
  private Boolean ignoreCase;

  //@ ensures this.getValidValue() != null && this.getIgnoreCase() !=
      null;
  public RegularExpressionRule() {
    this.validValue = "";
    this.ignoreCase = Boolean.valueOf(true);
  }

  public /*@ pure @*/ String getValidValue() { /* ... */ }

  public void setValidValue(String validValue) { /* ... */ }

  public /*@ pure @*/ Boolean getIgnoreCase() { /* ... */ }

  public void setIgnoreCase(Boolean ignoreCase) { /* ... */ }

  /*@ also
    @ requires \same;
    @ assignable \nothing;
    @ ensures \result.equals(ValidationType.REGULAR_EXPRESSION);
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof StringData);
    @ assignable \not_specified;
```

```
    @ ensures true || false;
    @ also
    @ requires !(data instanceof StringData);
    @ assignable \not_specified;
    @ ensures true;
    */
  private boolean validateRegularExpression(Data data) {
    boolean result = true;
    if (data instanceof StringData) {
      Pattern regexPattern;
      if (((RegularExpressionRule)this).getIgnoreCase().booleanValue())
          {
        regexPattern = Pattern.compile(((RegularExpressionRule)this).
            getValidValue(), Pattern.CASE_INSENSITIVE);
      } else {
        regexPattern = Pattern.compile(((RegularExpressionRule)this).
            getValidValue());
      }
      result = regexPattern.matcher(((String)((StringData)data).
          getValue())).matches();
    }
    return result;
  }

  public boolean validate(Data data) {
    /*@ assert (data instanceof Data); @*/
    return this.validateRegularExpression(data);
  }
}
```

C.4 Number-based validation rules classes after extracting superclass *AbstractNumberValidationRule*

```
public class AbstractNumberValidationRule extends
   AbstractValidationRule {
  //@ invariant this.getPurpose() != null;
  /*@ nullable @*/ Double referenceValue;

  public void setReferenceValue(Double referenceValue) { /* ... */ }

  public /*@ pure @*/ Double getReferenceValue() { /* ... */ }
}

public class MaxSizeRule  extends AbstractNumberValidationRule {
  public MaxSizeRule() { /* ... */ }

  /*@ also
    @ requires \same;
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

   /*@ requires (data instanceof StringData) && ((StringData)data).
       getValue() != null
      && ((MaxSizeRule)this).getReferenceValue() != null;
    @ ...
    @*/
```

```
  private /*@ pure @*/ boolean validateMaxSize(Data data) { /* ... */ }

  public boolean validate(Data data) { /* ... */ }
}

public class MaxValueRule extends AbstractNumberValidationRule {
  public MaxValueRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof DoubleData) && data.getValue() != null
     && ((MaxValueRule)this).getReferenceValue() != null;
    @ ...
    @*/
  private /*@ pure @*/ boolean validateMaxValue(Data data) { /* ... */
     }

  public boolean validate(Data data) { /* ... */ }
}

public class MinValueRule extends AbstractNumberValidationRule {
  public MinValueRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof DoubleData) && data.getValue() != null
     && ((MinValueRule)this).getReferenceValue() != null;
    @ ...
    @*/
  private /*@ pure @*/ boolean validateMinValue(Data data) { /* ... */
     }

  public boolean validate(Data data) { /* ... */ }
}

public class MaxToleranceRule extends AbstractNumberValidationRule {
  public MaxToleranceRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
     getDefaultValue() != null
      && data.getValue() != null;
  @ ...
  @*/
  private /*@ pure @*/ boolean validateMaxTolerance(Data data) { /* ...
      */  }

  public boolean validate(Data data) { /* ... */ }
```

```
}

public class MinToleranceRule extends AbstractNumberValidationRule {
  public MinToleranceRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && data.getValue() != null;
    @ ...
    @*/
  private /*@ pure @*/ boolean validateMinTolerance(Data data) { /* ...
      */ }

  public boolean validate(Data data) { /* ... */ }
}

public class MaxTolerancePercentageRule extends
    AbstractNumberValidationRule {
  public MaxTolerancePercentageRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && ((MaxTolerancePercentageRule)this).getReferenceValue() != null
          ;
  @ ...
  @*/
  private /*@ pure @*/ boolean validateMaxTolerancePercentage(Data data
      ) { /* ... */ }

  public boolean validate(Data data) { /* ... */ }
}

public class MinTolerancePercentageRule extends
    AbstractNumberValidationRule {
  public MinTolerancePercentageRule() { /* ... */ }

  /*@ also
    @ ...
    @*/
  public /*@ pure @*/ ValidationType getType() { /* ... */ }

  /*@ requires (data instanceof IntegerData) && data.getMetaData().
      getDefaultValue() != null
      && ((MinTolerancePercentageRule)this).getReferenceValue() != null
          ;
  @ ...
  @*/
```

```
  private /*@ pure @*/ boolean validateMinTolerancePercentage(Data data
      ) { /* ... */ }

  public boolean validate(Data data) { /* ... */ }
}
```

C.5 The new validation rule class: *NotNullRule*

```
public class NotNullRule extends AbstractValidationRule {

  /*@ also
    @ requires true;
    @ assignable \not_specified;
    @ ensures data != null && data.getValue() != null;
    @*/
  public boolean validate(Data data) {
    return data != null && data.getValue() != null;
  }
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo