On the Theoretical Foundations of LePUS3 and its Application to Object-Oriented Design Verification

Jonathan Nicholson

A thesis submitted for the degree of Doctor of Philosophy

School of Computer Science and Electronic Engineering

University of Essex

March 2011

Abstract

Software systems are the most complex artefacts ever produced by humans, and managing their complexity is one of the central challenges of software engineering. One major source of complexity arises from maintaining consistency between a program and its design documentation. Design verification reduces such complexity by improving consistency between design and implementation. Fully automated tools are of paramount importance so that the budget, time and quality trade-off is minimized.

This thesis combines aspects of both the theory and practise of design verification. We begin by defining a theory of classes abstracted and refined from the theoretical foundations of LePUS3, and developed in the Typed Predicate Logic. LePUS3 is a promising formal and visual design description language for the decidable aspects of object-oriented design. Our theory fixes many of the criticisms of LePUS3 and is also more expressive, rigorous, and extensible.

We then move onto the practise of design verification, which we define for a restricted subset of our theory. We demonstrate our design verification method in three case studies: verifying an implementation of the Composite design motif in Java's Abstract Window Toolkit, verifying a selection of requirements of a JUnit test case, and verifying instantiations of Java's generic lists. This form of design verification is fully automatic, which we illustrate in a proof-of-concept tool: the Two-Tier Programming Toolkit. The Toolkit, which is designed for round-trip engineering, has been examined in a very small empirical study that showed a marked improvement to the productivity of its users.

The result of this research is a powerful new rigorous design description language that can be easily extended and applied to both the theoretical investigation of object-oriented design, and practical application to the improvement and understanding of new and existing software.

Acknowledgements

Thank you to all who have helped me during my research, too many to address individually. I would like to give special thanks to the following people for all their support, guidance, constructive feedback and their vast patience: Professor Raymond Turner, Dr Amnon H. Eden, Dr Epameinondas Gasparis, Dr Rick Kazman, Professor Martin Henson, Professor Jonathan Bowen.

A personal thank you to my partner, friends, and family, your constant support and encouragement has kept me smiling.

Lastly, I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) UK for their funding, without which none of this work would have been possible.

Contents

List of Tables

List of Figures

Writing Conventions

What follows is a summary of the main writing conventions and symbols used throughout this thesis, loosely grouped according to theme. This list is not intended to be complete, or to define for any given symbol. Rather, this list should be treated as a quick reference to the most common symbols and some terminology. We adopt the shorthands listed below only when it is not ambiguous to do so, or unless otherwise stated.

1. **General**

   (a) We italicize the first occurrence of a new important term in this text

   (b) We use the `sanserif` font for program source code

   (c) We take our notion of computability and decidability from [Cutland, 1980], that is a function is *computable* if and only if one can define a Turing Machine that computes it (Turing computable). A computable function is always *decidable*, but a relation R is decidable if a computable *characteristic function* $\mathcal{C}_R$ can be defined, such that:

   $$\mathcal{C}_R\left(t\right) = \begin{cases} \textbf{True} \text{ if and only if } t \in R \\ \textbf{False} \text{ if and only if } t \notin R \end{cases}$$

   Or equivalently, R is said to be decidable if it and its complement are recursively enumerable. And finally, to be undecidable is to be not decidable

   (d) All functions are partial unless stated otherwise

   (e) **FOPL** is the First-Order Predicate Logic [Huth and Ryan, 2000]

   (f) **TPL** is the Typed Predicate Logic as defined in Chapters 5 and 6 [Turner, 2009]

   (g) **TC** is the Theory of Classes we develop through Chapter 7

   (h) **RTC** is the Restricted Theory of Classes, defined in Chapter 11 for the purposes of design verification

2. **Notation**

   (a) Lower-case Greek letters (ϕ, φ, ψ, ...) are reserved for propositions

   (b) We use upper-case Greek letters (Φ, Ψ, ...) for specifications (see p.59), with the exception of:

      i. Γ and Δ for contexts (see p.44)

      ii. Θ for judgments (see p.43)

      iii. Ω for contradiction (see p.45)

      iv. Σ for sigma propositions (see p.49) dependent types (see p.63)

   (c) Judgments in the form $\Theta_1[\Theta_2, t]$ are shorthand for a judgment Θ_1 in which the judgment Θ_2 and term t occurs

   (d) Sequents take the usual form of $\Gamma \vdash \Theta$

   (e) $\triangleq$ is a meta-operator that means defined as

   (f) $t_1, \ldots, t_n : T$ is shorthand for the series $t_1 : T, \ldots, t_n : T$

   (g) We write (t) in place of $((t))$ where it is not ambiguous to do so

   (h) Uppercase letters (X, Y, ...) are reserved for type variables (see §5.2)

   (i) The blackboard bold font ($\mathbb{X}$, $\mathbb{Y}$, ...) is reserved for types

   (j) τ as a meta-variable ranging over type constructors, where types of the form $A \tau B$ are shorthand for $\tau\,(A, B)$

   (k) Lowercase letters (u, v, w, x, y, z ...) are reserved for variables in the theory, where t is a meta-variable over terms

   (l) The typewriter font ($\texttt{0}$, $\texttt{x}$, ...) is reserved for constants

   (m) All operators have left-to-right associativity (operational precedence)

   (n) General symbol quick reference:

      i. $\neg$ is negation (see p.45)

      ii. Ω is contradiction (see p.45)

      iii. $\wedge$ is conjunction (see p.46)

      iv. $\vee$ is disjunction (see p.46)

      v. $\implies$ is implication (see p.46)

vi. $\iff$ is bi-conditional (see p.46)

vii. $=$ is equality (see p.46)

viii. $\neq$ is inequality (see p.47)

ix. $\forall x : T \bullet \phi$ means for all x of type T such that ϕ holds (see p.48)

x. $\exists x : T \bullet \phi$ means there exists an x of type T such that ϕ holds (see p.47)

xi. $\exists ! x : T \bullet \phi$ means there exists a unique x of type T such that ϕ holds (see p.48)

(o) Let $\mathcal{Q}$ be one of the above quantifiers, we write $\mathcal{Q}x_1, \ldots, x_n : T \bullet \phi$ as shorthand for $\mathcal{Q}x_1 : T \bullet \ldots \mathcal{Q}x_n : T \bullet \phi$

3. **Relations**

(a) We neglect the type arguments of relations where they are recoverable from the context and it is not ambiguous to do so

(b) Let R be a relation such that $\Gamma, a : T_{in}, b : T_{out} \vdash R(a, b)$ *prop*. R is a total functional relation, written $R : T_{in} \longmapsto T_{out}$ (see p.62), if the following holds:

$$\forall x : T_{in} \bullet Dom_R(x) \implies \exists ! y : T_{out} \bullet R(x, y)$$

(c) Let R be a total functional relation, we write $R(a)$ for the unique b such that $R(a, b)$ holds[1]

(d) Specifications in **TPL** are written in the following horizontal (inline) notation:

$$R \triangleq [t_1 : T_1, \ldots, t_n : T_n | \phi[t_1, \ldots, t_n]]$$

or the vertical notation:

$$\begin{array}{|c|} \multicolumn{1}{l}{R} \\ \hline t_1 : T_1, \ldots, t_n : T_n \\ \hline \phi[t_1, \ldots, t_n] \\ \hline \end{array}$$

[1] R does not have to be the name of the function, but generally this is the case.

(e) Specifications in **RTC** are written in the following horizontal (inline) notation:

$$R \triangleq \lfloor t_1 : T_1, \ldots, t_n : T_n | \phi\left[t_1, \ldots, t_n\right] \rfloor$$

or the vertical notation:

$$\begin{array}{|l}
\multicolumn{1}{l}{R} \\ \hline
t_1 : T_1, \ldots, t_n : T_n \\ \hline
\phi\left[t_1, \ldots, t_n\right] \\ \hline
\end{array}$$

where ϕ is a series of propositions that are implicitly joined by conjunction, each of which is restricted to the grammar presented at the beginning of Part III.

4. **Types**

(a) $A \times B$ is the type of pairs (see p.52) with selection operations π_1 and π_2 (see p.53). n-tuples are constructed from pairs in the form:

$$T_1 \times T_2 \times \ldots \times T_{n-1} \times T_n \triangleq T_1 \times (T_2 \times (\ldots \times (T_{n-1} \times T_n)))$$

with selection operator π_x^n as shorthand for the xth selection operator for an n-tuple

(b) $set\,(T)$ is the type of finite set terms containing elements of type T (see p.54) with the following symbols and shorthands:

i. $\varnothing_T$ is the empty set of type T (see p.54)

ii. $\oplus$ is set addition (see p.54)

iii. $\ominus$ is set subtraction (see p.55)

iv. $\in$ is set membership (see p.55)

v. $\notin$ is set non-membership (see p.55)

vi. $\subset$ is proper subset (see p.57)

vii. $\subseteq$ is set subset or equal (see p.56)

viii. $\cup$ is set union (see p.57)

ix. $\cap$ is set intersection (see p.57)

x. $\forall x \in s \bullet \phi$ means for all x in set s such that ϕ holds (see p.56)

xi. $\exists x \in s \bullet \phi$ means there exists an x in set s such that ϕ holds (see p.56)

xii. $\exists! x \in s \bullet \phi$ means there exists a unique x in set s such that ϕ holds (see p.56)

xiii. Let $\mathcal{Q}$ be one of the above quantifiers, we write $\mathcal{Q}x_1, \ldots, x_n \in t \bullet \phi$ as shorthand for $\mathcal{Q}x_1 \in t \bullet \ldots \mathcal{Q}x_n \in t \bullet \phi$

xiv. We adopt the shorthand $\{t_1, \ldots, t_n\} : \mathit{set}\,(T)$ for the set constructed by $t_1 \oplus \ldots \oplus t_n \oplus \varnothing_T$[2]

(c) *list* (T) is the type of finite lists containing elements of type T (see p.159) with the following symbols and shorthands:

i. $[]_T$ is the empty list of type T (see p.159)

ii. $\boxplus$ is list append (see p.159)

iii. *head* gets the head of a list (see p.160)

iv. *tail* gets the tail of a list (see p.160)

v. *concat* concatenates two lists (see p.160)

vi. $\forall x \in s \bullet \phi$ means for all x in set s such that ϕ holds (see p.161)

vii. $\exists x \in s \bullet \phi$ means there exists an x in set s such that ϕ holds (see p.161)

viii. $\exists! x \in s \bullet \phi$ means there exists an x in set s such that ϕ holds (see p.161)

ix. Let $\mathcal{Q}$ be one of the above quantifiers, we write $\mathcal{Q}x_1, \ldots, x_n \in t \bullet \phi$ as shorthand for $\mathcal{Q}x_1 \in t \bullet \ldots \mathcal{Q}x_n \in t \bullet \phi$

x. We adopt the shorthand $[t_1, \ldots, t_n] : \mathit{list}\,(T)$ for the list constructed by $t_1 \boxplus ... \boxplus t_n \boxplus []_T$[3]

(d) *schema* (T) is the type of relations that operate over type T (see p.68) with the following symbols and shorthands:

i. *Dom* gets the domain of a relation (see p.68)

ii. *Ran* gets the range of a relation (see p.68)

iii. $\wedge$ is relational conjunction (see p.69)

iv. $\vee$ is relational disjunction (see p.70)

v. $\circ$ is relational composition (see p.70)

vi. R^+ is the transitive closure of relation R (see p.71)

[2] Although similar, this shorthand is not to be confused with the notation for specifying subtypes.

[3] Although similar, this shorthand is not to be confused with the inline notation for specification.

(e) $\{x : T|\phi\}$ is a subtype of T (see p. 49)

(f) $\mathbb{N}$ is the type of natural numbers (see p.50) with the following symbols and shorthands:

i. n^+ is the successor of n (see p.50)

ii. $+$ is addition (see p.51)

iii. $-$ is subtraction (see p.51)

iv. $*$ is multiplication (see p.52)

v. / is division (see p.52)

vi. a^b is exponentiation: a to the power of b (see p.52)

vii. $a!$ is a factorial (see p.52)

viii. We adopt `1` as shorthand for $\mathtt{0}^+$, `2` is shorthand for $\mathtt{1}^+$, and so on

(g) $\mathbb{STRING}$ is the type of alphanumeric strings (equivalent to the $list\,(\mathbb{CHAR})$ type), where $\mathtt{c}_1 \ldots \mathtt{c}_n$ is shorthand for that string constructed by $\mathtt{c}_1 \boxplus \ldots \boxplus \mathtt{c}_n \boxplus empty_{\mathbb{CHAR}}$. For example `String` is shorthand for $\mathtt{S} \boxplus \mathtt{t} \boxplus \mathtt{r} \boxplus \mathtt{i} \boxplus \mathtt{n} \boxplus \mathtt{g} \boxplus empty_{\mathbb{CHAR}}$ (see p.162)

(h) $\mathbb{CLASS}$ is the type of classes in class-based object-oriented programs (see p.79) with the following relations:

i. $Inherit\,(a, b)$: Class a inherits from class b (see p.80)

ii. $Subclass\,(a, b)$: Class a inherits from, or is equal to, class b (see p.83)

iii. $DataMember\,(a, b)$: Class a has a data member of class b (see p.113), see also $Member$ (p.114)

iv. $Collection\,(c)$: c is a collection class (see p.115)

v. $Aggregate\,(a, b)$: Class c has an aggregation of class b (see p.116)

vi. $AbstractClass\,(c)$: c is an abstract class (see p.121), see also $Abstract$ (p.123)

(i) $\mathbb{HIERARCHY}$ is the type of sets of classes all related by inheritance (see p.84)

(j) $\mathbb{SIGNATURE}$ is the type of method signatures (a method's name and argument types) in class-based object-oriented programs:

i. $sig\,(cls_1, \ldots, cls_n)$ is the form of atomic terms of the unimproved type $\mathbb{SIGNATURE}$ (see p.86)

ii. $sig\,\langle cls_1, \ldots, cls_n\rangle$ is shorthand for the pair $(sig, [cls_1, \ldots, cls_n])$, a composite term of the improved $\mathbb{SIGNATURE}$ type (see p.162)

(k) $\mathbb{METHOD}$ is the type of methods in class-based object-oriented programs (see p.85) with the following relations:

   i. $\mathit{SignatureOf}\,(s,m)$: s is the signature of method m (see p.86), see also the derived function SigOf (p.86)

   ii. $\otimes\,(s,c,m)$: (Nested set of) signature s and (nested set of) class c identify (nested set of) method m (see p.96), see also the derived function $\otimes$ (p.102)

   iii. $\mathit{MethodMember}\,(c,m)$: Method m is a member of class c (see p.85), see also Member (p.114)

   iv. $\mathit{AbstractMethod}\,(m)$: Method m is abstract (see p.121), see also $\mathit{Abstract}$ (p.123)

   v. $\mathit{Method}\,(c,s)$: There exists a method m such that $m = s \otimes c$ (see p.112)

   vi. $\mathit{Overrides}\,(a,b)$: Method a overrides method b (see p.89)

   vii. $\mathit{Overloaded}\,(m)$: Method m is overloaded in some class (see p.165)

   viii. $\mathit{Call}\,(a,b)$: Method a calls method b (see p.117)

   ix. $\mathit{Forward}\,(a,b)$: Method a forwards the its invocation to method b (see p.118)

   x. $\mathit{Return}\,(m,c)$: Method m returns an instance of class c (see p.119)

   xi. $\mathit{Create}\,(m,c)$: Method m creates an instance of class c (see p.120)

   xii. $\mathit{Produce}\,(m,c)$: Method m produces an instance of class c (see p.120)

5. **Design Models**

(a) The fraktur font ($\mathfrak{A}$, $\mathfrak{B}$, $\ldots$) is reserved for design models, where their universes are denoted $\mathbf{A}$, $\mathbf{B}$, $\ldots$ respectively (see p.180)

(b) $\mathcal{I}_{\mathfrak{M}}$ is the interpretation function of model $\mathfrak{M}$, where $\Theta^{\mathfrak{M}}$ is shorthand for $\mathcal{I}_{\mathfrak{M}}\,(\Theta)$ (see p.180)

(c) $\mathcal{A}_{pl}$ is the abstract semantics function from programs articulated in programming language pl to design models (see p.179)

(d) $\mathfrak{M} \models \Psi$ is the satisfies relation between design model $\mathfrak{M}$ and **RTC** specification Ψ (see p.181 and p.183)

Part I

Background and Motivation

Chapter 1

Introduction

Software systems are the most complex artefacts ever produced by humans [Brooks, 1987, Gibbs, 1994]. But software systems are very rarely static artefacts, instead being in a continuous state of flux. This is best described by the laws of software evolution, which we summarize in Table 1.1 from [Lehman et al., 1997]. Specifically, Lehman provides laws for E-type systems: those software systems that are *embedded* in the world in which they operate.

Table 1.1: The eight laws of software evolution [Lehman et al., 1997]

No.	Year	Law
I	1974	*Continuing Change* E-type systems must be continually adapted else they become progressively less satisfactory.
II	1974	*Increasing Complexity* As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
III	1974	*Self Regulation* E-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV	1980	*Conservation of Organisational Stability* (invariant work rate) The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
V	1980	*Conservation of Familiarity* As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI	1980	*Continuing Growth* The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII	1996	*Declining Quality* The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII	1996	*Feedback System* (first stated 1974, formalised as law 1996) E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Managing complexity is therefore one of the central challenges of software engineering. One major source of complexity arises from maintaining consistency between a program[4] and its design documentation. This is due in part to a lack of adequate representations of the design, and of tools detecting violations thereto. Without these, software evolution devolves into a continuous process of introducing violations to the design, ultimately making the system more brittle and difficult to maintain. This is the problem commonly known as *architectural erosion* [Perry and Wolf, 1992], which is especially commonplace in large organizations with many developers. What is worse is that as developers leave the company, and new ones join, the design documentation fast becomes outdated, inaccurate, or is simply missing. Eventually, the complexity becomes unmanageable and the program too difficult and costly to maintain.

Maintaining consistency therefore requires that developers identify inconsistencies between the program and the design as early as possible, a task broadly known as *software verification*. But a developer must select an appropriate software verification technique that is adequate for their purposes. For example, it is unnecessary to ensure functional correctness in systems that do not require that level of scrutiny; similarly it would not be wise to perform only basic unit testing for safety critical systems. Yet no matter which methods of software verification are selected for a project, they should come at minimal cost by minimizing the budget, time and quality trade-off. Therefore tools that perform fully automated software verification are of paramount importance. This is our overarching motivation for this research, where we focus specifically on the problem of *design verification*. Design verification is a type of software verification driven by the creational and structural aspects of object-oriented program design. We choose the class-based object-oriented paradigm as it is so widely accepted in industry and includes languages such as Simula 67, Smalltalk (in its various versions), C++, Object Pascal, Beta, Java, Ada 95, CLOS, and C#.

One of the most basic requirements for tool support in fully automated design verification is an unambiguous language in which we may easily articulate our requirements. As such natural language is not appropriate in this context. For example, consider the English phrase “I made her duck” for which there are at least five interpretations [Jurafsky and Martin, 2000][5]. To clarify exactly which interpretation is the one intended requires further explanation, increasing its verbosity. Although natural languages are indispensable for documenting systems, they are not applicable where absolute clarity and rigour are required. Formal languages are, however, unambiguous by

[4] Source code is the means by which programmers encode programs; it consists of a detailed set of instructions dictating how (rightly or wrongly) the target computer operates. Therefore the source code is the least abstract specification of a program, so when speak of a *program* or *implementation* we refer to the associated source code.

[5] Which we paraphrase as: “I cooked waterfowl for her; I cooked waterfowl belonging to her; I created the (plaster?) duck she owns; I caused her to quickly lower her head or body; and I waved my magic wand; and turned her into undifferentiated waterfowl.”

design and therefore much better suited to the task at hand.

This thesis focuses on one such formal language, the third version of the Language for Pattern Uniform Specification known as LePUS3 and its schematic counterpart Class-Z[6]. LePUS3 is a mature visual design description language designed for the specification and reasoning over the structural and creational aspects of object-oriented design [Eden and Nicholson, 2011]. Through its evolution, LePUS3 has managed to retain a strong concept of elegance [Hoare, 1975]: "We need a puritanical rejection of the temptations of features and facilities, and a passionate devotion to the principles of purity, simplicity and elegance." However, despite our best efforts LePUS3 has suffered from the same issues that afflict evolving software; over time LePUS3 has grown more complex, the underlying theory has become obfuscated by its applications, and further extension is becoming progressively difficult. Therefore, LePUS3 must be redefined to manage its increasing complexity.

In the next section we discuss our objectives on these topics, and how the rest of the thesis is structured.

1.1 Objectives

The overarching aim of this research is to develop a new technique of fully automated design verification of the decidable properties of object-oriented design. As this research combines both theoretical and practical investigation, this aim is best broken into two more specific goals:

G1 Create a new design description language, a Theory of Classes (**TC**), by extracting, refining, and extending the underlying theory from LePUS3 within the framework of the Typed Predicate Logic (**TPL**).

G2 Define and demonstrate the practical use of **TC** in the form of design verification and associated tool support in the Two-Tier Programming Toolkit.

We further constrain each of these goals with a set of three more defined objectives, under which this work can ultimately be assessed:

G1.1 Being that **TC** is to be based on LePUS3, we will also pursue the same guiding principles:

[6] Class-Z differs to LePUS3 only in representation, having no distinction in their definition or expressiveness. Therefore, unless explicitly written otherwise, we will simply refer to both as LePUS3.

G1.1.1 *Object-orientation*—Be restricted to the object-oriented programming paradigm, to represent its core building blocks, namely classes, methods, sets thereof, and their relationships.

G1.1.2 *Rigour*—There are two aspects to this principle. Firstly, the intuitions that motivate the language must be accurate so that there are no inherent flaws in the language's definition. Secondly, that the language and the rules of inference must be formally defined in such a way that any conclusions drawn are done with clarity.

G1.1.3 *Scalability*—Economy of expression is a leading concern, where a language is measured by its ability to represent a lot of information compactly, and to use as few symbols as possible. As such this is closely tied to the notions of *minimality* and *elegance* [Eden and Nicholson, 2011], where for simplicity we consider these to be matters of scalability.

G1.1.4 *Genericity*—The principle of abstraction that governs the relationship between concrete instances and their more generic concepts. The spectrum of genericity is very broad, from the simple and consistent replacement of constants with variables, to full polymorphism. We must decide on an appropriate degree of genericity with respect to our problem domain.

G1.2 LePUS3's intuitions about object-oriented design shall be preserved in **TC**, and expanded upon to capture object-oriented design in greater detail. As such **TC** will be more expressive, while maintaining the existing relations and predicates of LePUS3. This will be demonstrated in part by the addition of several rules of object-oriented design previously inexpressible in LePUS3. We will also show this by articulating a selection of design motifs from [Gamma et al., 1994] and formally reasoning over them to show how their formulations in **TC** are related.

G1.3 Having defined **TC** based on LePUS3, we will extend it with regards to the principle of genericity. We shall make our language more generic by allowing specifications to dynamically declare new types. Doing so allows us to represent design at a higher level of abstraction through greater modularization and reuse.

G2.1 We will define design verification for a subset of **TC** that is similar to the expressiveness of LePUS3 that complies with the definition of a formal method as presented in [Wing, 1990].

G2.2 Demonstrate that the design verification algorithm can be implementable as a tool, that is fully automated, executable at the click of a button, and completes within reasonable time[7]. We will show this by discussing the Two-Tier Programming Toolkit, our proof of concept tool that supports design verification for LePUS3.

G2.3 We shall, based on a small empirical study, demonstrate that design verification as implemented in the Two-Tier Programming Toolkit improves the accuracy of software developers. We will also discuss the other findings of this study in more depth.

In the next section we discuss how the rest of our thesis is structured with respect to these objectives.

1.2 Structure

The main body of this thesis is split into three parts. The following two chapters (Chapters 2 and 3) discuss the context of our work, setting the scene for a detailed account of LePUS3 (Chapter 4). This is followed by a brief summary of the Typed Predicate Logic (**TPL**, Chapter 5) and how it can be used for specification (Chapter 6). As these two chapters for the most part from [Turner, 2009], with some variation in style and order of presentation, they should be considered background information. The remaining two parts should be considered to be the main contribution of the thesis described as follows.

In Part II we focus on the theoretical foundations of LePUS3. We create a new design description language, **TC,** (Chapter 7, **G1.1–2**) by teasing out the underlying theory of classes from the definitions of LePUS3 (provided in Appendix B), and articulated in **TPL**. Although this redefinition is based on LePUS3, to which we have already made contributions, it also includes several refinements and improvements. The applicability of **TC** to formalizing design motifs is then demonstrated (Chapter 8, **G1.2**) where we not only refine and improve previous attempts to formalize certain design motifs [Eden and Nicholson, 2011], but we also go on to make explicit relationships between them. The part ends (Chapter 10, **G1.3**) by extending our theory toward the principle of genericity so that we may represent design at a greater level of abstraction.

In Part III we focus on applying our theory to design verification. We define this for a subset of our theory (Chapter 11, **G2.1**) and provide three design verification case studies (Chapter 12). We conclude the part with a discussion on tool support for design verification (Chapter 13, **G2.2–3**) in

[7] This is the LePUS3 principle of *automated verifiability.*

the context of LePUS3 within the Two-Tier Programming Toolkit, and discuss our formal reverse engineering tool called the Design Navigator. We also evaluate the Two-Tier Programming Toolkit with a small empirical study.

We conclude this thesis (Chapter 14) with an assessment of **TC** in relation to the comparable aspects LePUS3. Having evaluated our contribution, we provide a brief discussion on future research that one may pursue with this work as a foundation.

Chapter 2

Background

The context of this research falls mainly on three central concepts that we summarize in this chapter. We begin with a discussion on object-orientation and identify the specific class of object-oriented implementation languages we will model. Next, we examine design patterns and identify which components of design patterns we intend to formalize. Finally, we discuss design verification, how it relates to software verification, and what it means for design verification to be fully automated.

2.1 Object-Orientation

Although the fundamental notion of an object is fairly universal, how they are treated varies wildly between programming languages; the term *object-oriented* has many interpretations in the literature. In [Craig, 2000] object-oriented programming languages are primarily divided into two broad categories, prototype and class-based languages. The general premise of a prototype language is that a new object is created by copying existing objects and being manipulated as desired. For example, an object that represents a chair could be copied and modified so that it now represents a table. As such the relationship between objects is fluid, being formed based on their current similarity. It is this notion of similarity that is a core concept to prototype languages.

Alternatively, class-based languages group objects through predetermined descriptions (classes). Rather than copying and modifying an existing object as with prototypes, class-based languages instantiate objects in accordance with a class definition. For example, an object that represents a chair does so because it is an instantiation of an appropriate class, and no amount of modification can turn it into a table. As opposed to similarity, the core concept in class-based languages are the inheritance mechanisms between classes [Taivalsaari, 1996], allowing instances of class c_{sub} to be treated as instances of another class c_{super} as long as c_{sub} inherits from c_{super}. For example, if the class that describes chairs inherits from that describing a tables[8], then any

[8] Assuming whoever defined such classes decided that chairs are kinds of (specialization of) tables.

chair object can indeed be treated as a table object.

However, "it appears prototype-based programming is better suited to exploratory or experimental programming than with production methods" [Craig, 2000]. Therefore we focus our attention on class-based, and therefore statically typed, languages. This is the more popular category of languages that includes Simula 67, Smalltalk (in its various versions), Object Pascal, Beta, Java, Ada 95, C++, and C#. Nonetheless, since the term *object-oriented* is more widely recognized than terms like *class-based* and *statically typed*, we employ the former even in contexts when the latter is more precise.

Now that we are clear what sort of object-oriented programs are of interest we briefly discuss the five central principles with which all such languages comply in varying degrees. These are the principles of *encapsulation*, *information hiding*, *inheritance*, *dynamic binding* and *polymorphism.* In the following subsections we outline what aspects of these principles we intend to capture in our design description language. Additionally, since it is well-known and exemplifies the properties we wish to discuss, we use the Java[9] programming language [Sun Microsystems Inc., 2006] as our primary source for examples.

2.1.1 Encapsulation

The distinction between encapsulation, sometimes referred to as modularity, and information hiding is often blurred. Being such a fundamental property of object-orientation, it is worth ensuring that it is a notion kept distinct from information hiding, which we discuss in the next subsection.

Encapsulation is the grouping of data members (fields) and associated behaviours (methods) into coherent units (classes). As such encapsulation allows program components to be logically divided by roles and responsibilities. Our design description language must therefore at least capture these same abstractions (data members, methods, and classes).

2.1.2 Information Hiding

Information hiding (aka data abstraction) is one of the fundamental principles of object-oriented programming. Mechanisms supporting information hiding are in place in all object-oriented programming languages. Essentially, it dictates that a class can be considered to be composed of two parts: its public part constitutes its functionality as a service provider, exposed via a

[9] Specifically we use Java 6 update 12 (JDK6u12).

public interface; and its private part constitutes its implementation, a set of operations and data kept hidden from its clients. Commonly, keywords such as **public** and **private** enforce this division explicitly. This distinction reduces dependencies between modules, thereby promoting modularity and changeability.

Unfortunately, information hiding statements are non-local [Eden et al., 2006]. This means that verifying statements about information hiding requires us to consider the entire implementation. The results of design verification are therefore only conclusive if the entire program has been examined. Even the smallest change to the program invalidates all previous verification results. As LePUS3 is committed to only representing local statements, and one of our objectives is to preserve such commitments (**G1.2**), we shall not attempt to capture information hiding.

2.1.3 Inheritance

> “The basic idea of inheritance is simple. Inheritance allows new object definitions to be based upon existing ones; when a new kind of an object class is to be defined, only those properties that differ from the properties of the specified existing classes need to be declared explicitly, while the other properties are automatically extracted from the existing classes and included in the new class. Thus, inheritance is a facility for differential, or incremental program development.” [Taivalsaari, 1996]

We introduced inheritance earlier in the chapter as a core mechanism in class-based object-oriented languages by which classes are grouped into hierarchical structures. We restrict ourselves to such a simplistic overview of inheritance, despite the many, sometimes conflicting, notions of inheritance [Taivalsaari, 1996]. Specifically, our notion of inheritance is that of *subclassing* and *subtyping* collectively, implemented in Java by the **extends** and **implements** keywords respectively. Inheritance should therefore be a strict order relation on classes [Craig, 2000, pp.32–38], where a strict order relation is irreflexive[10], asymmetric[11], and transitive[12]. Other notions of inheritance can be added, but these are the most interesting for our purposes.

Additionally, we try to preserve the same degree of conceptual correspondence found in object-oriented languages such as Java. That is, we require that a subclass maintain full syntactic (interface) compatibility with its superclass. This non-strict view of inheritance allows for greater expressivity as methods can be redefined with radically different behaviour than that of the

[10] Irreflexive: for any element e then $r(e,e)$ does not hold.

[11] Asymmetric: for any elements e_1, e_2 then $r(e_1,e_2)$ implies $r(e_2,e_1)$ does not hold.

[12] Transitive: for any elements e_1, e_2, e_3 then $r(e_1,e_2)$ and $r(e_2,e_3)$ implies $r(e_1,e_3)$.

superclass, as long as they maintain the same interface. Classes related by this notion of inheritance invariably form a tree structure—an inheritance class hierarchy. Inheritance class hierarchies are sets of classes that contain a single root class such that all other classes in the set inherits from it. As the interface of the root class must be preserved with each inheriting class, all classes in an inheritance class hierarchy must share at least the minimal interface as defined by the root class. Inheritance class hierarchies are ubiquitous in object-oriented programs and class libraries, and therefore should be considered as primitive components of object-oriented design.

To summarize, if a class c_{sub} (the subclass) inherits from another class c_{super} (the superclass) then they share some of their internal structure, extending and possibly overriding the components of the superclass. As the interface of c_{super} is preserved, instances of c_{sub} can be treated as instances of c_{super}. Sets of such classes, i.e. c_{super} and at least one c_{sub} class, are called inheritance class hierarchies, which all share at least the minimal interface of c_{super}.

2.1.4 Dynamic Binding

Dynamic binding is closely associated with the principle of inheritance, which we briefly discussed in the previous subsection. "Dynamic binding means that the operation that is executed when objects are requested to perform an operation is the operation associated with the object itself and not with one of its ancestors" [Craig, 2000]. For example, consider method m defined in c_{super} and overridden by m' in c_{sub}, and a variable x of type c_{super}. If x references an instance c_{super} at run-time then invoking method m on variable x will execute method m. Alternatively, if x references an instance of c_{sub} at run-time then dynamic binding ensures that m' is executed rather than m. This allows for an instance of c_{sub} to be dynamically substituted in place of an instance of c_{super}, thereby replacing methods with more specialized logic.

This seems very simple, but it is worth examining the conditions under which method m' overrides method m. For this to happen both methods must have the same interface (signature). A signature is often taken to be the method's name, the type of its arguments, and its return type. However, in our experience the return type does not help to distinguish one method from another. For example, in Java it would be a compile-time error for two methods to be given identical signatures that differ only in their respective return types. For this reason we consider signatures to be only those details that identify a method within a given class, their name and types of its arguments.

2.1.5 Polymorphism

Polymorphism has an immense body of work supporting it, and is a much bigger subject than can be completely addressed here. We will limit our discussions to the forms of polymorphism most commonly implemented in object-oriented programming languages. For example, the combination of inheritance and dynamic binding provide a simple form of polymorphism, called *inclusion polymorphism*, where subclasses can be interacted with using the interface of its superclass.

Another example is *ad-hoc polymorphism* [Craig, 2000], often interpreted as method overloading: methods that share the same method name, but differ in the types of their arguments. For example, the methods add(**int**, **int**) and add(String, String) defined in class c are overloaded as they share the same method name (add), but differ in their argument types. Exactly which is to be executed depends on the type of the arguments at the point of execution.

Another common form of polymorphism is *parametric polymorphism* [Craig, 2000], otherwise known as *genericity*. As with polymorphism, genericity also has an immense body of work supporting it. It would not be appropriate to provide a complete survey of the various mechanisms here, but we must be clear as to what form of genericity we focus on. We limit our discussion to the genericity mechanisms found in object-oriented programming languages like Java and C++. Specifically, we use the following quote to drive our work on genericity:

> "Genericity is considered to be a kind of polymorphism because it takes a piece of code—a class, procedure or function—which implements an algorithm or collection of algorithms in a type independent fashion (or: in terms of an abstract, most general type) and which employs a type substitution mechanism to produce instances which are specialized to particular types." [Craig, 2000, p.153]

Two of the most well known implementations of genericity are Java's *generics*, for which [Liang, 2006, Chapter 21] provides a good introduction, and C++'s *template* mechanism. Although these mechanisms exist to solve similar problems, and therefore are abstractly quite similar, they are implemented in very different ways.

> "While generics look like the C++ templates, it is important to note that they are not the same. Generics simply provide compile-time type safety and eliminate the need for casts. The main difference is encapsulation: errors are flagged where they occur and not later at some use site, and source code is not exposed to clients.

> Generics use a technique known as type erasure . . . , and the compiler keeps track of the generics internally, and all instances use the same class file at compile/run time.” [Mahmoud, 2004]

To demonstrate the compile-time type safety of Java’s generics, consider Figure 2.1, which shows a simple list class implemented without using Java’s generics. Note the need to cast the object returned from method `head()` to `Integer`. Compare this to Figure 2.2, which shows the same code re-implemented using Java’s generics. This version does not require any casting as Java’s generics ensures, at compile-time, that the return type of `head()` is `Integer`.

```
public class SimpleList {
  private Object head = null;
  private SimpleList tail = null;

  public Object head() { return head; }

  public SimpleList tail() { return tail; }

  public void append(Object t) {
    SimpleList newtail = new SimpleList();
    newtail.head = head;
    newtail.tail = tail;
    head = t;
    tail = newtail;
  }
}

public class Test {
  public static void main(String[] args) {
    SimpleList list = new SimpleList();
    list.append(new Integer(0));  // [0]
    list.append(new Integer(1));  // [1,0]
    Integer head = (Integer)list.head();  // 1
  }
}
```

Figure 2.1: A simple non-generic list implementation written in Java

It is important to address the matter of type erasure, a mechanism that translates generic code to its non-generic equivalent [Gosling et al., 2005]. Once compiled to bytecode, the code in Figure 2.2 is translated to equivalent code to that in Figure 2.1. The benefit of this is the compatibility of new bytecode with the existing Java Virtual Machine specification (backward compatibility). The consequence of this is that, at run-time, there is no distinction between instantiations of a generic class. The implication being an isomorphism between generic classes and classes in the bytecode, no matter how many times they may have been instantiated or with what arguments.

Compare this to C++’s template mechanism, which is “a fancy macro processor; whenever a template class is instantiated with a new class, the entire code for the class is reproduced and

```
public class SimpleList<T> {
  private T head = null;
  private SimpleList<T> tail = null;

  public T head() { return head; }

  public SimpleList<T> tail() { return tail; }

  public void append(T t) {
    SimpleList<T> newtail = new SimpleList<T>();
    newtail.head = head;
    newtail.tail = tail;
    head = t;
    tail = newtail;
  }
}

public class Test {
  public static void main(String[] args) {
    SimpleList<Integer> list = new SimpleList<Integer>();
    list.append(new Integer(0));  // [0]
    list.append(new Integer(1));  // [1,0]
    Integer head = list.head(); // 1
  }
}
```

Figure 2.2: A simple generic list implementation written using Java's generics

recompiled for the new class" [Mahmoud, 2004]. Therefore, a template class that is instantiated x unique times will result in x new classes in the compiled program. C++'s templates also allows for compile-time computation (template metaprogramming).

This is a very simplistic summary of these two approaches to genericity mechanisms in Java and C++, but it serves to make a point: each language implements genericity very differently. Attempting to capture genericity at the level of the implementation language means adopting the assumptions of the respective implementation language to the exclusion of others. We wish to capture object-oriented design at a level of abstraction that does not tie us to any single implementation language. Therefore, we argue that the use (or not) of implementation specific genericity mechanisms is a matter left to either a lower level of design than we are interested in, or directly in the hands of the programmer.

In summary, we shall represent inclusion and ad-hoc polymorphism mechanisms as used in implementation languages. However, we shall not attempt to represent genericity mechanisms employed at the implementation level[13]. We will, however, employ a degree of genericity within our design description language itself, such as type variables. We discuss these features later in this thesis as it becomes appropriate (Chapter 5, and Chapter 10).

[13] This is not to say that design description languages should not attempt to capture any genericity mechanisms of programming languages, only that we will not do so.

2.1.6 Conclusion

We have discussed the guiding principles of object-oriented languages, and outlined what level of detail we intend to capture in our design description language. To summarize, we limit our discussion of inheritance to subclassing/subtyping with syntactic (interface) compatibility. Our design description language should at least capture data members, methods, classes, and inheritance class hierarchies. We will not attempt to capture information hiding statements, as explained in §2.1.2. We will capture inclusion and ad-hoc polymorphism, but not genericity mechanisms employed at the implementation level. We will, however, add genericity mechanisms to our design description language.

To accomplish this, types must be treated as first class citizens of our design description language. The Typed Predicate Logic (**TPL**) is a logical choice as it agrees with our requirement for rigour and is a convenient framework in which to formulate type systems; we discuss **TPL** in Chapters 5 and 6. **TPL** allows us to focus our attention on which types we should employ and their relationships, while keeping our requirements in mind. For example, should we reason over only classes, objects, or both? We find clues to answer these questions not only in implementation languages, but also with at design patterns and their participants.

2.2 Design Patterns

The introduction of object-oriented programming caused a paradigm shift, and has proved to be an effective mechanism for encoding most modern programming problems. This exemplifies the notion that better mechanisms of abstraction offer effective methods of addressing the problem of complexity. Similarly, design patterns [Beck and Cunningham, 1987] are arguably one of the most important advancements in software engineering of the past half century. [Schmidt et al., 1996] describes them as:

> "...a recurring solution to a standard problem. When related patterns are woven together they form a language that provides a process for the orderly resolution of software development problems. ... Both patterns and pattern languages help developers communicate architectural knowledge, help people learn a new design paradigm or architectural style, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience."

Design patterns (henceforth simply patterns) can therefore be used to document any sort of recurring design solution and are found in many different programming paradigms. For example, service oriented architecture [SOA Systems Inc., 2010] and agent based systems [Sauvage, 2004] have both had attention in this area. However, as our work focuses on the object-oriented paradigm, it is the object-oriented patterns to which we direct our attention.

As a pattern can document any solution to a standard problem, what or who then decides which patterns are more appropriate than others? The success of a pattern comes from it being tried and tested in the wider community:

> "*Success is more important than novelty.* The longer a pattern has been used successfully, the more valuable it tends to be. In fact, novelty can be a liability, because new techniques are often untested. Finding a pattern is a matter of discovery and experience, not invention. A new technique can be documented as a pattern, but its value is known only after it has been tried. This is why most patterns describe several uses." [Schmidt et al., 1996]

This accounts for the immense success of the "Gang of Four" design pattern catalogue [Gamma et al., 1994], which struck a chord with many experienced object-oriented programmers, designers, and architects. The "Gang of Four" catalogue is perhaps the most commonly cited pattern catalogue, and the one in which we will also place our attention.

The "Gang of Four" divides a pattern into four essential aspects, each generally documented in natural language and illustrated with clear examples where appropriate:

Name Identifies the pattern, allowing it and all that it entails to be discussed with clarity.

Problem Describes when to apply the pattern, and is often split into the smaller categories of *intent*, *motivation*, and *applicability*.

Solution Describes what design (or variations of) should be employed, and is often split into the smaller categories of *structure*, *participants*, *responsibilities*, and *collaborations*.

Consequences Arguments for and against applying the pattern.

For example, Figure 2.3 (see p.18) summarizes the Abstract Factory pattern defined in [Gamma et al., 1994][14]. We do not attempt to formalize every aspect of design patterns, rather

[14] Collections of pattern definitions like that of Figure 2.3 provide an informal vocabulary for communicating common problems, a *pattern language* [Schmidt et al., 1996].

we focus our attention on *design motifs* (or simply motifs): good solutions advocated by design patterns i.e. their structure, participants, responsibilities and collaborations. This is an area which has received a great deal of attention as demonstrated in the collection of techniques contained in [Taibi, 2007a]. Ideally, motifs should be represented formally so that examples serve to illustrate and not define. Therefore a language for specifying motifs must be appropriately abstract to capture them at right level of detail, without losing sight of its purpose. The matter of abstraction is no easy subject, perhaps best summarized by Gottlob Frege (quoted from [Shapiro, 2000]):

> "Inattention is a very strong lye; it must not be applied at too great a concentration, so that everything does not dissolve, and likewise not too dilute, so that it effects a sufficient change in the things. Thus it is a question of getting the right degree of dilution; this is difficult to manage, and I at any rate have never succeeded."

Therefore our attention must be drawn to design languages that capture motifs at what we consider an appropriate level of abstraction, where we pay specific attention to those that have demonstrated real world practical application in the form of design verification.

2.3 Design Verification

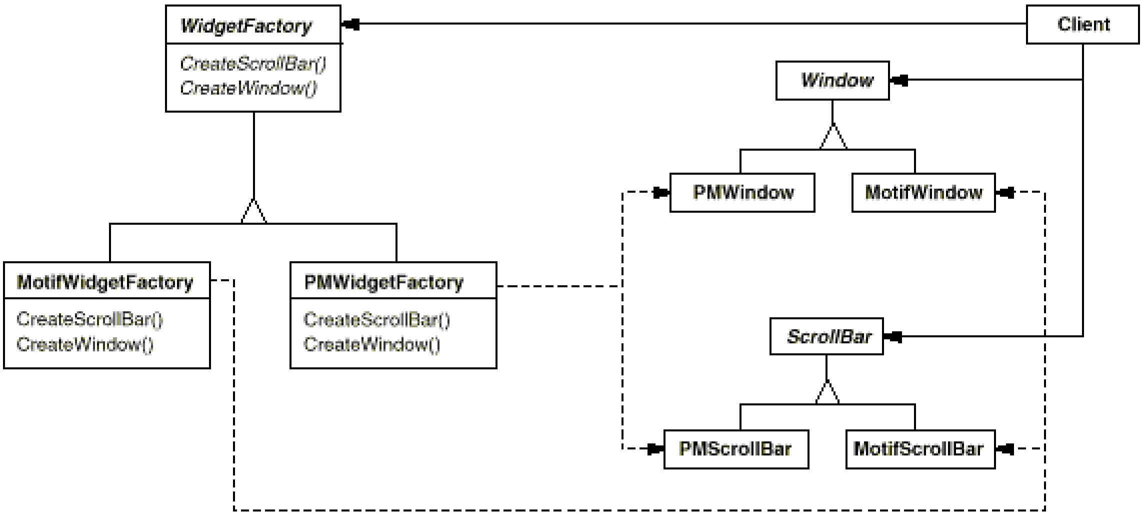
A common occurrence in large software projects is a constant stream of changes to the operational environment and end user requirements throughout the projects lifetime. Any change must therefore be carefully implemented so as to avoid violating existing design decisions. In Java's Abstract Window Toolkit (AWT) for example, it is often asserted that the classes Component, Container, and the set of concrete components (i.e., Button, Scrollbar , Canvas, etc.) constitute an implementation of the Composite design motif [Seemann and von Gudenberg, 1998, Dong and Zhao, 2007, Stelting and Maassen, 2002]. As Java evolves every change to AWT must continue to uphold this design decision.

Unfortunately, even the most carefully documented and detailed design documentation will either be lost, ignored, or most frequently become irrelevant through many iterations of implementation updates. Worse, through years of development and evolution projects tend to amass several millions of lines of code, not all of which will be properly documented. This is compounded by the movement of developers to and from the project, and implicit design information is lost. Unless effort is taken to minimize these issues the project becomes progressively more complex,

Name Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Structure Defined in Object-Modelling Technique (OMT) notation as:



Participants Defined as:

AbstractFactory (WidgetFactory) declares an interface for operations that create abstract product objects.

ConcreteFactory (MotifWidgetFactory, PMWidgetFactory) implements the operations to create concrete product objects.

AbstractProduct (Window, ScrollBar) declares an interface for a type of product object.

ConcreteProduct (MotifWindow, MotifScrollBar) defines a product object to be created by the corresponding concrete factory, and implements the AbstractProduct interface.

Client uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations Defined as:

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Figure 2.3: Summary of the Abstract Factory pattern [Gamma et al., 1994]

brittle, and difficult to manage [Lehman et al., 1997, Perry and Wolf, 1992]. How can changes be applied without violating existing design decisions? How can specifications remain current and accurate throughout the lifetime of a program? These are questions answered by the broad category of *software verification* techniques.

Software verification techniques report how well a software component satisfies a specification, but what this entails varies greatly according to the employed notion of *correctness*. Commonly, the term correctness refers to the input/output behaviour of a program, which is almost without exception undecidable and more appropriately termed *functional correctness*[15]. However, software verification is a very broad topic where functional correctness is often too strong a notion of correctness. Our research centres on *design verification*, which uses a weaker and more limited notion of correctness that we call *structural correctness*. Structural correctness ensures that the decidable (structural and creational) properties of a program implement the given specification. The significance of this is that we cannot verify other aspects of programs, such as their behaviour, events, or state.

Verifying programs against design motifs, however, requires the additional step of identifying the components of the program that qualify as participants of the motif. In our case identifying participants is a manual step as the those participants are already identified during the design process. Although this is not the focus of our work, when the use of a pattern is unknown then it is possible to find appropriate participants automatically[16], for example [Dong and Zhao, 2007]. As such, we consider the *fully automated design verification* of a program against a design motif to be the uninterrupted[17] and conclusive process of verifying the implementation once the participants of the motif have been identified.

[15] Functional correctness falls into two categories. *Partial* correctness requires that when a program calculates a result that it be the one specified. *Total* correctness additionally requires that the program terminates in all cases.

[16] Although false positives are always an issue here.

[17] The process does not prompt the user for additional information.

Chapter 3

Related Work

There are many classifications of design languages. For example, Architecture Description Languages (ADLs) [Medvidovic and Taylor, 2000] and formal pattern specification languages [Taibi, 2007a]. We focus on *Design Description Languages* (DDLs) defined as "any modelling or specification language for representing the non-functional specifications of software design. These statements may describe programs at any level of abstraction: strategic (or architectural) design, tactical design[18], and implementation minutia." [Eden and Nicholson, 2011] Specifically, we focus on those that are (1) designed for capturing object-oriented design, (2) can capture design motifs, and (3) have tool support. Tools that by a click of a button can conclusively establish whether a given implementation conforms to (*satisfies*) the design. LePUS3 achieves all these goals, so it is on LePUS3 that we base our research. But to put LePUS3 in context we discuss three other languages in some detail in the following sections, before discussing LePUS3 in Chapter 4.

3.1 UML: Unified Modelling Language

As UML is the successor to the Object-Modelling Technique notation (OMT), and is the de facto design language in industry, it is a logical place to begin our discussion. UML [Object Management Group, 2003, Object Management Group, 2005] is a broad collection of languages spanning many aspects of software development, such as specifying a systems structure (e.g. class and package diagrams) and behaviour (e.g. sequence and activity diagrams). It is the class/package diagrams on which we will focus our attention. UML is a very powerful and expressive language that has a range of tool support, from open-source tool that focus on diagramming [ArgoUML Open Source Community, 2010], to more advanced tools that target problems such as reverse engineering [Fujaba Development Group, 2009].

Although UML has had a great deal of attention over the years, not all of it has been positive. For example, Bell's satirical article highlights many concerns about the current state of use of

[18] See [Eden, 2005] for a discussion in the distinction between strategic and tactical design.

the language [Bell, 2004]. From our perspective, although UML was designed with object-oriented programming in mind, it was not designed for the articulation of abstract notions such as design patterns. We illustrate this point in Figure 3.1, a class diagram version of the OMT diagram presented previously (Figure 2.3) where Figure 3.2 is provided for reference.

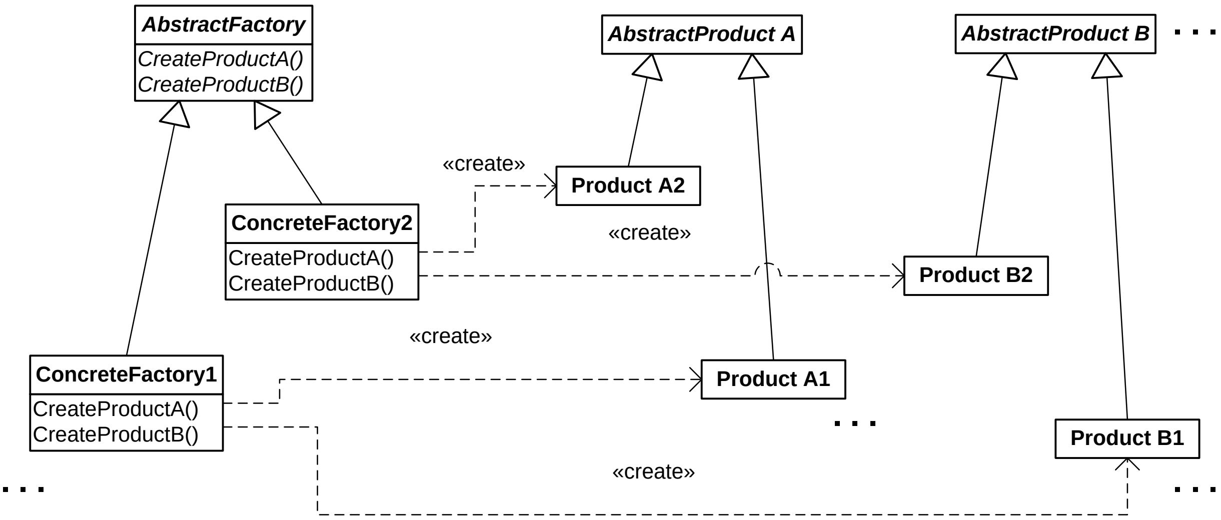


Figure 3.1: The Abstract Factory motif in UML [Eden and Nicholson, 2011]

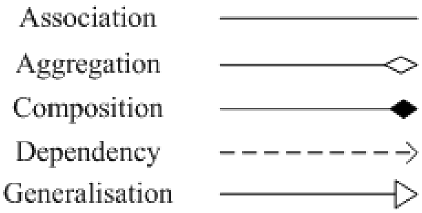


Figure 3.2: Legend of basic UML relations

Figure 3.1 attempts to convey the abstract nature of the Abstract Factory pattern. However, UML does not have an appropriate vocabulary for doing so. This results in a diagram that is not a specification for a specific program, nor is it a specification of the pattern. For example, the pattern does not state that the classes must be named as indicated, yet the class diagram does. The diagram also uses non-standard ellipsis notation to try to indicate abstraction on multiple elements. UML is just not abstract enough to capture the actual design motif—the more abstract notion of which Figure 3.1 is an instance. If specification of a motif is not possible then neither is design verification of motifs. "UML cannot be used to describe an infinite set of pattern instances because the language is not designed for that purpose. If there were some higher-level metaUML, then perhaps this would be appropriate, but the UML itself does not provide this." [Blewitt, 2007] To this end, many research projects have set about formalizing and/or extending subsets of UML for the purpose of capturing motifs and verifying their implementations, for example [France et al., 2004], [Mak et al., 2004], and [Dong et al., 2007]. However, such an approach is really an investigation on how to use UML for the problem of representing motifs, rather than investigating motifs themselves.

3.2 DPML: Design Pattern Modelling Language

An alternative to extending UML is to define the semantics of a language in UML. This results in a language that is not directly constrained by the definitions of UML. Such languages can allow for abstract specifications to be written, such as motifs, and then transformed (instantiated) into UML models as appropriate. This is the approach of the Design Pattern Modelling Language (DPML) [Mapelsden et al., 2002, Maplesden et al., 2007]. We focus our attention on DPML specification diagrams, a visual language for modelling the structural aspects of design motifs with the admirable aim for "sufficient formalism to provide a robust representation, while avoiding complex mathematical formalisms that restrict the use of our approach to a very small set of mathematically inclined programmers." [Maplesden et al., 2007].

DPML models participants, "a structurally significant feature of a Design pattern" [Maplesden et al., 2007]. Participants can be interfaces, implementations, methods, operations and attributes; notions taken directly from [Gamma et al., 1994]. Attributes represent the state of the system held by some other participant. Interfaces are roles that *declare* a set of behaviours, and may correspond to Java classes and interfaces. Implementations are roles that *define* a set of behaviours, and corresponds to Java classes. The intention of this is the separation of concerns between those participants that declare a type or behaviour. This is mirrored in operations (*declare* behaviour) and methods (*define* behaviour). Although such distinctions between declaration and definition are appropriate, the way in which they are presented feels artificial. We believe that a more appropriate abstraction are class and method participants with a constraint to indicate which are abstract.

DPML also introduces a notion of *dimension*, which allow similar participants in a motif to be collected together. These translate to sets of elements playing the desired participant role, a simple and important mechanism of abstraction not present in UML. A dimension can be associated with more than one participant, indicating that these sets have the same number of elements, but exactly how many is abstracted. The addition of sets requires quantification to facilitate constraints on participants in dimensions. In DPML, constraints are essentially either unary relations on a participant or binary relations between them, each intuitively named. Constraints on participants with dimensions are a little more complex. A *total* relation requires that the relation holds between every possible pair of elements; a *regular* relation requires exactly one relationship holds for each element; a *complete* relation requires that all elements in one dimension relate to something in the other; and finally an *incomplete* relation exists between only one pair of elements in each dimension. Relations can be composed when dealing with multiple dimensions, which are called *extended* relations, e.g. a regular complete relation. The visual notation of these elements are summarized in Figure 3.3.

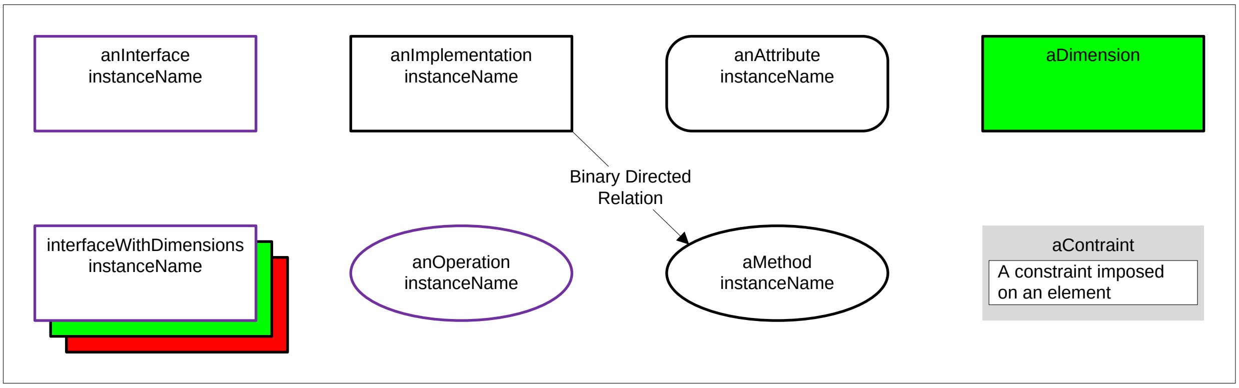


Figure 3.3: The basic notation of DPML [Maplesden et al., 2004]

Note the significance of colour in the above notation. A dimension has a unique colour by which it can be identified, such as green for the dimension "aDimension". Identifying a dimension by colour minimizes the amount of text in a diagram without losing information. However, reliance on colour raises serious accessibility issues, such as for colour blind readers. Greyscale printing can also render DPML specifications unreadable.

Let us now demonstrate DPML in specifying the Abstract Factory motif (Figure 3.4) mentioned previously, which specifies the following participants:

- The interfaces $AbstractFactory$ and $Products$
- The implementations $ConcreteFactories$ and $ConcreteProducts$
- The operation *createOps*
- The method $ConcreteCreateOps$

Figure 3.4 also depicts the dimensions Products, which is associated with the participants *createOps*, *Products*, *concreteCreateOps*, and $ConcreteProducts$; and Factories, which is associated with the participants $ConcreteFactories$, *concreteCreateOps*, and $ConcreteProducts$.

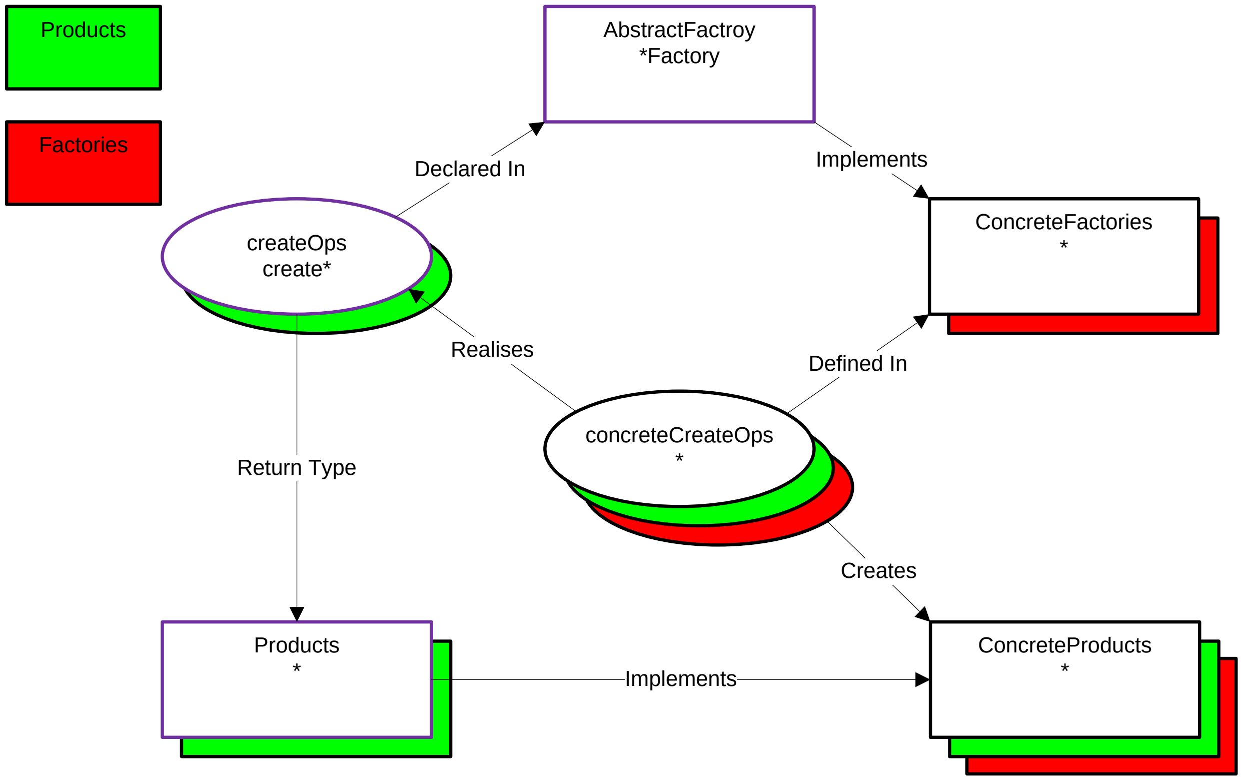


Figure 3.4: The Abstract Factory motif in DPML [Maplesden et al., 2004]

These dimensions ensure that there are as many *createOps* operations as there are *Products*. Each set of *createOps* operations is realized by a set of *concreteCreateOps*, where there are as many sets of such methods as there are *ConcreteFactories*, and finally for each *ConcreteFactory* there is a set of *ConcreteProducts*. Although Figure 3.4 provides very few visual clues to these details, [Maplesden et al., 2007] does state the types of each relation. For example *Return_Type* is a regular relation (between *createOps* and *Products*); *Declared_In* and *Implements* (between *AbstractFactory* and *ConcreteFactories*) are complete relations; *Defined_In*, *Realises*, and *Implements* (between *Products* and *ConcreteProducts*) are extended relations (regular, complete); and finally *Creates* is an extended relation (regular, regular).

Design verification in DPML is a process of instantiation, as illustrated in Figure 3.5. Specification diagrams are refined to instantiation diagrams, which map (bind) the participants in the specification diagram to components of the UML Object Model. That is, the elements in a specification diagram are *instantiated* (represented by the solid lines) to elements in an instantiation diagram, which in turn are *realized* (represented by the broken lines) in the UML Object Model. In addition, instantiation diagrams can be tailored for the specific application the motif is to be used for. Therefore, in this case design verification checks the consistency of the UML Object Model with a specification diagram, via instantiation diagrams. To accomplish design verification of the actual software, the UML Object Model would have to be verified against some implementation. Forward engineering an implementation would be one approach, but one that this is not appropriate considering part of our focus is on the evolution of *existing* systems. To our understanding, at this time, verifying a UML Object Model against an implementation is still an open problem. “No formal definition exists of how the UML maps to any particular programming language. You cannot look at a UML diagram and say exactly what the equivalent code would look like. However, you can get a rough idea of what the code will look like” [Fowler, 2003]. To our knowledge two tools exist that support design verification in DPML as we discussed, DPTool and the more recent MaramaDPTool [Maplesden et al., 2004].

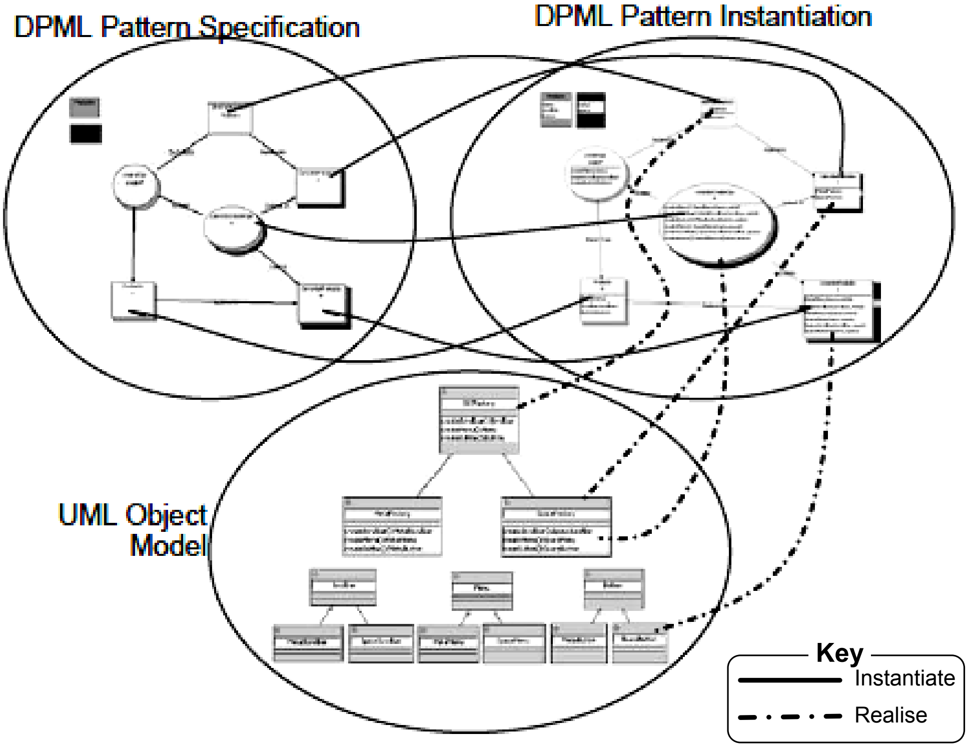


Figure 3.5: Mapping DPML to the UML Object Model [Maplesden et al., 2007]

To summarize, DPML specification diagrams capture more about the Abstract Factory motif (Figure 2.3) than we were able to in UML (Figure 3.1). That is, the notion of dimension (sets of participants) facilitates the specification of an unbounded number of possible implementations. We found the notation to be relatively unclear, and the reliance on colour does raise certain accessibility problems[19]. Design verification exists in some form, although it is not appropriate in our context. For these reasons DPML is much better at expressing motifs at an appropriate level of abstraction in comparison with UML, but still does not meet our criteria in regards to its practical application

[19] Obviously any visual language has accessibility problems for blind users, but as DPML relies on colour there are additional concerns for colour blind readers.

to design verification. In the next section we discuss a language that completely departs from the cloud of UML related technologies.

3.3 Spine

First appearing in [Blewitt et al., 2001], and most recently in [Blewitt, 2007], Spine is a language for capturing object-oriented design patterns based on the logic programming language Prolog. There are three reasons for this choice as given in [Blewitt, 2006]:

- "Prolog rules are a natural way of defining rule-based proof systems
- The syntax of Prolog is very simple (and therefore easy to parse, understand and extend)
- The language allows other patterns to be easily defined"

The expressive and intuitive notation of Spine is one of its strong points. Like DPML, Spine has the ability to group participants into sets, although it does so indirectly, and simple quantification. To demonstrate this we present the truth conditions for the basic formulas (φ) of the language, summarized from [Blewitt, 2006][20]:

- **true** is true
- **false** is false
- $\mathsf{and}([\varphi_1, ..., \varphi_n])$ is true iff every formula φ_i is true for $1 \leq i \leq n$
- $\mathsf{or}([\varphi_1, ..., \varphi_n])$ is true iff at least one formula φ_i is true for $1 \leq i \leq n$
- $\mathsf{not}(\varphi)$ is true iff formula φ is false
- $\mathsf{forAll}(S, x . \varphi(x))$ is true iff for all x in set S, formula $\varphi(x)$ is true
- $\mathsf{exists}(S, x . \varphi(x))$ is true iff there exists at least one x in set S where formula $\varphi(x)$ is true
- $\mathsf{realises}(P, [v_1, ..., v_n]) :- \varphi_1, ..., \varphi_m$ defines a Spine *rule* called P iff P is an unused constant name, every v_i $(1 \leq i \leq n)$ is a variable name, and every φ_j $(1 \leq j \leq m)$ is a formula over $v_1, \ldots, v_n$
- $\mathsf{realises}(P, [c_1, ..., c_n])$ is true iff P is a Spine rule that takes n arguments, and every formula defined in P is true for values $c_1, \ldots, c_n$

[20] In the interests of simplicity, we have taken the liberty of making modifications to how these rules are presented.

Where SPINE rules are definitions of patterns, i.e. they are SPINE specifications. This is then built on with a collection of domain specific relationships (*evaluable propositions*), a small selection of these are paraphrased here:

- instantiates (m,c) — is true iff the member m creates an instance of class c
- isAbstract (m) — is true iff the member m has the Java modifier **abstract**
- removes(m,c,f) — is true iff the method m removes an instance of class c from the collection referenced by field f
- sameSignature(m_1,m_2) — is true iff member m_1 has the same signature as m_2
- subtypeOf(c_1,c_2) — is true iff c_1 is a subtype of c_2
- typeOf(m,c) — is true iff member m is of type c, if m is a method then c is its return type

Where a member could be a method, field, class, and so on. Although SPINE is designed for representing motifs, it is clearly heavily tied to Java. This is exemplified by the choice in relations and how they are defined. For example, the semantics of many relationships are given only in terms of Java keywords, such as the isAbstract relation above. Additionally, some relations are a vague in their definition and have questions regarding their decidability, such as the removes relation[21].

SPINE also includes *evaluable sets*, which are derivable directly from the source code of a program. They are always finite as a program is always considered to be finite. For example:

- methodsOf(c) — evaluates to the set of methods in class c
- subclassesOf(c) — evaluates to the set of subclasses of class c

Having briefly summarized only a small subset of SPINE, we are able to demonstrate its applicability to specifying motifs. Figure 3.6 specifies the Abstract Factory motif (Figure 2.3). We believe that AbstractFactory2 essentially specifies the Factory Method pattern, although this observation is not shared in [Blewitt, 2006][22]. In the terminology of [Gamma et al., 1994] the four class participants AF, AP, CF and CP are "Creator", "Product", "ConcreteCreator" and "ConcreteProduct" respectively. This definition obfuscates exactly which method is a factory method, but indicates

[21] E.g. what constitutes a collection, e.g. is an array a collection? What does it mean for an instance to be removed from that collection? Is it decidable that the instance is always removed, or is it good enough to capture the possible attempt to remove an instance?

[22] This is not the first time seeing this hypothesis. [Eden and Nicholson, 2011] also states that the Abstract Factory is a collection (abstraction) of many Factory Method instances.

that one must exist. It must be defined in AF, overridden in CF, instantiates CP and returns it. Interestingly, the definition of AbstractFactory requires that all subclasses of the class AF must comply with AbstractFactory2 with respect to CF and one of the subclasses of CP.

```
realises(‘AbstractFactory‘,[AF,AP]) :-
  forAll(subclassesOf(AF),
    CF.exists(subclassesOf(AP),
      CP.realises(‘AbstractFactory2‘,[AF,CF,AP,CP])
    )
  )

(* CF is a sub-type of AF and CP is a sub-type of AP, such that CF
    generates CP *)

realises(‘AbstractFactory2‘,[AF,CF,AP,CP]) :-
  subtypeOf(CF,AF),
  subtypeOf(CP,AP),
  exists(methodsOf(AF),
    M1.and([
      typeOf(M1,AP),
      isAbstract(M1)
      exists(methodsOf(CF),
        M2.and([
          sameSignature(M1,M2),
          typeOf(M2,AP),
          instantiates(M2,CP),
        ]))
    ]))
```

Figure 3.6: The Abstract Factory motif in Spine [Blewitt, 2006]

This is not only a good demonstration of using Spine to specify motifs, but also of using it to compose larger and more complex motifs from smaller ones. Spine also has fully automated design verification against Java implementations as provided by the Hedgehog proof engine. Although Hedgehog is not as conclusive as we would like as in some cases an “unknown” result can occur. When such inconclusive or failure results are reached, user readable messages are presented so that they may be appropriately addressed. Documenting errors in this way is essential for the evolution of programs. Without accurate indications of faults they become increasingly difficult to resolve.

To summarize, Spine is certainly an improvement over the both DPML and UML, but it is not without its drawbacks. For example, Spine is designed to be used with Java[23] rather than for object-oriented languages in general. As design patterns are an abstraction it seems counter to their purpose to limit their application to only one implementation language. As Spine is based on Prolog, it also has the same formal background. However, Prolog only allows terms to be atoms, numbers, variables or compound terms, yet several relations in Spine suggest a more rich set of types. From those relations we have mentioned above it can be seen that there is implicit notion

[23] “...all of the Spine predicates are tightly focussed on the Java implementation.” [Blewitt, 2006]

of classes, methods, method signatures, and fields. It is our opinion that such types should be explicit in our language. In terms of tool support for design verification, Hedgehog is the most advanced we have examined so far. However, due to some undecidable predicates it cannot always conclusively return a pass or fail result.

3.4 Summary

From the discussion on existing languages above we collect a brief collection of summary points:

1. *Visuality* — Languages for the specification of design motifs (see p.17) need not be visual (graphical, or diagrammatic), as exemplified by Spine. Both DPML and Spine also demonstrate that formal languages need not be difficult to understand if care is taken in their definition. However, for our purposes this aspect is less important as we focus on the theory and its applications. How that theory may be presented to an eventual user, perhaps in some graphical form, is the concern of a different research project.

2. *Expressivity* — Although UML is very good at representing object-oriented design of specific programs, it lacks appropriate mechanisms for expressing motifs. DPML, which sits on top of UML, makes a clear distinction between the declaration and definition between participants, but our intuition is that such distinctions over-specify the problem. Spine is an improvement, but being Java-centric and including a great deal of implementation detail makes it difficult to reuse for any other programming language. What is needed is a good balance of form and function; a language that is simple and intuitive to learn and use, yet expressive enough to capture sufficiently interesting design decisions.

3. *Reusability* — Spine demonstrates that it is extremely desirable to be able to reuse specifications. The ability to dynamically extend the language is a useful feature not present in any of the other languages discussed, and one which we shall incorporate into our theory.

4. *Deviation from UML* — Spine demonstrates that deviating from the cloud of UML extensions and variations yield very positive results, so we need not limit ourselves to this class of languages.

5. *Design verification* — UML does not currently provide design verification as we defined in §2.3, although there is active research into this area. “No formal definition exists of how the UML maps to any particular programming language. You cannot look at a UML diagram and

say exactly what the equivalent code would look like. However, you can get a rough idea of what the code will look like" [Fowler, 2003]. DPML sits on top of UML and facilitates design verification against the UML Object Model of a program, but not against the implementation as desired. SPINE is a step closer to our definition, however as not all components of the language are decidable the result of performing design verification can be inconclusive.

6. *Automated tool support* — All languages discussed had a degree of tool support, UML having the greatest repository of tools from those that focus on diagramming [ArgoUML Open Source Community, 2010], to ones that target problems such as reverse engineering [Fujaba Development Group, 2009]. Although UML has so many tools available, as the language does not yet support design verification there are no tools that support automating the process. As for the other languages we discussed, DPML has two tools we know of: DPTool, and the more recent MaramaDPTool [Maplesden et al., 2004]. Both of these tools automate their version of design verification against UML Object Models. And finally, SPINE has the HEDGEHOG [Blewitt, 2006] proof engine. HEDGEHOG automates SPINE's version of design verification, but as stated previously, some properties in the language are not decidable and therefore the tool can return inconclusive results. We require that tool support for our language always be conclusive, requiring that the language it supports be fully decidable.

7. Where appropriate, each tool supports its own language's mechanism of design verification.

In the next chapter we will introduce LePUS3, and discuss it in light of the above.

Chapter 4

LePUS3: Language for Pattern Uniform Specification

> "We have only recently come to a realization of the mathematical and logical basis of computer programming: we can now begin to construct program specifications with the same accuracy as an engineer can survey a site for a bridge or road, and on this basis we can now construct programs proved to meet their specification with as much certainty as the engineer assures us his bridge will not fall down. Introduction of these techniques promises to transform the arcane and error-prone craft of computer programming to meet the highest standards of a modern engineering profession." [Hoare, 1989]

According to Robin Milner, the problems related to the design of software systems can only be tackled successfully by combining theoretical investigation with practical experience:

> "The design of computing systems can only properly succeed if it is well grounded in theory, and that the important concepts in a theory can only emerge through protracted exposure to application." [Milner, 1987]

Christopher Strachey was even more explicit: the very reason why current approaches fail is because they are either impractical or because they lack a sound theoretical underpinning:

> "It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing." [Hoare, 2000]

The requirement that Strachey and Milner have expressed is not surprising. After all, the combination of theoretical insight with usability is the hallmark of every successful engineering discipline. Unfortunately, most existing design description languages are lacking on at least one of these aspects.

Balancing theory and practice is a driving principle of LePUS3—the third version of the oft cited LePUS design description language. LePUS3 is a visual design description language for representing object-oriented design, specifically design motifs, at an appropriate level of abstraction. LePUS3 brings together strong theoretical background, being well grounded in the theory of object-oriented languages and formally defined in the First-Order Predicate Logic (**FOPL**), and practical application to software development, primarily in the form of design verification.

Unfortunately, the history and evolution of LePUS3 is not well publicized and many still reference the original LePUS as the current state of the language. In order to help explain the evolution from LePUS to LePUS3 we present a time line that summarizes the development of LePUS and associated tool support.

1999 LePUS—Language for Pattern Uniform Specification—appears [Eden, 1999]. LePUS was a formal language, defined in higher-order monadic logic, with both visual and formulaic representations for specifying design pattern motifs at an appropriate level of abstraction. Tool support and fully automated design verification was always one of the main motivations of LePUS, for which Prolog was indicated as a candidate implementation language. LePUS has received a great deal of attention, being ubiquitous in the field [Taibi, 2007a], and having accumulated a large body of derived work. For example ExLePUS [Mak et al., 2003], eLePUS [Raje et al., 2007], and BPSL [Taibi, 2007b].

2005 Through the years LePUS had continued to develop to the point that it required a new name to avoid ambiguities between publications. This language became known as LePUS2 [Gasparis, 2005], and was now defined in the First-Order Predicate Logic (**FOPL**). The visual notation was simplified, and perhaps most importantly introduced constants allowing the language to be used to specify specific programs and frameworks alongside pattern motifs. However, the language was little publicized and was quickly superseded. It now stands as a widely undocumented and extinct language. It was also around this time that the schematic representation of LePUS2 was named Class-Z.

2006 Version 0.3 of the Two-Tier Programming Toolkit (the Toolkit) released, created by E. Gasparis, was loosely based on a proof of concept prototype [Iyaniwura, 2003] and incorporated rewrites of previous student projects [Bo, 2004, Liang, 2004]. It was mainly used for student research, facilitating practical experimentation with design verification [Nicholson, 2006] and a graphical editing [Fragkos, 2006, Ayodeji, 2006]. It was at this time that J. Nicholson joined the team and suggested changes that were to be incorporated in LePUS3 the following year. For example, J. Nicholson suggested what became the fourth axiom of class-based programming (LePUS3 Definition VIII, Appendix B), numerous small changes to many existing definitions, and stylistic changes.

2007 LePUS3 first appeared in [Eden et al., 2007b, Gasparis et al., 2008b], which greatly revised the definition of LePUS2. To improve the visibility of LePUS3, we launched improved websites that distinguished the language [Eden et al., 2008] from the Two-Tier Programming Project [Nicholson et al., 2008]. LePUS3 spawned further publication on the problems of design verification [Nicholson et al., 2009] and program visualization [Gasparis et al., 2008a, Gasparis, 2010]. LePUS3 has also been used as a teaching tool by A.H. Eden in postgraduate courses for several years.

2008 Version 0.5.1 of the Toolkit released, a complete redesign of version 0.3, written by E. Gasparis and J. Nicholson. Includes more advanced implementations of features from version 0.3 such as the Java 1.4 abstract semantics and XML import/export. It also included a redeveloped design verification algorithm and chart editor, with the first implementation of a new design recovery tool called the Design Navigator. Since its release it has had additional features and updates made in accordance with user feedback, such as a pattern library feature.

2009 We conducted a small controlled experiment that shows significant evidence of productivity gains when using the Toolkit (version 0.5.3) over an alternative industrial tool. These results of this experiment were originally published in [Eden and Gasparis, 2009], and will be reexamined in §13.3.

2010 Specifications articulated as LePUS3 charts became officially referred to as Codecharts [Eden and Nicholson, 2011].

This timeline should make clear that very little of the original LePUS still holds for LePUS3. In the remainder of this chapter we will describe LePUS3 in more detail, where the definitions of LePUS3 are provided in Appendix B. As it is the most publicized and recognisable, we primarily discuss LePUS3's visual notation rather than the schematic notation of Class-Z.

4.1 Notation

LePUS3 has an elegant and minimal vocabulary, summarized in Figure 4.1. Constants, which are grey filled shapes in a monospace font, represent specific identified (bound) program components, whereas variables, which are unfilled shapes in an italic roman font, represent as yet unidentified (unbound) program components. Each shape represents a different sort of program component, where ellipses represent method signatures, rectangles represent classes, and triangles represent inheritance hierarchies. Methods are represented by superimposing (overlaying) a method signature over a class or hierarchy, which requires that the methods interface complies with the signature and it is either defined in or inherited by the class or hierarchy. This is a notion that expands naturally when sets are introduced in the form of *dimensions*.

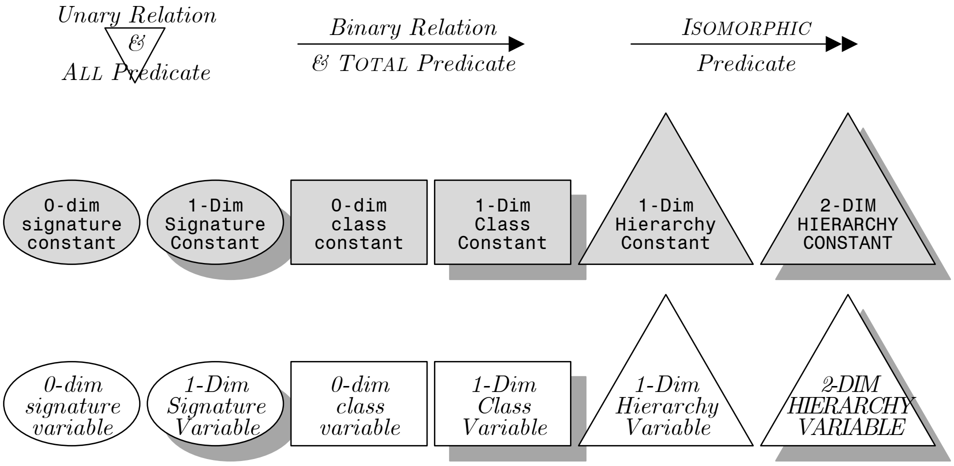


Figure 4.1: The vocabulary of LePUS3

Dimensions occur in both LePUS3 and DPML, but the concepts are distinct. In DPML, dimensions are named visual primitives that are associated with participants and represent a set. Additionally, when a dimension is associated with several participants, each set thereof must have the same number of elements. In LePUS3, dimensions simply stand for sets of terms, and is represented by decorating a symbol with a shadow. This terminology originates from geometry, where a 0-dimensional term is a single term (point), a 1-dimensional term is a set of terms (line), a 2-dimensional term is a set of sets (plane), and so on. In our experience 3-dimensional or higher terms are rarely needed.

As already discussed in §2.1.3, an inheritance hierarchy is an important concept in object-oriented design: a set of classes that contains a superclass such that all other members inherit (directly or indirectly) from it, as in Figure 4.2 for example. As hierarchies are sets, they are interpreted as 1-dimensional terms that are not decorated with a shadow. Triangles with shadows are therefore 2-dimensional (sets of) hierarchies.

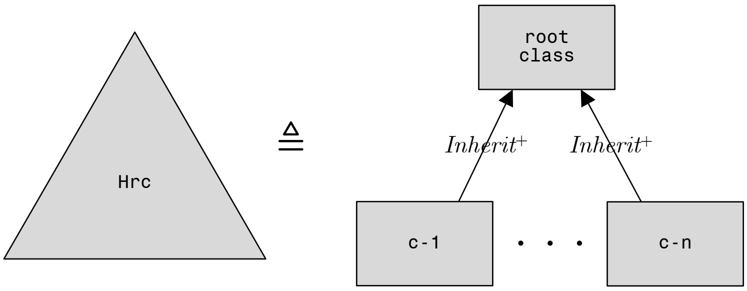


Figure 4.2: Illustration of hierarchies

LePUS3 is restricted to decidable relations, those that can always be conclusively identified in a program via static analysis. Examples of such relations are *Inherit*, *Abstract*, *Member*, and *Aggregate*, where unary relations are represented by inverted triangles, and binary relations by directed edges. This limitation therefore makes LePUS3 less expressive than Spine, yet it still captures enough sufficiently interesting detail of object-oriented design. As in DPML, LePUS3 provides complex relations (predicates) for dealing with constraints on and between sets, and extend naturally for nested sets. LePUS3 does not allow the composition of predicates as one can in DPML, but in our experience only three such predicates are required for all but the most obscure cases.

All is a simple unary predicate, requiring that all elements of a given set are in the specified relation. This is exemplified by Figure 4.3. As with all aspects of LePUS3, care is taken to avoid cluttering the notation and to keep close to the principle of elegance. For this reason the unary relation and All predicate symbols are intentionally indistinguishable. This is permissible while maintaining an unambiguous language because that which is specified is always derivable from the context of its use. The unary relation symbol is only interpreted for 0-dimensional terms, and the All predicate in all other cases.

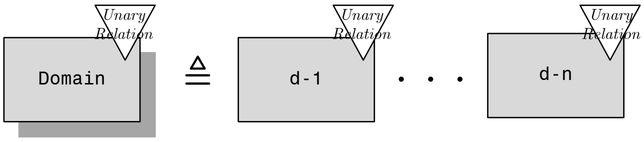


Figure 4.3: Illustration of the All predicate using classes

Total is a binary predicate, the intention of which originates in the definition of total functional relations. Abstractly, this predicate specifies that every element of the set on the domain is related by the given relation to some specific element of the range. As with the All predicate and unary relation symbols, in the interests of elegance the symbol for the Total predicate is indistinguishable from the directed edge symbol of binary relations, where that which is specified is always derivable from the context. Specifically, the TOTAL predicate is only interpreted when at least one of the symbols it is connected to represents an n-dimensional term, where $n > 0$.

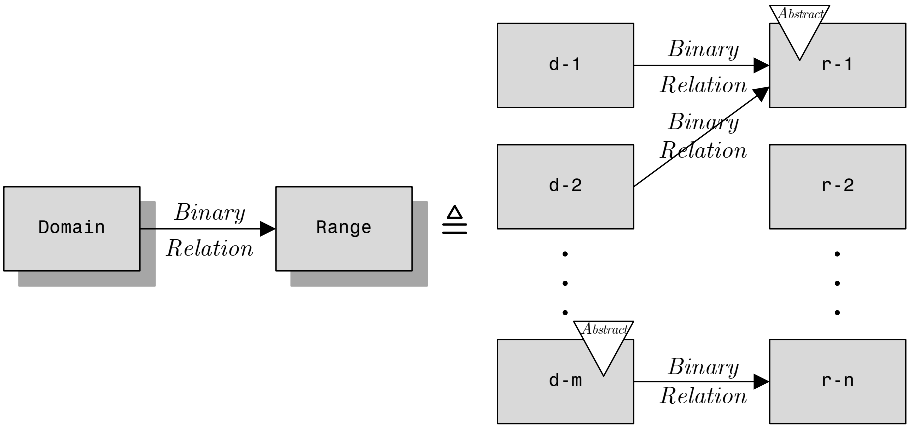


Figure 4.4: Illustration of the TOTAL predicate using classes

ISOMORPHIC is a binary predicate, the intention of which originates in the definition of isomorphisms. Abstractly, this predicate specifies a pairing of elements from the domain and range, with respect to the given relation. There is an exception made for abstract terms, which are intentionally excluded. The ISOMORPHIC predicate is represented with a double headed directed edge and associated relation symbol.

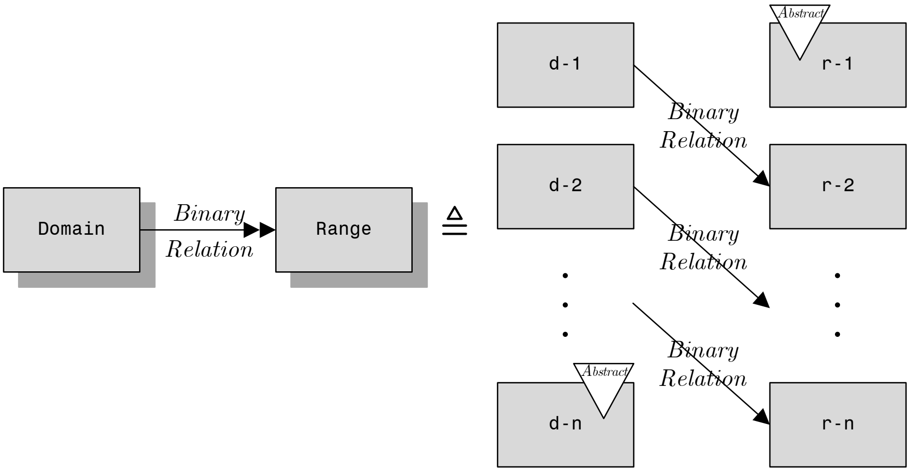


Figure 4.5: Illustration of the ISOMORPHIC predicate using classes

We will now demonstrate the notation of LePUS3 in Figure 4.6 by specifying the Abstract Factory motif (Figure 2.3).

This is the simplest and most elegant representation of the Abstract Factory motif we have seen so far, having only three sets of participants. To summarize, the Codechart states that there is a hierarchy of factories, a set of classes that all inherit from a single root class also contained in

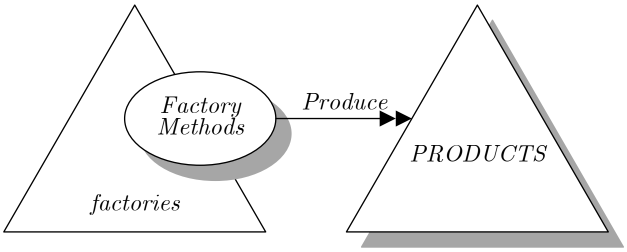


Figure 4.6: The Abstract Factory motif in LePUS3

that set. Each factory therein has a set of dynamically bound factory methods. For each factory class there is a, 2-dimensional, hierarchy of products (a set of hierarchies). Each product hierarchy contains exactly one product class that is produced (created and returned) by each factory method. That is, there is a one-to-one correspondance between a product hierarchy and a set of factory methods in a factory.

4.2 Design Verification and Tool Support

Since Codecharts are decidable statements about classes and methods, design verification in this context merely requires comparing specifications with the collection facts about the program that may potentially be relevant. Design models represent these decidable program facts. A design model is a simplification of a program—its abstract semantics—because it strips the source code from its complex grammatical structure. Instead, design models define a universe of primitive entities and relationships that can easily be reasoned over. Tools that generate design models, such as the Two-Tier Programming Toolkit which we discuss in detail in Chapter 13, use static analyzers to reverse-engineer them from the source code in the form of a relational database.

The Codechart is then bound (assigned) to elements of that design model, at which point design verification becomes mostly a matter of quantification and set membership. Design verification in LePUS3 is always conclusive, as opposed to Spine. LePUS3 also has tool support for design verification as demonstrated in the Two-Tier Programming Toolkit [Nicholson et al., 2009]—the Verifier. The Two-Tier Programming Toolkit also demonstrates that the language can also be used in other fields, for example the Design Navigator is a design recovery [Gasparis, 2010] tool which we show in Figure 4.7. The Design Navigator, which we discuss in more detail in §13.1, allows for a user-driven step-wise bird's eye view of large programs. Figure 4.7 shows this process being applied to Java's Abstract Window Toolkit. Users can selectively show greater levels of detail of a program (*zoom-in*), or abstract away from details that they are not interested in (*zoom-out*). This is achieved by selecting a symbol on the Codechart, and applying one of the navigation operators, shown on the left hand side of Figure 4.7. The Design Navigator and the Verifier have both been shown to make significant productivity gains in a small empirical study [Eden and Gasparis, 2009], which we will discuss more in §13.3.

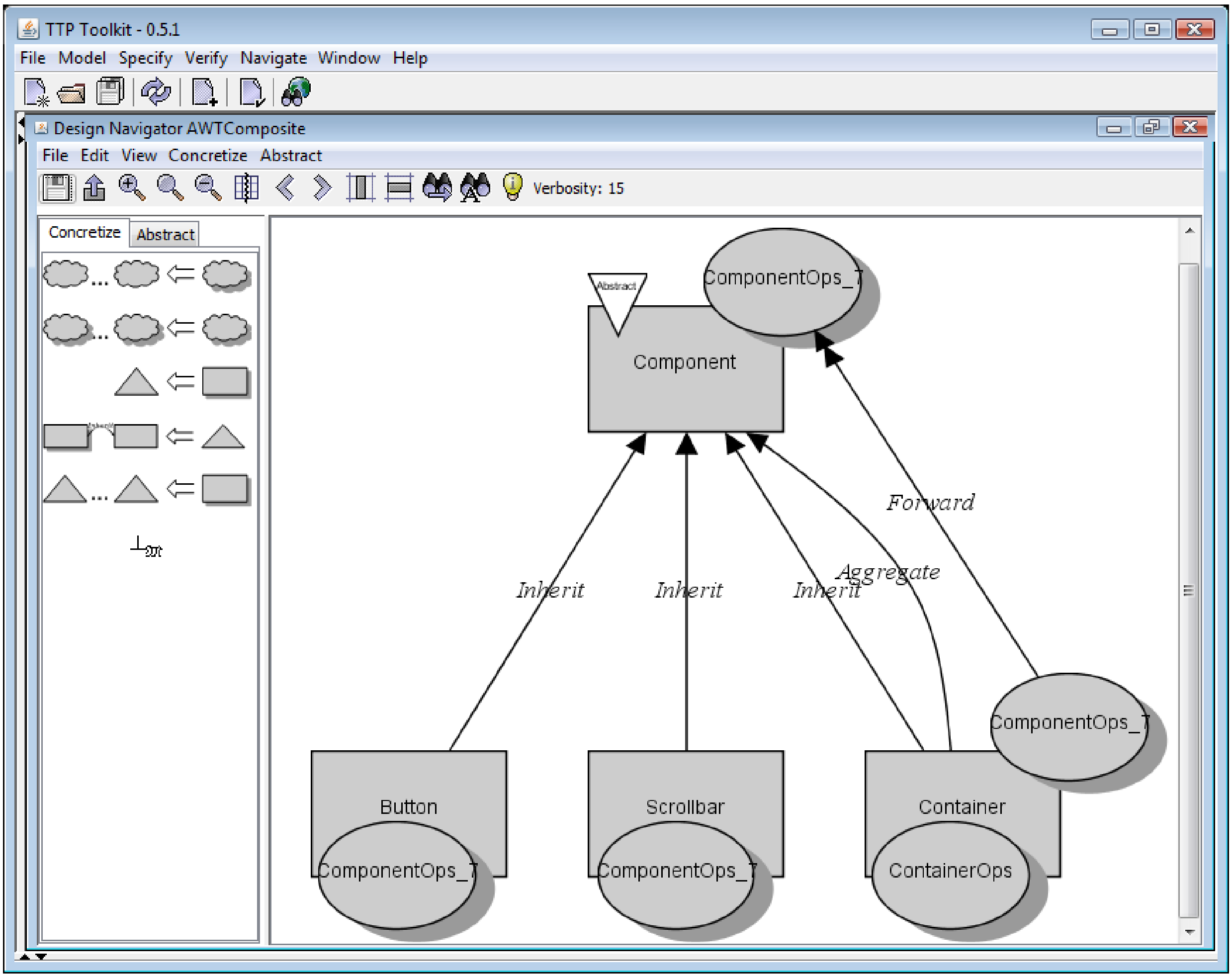


Figure 4.7: The Two-Tier Programming Toolkit in action

4.3 Issues and Criticisms

One of the common criticisms of LePUS3[24] is that it is not as expressive as UML. As LePUS3 is restricted to decidable properties of object-oriented design it does not capture notions such as behaviour, events, or state. LePUS3 is so restricted as its charter places a great deal of its emphasis on automated and conclusive tool support for design verification at the sacrifice of expressivity. Despite this, the language has demonstrated that it is expressive enough to

[24] We make this observation from reviews and feedback received after attempts (successful and otherwise) to publish material on LePUS3.

capture interesting structural and creational properties of object-oriented design, as exemplified by [Eden et al., 2007a]. LePUS3 therefore serves a different, but complementary, role to UML.

Another more recent criticism has been brought to our attention with respect to our terminology, specifically using the words "total" and "isomorphic" for the names of the respectively named predicates. This is a recent criticism as only in LePUS3 has the definition of these predicates become more specialized. We appreciate this criticism, and therefore will change the names of the predicates to Tot and Iso respectively (Chapter 7). These new names should still convey a hint toward their origins, but are distinct from the well known mathematical terms.

Terminology is not the only issue that has arisen through the evolution of LePUS3. Despite our best efforts LePUS3 has suffered from the same increase in complexity as occurs with evolving software. Over time the language has grown more complex and fragile; the underlying theory has become cumbersome and obfuscated. This means that further extension is getting progressively difficult, and that steps must be taken to manage its increased complexity.

One of the main causes of this complexity is that the definitions (Appendix B) do not make a clear distinction between the definition of the language and its application to design verification. For example, what the *Inherit* relation means depends entirely on how a program is abstracted to a design model. Moving the semantics of the language to the design model allows LePUS3 to be easily applied to other programming languages, and as such is more flexible than languages like Spine. However, this greatly limits what reasoning can be done within the language itself; Codecharts only really have a meaning once they are considered with respect to a given design model. The definitions do begin to address this issue by providing four axioms of class-based object-oriented languages, LePUS3 Definition VIII (Appendix B), which we reiterate here:

Axiom 1 No two methods with the same signature are members of the same class.

Axiom 2 There are no cycles in the inheritance graph.

Axiom 3 Every method has exactly one signature.

Axiom 4 If a method produces instances of a class it also creates it and returns it; if one method forwards the call to another it can be said to call it; and if one class holds an aggregate of another, it can also be said to hold a member of it.

These axioms articulate some of the basic conditions that any design model must satisfy. As such they are meta-constraints over the validity of design models, and not a part of the LePUS3 language. This is a severe limitation on the reasoning capability of LePUS3, as any reasoning requiring these axioms is only achievable at the meta-level. For example, consider Figure 4.8, a motif that presents two classes that inherit from each other. Intuitively this Codechart is invalid as it violates the second axiom, i.e. one of the *Inherit* relationships must be false in any given design model. Ideally this conclusion should be drawn by simply examining the rules that govern the *Inherit* relation as defined within the language. In LePUS3, however, the language does not have any such rules thereby making it impossible to draw this conclusion from the language alone. Rather we can only reach this conclusion by reasoning about the Codechart at the meta-level, i.e. outside the language where we can make use of the above axioms. The ability to reason over Codecharts in LePUS3 is therefore extremely limited and restricted to the meta-level. We must address this issue by not only redefining LePUS3 so that specifications can be reasoned over in the language, but also by extending and refining it to articulate more axioms of class-based object-oriented design.

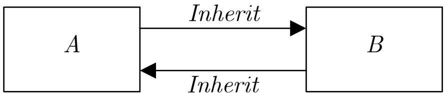


Figure 4.8: A contradiction in a LePUS3 Codechart

Our final criticism of LePUS3 is its treatment of types. The type system in LePUS3 is fixed, where the static nature of these types make the language very inflexible, and prevents polymorphism within the language. For example, [Gasparis, 2010] adopted a cloud-like symbol as notation for a “term of any permissible type”, but this was only meta-notation as LePUS3 does not facilitate variables over types. Another problem with such a static approach to types is the lack of subtyping. For example, a hierarchy is a static type defined as a set of classes that comply with certain constraints[25]. The definition of hierarchies, and the way in which they are used, might imply that they are subtypes of sets of classes. However, the type system of LePUS3 lacks subtyping, so such an interpretation is purely informal. To address this criticism requires radically changing the type system of LePUS3, to introduce both subtyping and polymorphism.

4.4 Summary

LePUS3 admirably balances the theory and practice of object-oriented design. LePUS3 is well suited to articulating the decidable structural and creational details of design motifs, application

[25] See LePUS3 Definition IV.

frameworks, and concrete programs. The notation is simple and elegant, capable of representing sufficiently interesting aspects of object-oriented design. The notation is also intuitive and a good teaching tool, having been used by A.H. Eden in postgraduate courses for several years. The broad body of work on, or affected by, the original LePUS is an indication of the impact on the field that language made, which LePUS3 inherits. Design verification is decidable and therefore implementable as a fully automated and conclusive tool, which is demonstrated in the Two-Tier Programming Toolkit. The practical applications of LePUS3 are not limited to design verification however, for example it has been applied to the field of design recovery with the Design Navigator tool [Gasparis et al., 2008a].

Although LePUS3 is not without its limitations, such as its dedication to decidability, its complicated and brittle definition, inadequate type system, and lack of reasoning capability in the language, we believe it is the most appropriate choice for our investigation. In the next chapter we introduce and discuss the Typed Predicate Logic (**TPL**), which we will use in Part II to define, refine, and extend LePUS3 to address our discussed criticisms.

Chapter 5

Typed Predicate Logic

Types are a fundamental concept in our investigation into object-oriented design. Our notion of type originates with *data types* in computer science [Pierce, 2002] that are operated over by computable relations and functions. Therefore, programming languages afford our first set of intuitions about such types from how they form their type systems around the data they operate over:

> "Natural numbers, characters, tuples of numbers, lists of numbers, finite sets of characters, classes of objects, and stacks of characters are all examples of types of data. In particular, they are the kinds of things that occur as data types in programming languages." [Turner, 2009]

However, to limit our understanding to this perspective would imply that implementation languages dictate what is to be considered a type. Clearly such a characterization is too specialized as it does not capture the abstract nature of types. Alternatively, we could allow for types that can be interpreted into a programming language, i.e. we allow a new type as long as we can implement it in some programming language. In this way a programming language is a model for a type, in which all axioms governing that type must hold. Unfortunately, programming languages do not generally have an adequate mathematical characterization in which to prove this is the case.

> "In isolation, a programming language is just that, i.e., a language. And without some mathematical interpretation of its constructs, aside from the formal nature of its grammar, it has no mathematical content. And neither do the programs written in it. Such content has to be imposed upon it via a semantic account (of some kind) and it is this that renders it amenable to mathematical analysis." [Turner, 2009]

Therefore, a more appropriate perspective is that types are determined by axiomatic theories, and not syntactic structures afforded by programming languages. The Typed Predicate Logic (**TPL**) captures exactly this notion of type. **TPL** is "a broad framework in which a rich variety of theories [of data types] can be easily and elegantly formulated" that is "sufficiently flexible to elegantly support a wide range of such constructors, including dependent types, subtypes, and polymorphism" [Turner, 2009]. As such, **TPL** is an ideal framework on which to redefine, clean, and extend the type theory of LePUS3. We devote this and the following chapter to a brief summary of **TPL**.

Broadly speaking, **TPL** differs from the First-Order Predicate Logic (**FOPL**) primarily in its treatment of types. In **TPL** the role that type theories play is parallel to the role of first-order theories in **FOPL**. To clarify, in **FOPL** "the notions of *operation*, *property*, *relation*, *object* are bare notions. They gain substance in the context of first order theories" [Turner, 2010]. Although **FOPL** can be extended to admit types, these are usually based on extensional set-theoretic notions [Huth and Ryan, 2000, Pierce, 2002]. As such we do not believe they put sufficient emphasis on the role that types play. In comparison, in **TPL** "the notions of *operation*, *property*, *relation*, *object* and *type* are bare notions. They gain substance in the context of type theories" [Turner, 2010]. **TPL** thereby treats types as primitive intensional notions.

Another important distinction between **FOPL** and **TPL** is that of the number and use of judgments admitted. Traditionally, the syntax of **FOPL** is defined using an over-generating context-free grammar. Syntactically valid sentences are then given meaning and pruned by semantic rules that have only one judgment form in their conclusions—that a proposition holds. In **TPL** the syntax is not predetermined by a context-free grammar. **TPL** is a many-sorted natural deduction system where the syntax and semantics of **TPL** are all defined within the rules of the system, "a type-inference system that is constituted by the *membership* and *formation* rules for types and propositions" [Turner, 2009]. To accomplish this, **TPL** requires the following four judgments (Θ):

$$\begin{array}{rl} T\ \mathit{type} & T \text{ is a type} \\ t : T & \text{Term } t \text{ is of type } T \\ \phi\ \mathit{prop} & \phi \text{ is a proposition} \\ \phi & \phi \text{ holds} \end{array}$$

Where a term t is either a variable (denoted t), or the result of a function[26] (such as $f\,(t_1, \ldots t_n)$). As is traditional we consider constants to be the result of nullary functions (denoted $\mathtt{t}$). The membership rules mentioned earlier are those that conclude a judgment of the form T *type*, whereas formation rules conclude judgments in the form of ϕ *prop*. We show what conclusions follow from

[26] We write *function* (*functional*) to indicate a partial function, unless otherwise stated.

a set of premises using the normal sequent notation:

$$\Gamma \vdash \Theta$$

where Θ is one of the aforementioned judgments, and Γ is a *context*. A context is a finite sequence of judgments of the form $t : T$, i.e. variable t is *declared* to be of type T, and ϕ, i.e. the assumption that ϕ is true. The order in which judgments occur within contexts are significant, every variable must be declared before it appears in a proposition. That is, the declaration $t : T$ must occur in the context before the variable t occurs in some proposition ϕ, written $\phi\,[t]$.

Next we present the basic structural rules of **TPL**: assumption ($\mathbf{A}_{1-2}$), thinning ($\mathbf{W}_{1-2}$), and substitution (**Sub**). The assumption rules allow only grammatically acceptable assumptions to be permitted, and that every term is attached to a type. The thinning rules allow weakening under the same grammatical conditions. The substitution rule allows a variable, x, to be replaced (substituted) by a term, t.

$$\mathbf{A}_1\ \frac{\Gamma \vdash T\ \mathit{type}}{\Gamma, x : T \vdash x : T} \qquad \mathbf{W}_1\ \frac{\Gamma, \Delta \vdash \Theta \quad \Gamma \vdash T\ \mathit{type}}{\Gamma, x : T, \Delta \vdash \Theta}$$

$$\mathbf{A}_2\ \frac{\Gamma \vdash \phi\ \mathit{prop}}{\Gamma, \phi \vdash \phi} \qquad \mathbf{W}_2\ \frac{\Gamma, \Delta \vdash \Theta \quad \Gamma \vdash \phi\ \mathit{prop}}{\Gamma, \phi, \Delta \vdash \Theta}$$

$$\mathbf{Sub}\ \frac{\Gamma, x : T, \Delta \vdash \Theta\,[x] \quad \Gamma \vdash t : T}{\Gamma, \Delta\,[t/x] \vdash \Theta\,[t/x]}$$

where in the rules $\mathbf{A}_1$ and $\mathbf{W}_1$, x is fresh (it does not already exist in the contexts Γ or Δ), and the notation $\Theta\,[t/x]$ indicates the consistent replacement of every occurrence of variable x with the term t.

At this point it is important to mention the matter of coherence, which ensures that provable statements must also be grammatically acceptable. This is an important concept, but discussing it fully goes beyond the scope of this brief summary of **TPL**. Therefore, we only summarize the a few coherence rules, and leave their proof in [Turner, 2009]. Below we present **TPL**'s intial theorem of coherence. The first two points refer to propositions; if one can conclude a proposition holds (1), or makes the assumption that one does (2), then such propositions must also be grammatically acceptable. The last two refer to types in the same way; if one can conclude a term is of some type T (3), or make a declaration that a variable is of some type T (4), then T must be a type.

Theorem 1 (Coherence)

1. If $\Gamma \vdash \phi$, then $\Gamma \vdash \phi\ \mathit{prop}$,

2. If $\Gamma, \phi, \Gamma' \vdash \Theta$, then $\Gamma \vdash \phi\ \mathit{prop}$,

3. If $\Gamma \vdash t : T$, then $\Gamma \vdash T\ \mathit{type}$,

4. If $\Gamma, t : T, \Gamma' \vdash \Theta$, then $\Gamma \vdash T\ \mathit{type}$

We continue to flesh out **TPL** with types and propositions, such as the familiar logical connectives and quantifiers. We opt to discuss the latter first as most primitive propositions are not dependent on any specific type. Type specific propositions are discussed when we introduce that type.

5.1 Propositions

Although the format of presentation in this section may not be familiar, the rules below provide the standard notions of the respective symbols. We begin our overview of simple propositions with *negation* ($\mathbf{L}_{1-3}$), and *contradiction* ($\mathbf{L}_{4-7}$). Note that the formation rule $\mathbf{L}_1$ requires ϕ to be a proposition before we can conclude $\neg\phi$ is a proposition. This ensures coherence, i.e. the negation of a proposition ϕ is well formed if and only if ϕ is already grammatically acceptable. Rules $\mathbf{L}_2$ and $\mathbf{L}_3$ are the obvious negation introduction and elimination rules respectively. Contradictions, provided by the formation rule $\mathbf{L}_4$, afford us the remaining rules regarding negation. $\mathbf{L}_5$ states that if a proposition and its negation hold, then a contradiction can be concluded. Similarly, $\mathbf{L}_6$ states that if a contradiction can be concluded under the assumption of some proposition ϕ, $\neg\phi$ must logically follow. Finally, $\mathbf{L}_7$ states that anything can be concluded from a contradiction.

$$\mathbf{L}_1\ \frac{\Gamma \vdash \phi\ \mathit{prop}}{\Gamma \vdash \neg\phi\ \mathit{prop}} \qquad \mathbf{L}_4\ \Gamma \vdash \Omega\ \mathit{prop}$$

$$\mathbf{L}_2\ \frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\neg\phi} \qquad \mathbf{L}_5\ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \neg\phi}{\Gamma \vdash \Omega}$$

$$\mathbf{L}_3\ \frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi} \qquad \mathbf{L}_6\ \frac{\Gamma, \phi \vdash \Omega}{\Gamma \vdash \neg\phi}$$

$$\mathbf{L}_7\ \frac{\Gamma \vdash \Omega \quad \Gamma \vdash \phi\ \mathit{prop}}{\Gamma \vdash \phi}$$

Moving on, we introduce the standard rules for *conjunction* ($\mathbf{L}_{8-11}$) and *disjunction* ($\mathbf{L}_{12-16}$). As we saw in the previous set of rules, $\mathbf{L}_8$ and $\mathbf{L}_{12}$ are formation rules. To reiterate, formation rules require that there is grammatical coherence when introducing new symbols. In this case, the propositions ϕ and φ must both be grammatically acceptable before we can conclude that $\phi \wedge \varphi$ or $\phi \vee \varphi$ are well formed propositions. The rest of the rules should be familiar to the reader. $\mathbf{L}_9$ is conjunction introduction, as opposed to $\mathbf{L}_{10}$ and $\mathbf{L}_{11}$ which are the usual conjunction elimination rules. $\mathbf{L}_{13}$ and $\mathbf{L}_{14}$ are the usual disjunction introduction rules, but where the truth of a proposition is unknown, we still require that such propositions are grammatically valid. Finally, $\mathbf{L}_{16}$ is the usual disjunction elimination rule.

$$\mathbf{L}_8 \ \frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \varphi \ \mathit{prop}}{\Gamma \vdash \phi \wedge \varphi \ \mathit{prop}}$$

$$\mathbf{L}_9 \ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \varphi}{\Gamma \vdash \phi \wedge \varphi}$$

$$\mathbf{L}_{10} \ \frac{\Gamma \vdash \phi \wedge \varphi}{\Gamma \vdash \phi}$$

$$\mathbf{L}_{11} \ \frac{\Gamma \vdash \phi \wedge \varphi}{\Gamma \vdash \varphi}$$

$$\mathbf{L}_{12} \ \frac{\Gamma, \varphi \vdash \phi \ \mathit{prop} \quad \Gamma, \phi \vdash \varphi \ \mathit{prop}}{\Gamma \vdash \phi \vee \varphi \ \mathit{prop}}$$

$$\mathbf{L}_{13} \ \frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \varphi}{\Gamma \vdash \phi \vee \varphi}$$

$$\mathbf{L}_{14} \ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \varphi \ \mathit{prop}}{\Gamma \vdash \phi \vee \varphi}$$

$$\mathbf{L}_{16} \ \frac{\Gamma \vdash \phi \vee \varphi \quad \Gamma, \phi \vdash \eta \quad \Gamma, \varphi \vdash \eta}{\Gamma \vdash \eta}$$

Next, we present the *Implication* ($\mathbf{L}_{17-19}$) and *biconditional* ($\mathbf{L}_{20-22}$) symbols. Rules $\mathbf{L}_{17}$ and $\mathbf{L}_{20}$ are the implication and biconditional formation rules respectively. What is interesting here is that rule $\mathbf{L}_{17}$ not only allows us to introduce the implication notation, but that the implied proposition φ need only be grammatically acceptable if ϕ holds. This is an essential change to the rule that allows our work in §7.6 to type check correctly. $\mathbf{L}_{18}$ is the implication introduction rule, and $\mathbf{L}_{19}$ is the obvious implication elimination rule. Our biconditional connectives are defined on top of implications, as exemplified by its introduction rule ($\mathbf{L}_{21}$) and elimination rule ($\mathbf{L}_{22}$).

$$\mathbf{L}_{17} \ \frac{\Gamma, \phi \vdash \varphi \ \mathit{prop}}{\Gamma \vdash \phi \implies \varphi \ \mathit{prop}}$$

$$\mathbf{L}_{18} \ \frac{\Gamma, \phi \vdash \varphi}{\Gamma \vdash \phi \implies \varphi}$$

$$\mathbf{L}_{19} \ \frac{\Gamma \vdash \phi \implies \varphi \quad \Gamma \vdash \phi}{\Gamma \vdash \varphi}$$

$$\mathbf{L}_{20} \ \frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \varphi \ \mathit{prop}}{\Gamma \vdash \phi \iff \varphi \ \mathit{prop}}$$

$$\mathbf{L}_{21} \ \frac{\Gamma \vdash \phi \implies \varphi \quad \Gamma \vdash \varphi \implies \phi}{\Gamma \vdash \phi \iff \varphi}$$

$$\mathbf{L}_{22} \ \frac{\Gamma \vdash \phi \iff \varphi}{\Gamma \vdash \phi \implies \varphi \wedge \varphi \implies \phi}$$

We are nearly to the point of introducing quantifiers, but before we do so we present some type independent rules for *equality* ($\mathbf{L}_{23-25}$). With the introduction of further types we define special cases of the equality relationship, but often we rely on these rules alone. The first rule, $\mathbf{L}_{23}$, is our equality formation rule where both a and b must be of the same type for the equality proposition to be well formed. $\mathbf{L}_{24}$ is the traditional introduction rule that articulates that any term must be equal to itself. Finally, $\mathbf{L}_{25}$ is the elimination rule, which states that equal objects can be substituted for each other in any context.

$$\mathbf{L}_{23}\ \frac{\Gamma \vdash a, b : T}{\Gamma \vdash a =_T b\ prop}$$

$$\mathbf{L}_{24}\ \frac{\Gamma \vdash a : T}{\Gamma \vdash a =_T a}$$

$$\mathbf{L}_{25}\ \frac{\Gamma \vdash a =_T b \quad \Gamma \vdash \Theta\,[a/x]}{\Gamma \vdash \Theta\,[b/x]}$$

Note that we do not define inequality as a primitive symbol as it can be easily defined by a combination of negation and equality. It is therefore a prime candidate to be defined using specifications, but we are not yet to the point of discussing these. For now we informally define equality as follows, and formalize it as a specification when it is appropriate to do so:

- For $a, b : T$, we define $a \neq b$ as $\neg\,(a =_T b)$

Indeed there are many cases where specifications are the most appropriate form of definition. In these cases we will use the same format of presentation, and use them as examples of the specification notation.

Having finished with equality, we have defined a form of propositional logic, which we want to enrich with quantification (a predicate logic). Specifically we will introduce the *existential* ($\mathbf{L}_{26-28}$), *universal* ($\mathbf{L}_{29-31}$), and *uniqueness* ($\mathbf{L}_{32-35}$) quantifiers. These rules articulate the traditional notion of each quantifier, even if their method of presentation is unfamiliar.

We begin with the existential quantifier (*there exists*) and its formation rule $\mathbf{L}_{26}$. This rule allows us to conclude that the existential quantification notation is well formed only if for all terms of a given type the given proposition is also well formed. $\mathbf{L}_{27}$ is the introduction rule, where we may introduce the existential quantifier if it is grammatically acceptable to do so, and we have a term of the right type that can replace our variable in the given proposition. $\mathbf{L}_{28}$ is the elimination rule, where if for any $x : T$ where ϕ holds we may deduce η, then we may conclude η. The usual assumptions are made here, i.e. that x must not be free in Γ, T, or η.

$$\mathbf{L}_{26}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop}{\Gamma \vdash \exists x : T \bullet \phi\ prop}$$

$$\mathbf{L}_{27}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash t : T \quad \Gamma \vdash \phi\,[t/x]}{\Gamma \vdash \exists x : T \bullet \phi}$$

$$\mathbf{L}_{28}\ \frac{\Gamma \vdash \exists x : T \bullet \phi \quad \Gamma, x : T, \phi \vdash \eta}{\Gamma \vdash \eta}$$

Next we examine the rules for universal quantification (*for all*), where its formation rule ($\mathbf{L}_{29}$) is almost identical to the existential quantifier ($\mathbf{L}_{26}$, p.47). That is, the universal quantification notation is well formed if for all $x : T$ the proposition is also well formed. $\mathbf{L}_{30}$ is the introduction rule, where if for all $x : T$ the proposition holds then we can conclude the same written in the appropriate notation, with the side condition that x must not be free in any proposition in Γ. Finally, $\mathbf{L}_{31}$ is the elimination rule where we simply replace the variable in the proposition by a term of the right type.

$$\mathbf{L}_{29}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop}{\Gamma \vdash \forall x : T \bullet \phi\ prop}$$

$$\mathbf{L}_{30}\ \frac{\Gamma, x : T \vdash \phi}{\Gamma \vdash \forall x : T \bullet \phi}$$

$$\mathbf{L}_{31}\ \frac{\Gamma \vdash \forall x : T \bullet \phi \quad \Gamma \vdash t : T}{\Gamma \vdash \phi\,[t/x]}$$

Finally, we examine the uniqueness quantifier (*there exists exactly one*), where its formation rule ($\mathbf{L}_{32}$) is again almost identical to the existential quantifier ($\mathbf{L}_{26}$, p.47). That is, the uniqueness notation is well formed if for all $x : T$ the proposition is also well formed. $\mathbf{L}_{33}$ is the introduction rule, which is more complicated than those we have seen so far. The first premise helps to ensure that what we conclude will be well formed, whereas the remaining premises do the logical work. It requires that we can introduce the notation if we know that given a term $t : T$ such that the proposition $\phi\,[t/x]$ holds, and that for all other $y : T$ if $\phi\,[y/x]$ holds then y must be equal to t. The final rule, $\mathbf{L}_{34}$, is an elimination rule. Assuming the uniqueness quantification holds, and that we have two terms of the right type, we can conclude that if the proposition holds for both terms then they must be equal.

$$\mathbf{L}_{32}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop}{\Gamma \vdash \exists! x : T \bullet \phi\ prop}$$

$$\mathbf{L}_{33}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash t : T \quad \Gamma \vdash \phi\,[t/x] \quad \Gamma \vdash \forall y : T \bullet \phi\,[y/x] \implies y = t}{\Gamma \vdash \exists! x : T \bullet \phi}$$

$$\mathbf{L}_{34}\ \frac{\Gamma \vdash \exists! x : T \bullet \phi \quad \Gamma \vdash u, v : T}{\Gamma \vdash \phi\,[u/x] \wedge \phi\,[v/x] \implies u = v}$$

This concludes our discussion on the type independent propositions. The next section introduces a few types and their associated rules, relations and functions.

5.2 Types

As we mentioned at the start of the chapter, one of the fundamental bare notions of **TPL** are *types*, to which this section is devoted. The treatment of types in **TPL** is non-standard, they are primitive intensional notions whose substance is given by type theories. This will be made clearer as we progress through this section.

We start by introducing a type of types (a universe of types), $\mathcal{U}$. The universe of types is itself a type, and is introduced as such in $\mathbf{T}_1$. All elements of $\mathcal{U}$ are themselves types ($\mathbf{T}_2$), and all types are elements of $\mathcal{U}$ ($\mathbf{T}_3$). This means that the universe of types is closed to the type system, where no other types exist outside of it. This allows us to treat types as first class citizens of the theory, i.e. type variables[27].

$$\mathbf{T}_1\ \Gamma \vdash \mathcal{U}\ type \quad \mathbf{T}_2\ \frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash T\ type} \quad \mathbf{T}_3\ \frac{\Gamma \vdash T\ type}{\Gamma \vdash T : \mathcal{U}}$$

The universe of types is a powerful abstraction mechanism as to reason over types we only need apply $\mathcal{U}$ to the existing propositions. For example, consider again the introduction rule for equality ($\mathbf{L}_{24}$, p.46), which we reiterate below on the left, and the result of using the universe of types below on the right:

$$\frac{\Gamma \vdash a : T}{\Gamma \vdash a =_T a} \quad \frac{\Gamma \vdash a : \mathcal{U}}{\Gamma \vdash a =_{\mathcal{U}} a}$$

As the derived rule on the right shows, the universe of types allows us to reason over types as terms. The same technique applies to our other propositions, such as quantification over types.

We add to the universe of types the rules of *separation*, which facilitate the addition of *subtypes* based on the truth of propositions. $\mathbf{T}_4$ is a membership rule, which dictates what is grammatically acceptable. $\mathbf{T}_5$ is an introduction rule, where if a proposition ϕ holds for a term of type T, then we may introduce a new subtype of T that depends on ϕ. $\mathbf{T}_6$ is the first elimination rule where we can strip away the subtype notation to leave a term of the supertype, in this case $a : T$. Similarly, $\mathbf{T}_7$ is also an elimination rule where we can strip away the subtype notation to leave the proposition that holds, in this case $\phi\,[a/x]$.

$$\mathbf{T}_4\ \frac{\Gamma, T : \mathcal{U}, x : T \vdash \phi\ prop}{\Gamma \vdash \{x : T|\phi\} : \mathcal{U}} \qquad \mathbf{T}_6\ \frac{\Gamma \vdash a : \{x : T|\phi\}}{\Gamma \vdash a : T}$$

$$\mathbf{T}_5\ \frac{\Gamma, T : \mathcal{U}, x : T \vdash \phi\ prop \quad \Gamma \vdash a : T \quad \Gamma \vdash \phi\,[a/x]}{\Gamma \vdash a : \{x : T|\phi\}} \qquad \mathbf{T}_7\ \frac{\Gamma \vdash a : \{x : T|\phi\}}{\Gamma \vdash \phi\,[a/x]}$$

The side condition of these rules is that ϕ is restricted to Σ propositions, which we define as:

Definition 1 Σ *propositions* characterize those semi-decidable propositions constructed from the

[27] The type variables we saw previously were informal, but the universe of types provides us means to formalize them appropriately. That is for any type variable T in a premise of a rule, that rule should also include a premise of the form $\Gamma \vdash T : \mathcal{U}$. However, for the sake of berevity, we shall neglect to do so when it is not ambiguous to do so.

basic relations, functions, and types of the theory with the logical connectives Ω, $\wedge$, $\vee$, and $\exists$. As our sets are finite, we also admit all our set quantifiers as Σ propositions since they are always decidable.

Given this definition, the fully-decidable propositions "are characterized as those Σ propositions whose negations are also Σ" [Turner, 2009]. This restriction ensures computability of our subtypes. Whenever we need to ensure that a proposition is computable, we shall restrict it to Σ. Doing so in the case of subtypes ensures that they are always conservative extensions (see §5.3). In the next chapter (Chapter 6) we introduce specifications, which have the same restriction to Σ propositions, also making our specifications conservative extensions. However, as it is outside the scope of this brief introduction to **TPL** we leave the proofs of these statements in [Turner, 2009].

This concludes our initial set of rules that govern types, we will demonstrate more concrete examples of types in the following subsections. As we proceed through Part II of this thesis we will build our collection of types for object-oriented design.

5.2.1 Natural Numbers

We begin with a simple example of a basic type that illustrates the ease with which types can be added to the **TPL** framework, the *natural numbers*[28]. The addition of the natural number type ($\mathbf{N}_1$), the constant zero ($\mathbf{N}_2$), and the associated rules that we discuss, is a formalization of Peano Arithmetic in **TPL**.

$$\mathbf{N}_1\ \ \Gamma \vdash \mathbb{N} : \mathcal{U} \qquad \mathbf{N}_2\ \ \Gamma \vdash \mathtt{0} : \mathbb{N}$$

We do not create a constant for every natural number, as each is derivable by getting the next value in the sequence. Therefore, any natural number is obtainable through any finite series of applications of the *successor* function on the constant $\mathtt{0}$. $\mathbf{N}_3$ is the formation rule for the successor operation, which states that the successor of a natural number is itself also a natural number. $\mathbf{N}_4$ articulates that $\mathtt{0}$ cannot be the successor to another natural number, meaning that the successor operation is injective. Equality of natural numbers ($\mathbf{N}_5$) is defined as the same number of applications of the successor operation. Finally, $\mathbf{N}_6$ is an induction principle, with the usual interpretation. This allows us to reason about natural numbers.

[28] For which we include zero, i.e. the type of non-negative integers.

$$\mathbf{N}_3\ \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash n^+ : \mathbb{N}} \qquad \mathbf{N}_5\ \frac{\Gamma \vdash a^+ =_{\mathbb{N}} b^+}{\Gamma \vdash a =_{\mathbb{N}} b}$$

$$\mathbf{N}_4\ \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash n^+ \neq \mathtt{0}} \qquad \mathbf{N}_6\ \frac{\begin{array}{c}\Gamma, n : \mathbb{N} \vdash \phi\,[n]\ \ prop \\ \Gamma \vdash \phi\,[\mathtt{0}] \quad \Gamma, n : \mathbb{N}, \phi\,[n] \vdash \phi\,[n^+]\end{array}}{\Gamma, n : \mathbb{N} \vdash \phi\,[n]}$$

To simplify our use of numbers, we employ a simple shorthand that allows us to write a number in the intuitive way where we interpret a natural number as an appropriate series of applications of the successor operation on 0. This is demonstrated as follows:

$$\begin{aligned} \mathtt{1} &\triangleq \mathtt{0}^+ \\ \mathtt{2} &\triangleq \mathtt{1}^+ \\ &= \mathtt{0}^{++} \\ &\ldots \end{aligned}$$

Now that we have laid the groundwork for reasoning over natural numbers, we add the usual operations to do so. We begin with *addition*, where rule $\mathbf{N}_9$ is our formation rule. $\mathbf{N}_8$ shows that the addition operation is commutative. $\mathbf{N}_9$ states that adding 0 to any natural number is equal to itself. Finally, rule $\mathbf{N}_{10}$ tells us that the successor of an addition operation can be applied to its result, or any one of its arguments. The combination of the last two rules allow us to compile addition away into a finite series of successor operations.

$$\mathbf{N}_7\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a + b : \mathbb{N}} \qquad \mathbf{N}_9\ \frac{\Gamma \vdash a : \mathbb{N}}{\Gamma \vdash a + \mathtt{0} = a}$$

$$\mathbf{N}_8\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a + b = b + a} \qquad \mathbf{N}_{10}\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a + b^+ = (a + b)^+}$$

Defining addition provides means to define *subtraction*, and the natural *ordering* relations. For now, we informally define these operations and relations in the same way we did for inequality. In the next chapter (Chapter 6) we will show how to formalize them in specifications.

- For $a, b, c : \mathbb{N}$, we define $a - b = c$ as $a = c + b$
- For $a, b : \mathbb{N}$, we define $a \leq b$ as $\exists x : \mathbb{N} \bullet a + x = b$
- For $a, b : \mathbb{N}$, we define $a < b$ as $a \leq b \wedge a \neq b$

Mirroring the addition relation, we also introduce *multiplication*. $\mathbf{N}_{11}$ is the formation rule for multiplication, and like addition it is also a commutative operation ($\mathbf{N}_{12}$). $\mathbf{N}_{13}$ states that multiplying any natural number by $\mathtt{0}$ results in $\mathtt{0}$. Finally, $\mathbf{N}_{14}$ shows how multiplication can be compiled into a finite series of addition operations.

$$\mathbf{N}_{11}\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a * b : \mathbb{N}} \qquad \mathbf{N}_{13}\ \frac{\Gamma \vdash a : \mathbb{N}}{\Gamma \vdash \mathtt{0} * a = \mathtt{0}}$$

$$\mathbf{N}_{12}\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a * b = b * a} \qquad \mathbf{N}_{14}\ \frac{\Gamma \vdash a, b : \mathbb{N}}{\Gamma \vdash a^{+} * b = (a * b) + b}$$

As with addition, multiplication provides means to define other operations. In this case the *division*, *exponentiation*, and *factorial* functions, which again we define informally for now:

- For $a, b, c : \mathbb{N}$, we define $a/b = c$ as $b \neq \mathtt{0} \wedge a = b * c$
- For $a, b, c : \mathbb{N}$, we define $a^b = c$ as either $b = \mathtt{0}$ and $c = \mathtt{1}$, or $\exists x, y : \mathbb{N} \bullet x^{+} = b \wedge a^x = y \wedge c = a * y$
- For $a, b : \mathbb{N}$, we define $a! = b$ as either $a = \mathtt{0}$ and $b = \mathtt{1}$, or $\exists x, y : \mathbb{N} \bullet x^{+} = a \wedge x! = y \wedge b = a * y$

This concludes our discussion of natural numbers, a formalization of Peano Arithmetic within **TPL**. This is a simple demonstration of the ease and elegance afforded by **TPL**. But, before we move on to discussing some of the more interesting types, such as pairs and finite sets, let us see a simple example of using subtypes. For example, consider adding a type of all even natural numbers, and a type for all odd natural numbers. Obviously any odd natural number divided by $\mathtt{2}$ will not result in another natural number, so we can use this knowledge to construct the propositions that define our new subtypes. Let us call the types $\mathbb{EVEN}$ and $\mathbb{ODD}$ respectively, which we can very easily define as follows:

$$\begin{aligned} \mathbb{EVEN} &\triangleq \{x : \mathbb{N} | \exists y : \mathbb{N} \bullet y = x/\mathtt{2}\} \\ \mathbb{ODD} &\triangleq \{x : \mathbb{N} | \neg \exists y : \mathbb{N} \bullet y = x/\mathtt{2}\} \end{aligned}$$

The advantage of subtypes is that we may use terms of these types anywhere a natural number is expected. For example, we can take successors of even and odd natural numbers, add them together, or multiply them in the same way we can any other natural number. This helps to demonstrate the power, flexibility, and elegance of the subtype mechanism in **TPL**.

5.2.2 Pairs and Tuples

There are two approaches to defining *tuples* and *pairs*. The first is to define a general notion of n-tuple, of which pairs are obviously a special case. Alternatively, define a notion of pair from which n-tuples can be composed. Either method is acceptable, but we opt for the latter as it ensures that tuples must be constructed by a finite series of the pair type constructor. Specifically, we adopt the standard notion of Cartesian products for our definition of the pair type constructor. A type constructor takes existing types and produces new types, where basic type are considered nullary type constructors. Type constructors and basic types are therefore analogous to functions and constants.[29].

The pair type constructor is added to the theory with the formulation rule $\mathbf{P}_1$, written in the infix notation. $\mathbf{P}_2$ is our introduction rule, allowing us to construct a pair using the round brackets notation. Rules $\mathbf{P}_{3-6}$ define the *selection* operation, which allows us to reason over the first and second components of the pair. Rules $\mathbf{P}_3$ and $\mathbf{P}_4$ introduce selection operation for each respective component of a pair, and as such are elimination rules for the pair type. Finally, the equality rules $\mathbf{P}_5$ and $\mathbf{P}_6$ both ensure that the selection operations retrieve the expected terms.

$$\mathbf{P}_1\ \frac{\Gamma \vdash A, B : \mathcal{U}}{\Gamma \vdash A \times B : \mathcal{U}} \qquad \mathbf{P}_5\ \frac{\Gamma \vdash (a, b) : A \times B}{\Gamma \vdash \pi_1\,(a, b) = a \wedge \pi_2\,(a, b) = b}$$

$$\mathbf{P}_2\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \mathbf{P}_6\ \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t = (\pi_1\,(t)\,, \pi_2\,(t))}$$

$$\mathbf{P}_3\ \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1\,(t) : A}$$

$$\mathbf{P}_4\ \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2\,(t) : B}$$

As previously mentioned, we can construct n-tuples through a finite sequence of the pair type constructor using the following shorthand:

$$\begin{aligned} T_1 \times T_2 \times T_3 &\triangleq T_1 \times (T_2 \times T_3) \\ T_1 \times T_2 \times T_3 \times T_4 &\triangleq T_1 \times (T_2 \times T_3 \times T_4) \\ &= T_1 \times (T_2 \times (T_3 \times T_4)) \\ &\dots \end{aligned}$$

Now that we have this shorthand notation to construct n-tuples, we need to address how to select the terms that they contain. This requires adding another shorthand notation. To accomplish this we decorate the selection operator with a superscript indicating the size of the

[29] We will use τ as a meta-variable over type constructors as the need arises, where type constructors of the form $A \tau B$ are a shorthand for $\tau\,(A, B)$.

tuple. For example if t is a 3-tuple, then we use π^3 as follows:

$$\begin{aligned}\pi^3_1(t) &\triangleq \pi_1(t)\\ \pi^3_2(t) &\triangleq \pi_2(\pi_1(t))\\ \pi^3_3(t) &\triangleq \pi_2(\pi_2(t))\end{aligned}$$

Let us ensure this is clear by providing a simple example. Consider representing a rectangle to be drawn on a computer screen using the pair type constructor. The rectangle's top left corner is to rest on the origin and has a width of $\mathtt{10}$ and a height of $\mathtt{20}$. We can represent this using a 4-tuple $r : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, which is shorthand for $r : \mathbb{N} \times (\mathbb{N} \times (\mathbb{N} \times \mathbb{N}))$. The values of the 4-tuple in this case are $r = (\mathtt{0}, \mathtt{0}, \mathtt{10}, \mathtt{20})$, which is shorthand for $r = (\mathtt{0}, (\mathtt{0}, (\mathtt{10}, \mathtt{20})))$. Therefore, to retrieve the x coordinate of the rectangle we simply use the right selection operator, $\pi^4_1(r) = \pi_1(r) = \mathtt{0}$. Similarly to retrieve the width of the rectangle we would use $\pi^4_3(r) = \pi_1(\pi_2(\pi_2(r))) = \mathtt{10}$.

5.2.3 Finite Sets

The final type we introduce in Part I of this thesis is a type constructor for *sets*. As our requirement is that the operations and relations on types be computable, we focus on unbounded, but *finite*, sets. Our treatment of finite sets is influenced bounded set theory [Turner, 2009].

We introduce finite sets in much the same way as we did natural numbers, with a type constructor ($\mathbf{S}_1$) and a constant for the empty set ($\mathbf{S}_2$).

$$\mathbf{S}_1 \;\frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash \mathit{set}(T) : \mathcal{U}} \qquad \mathbf{S}_2 \;\frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash \varnothing_T : \mathit{set}(T)}$$

A finite set (or simply set) is then constructed by a finite series of *set addition* operations, i.e. we can always make a bigger set by adding another element to an existing set. Indeed, in this way our treatment of sets is very similar to our treatment of natural numbers. This similarity is reflected in the rules for set addition we present here, and quantification over sets presented shortly.

The formation rule for set addition is given in $\mathbf{S}_3$, where set addition carries the standard notions. That is, $\mathbf{S}_4$ states that adding a new element to a set results in a set that is not equal to the empty set. $\mathbf{S}_5$ states that set addition is commutative. And $\mathbf{S}_6$ states that an element is always unique in a set, i.e. adding an element multiple times is a redundancy. Finally, $\mathbf{S}_7$ is an induction rule allowing us to reason over sets.

$$\mathbf{S}_3\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{set}\,(T)}{\Gamma \vdash a \oplus b : \mathit{set}\,(T)} \qquad \mathbf{S}_6\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{set}\,(T)}{\Gamma \vdash a \oplus a \oplus b = a \oplus b}$$

$$\mathbf{S}_4\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{set}\,(T)}{\Gamma \vdash a \oplus b \neq \varnothing_T} \qquad \mathbf{S}_7\ \frac{\begin{array}{c}\Gamma, s : \mathit{set}\,(T) \vdash \phi\,[s]\ \mathit{prop} \\ \Gamma \vdash \phi\,[\varnothing_T] \quad \Gamma, s : \mathit{set}\,(T)\,, t : T, \phi\,[s] \vdash \phi\,[t \oplus s]\end{array}}{\Gamma, s : \mathit{set}\,(T) \vdash \phi\,[s]}$$

$$\mathbf{S}_5\ \frac{\Gamma \vdash a, b : T \quad \Gamma \vdash c : \mathit{set}\,(T)}{\Gamma \vdash a \oplus b \oplus c = b \oplus a \oplus c}$$

Set addition allows use to define *set membership*, as presented in its formation ($\mathbf{S}_8$) and introduction ($\mathbf{S}_9$) rules. Rules $\mathbf{S}_{10}$ and $\mathbf{S}_{11}$ characterize the set membership relation, and are also therefore elimination rules. $\mathbf{S}_{10}$ states that no element is a member of the empty set, while $\mathbf{S}_{11}$ tells us how we may search through a non-empty set to find the element in question. Finally, $\mathbf{S}_{12}$ states that if a term is a member of a set, then it must also be a member any extension thereof.

$$\mathbf{S}_8\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{set}\,(T)}{\Gamma \vdash a \in b\ \mathit{prop}} \qquad \mathbf{S}_{11}\ \frac{\Gamma \vdash a \in b \oplus c}{\Gamma \vdash a = b \vee a \in c}$$

$$\mathbf{S}_9\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{set}\,(T)}{\Gamma \vdash a \in a \oplus b} \qquad \mathbf{S}_{12}\ \frac{\Gamma \vdash a, b : T \quad \Gamma \vdash c : \mathit{set}\,(T) \quad \Gamma \vdash a \in c}{\Gamma \vdash a \in b \oplus c}$$

$$\mathbf{S}_{10}\ \frac{\Gamma \vdash a \in \varnothing_T}{\Gamma \vdash \Omega}$$

We also introduce a little shorthand to simplify the membership relation in our later use; we take $r\,(t)$ to be shorthand for $(t) \in r$ when it is not ambiguous to do so.

Set addition allows us to now informally define *non-membership*, which mirrors our definition of inequality, and *set subtraction*, which mirrors subtraction for natural numbers. These will be formalized as specifications in the next chapter (Chapter 6).

- For $a : T$ and $b : \mathit{set}\,(T)$, we define $a \notin b$ as $\neg\,(a \in b)$
- For $a : T$ and $b, c : \mathit{set}\,(T)$, we define $a \ominus b = c$ as $a \notin c \wedge b = a \oplus c$

We now have the ability to add and remove elements from a set, and decide if a term is a member of a set or not. On top of this we must define mechanisms for quantifying over sets. We define the same quantifiers as for types, the *existential* ($\mathbf{S}_{13-15}$), *universal* ($\mathbf{S}_{16-18}$) and *uniqueness* ($\mathbf{S}_{19-22}$) quantifiers, and employ the same structure and symbols. The result of adding these quantifiers allows us to reason over sets, and define more relationships between them. For example, union and intersection, equality, and subsets. However, as the rules for the set quantifiers are almost identical to those for types, we will keep our description of each rule to a minimum for the sake of brevity.

As previously, we begin with the existential quantifier for sets. Its formation rule is given in $\mathbf{S}_{13}$, the introduction rule in $\mathbf{S}_{14}$, and the elimination rule in $\mathbf{S}_{15}$.

$$\mathbf{S}_{13}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash s : set\,(T)}{\Gamma \vdash \exists x \in s \bullet \phi\ prop}$$

$$\mathbf{S}_{14}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash s : set\,(T) \quad \Gamma \vdash \phi\,[t/x] \quad \Gamma \vdash t \in s}{\Gamma \vdash \exists x \in s \bullet \phi}$$

$$\mathbf{S}_{15}\ \frac{\Gamma \vdash \exists x \in s \bullet \phi \quad \Gamma, x \in s, \phi \vdash \eta}{\Gamma \vdash \eta}$$

Next is the universal quantifier for sets. Its formation rule is given in $\mathbf{S}_{16}$, the introduction rule in $\mathbf{S}_{17}$, and the elimination rule in $\mathbf{S}_{18}$.

$$\mathbf{S}_{16}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash s : set\,(T)}{\Gamma \vdash \forall x \in s \bullet \phi\ prop}$$

$$\mathbf{S}_{17}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash s : set\,(T) \quad \Gamma, x \in s \vdash \phi\,[x]}{\Gamma \vdash \forall x \in s \bullet \phi}$$

$$\mathbf{S}_{18}\ \frac{\Gamma \vdash \forall x \in s \bullet \phi \quad \Gamma \vdash t \in s}{\Gamma \vdash \phi\,[t/x]}$$

And finally the uniqueness quantifier for sets. Its formation rule is given in $\mathbf{S}_{19}$, the introduction rule in $\mathbf{S}_{20}$, and the elimination rule in $\mathbf{S}_{21}$.

$$\mathbf{S}_{19}\ \frac{\Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash s : set\,(T)}{\Gamma \vdash \exists! x \in s \bullet \phi\ prop}$$

$$\mathbf{S}_{20}\ \frac{\begin{array}{c}\Gamma \vdash s : set\,(T) \quad \Gamma \vdash t : T \quad \Gamma \vdash t \in s \\ \Gamma, x : T \vdash \phi\,[x]\ \ prop \quad \Gamma \vdash \phi\,[t/x] \quad \Gamma \vdash \forall y \in s \bullet \phi\,[y/x] \implies y = t\end{array}}{\Gamma \vdash \exists! x \in s \bullet \phi}$$

$$\mathbf{S}_{21}\ \frac{\Gamma \vdash \exists! x \in s \bullet \phi \quad \Gamma \vdash s : set\,(T) \quad \Gamma \vdash u, v : T}{\Gamma \vdash \phi\,[u/x] \wedge \phi\,[v/x] \implies u = v}$$

These quantifiers allow us to define further operations and relationships between sets. Specifically, we informally define the *subset*, (extensional) *equality*, and *proper subset* relations, with the set *union* and *intersection* functions; all of these new functions and relations over sets have the standard meaning. As with other definitions of this form, they will be formalized as specifications in the next chapter (Chapter 6).

- For $a, b : set\,(T)$, we define $a \subseteq b$ as $\forall x \in a \bullet x \in b$

- For $a, b : set\,(T)$, we define $a =_{set(T)} b$ as $a \subseteq b \wedge b \subseteq a$

- For $a, b : set\,(T)$, we define $a \subset b$ as $a \subseteq b \wedge a \neq b$
- For $a, b, c : set\,(T)$, we define $a \cup b = c$ as $a \subseteq c \wedge b \subseteq c \wedge \forall x \in c \bullet x \in a \vee x \in b$
- For $a, b, c : set\,(T)$, we define $a \cap b = c$ as:
  $(\forall x \in c \bullet x \in a \wedge x \in b) \wedge (\forall x \in a \cup b \bullet x \in a \wedge x \in b \implies x \in c)$

With these new functions and relations, we have come to the end of our discussion on sets. In the next section we discuss the different sorts of extensions that we can make to our type theory.

5.3 Extensions

One of the reasons **TPL** is so expressive is that it is not designed to accommodate any one given type theory, but rather a framework in which any type theory can be easily developed. Type theories can then be glued together to form new more expressive theories. In this section we discuss several types of extensions to theories, give some formal definitions for them, and discuss their connotations. Specifically, our attention will be focused on grammatical, conservative, proper and elimination extensions.

We begin with grammatical extensions, which are the weakest and most common form of extension. It simply requires that what can be said in one theory can be said in its extension, and formally defined as follows:

Definition 2 Let T_1 and T_2 be two theories. T_2 is a *grammatical extension* of T_1 if and only if each of the following holds:

1. If $T_1 \vdash \phi\ prop$ then $T_2 \vdash \phi\ prop$
2. If $T_1 \vdash T\ type$ then $T_2 \vdash T\ type$
3. If $T_1 \vdash t : T$ then $T_2 \vdash t : T$

Grammatical extensions are weak notions as they do not make any commitments about the truth of propositions in either the original theory or its extension. But it is the foundation on which we can define conservative and proper extensions. Conservative extensions ensure that if a proposition is well formed in the original theory, and holds in its extension, then it must also hold in the original theory. In other words, conservative extensions cannot say anything new about the original theory that can be said therein. The benefit of this is that a conservative extension preserves consistency, i.e. the lack of a contradiction, between the original theory and its extension. That is, if a theory is, or is assumed to be, consistent then so are its conservative extensions.

Definition 3 Let T_1 and T_2 be two theories such that T_2 is a grammatical extension of T_1. T_2 is a *conservative extension* of T_1 if and only if $T_1 \vdash \phi$ *prop* then $T_2 \vdash \phi$ implies $T_1 \vdash \phi$

There is a special case of conservative extensions, which we call elimination extensions. Elimination extensions are grammatical extensions where propositions that hold in the new theory can be compiled away into usually much more complex propositions that hold in the original theory. As such, elimination extensions usually afford propositions that are more syntactically succinct, but are not more expressive than the original theory. Therefore, elimination extensions are defined as follows:

Definition 4 Let T_1 and T_2 be two theories such that T_2 is a grammatical extension of T_1. T_2 is a *elimination extension* of T_1 if and only if $T_2 \vdash \phi$ *prop* then for some φ we have $T_1 \vdash \varphi$ *prop* and $T_2 \vdash \phi \iff \varphi$

Finally, proper extensions are simply defined as not conservative extensions, i.e. those theories that say something new about the original theory that was previously expressible therein. Proper extensions do not necessarily preserve the properties of the original theory, such as consistency. Formally we define proper extensions as follows:

Definition 5 Let T_1 and T_2 be two theories such that T_2 is a grammatical extension of T_1. T_2 is a *proper extension* of T_1 if and only if it is not a conservative extension.

With this we conclude our brief summary of **TPL**. In the next chapter (Chapter 6) we introduce the schema notation of **TPL**, on which we base our notion of specification.

Chapter 6

Specification

Some of the relations in the previous chapter appeared in the form "for $t : T$ we define $R(t)$ as $\phi[t]$", and stated that we would demonstrate how these can be presented formally. In this chapter we provide and discuss the rigorous means for such definitions. We introduce **TPL**'s schema notation, whose box-style layout was inspired by the Z specification language [Spivey, 1992], and form the foundation of *specifications*.

We begin with discussing simple specification, and then refine it in the context of dependent types and recursion in the next two sections. Specifications in **TPL** elegantly introduce new relations into the theory. They consist of a series of declarations, where variables are associated with types, and a proposition, which articulates the properties of and relationships between the variables. We write these using either the following horizontal (inline) notation:

$$R \triangleq [t_1 : T_1, \ldots, t_n : T_n | \phi[t_1, \ldots, t_n]]$$

or the vertical notation:

$$\begin{array}{l} R \\ \begin{array}{|c|} \hline t_1 : T_1, \ldots, t_n : T_n \\ \hline \phi[t_1, \ldots, t_n] \\ \hline \end{array} \end{array}$$

where which we use in any particular case is a matter of presentation. We opt to use the horizontal notation in simple cases, and the vertical otherwise.

But what is the logical foundation of specifications, i.e. how are they unpacked into the theory? Our approach differs from that in [Turner, 2009], which defines general rules for n-ary specifications. Instead, we observe that any n-ary specification may be rewritten as a unary specification using tuples (and later using dependent types in §6.1). A positive by-product of this is that a relation must have a finite number of arguments. Therefore, a specification of the form

$$R \triangleq [t_1 : T_1, \ldots, t_n : T_n | \phi [t_1, \ldots, t_n]]$$

which has no dependencies between the declarations, can be rewritten to be of the form

$$R \triangleq [t : T_1 \times \ldots \times T_n | \phi [\pi_1^n (t), \ldots, \pi_n^n (t)]]$$

Dependencies in the declarations and contexts are common, but require a slightly different treatment. To deal with such cases we employ dependent types, which we will introduce and discuss in the next section (§6.1). We also leave discussion of recursive schema to §6.3. For now we take all specifications to be meta-notation for the introduction of simple unary relations[30], which may be unpacked by the following rules. $\mathbf{R}_1$ is the formation rule for the new relation under the assumption of the proposition being well formed under the declaration. $\mathbf{R}_2$ allows us to introduce the relation on $x : T$ if its proposition is well formed for all terms of the type and holds for x. Finally, $\mathbf{R}_3$ allows us to eliminate the relation by replacing it with its proposition.

$$\mathbf{R}_1 \ \frac{\Gamma, x : T \vdash \phi \ \mathit{prop}}{\Gamma \vdash R(x) \ \mathit{prop}}$$

$$\mathbf{R}_2 \ \frac{\Gamma, x : T \vdash \phi \ \mathit{prop} \quad \Gamma \vdash x : T \quad \Gamma \vdash \phi [x]}{\Gamma \vdash R(x)}$$

$$\mathbf{R}_3 \ \frac{\Gamma \vdash x : T \quad \Gamma \vdash R(x)}{\Gamma \vdash \phi [x]}$$

where the new relation R must not already exist in the theory, and ϕ is restricted to a Σ proposition[31] (see Definition 1). That is, our specifications must always be computable. Additionally, our specifications preserve the theory on which they are defined, i.e. they are always conservative extensions. However, as with subtypes, we will not prove that this is the case. Instead we direct the reader to [Turner, 2009] for the proof.

[30] LePUS3 also admits specifications with no declarations, which operate only over constants and therefore introduce nullary relations that must always hold. To differentiate these special kind of specifications we adopt the LePUS3 terminology, borrowed from sentential formulas. We say a specification Ψ is *closed* if it does not have any declarations, i.e. it introduces a new nullary relation into the language. Such specifications are therefore equivalent to their proposition. That is they are a method of naming a proposition that is unpacked into **TPL** with the following rules:

$$\mathbf{R}_1' \ \frac{\Gamma \vdash \phi \ \mathit{prop}}{\Gamma \vdash R \ \mathit{prop}} \qquad \mathbf{R}_2' \ \frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \phi}{\Gamma \vdash R} \qquad \mathbf{R}_3' \ \frac{\Gamma \vdash R}{\Gamma \vdash \phi}$$

Conversely, a specification Ψ is *open* if it has one or more declarations, i.e. it introduces a new n-place relation into the language. We will not see closed specifications until Part III.

[31] It is possible to lift this constraint in particular cases, but we shall not need to do so within this work. See [Turner, 2009] for further discussion on this.

To demonstrate how specifications are applied, we present a selection of examples taken from the informal definitions from the previous chapter. However, our understanding of specification is limited as we have not yet addressed dependent types or recursive specifications. As we refine the schema notation to accommodate for these issues, we will be able to express more informal definitions as formal specifications. Indeed, by the end of this chapter we will have presented a specification for every informal definition found in the previous chapter.

We begin with the simple cases of inequality and ordering over natural numbers:

Example 1 Previously we wrote "for $a, b : T$, we define $a \neq b$ as $\neg\,(a =_T b)$", which we can now rewrite as:

$$\neq_T \triangleq [a, b : T | \neg\,(a =_T b)]$$

This example shows how easily the informal definitions are translated into the specification notation. That is, the informal definition states the required declarations ("$a, b : T$"), the relation name ("we define $a \neq b$"), and the proposition ("as $\neg\,(a =_T b)$"). Note that we have not put in the declaration for the type variable T. Doing so keeps the specifications simple and intuitive to understand, but we may only do this where such type declarations are recoverable from the context or not ambiguous to do so. Next we consider formalising the ordering relationships for natural numbers.

Example 2 Previously we wrote "for $a, b : \mathbb{N}$, we define $a \leq b$ as $\exists x : \mathbb{N} \bullet a + x = b$", which we can now rewrite as:

$$\leq \triangleq [a, b : \mathbb{N} | \exists x : \mathbb{N} \bullet a + x = b]$$

where $\leq$ is a binary relation written in the infix notation.

Despite this example having a slightly more complicated proposition, it need be treated no differently from the example above. That is, the informal definition states the required declarations ("$a, b : \mathbb{N}$"), the relation name ("we define $a \leq b$"), and the proposition ("as $\exists x : \mathbb{N} \bullet a + x = b$"). The next example follows exactly the same method of translation:

Example 3 Previously we wrote "for $a, b : \mathbb{N}$, we define $a < b$ as $a \leq b \wedge a \neq b$", which we can now rewrite as:

$$< \triangleq [a, b : \mathbb{N} | a \leq b \wedge a \neq b]$$

where $<$ is a binary relation written in the infix notation.

These are clear examples of defining relations using specification, but what about functions? A relation may describe one or more functions provided that they are functional in some context, i.e. everything in the relations domain relates to exactly one thing in its range. To accomplish this we need a way to reason over the elements in the relations domain. To this end, we introduce the *Dom* and *Ran* relations[32] from [Turner, 2009]. There is a condition to the *Dom* and *Ran* relations, which is that we must be able to treat the relation in question as a well formed binary relation, and is already in the context[33]. That is, the following sequent holds:

$$\Gamma, x : A, y : B \vdash R\,(x, y) \;\; prop$$

and given this, we may simply define *Dom* and *Ran* in the following specifications:

Dom_R

$x : A$

$\exists y : B \bullet R\,(x, y)$

Ran_R

$y : B$

$\exists x : A \bullet R\,(x, y)$

These specifications provide the formal means of reasoning over the domain and range of a relation. For example, consider applying these relations to the $<$ relation. The resulting relations, $Dom_<$ and $Ran_<$, both reason over the natural numbers. However, where any natural number is a member of $Dom_<$, as 0 is not greater than any other number it cannot be a member of $Ran_<$.

We are now able to employ the *Dom* relation in the definition of total functional relations. Formally we define these as follows:

Definition 6 R is a *total functional relation* from T_{in} to T_{out}, written $R : T_{in} \longmapsto T_{out}$, if and only if both of the following hold:

1. $\Gamma, a : T_{in}, b : T_{out} \vdash R\,(a, b) \;\; prop$

2. $\forall x : T_{in} \bullet Dom_R\,(x) \implies \exists! y : T_{out} \bullet R\,(x, y)$

[32] Actually, we will not need to use the *Ran* relation. We include it so that although our discussion is brief, it is relatively complete.

[33] We will show how to formally reason over relations in §6.2, and redefine these specifications accordingly.

That is, R is well formed, and for every possible x of type T_{in} if x is in the domain of R then there exists a unique y of type T_{out} between which R holds. This definition ensures that all functions introduced this way are total, as opposed to partial. This is preferable as although the theory remains silent (unprovable) when the relation does not hold, it is still a well formed proposition.

When $R : T_{in} \longmapsto T_{out}$ holds we may introduce a new function symbol f, where we write $f(x)$ for the unique y such that $R(x, y)$ holds. Generally, we will not prove functionality for the obvious cases, i.e. those which follow the standard definitions. In most other cases such proofs are provided in [Turner, 2009], which we shall redirect the reader for the sake of brevity. Having said that, we will prove those few cases unique to our work.

To demonstrate how we may define new function symbols in this way, we apply our new understanding to our informal definitions. For example, subtraction in the natural numbers:

Example 4 Previously we wrote "for $a, b, c : \mathbb{N}$, we define $a - b = c$ as $a = c + b$", which we can now rewrite as:

$$- \triangleq [a, b, c : \mathbb{N} | a = c + b]$$

As the above relation is functional we write $a-b$ for the unique c. To clarify, $-$ is a binary function written in the infix notation such that $- : \mathbb{N} \times \mathbb{N} \longmapsto \mathbb{N}$ (see p.62) holds.

We first define the relationship between the arguments of subtraction and its result as a specification, as we saw with the previous examples. Then, assuming that the introduced relation is functional, we define a new function symbol. The next example follows exactly the same method of translation, and introduction of a new function symbol:

Example 5 Previously we wrote "for $a, b, c : \mathbb{N}$, we define $a/b = c$ as $b \neq \mathsf{0} \wedge a = b * c$", which we can now rewrite as:

$$/ \triangleq [a, b, c : \mathbb{N} | b \neq \mathsf{0} \wedge a = b * c]$$

As the above relation is functional we write a/b for the unique c. To clarify, $/$ is a binary function written in the infix notation such that $/ : \mathbb{N} \times \mathbb{N} \longmapsto \mathbb{N}$ (see p.62) holds.

6.1 Dependent Schema

We will now consider some of the more complex cases that are inexpressible with our current concept of specification. In this section we first deal with dependencies between types within contexts and then move on to show how this helps solve dependencies within declarations of specifications. A dependency is introduced when a type constructor contains one or more variables. That is, the legitimacy of the type resulting from application of the type constructor is dependent on the values of the variables with which it has been provided. We have already seen this with the set and pair type constructors. For example, consider the following sequent for the introduction of the set type constructor:

$$\Gamma, T : \mathcal{U} \vdash set\,(T) : \mathcal{U}$$

where the legitimacy of the type $set\,(T)$ depends on the value of variable T. If T does not contain an acceptable value, then $set\,(T)$ may not be a legitimate type.

The solution to this comes from a generalization of the pair type constructor to allow us to construct dependent products. We introduce the dependent product notation with the following six rules, which mirror the rules of the pair type constructor. $\mathbf{T}_{14}$ is the formation rule for dependent products, which dictates that the dependent product $\Sigma x : T \cdot S$ is well formed only if the dependent type $S\,[x]$ is a type for all $x : T$. $\mathbf{T}_{15}$ is the introduction rule, where we see that terms of a dependent product are pairs; the first component of which is the argument for the dependent type, $a : T$, and the second is a term of the dependent type itself, $b : S\,[a]$. As with pairs, we introduce the selection operations for dependent products, $\mathbf{T}_{16-17}$, and include the same shorthand notation as for multiple dependent products. Notice that in rule $\mathbf{T}_{17}$ the type of the second component is reconstructed from the first component. Finally, and as with normal pairs, the equality rules $\mathbf{T}_{18}$ and $\mathbf{T}_{19}$ both ensure that the selection operations retrieve the expected terms.

$$\mathbf{T}_{14}\ \frac{\Gamma, x : T \vdash S\,[x] : \mathcal{U}}{\Gamma \vdash \Sigma x : T \cdot S : \mathcal{U}}$$

$$\mathbf{T}_{15}\ \frac{\Gamma, x : T \vdash S\,[x] : \mathcal{U} \quad \Gamma \vdash a : T \quad \Gamma \vdash b : S\,[a/x]}{\Gamma \vdash (a, b) : \Sigma x : T \cdot S}$$

$$\mathbf{T}_{16}\ \frac{\Gamma \vdash p : \Sigma x : T \cdot S}{\Gamma \vdash \pi_1\,(p) : T}$$

$$\mathbf{T}_{17}\ \frac{\Gamma \vdash p : \Sigma x : T \cdot S}{\Gamma \vdash \pi_2\,(p) : S\,[\pi_1\,(p)]}$$

$$\mathbf{T}_{18}\ \frac{\Gamma \vdash (a, b) : \Sigma x : T \cdot S}{\Gamma \vdash \pi_1\,(a, b) = a \wedge \pi_2\,(a, b) = b}$$

$$\mathbf{T}_{19}\ \frac{\Gamma \vdash p : \Sigma x : T \cdot S}{\Gamma \vdash p = (\pi_1\,(p)\,, \pi_2\,(p))}$$

To revisit our simple example, the sequent that introduces the set type can now be rewritten as:

$$\Gamma \vdash \Sigma T : \mathcal{U} \cdot set\,(T) : \mathcal{U}$$

where any $x : \Sigma T : \mathcal{U} \cdot set\,(T) : \mathcal{U}$ is a pair such that $\pi_1\,(x)$ is the type of elements in the finite set, and $\pi_2\,(x)$ is the set itself.

This solves dependencies in contexts, so now we show how we deal with dependencies in a specification's declarations. We call such specifications *dependent specifications*, which are most common when the specification declares type variables. We have already shown that a specification that introduces an n-ary relation can be rewritten as a unary specification using the pair type constructor. That is, a specification of the form:

$$R \triangleq [x_1 : T_1, \ldots, x_n : T_n | \phi\,[x_1, \ldots, x_n]]$$

is taken to be shorthand for a specification of the form:

$$R \triangleq [x : T_1 \times \ldots \times T_n | \phi\,[\pi_1^n\,(x), \ldots, \pi_n^n\,(x)]]$$

Employing dependent products is no different to this, i.e. any n-ary specification that contains dependencies in its declaration can be rewritten as a unary specification using the dependent product type constructor. In general, a specification of the form[34]:

$$R \triangleq [x_1 : T_1, \ldots, x_n : T_n\,[x_1, \ldots, x_{n-1}]\,| \phi\,[x_1, \ldots, x_n]]$$

is taken to be shorthand for a specification of the form:

$$R \triangleq [x : \Sigma y_1 : T_1 \cdot \ldots \cdot \Sigma y_{n-1} : T_{n-1} \cdot T_n | \phi\,[\pi_1^n\,(x), \ldots, \pi_n^n\,(x)]]$$

Let us demonstrate this with an example taken from our informal definitions of the previous chapter. We begin with the simple example of non-membership of sets, which we informally defined previously as "for $a : T$ and $b : set\,(T)$, we define $a \notin b$ as $\neg\,(a \in b)$". We may write a naive specification for this as follows:

$$\notin \triangleq [T : \mathcal{U}, a : T, b : set\,(T)\,| \neg\,(a \in b)]$$

[34] Where each type T_i $(1 < i \leq n)$ may depend on any of the variables declared before it $(T_i\,[x_1, \ldots, x_{i-1}])$. For example, T_1 must not depend on anything, and T_3 may depend on the values of x_1 and x_2.

where we include the declaration $T : \mathcal{U}$ to be explicit and unambiguous[35]. Here, the legitimacy of $\mathit{set}\,(T)$ type is dependent on T, which we must use the dependent product type to resolve. Therefore, this specification is a shorthand for the specification rewritten:

$$\notin \triangleq \left[x : \Sigma y_1 : \mathcal{U} \cdot \Sigma y_2 : T \cdot \mathit{set}\,(T)\,|\neg\left(\pi_2^3\,(x) \in \pi_3^3\,(x)\right)\right]$$

where $\pi_1^3\,(x)$ is T, $\pi_2^3\,(x)$ is a, and $\pi_3^3\,(x)$ is b.

With this we conclude our discussion on dependent types. We have shown how to use dependent products to resolve dependencies both in contexts and in specifications. As before, we will continue to write our specifications in the n-ary form, as we find them much more intuitive and simple to understand. We are now capable of formalizing almost all of our informal definitions of the previous chapter, with the exception of those that require recursive definitions which we examine in the next section (§6.3). For consistency we reiterate our inequality example.

Example 6 Previously we wrote "for $a : T$ and $b : \mathit{set}\,(T)$, we define $a \notin b$ as $\neg\,(a \in b)$", which we can now rewrite as[36]:

$$\notin \triangleq [a : T, b : \mathit{set}\,(T)\,|\neg\,(a \in b)]$$

Example 7 Previously we wrote "for $a : T$ and $b, c : \mathit{set}\,(T)$, we define $a \ominus b = c$ as $b = a \oplus c$", which we can now rewrite as:

$$\ominus \triangleq [a : T,\;\; b, c : \mathit{set}\,(T)\,|a \notin c \wedge b = a \oplus c]$$

As the above relation is functional we write $a \ominus b$ for the unique c. To clarify, $\ominus$ is a binary function written in the infix notation such that $\ominus : \mathit{set}\,(T) \times T \longmapsto \mathit{set}\,(T)$ (see p.62) holds where $T : \mathcal{U}$

Example 8 Previously we wrote "for $a, b : \mathit{set}\,(T)$, we define $a \subseteq b$ as $\forall x \in a \bullet x \in b$", which we can now rewrite as:

$$\subseteq \triangleq [a, b : \mathit{set}\,(T)\,|\forall x \in a \bullet x \in b]$$

Example 9 Previously we wrote "for $a, b : \mathit{set}\,(T)$, we define $a =_{\mathit{set}(T)} b$ as $a \subseteq b \wedge b \subseteq a$", which we can now rewrite as:

$$=_{\mathit{set}(T)} \triangleq [a, b : \mathit{set}\,(T)\,|a \subseteq b \wedge b \subseteq a]$$

[35] Such declarations are usually neglected if they are recoverable form the context, or not ambiguous to do so.

[36] This is the same specification as we saw previously, with the exception that the declaration of T has been abstracted.

Example 10 Previously we wrote "for $a, b : \mathit{set}\,(T)$, we define $a \subset b$ as $a \subseteq b \land a \neq b$", which we can now rewrite as:

$$\subset \triangleq [a, b : \mathit{set}\,(T)\,|a \subseteq b \land a \neq b]$$

Example 11 Previously we wrote "for $a, b, c : \mathit{set}\,(T)$, we define $a \cup b = c$ as $a \subseteq c \land b \subseteq c \land \forall x \in c \bullet x \in a \lor x \in b$", which we can now rewrite as:

$\cup$

$a, b, c : \mathit{set}\,(T)$

$$a \subseteq c \land b \subseteq c \land$$
$$\forall x \in c \bullet x \in a \lor x \in b$$

As the above relation is functional we write $a \cup b$ for the unique c. To clarify, $\cup$ is a binary function written in the infix notation such that $\cup : \mathit{set}\,(T) \times \mathit{set}\,(T) \longmapsto \mathit{set}\,(T)$ (see p.62) holds where $T : \mathcal{U}$

Example 12 Previously we wrote "for $a, b, c : \mathit{set}\,(T)$, we define $a \cap b = c$ as $(\forall x \in c \bullet x \in a \land x \in b) \land (\forall x \in a \cup b \bullet x \in a \land x \in b \implies x \in c)$", which we can now rewrite as:

$\cap$

$a, b, c : \mathit{set}\,(T)$

$$(\forall x \in c \bullet x \in a \land x \in b) \land$$
$$(\forall x \in a \cup b \bullet x \in a \land x \in b \implies x \in c)$$

As the above relation is functional we write $a \cap b$ for the unique c. To clarify, $\cap$ is a binary function written in the infix notation such that $\cap : \mathit{set}\,(T) \times \mathit{set}\,(T) \longmapsto \mathit{set}\,(T)$ (see p.62) holds where $T : \mathcal{U}$

6.2 Schema Type

Before we progress onto the matter of recursive specifications, we need to add a mechanism to reason over them. In Chapter 5 we demonstrated how we can develop type theories with their primitive relations and operations. In this section we develop a theory of relations, i.e. elevating relations to first class citizens of the theory, just as we did with types. We will then be able to reason over relations as we do any other terms in the theory, allowing us to define operations on relations. We have already seen two such operations, *Dom* and *Ran*, where the relation in question was assumed to be in the context as it could not be an argument. That is, the relation is provided as an argument informally. With a type of relations we can move the relation from its position as meta-notation to a formal argument. We proceed to then demonstrate a few more examples of using this type constructor by defining some notions from the *Z* specification language: schema conjunction/disjunction and schema composition. We then conclude this section with the transitive operator on relations.

We introduce our new type constructor for specifications, *schema*, with the formation rule $\mathbf{R}_4$. In $\mathbf{R}_5$ is an introduction rule that tells us that the well formed specification $[x : T|\phi]$ is of type $\mathit{schema}\,(T)$. Rule $\mathbf{R}_6$ shows that a term of the *schema* type forms a Σ proposition (see Definition 1) under application. $\mathbf{R}_7$ and $\mathbf{R}_8$ ensure that specifications behave as we would expect given the original specification introduction ($\mathbf{R}_3$) and elimination ($\mathbf{R}_4$) rules. Finally, $\mathbf{R}_9$ shows us that renaming a declared (bound) variable results in an equal specification, where equality of specifications is taken to be intensional. The side condition of these rules, as with specifications in general, is that ϕ must be in Σ.

$$\mathbf{R}_4\ \frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash \mathit{schema}\,(T) : \mathcal{U}} \qquad \mathbf{R}_7\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop} \quad \Gamma \vdash t : T \quad \Gamma \vdash \phi\,[t/x]}{\Gamma \vdash [x : T|\phi]\,(t)}$$

$$\mathbf{R}_5\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop}}{\Gamma \vdash [x : T|\phi] : \mathit{schema}\,(T)} \qquad \mathbf{R}_8\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop} \quad \Gamma \vdash t : T \quad \Gamma \vdash [x : T|\phi]\,(t)}{\Gamma \vdash \phi\,[t/x]}$$

$$\mathbf{R}_6\ \frac{\Gamma \vdash s : \mathit{schema}\,(T) \quad \Gamma \vdash t : T}{\Gamma \vdash s\,(t)\ \ \mathit{prop}} \qquad \mathbf{R}_9\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop}}{\Gamma \vdash [x : T|\phi] =_{\mathit{schema}(T)} [y : T|\phi\,[y/x]]}$$

We are now able to show how this new type can be used in practice, and we begin with reformulating the *Dom* and *Ran* specifications we originally presented at the start of this chapter:

Dom

$R : schema\,(A \times B)$
$x : A$

$\exists y : B \bullet R\,(x, y)$

Ran

$R : schema\,(A \times B)$
$y : B$

$\exists x : A \bullet R\,(x, y)$

The difference here is that we no longer have the subscript meta-notation to specify which relation we are interested in, i.e. Dom_R. Rather, the relation is now a formal argument of the specified relation. So when we previously wrote:

$$\forall x : T_{in} \bullet Dom_R\,(x) \implies \exists! y : T_{out} \bullet R\,(x, y)$$

when we defined total functional relations, we may now write this more formally as:

$$\forall x : T_{in} \bullet Dom\,(R, x) \implies \exists! y : T_{out} \bullet R\,(x, y)$$

Our other examples come from the Z specification language, schema conjunction, disjunction and composition. Schema conjunction makes "a new schema with the declarations of the two component schemas merged and their predicates conjoined" [Lightfoot, 2001]. We can define a version[37] of this in our specification notation as follows:

$\wedge$

$R_a : schema\,(A \times B)$
$R_b : schema\,(B \times C)$
$R_{out} : schema\,(A \times B \times C)$

$R_{out} = [x : A, y : B, z : C | R_a\,(x, y) \wedge R_b\,(y, z)]$

where clearly the $\wedge$ relation is functional as R_{out} is defined as a schema within the spe-

[37] Our versions are loose interpretations of their respective notions from the Z specification language.

cification. Therefore, we can define a new function symbol $\wedge$ where we write $R_a \wedge R_b$ for the unique R_{out}. To clarify, $\wedge$ is a binary function written in the infix notation such that $\wedge : \mathit{schema}\,(A \times B) \times \mathit{schema}\,(B \times C) \longmapsto \mathit{schema}\,(A \times B \times C)$ (see p.62) holds where $A : \mathcal{U}$, $B : \mathcal{U}$ and $C : \mathcal{U}$.

The case of schema disjunction is the mirror image of this, i.e. schema disjunction makes "a new schema with the declarations of the two component schemas merged and their predicates disjoined" [Lightfoot, 2001]. Therefore, as previously, we can define a version of this in our specification notation as follows:

$\vee$

$$R_a : \mathit{schema}\,(A \times B)$$
$$R_b : \mathit{schema}\,(B \times C)$$
$$R_{out} : \mathit{schema}\,(A \times B \times C)$$

$$R_{out} = [x : A, y : B, z : C | R_a\,(x, y) \vee R_b\,(y, z)]$$

where, again, the relation $\vee$ is functional as R_{out} is defined within the specification. Therefore, we can define a new function symbol $\vee$ where we write $R_a \vee R_b$ for the unique R_{out}. To clarify, $\vee$ is a binary function written in the infix notation such that $\vee : \mathit{schema}\,(A \times B) \times \mathit{schema}\,(B \times C) \longmapsto \mathit{schema}\,(A \times B \times C)$ (see p.62) holds. The overloaded $\wedge$ and $\vee$ symbols do not pose a problem with those previously defined as the contexts in which they are used are completely distinct.

Let us provide one final example of using the specification type in the form of relational composition. Relation composition is a form of what is called *schema hiding* in the Z specification language. We take relational composition to result in a relation where the intermediate terms between the two have been hidden by existential quantification. Therefore, we define this as follows:

$\circ$

$$R_a : schema\,(A \times B)$$
$$R_b : schema\,(B \times C)$$
$$R_{out} : schema\,(A \times C)$$

$$R_{out} = [x : A, y : C | \exists z : B \bullet (R_a \wedge R_b)\,(x, z, y)]$$

where, as with the previous two examples, this relation is functional as R_{out} is defined within the specification. Therefore, we can define a new function symbol $\circ$ where we write $R_a \circ R_b$ for the unique R_{out}. To clarify, $\circ$ is a binary function written in the infix notation such that $\circ : schema\,(A \times B) \times schema\,(B \times C) \longmapsto schema\,(A \times C)$ (see p.62) holds where $A : \mathcal{U}$, $B : \mathcal{U}$ and $C : \mathcal{U}$.

Our final example is the *transitivity* operator for relations[38], which is an essential component in our definition of LePUS3's type theory in **TPL** (Chapter 7). That is, an operation on relations that when applied results in the transitive closure of that relation. The transitivity operator, a superscript plus symbol ($^+$), is formulated in rule $\mathbf{R}_{10}$. The operator may only be applied to relations whose first and second arguments are of the same type. $\mathbf{R}_{11}$ is an introduction rule, where the operator if a pair of terms is in a relation, then it is in the transitive closure of that relation. The actual content of transitive closures is defined in the standard way ($\mathbf{R}_{12}$). Finally, $\mathbf{R}_{13}$ is a simple elimination rule.

$$\mathbf{R}_{10}\ \frac{\Gamma \vdash R : schema\,(T \times T)}{\Gamma \vdash R^+ : schema\,(T \times T)} \qquad \mathbf{R}_{12}\ \frac{\Gamma \vdash R^+\,(a, b) \quad \Gamma \vdash R^+\,(b, c)}{\Gamma \vdash R^+\,(a, c)}$$

$$\mathbf{R}_{11}\ \frac{\Gamma \vdash R : schema\,(T \times T) \quad \Gamma \vdash R\,(a, b)}{\Gamma \vdash R^+\,(a, b)} \qquad \mathbf{R}_{13}\ \frac{\Gamma \vdash R^+\,(a, b)}{\Gamma \vdash R\,(a, b) \vee (\exists x : T \bullet R\,(a, x) \wedge R^+\,(x, b))}$$

6.3 Recursive Schema

Our final discussion on specification focusses on *recursive specifications.* Recursive specifications are those that are defined in terms of themselves, and are an essential component to this work.

[38] Although occurring prior to teasing out our theory of classes from LePUS3, this definition accounts for LePUS3 Definition III, Appendix B.

For example, consider exponentiation of natural numbers, which we defined informally as "for $a, b, c : \mathbb{N}$, we define $a^b = c$ as either $b = \mathtt{0} \wedge c = \mathtt{1}$, or $\exists x, y : \mathbb{N} \bullet x^+ = b \wedge a^x = y \wedge c = a * y$". Notice that the proposition used to define exponentiation uses exponentiation too. Generally, a recursive specification is of the form:

$$R \triangleq [x : T | \phi [R, x]]$$

where the relation R appears in its own definition. Unpacking such relations therefore needs a different treatment than those we have seen previously. As such, we introduce three new rules that govern these recursive relations. These rules are for unary recursive relations, where the use of pairs and dependent products allow us to extend this definition naturally to n-ary relations. First, we assume that

$$\Gamma, x : T, \delta : \mathit{schema}\,(T) \vdash \phi\,[\delta, x]\ \ \mathit{prop}$$

holds where when δ occurs in ϕ it does so as a predicate. $\mathbf{R}_{14}$ is the formation rule, which is very similar to our formation rule for general specifications ($\mathbf{R}_1$) with the exception that we explicitly state that it is well formed for R to appear in its own proposition. $\mathbf{R}_{15}$ is the introduction rule and a closure condition, where if there is an x of the right type for which our recursive proposition holds then we may introduce $R\,(x)$, assuming it is well formed to do so. Finally, $\mathbf{R}_{16}$ is our elimination rule and an induction principle, which we use to compile away the recursive relation into a non-recursive proposition.

$$\mathbf{R}_{14}\ \ \frac{\Gamma, x : T, \delta : \mathit{schema}\,(T) \vdash \phi\,[\delta, x]\ \ \mathit{prop}}{\Gamma \vdash R\,(x)\ \ \mathit{prop}}$$

$$\mathbf{R}_{15}\ \ \frac{\Gamma, x : T, \delta : \mathit{schema}\,(T) \vdash \phi\,[\delta, x]\ \ \mathit{prop} \quad \Gamma \vdash x : T \quad \Gamma \vdash \phi\,[\delta, x]}{\Gamma \vdash R\,(x)}$$

$$\mathbf{R}_{16}\ \ \frac{\Gamma, x : T \vdash \varphi\,[x]\ \ \mathit{prop} \quad \Gamma, x : T \vdash \phi\,[\varphi, x] \implies \varphi\,[x]}{\Gamma, x : T \vdash R\,(x) \implies \varphi\,[x]}$$

These rules come with the same side condition as the general form of specification we have already seen; that the propositions ϕ and φ are restricted to Σ propositions (see Definition 1). However, where previously this ensured that every relation introduced by specification is a conservative extension, this is not the case for recursive specifications. To prove that a recursive definition is a conservative extension of a theory is a much more complex task. We must show that these specifications can always be compiled away to a proposition provable in the original theory. An algorithm for this is given in Theorem 231 in [Turner, 2009], but requires going into much more detail of **TPL** than we are capable of doing in this brief summary. Therefore, we will operate under the assumption that what recursive specifications we provide in the later chapters are conservative extensions.

To demonstrate specifications of this form, consider again the case of exponentiation that we are now able to define succinctly as:

Example 13 Previously we wrote "for $a, b, c : \mathbb{N}$, we define $a^b = c$ as either $b = \mathtt{0} \land c = \mathtt{1}$, or $\exists x, y : \mathbb{N} \bullet x^+ = b \land a^x = y \land c = a * y$", which we can now rewrite as:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{exp}} \\
a, b, c : \mathbb{N} \\
\hline
(b = \mathtt{0} \land c = \mathtt{1}) \lor \\
(\exists x, y : \mathbb{N} \bullet x^+ = b \land \mathit{exp}\,(a, x, y) \land c = a * y) \\
\hline
\end{array}$$

As exp is functional, such that $\mathit{exp} : \mathbb{N} \times \mathbb{N} \longmapsto \mathbb{N}$ (see p.62) holds, we write a^b for the unique c.

Similarly, we can now do the same for our last informal definition, factorial:

Example 14 Previously we wrote "for $a, b : \mathbb{N}$, we define $a! = b$ as either $a = \mathtt{0} \land b = \mathtt{1}$, or $\exists x, y : \mathbb{N} \bullet x^+ = a \land x! = y \land b = a * y$", which we can now rewrite as:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{fac}} \\
a, b : \mathbb{N} \\
\hline
(a = \mathtt{0} \land b = \mathtt{1}) \lor \\
(\exists x, y : \mathbb{N} \bullet x^+ = a \land \mathit{fac}\,(x, y) \land b = a * y) \\
\hline
\end{array}$$

As fac is functional, such that $\mathit{fac} : \mathbb{N} \longmapsto \mathbb{N}$ (see p.62) holds, we write $a!$ for the unique b.

This concludes our definition of specifications, in its general, dependent and recursive form, and a type of relations. In the following chapter we introduce the types and relations that form the theoretical foundations of LePUS3, which we refine and extend from our research in object-oriented programming.

Part II

Theoretical Investigation

LePUS3 [Eden and Nicholson, 2011] is discussed in detail in Chapter 4. In summary, LePUS3 is a visual language whose logical foundations are defined in the First-Order Predicate Logic (**FOPL**) (see Appendix B). It is designed for the specification and automated verification of object-oriented design motifs, application frameworks, and programs. LePUS3 has evolved and matured since its first appearance in [Eden et al., 1997], but despite our best efforts it has suffered from increasing complexity as experienced in evolving software. That is, over time the language has grown more complex and fragile; the underlying theory has become cumbersome and obfuscated. The consequence of this is that further extension is getting progressively difficult, and that steps must be taken to manage its increased complexity.

One of the main causes of complexity is that the definitions (Appendix B) combine components from several aspects of the language. For example, LePUS3 Definition X–XIII define the meaning of formulas by how they are used for design verification. Additionally, LePUS3 Definition IX defines terminology with respect to their visual representation, rather than in general terms. These, and other, limitations are discussed in more detail in §4.3.

We divide LePUS3 into three main components: its visual representation, theoretical foundations, and practical applications. We choose to ignore the visual representation for the most part, as this is outside the scope of this thesis[39]. The practical applications of LePUS3, i.e. the definition of design verification, shall be addressed in Part III. It is the theoretical foundations of LePUS3 that we focus on in this part: to tease out, refine, and fix, all relevant definitions, axioms and assumptions of LePUS3. We choose to develop our new theory using the Typed Predicate Logic (**TPL**) framework (Chapters 7 and 6) as it facilitates defining rich type theories in a clean and elegant fashion. We demonstrate our refined theory's expressive power by using it to specify design motifs and to reason about them (Chapter 8). Finally we extend our theory with respect to the principle of genericity (Chapter 10).

[39] Although Appendix C contains some minimal discussion of the visual notation of LePUS3.

Chapter 7

A Theory of Classes

The purpose of this chapter is the redefinition of the theoretical foundations of LePUS3 within the **TPL** framework[40]. By the end of this chapter we will have constructed our new Theory of Classes (**TC**). **TC** is far more rigorous and expressive than LePUS3, as demonstrated in the next chapter (Chapter 8), but comes at the sacrifice of a visual interpretation for every aspect of the language.

As we progress through this chapter we use simple source code examples to illustrate what we are introducing and why. These examples are written in the widespread and well-known Java programming language [Sun Microsystems Inc., 2006], many of which are taken directly from the Java Development Kit (JDK) itself. We also use Java as a source for inspiration with regards to the new rules and relationships we introduce in **TC**. However, we always strive to ensure our theory describes only those properties and relationships common to object-oriented languages. To keep a fairly unbiased perspective we also make heavy use of [Craig, 2000, p.144], which discusses the common features and distinctions of some of the most common object-oriented languages.

So how do we go about redefining LePUS3? We could define **TC** as a literal definition for definition, and rule for rule, translation from LePUS3 into **TPL**. But we learn nothing new from such an approach, and it does not easily facilitate any exploration beyond the boundaries of the original language. Instead, we base **TC** on the same assumptions and definitions of LePUS3, but rebuilt from the ground up. That is, we use LePUS3 as a guide, but not as gospel. This allows us more flexibility and freedom in our interpretation of LePUS3. As we proceed through the chapter we will discuss key differences in the two definitions.

We discussed the fundamental building blocks of LePUS3 [Eden and Nicholson, 2011] in Chapter 4, an ontology that is expressive enough for most of the structural and creational properties of object-oriented design. We summarize this ontology in Table 7.1[41], where we treat classes and methods as atomic notions rather than being derivable or constructed.

Committing ourselves to such a minimal ontology leads to a clear, elegant, and simple theory.

[40] Our reasons for doing so are summarized at the beginning of this Part (see p.76).

[41] As in LePUS3 Definition VI (Appendix B).

Table 7.1: The fundamental building blocks of object-oriented design

- Classes
- (Nested) sets of classes, where particular attention is given to inheritance hierarchies
- Methods (and their signatures)
- (Nested) sets of methods, where particular attention is given to sets of dynamically-bound methods
- Simple relations on or between classes and methods
- Complex relations on or between (nested) sets of classes or methods

For example, we do not over complicate our theory by differentiating interfaces from implementations (as in DPML, Chapter 3). Both interfaces and implementations are essentially classes whose roles are never as clear cut as those distinctions indicate. However, one of the natural consequences of this is that we are limited from specifying other aspects of software design, such as other programming paradigms, architectural styles, objects, temporal or behavioural relations. For example, we could make our theory more expressive with the added notion of object (as in UML, Chapter 3), but doing so radically changes the focus of our theory and would make it undecidable in the general case. We must remember that the our aim of this work is in automated program verification, and therefore we cannot afford undecidability here. However, this does not prohibit our new theory from being extended in the future to include aspects that we currently choose to ignore. Indeed our work facilitates this to a much greater degree than the current definition of LePUS3, as this chapter will show.

The sections that compose this chapter are dedicated to discussing each of the above building blocks in order. In doing so we define a logical type theory (**TC**) for object-oriented design in the **TPL** framework[42]. That is, in §7.1 we discuss classes and inheritance. In §7.2 we discuss special sets of classes related by inheritance. In §7.3 we discuss methods and their signatures. In §7.4 we discuss sets of dynamically bound methods. In §7.5 we discuss simple relationships on and between classes and methods, such as data members, aggregations, and method invocations. Finally, in §7.6 we discuss complex relationships, those that take a simple relation as an argument, on and between sets of classes or methods.

[42] We build our type theory (**TC**) directly on top of the type theory we constructed in Part I.

7.1 Classes

Classes are the most fundamental building block of object-oriented design. As stated previously, we take classes to be atomic elements of our theory. That is to say, classes cannot be decomposed into constituent components. With this assumption in mind, we begin building our theory of classes (**TC**) by introducing a type of classes into our universe of types ($\mathbf{TC}_1$):

$$\mathbf{TC}_1 \quad \Gamma \vdash \mathbb{CLASS} : \mathcal{U}$$

A term of the $\mathbb{CLASS}$ type can be interpreted in Java as a class, inner class, anonymous inner class, interface, array, or a primitive type. Table 7.2[43] illustrates how this is interpreted against Java by discussing some simple code fragments from the Java SDK[44]. Each line of code introduces a new class into the program, which we articulate in **TC** with the respective representation[45].

The reason for abstracting the distinction between classes, interfaces, and primitive types is because such distinctions are largely implementation details that rarely serve the software designer. For example, Java has both primitive types (**int**, **float**, **double**, etc.) and respective class implementations (Integer, Float, Double, etc.). A designer may wish to specify the need for integers, but should not need to specify if the primitive type (**int**) or the class implementation (Integer) is employed in the final implementation. If, however, such subtleties must be represented and enforced, then we could easily introduce a set of appropriate subtypes as required.

Table 7.2: Modelling Java classes

Code	Description and Representation
int elementCount ...	**int** is a primitive type. $\mathtt{int} : \mathbb{CLASS}$
class Vector ...	Vector is a class $\mathtt{Vector} : \mathbb{CLASS}$
abstract class AbstractList ...	AbstractList is an abstract class $\mathtt{AbstractList} : \mathbb{CLASS}$
interface List ...	List is an interface $\mathtt{List} : \mathbb{CLASS}$
Object[] elementData ...	Object[] is an array $\mathtt{Object[]} : \mathbb{CLASS}$

[43] We simplify the source code used in our illustrations, e.g. by abbreviating the class AbstractList as AList.

[44] Note the distinction between source code (sanserif font) and mathematical `constants` (typewriter font).

[45] We've modified the code in these tables slightly to remove the Java's generics, which at this point may be confusing, and simplified variable names. So that we can discuss components and properties of a program in our theory, we informally introduce representative constants. However, we do not actually admit these constants into our theory. We leave the discussion of how to address specific program elements to Part III of this thesis. Usually we'll use the same text for constant names as their source code counterparts, but where such text is too long we will use an appropriate alternative.

As discussed in §2.1.3, inheritance is a core mechanism in class-based object-oriented languages. It groups classes into hierarchies, and facilitates a form of polymorphism. There are many interpretations as to what inheritance means (see [Taivalsaari, 1996]), and we will not attempt to capture them all. Instead we restrict ourselves to the two most common interpretations of inheritance, *subclassing* and *subtyping*, as implemented in Java by the **extends** and **implements** keywords respectively.

Additionally, we opt to group these two notions together as a single binary relation over classes. We do this for the same reason we abstract from the distinctions between classes, interfaces and primitive types. That is, a designer needs to specify an inheritance relationship, but often will not care exactly which mechanism is used in the final implementation. Decisions like this keep our theory small and elegant, yet expressive enough for articulating object-oriented design at our level of abstraction. And as previously, if such a distinction is required, then we can always add more relations to that effect.

We introduce inheritance in **TC** with the binary relation *Inherit* ($\mathbf{TC}_2$). This is accompanied with rules that articulate our basic assumptions of inheritance. Firstly, a class may not inherit from itself ($\mathbf{TC}_3$), and secondly there are no cycles in the inheritance graph ($\mathbf{TC}_4$)[46]. Notice that unlike our previous relations, we do not have clear introduction or elimination rules. This is because what can be introduced or eliminated depends on the program we are reasoning over. We will come back to this matter in Part III (see p.180), where we discuss how a program can be used to introduce or eliminate relations in our theory. For now we allude to this information with examples. For example, consider Table 7.3, which shows lines of code from the Java SDK[47] that introduce inheritance relationships into a program (either explicitly or implicitly) against their respective representative relation in **TC**.

$$\mathbf{TC}_2 \quad \Gamma \vdash \mathit{Inherit} : \mathit{schema}\,(\mathbb{CLASS} \times \mathbb{CLASS})$$

$$\mathbf{TC}_3 \quad \frac{\Gamma \vdash c : \mathbb{CLASS}}{\Gamma \vdash \neg \mathit{Inherit}\,(c, c)}$$

$$\mathbf{TC}_4 \quad \frac{\Gamma \vdash \mathit{Inherit}\,(a, b)}{\Gamma \vdash \neg \mathit{Inherit}\,(b, a)}$$

We now have the vocabulary to discuss one of the fundamental principles of inheritance [Eden and Nicholson, 2011]. That is, a subclass inherits the properties of its superclass, and is

[46] This is the second axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII in Appendix B.

[47] Taken from the java.util package.

Table 7.3: Modelling inheritance

Code	Description and Representation
interface List ...	Interface List implicitly subclasses Object $\mathit{Inherit}(\texttt{List}, \texttt{Object})$
class AbstractList ... **implements** List ...	Class AbstractList is a subtype of interface List $\mathit{Inherit}(\texttt{AbstractList}, \texttt{List})$
class Vector ... **extends** AbstractList ...	Class Vector subclasses AbstractList $\mathit{Inherit}(\texttt{Vector}, \texttt{AbstractList})$
Object[] elementData ...	Array Object[] implicitly subclasses Object $\mathit{Inherit}(\texttt{Object[]}, \texttt{Object})$

an instance of all its superclasses. For example, if a class c contains a data member of some other class d then so does every subclass of c. Similarly, if a method m returns an instance of class c then by subsumption it returns an instance of all superclasses of c. Therefore, we introduce rules to this effect[48]. The rule $\mathbf{TC}_5$ formalizes our example of data members, and is only concerned with the first argument of the relation. If the relation holds with the superclass as the first argument, then it holds for the subclass too. The rule $\mathbf{TC}_6$ formalizes our example of a methods returned instance, and is the mirror image of $\mathbf{TC}_5$. That is, it is only concerned with the last argument of the relation, and if the relation holds for the subclass as the last argument, then it holds for the superclass too.

$$\mathbf{TC}_5 \quad \frac{\begin{array}{c}\Gamma \vdash T : \mathcal{U} \quad \Gamma \vdash R : \mathit{schema}\,(\mathbb{CLASS} \times T) \\ \Gamma \vdash t : T \quad \Gamma \vdash \mathit{super}, \mathit{sub} : \mathbb{CLASS} \\ \Gamma \vdash \mathit{Inherit}\,(\mathit{sub}, \mathit{super}) \quad \Gamma \vdash R\,(\mathit{super}, t)\end{array}}{\Gamma \vdash R\,(\mathit{sub}, t)}$$

$$\mathbf{TC}_6 \quad \frac{\begin{array}{c}\Gamma \vdash T : \mathcal{U} \quad \Gamma \vdash R : \mathit{schema}\,(T \times \mathbb{CLASS}) \\ \Gamma \vdash t : T \quad \Gamma \vdash \mathit{sub}, \mathit{super} : \mathbb{CLASS} \\ \Gamma \vdash \mathit{Inherit}\,(\mathit{sub}, \mathit{super}) \quad \Gamma \vdash R\,(t, \mathit{sub})\end{array}}{\Gamma \vdash R\,(t, \mathit{super})}$$

There are two important side conditions of these rules:

1. We restrict T to be either $\mathbb{CLASS}$ or $\mathbb{METHOD}$[49]. This ensures that we are only talking about binary relations over the types where these rules make sense.
2. We restrict R to be one of the relations defined in this chapter (see §7.5). This ensures the effects of the rules stays within our scope. For example, consider allowing equality to be R

[48] This formalizes LePUS3 Definition X (Appendix B), to the exclusion of the model theoretic/practical application component discussed in Chapter 11.

[49] We introduce the $\mathbb{METHOD}$ type in the next section of this chapter.

in rule $\mathbf{TC}_5$. It is not difficult to see here that all hell breaks loose as it allows us to prove

$$\Gamma a, b : \mathbb{CLASS} \vdash \mathit{Inherit}\,(a, b) \implies a = b$$

which is a contradiction by rule $\mathbf{TC}_3$ (p.80).

These side conditions demonstrate that, although we maintain a tight control over how these rules can be used, they are fragile and their misuse can be very dangerous. These rules are not elegant, and they suggest that there is a much more logical way in which to achieve the same goals.

An alternative method of accomplishing this is for instances of the class type to be types themselves[50]. That is, that if $c : \mathbb{CLASS}$ then $c : \mathcal{U}$, and *Inherit* can be used to deduce the subtype relation. This would give us rules $\mathbf{TC}_{5-6}$ for free, but comes at the consequence of introducing objects into our theory. Such a radical change is beyond the scope of this thesis. For this reason we opt to maintain the rules above, which are adequate for our purposes.

There is a very interesting outcome from either rule $\mathbf{TC}_5$, $\mathbf{TC}_6$, or even from turning class instances into types. They all dramatically change the meaning of the *Inherit* relation. Take rule $\mathbf{TC}_5$ for example, if we apply *Inherit* to R we end up with the following (simplified) rule:

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{super}, \mathit{sub}, t : \mathbb{CLASS} \\ \Gamma \vdash \mathit{Inherit}\,(\mathit{sub}, \mathit{super}) \quad \Gamma \vdash \mathit{Inherit}\,(\mathit{super}, t)\end{array}}{\Gamma \vdash \mathit{Inherit}\,(\mathit{sub}, t)}$$

The conclusion we draw from this is that the *Inherit* relationship is a transitive one by nature, as summarized in the following remark.

Remark 1 The *Inherit* and *Inherit*$^+$ relations are equivalent:

$$\Gamma, a, b : \mathbb{CLASS} \vdash \mathit{Inherit}\,(a, b) \iff \mathit{Inherit}^+\,(a, b)$$

But is this consequence desirable? Indeed it is, and is a concept supported by [Craig, 2000, pp.32–38]. When we specify that one class inherits from another it does not usually matter if that relationship is direct or otherwise. If we must specify a direct inheritance relationship, then we can always introduce a new relation to that effect.

Craig also states that inheritance is a strict order relation on classes [Craig, 2000, pp.32–38],

[50] This would have been very difficult to introduce into LePUS3, and even then the solution would not be clean or elegant. However, in **TC** it would only take one small rule.

which we can now very easily prove. To do so we must show that *Inherit* is irreflexive, that for any class c then $\mathit{Inherit}\,(c,c)$ does not hold; asymmetric, that for any elements c_1, c_2 then $\mathit{Inherit}\,(c_1,c_2)$ implies $\mathit{Inherit}\,(c_2,c_1)$ does not hold; and transitive, that for any elements c_1, c_2, c_3 then $\mathit{Inherit}\,(c_1,c_2)$ and $\mathit{Inherit}\,(c_2,c_3)$ implies $\mathit{Inherit}\,(c_1,c_3)$.

Proposition 1 *Inherit* is a strict order relation.

Proof. *Inherit* is irreflexive ($\mathbf{TC}_3$, p.80), asymmetric ($\mathbf{TC}_5$, p.81) and transitive ($\mathbf{TC}_5$ and/or $\mathbf{TC}_6$) ■

Since *Inherit* is a strict order relation on the $\mathbb{CLASS}$ type[51], the respective partial order relation is easily definable. We call the partial order relation *Subclass*, and define it as follows:

$$\mathit{Subclass} \triangleq [a,b : \mathbb{CLASS} | \mathit{Inherit}\,(a,b) \vee a = b]$$

7.2 Inheritance Hierarchies

We introduced the notion of an inheritance hierarchy (or, simply *hierarchy*) in §2.1.3 as a set of classes that contains a superclass such that all other members inherit (directly or indirectly) from it. In other words, hierarchies represent a branch in the inheritance tree. For example, the set of Java collections (and their interfaces) all inherit from the Collection interface. This is illustrated in Table 7.4, where all classes mentioned constitute a hierarchy.

Table 7.4: Modelling the Java Collections hierarchy

Code	Description and Representation
interface Collection ...	Collection is an interface $\mathtt{Collection} : \mathbb{CLASS}$
interface List ... **implements** Collection ...	Interface List is a subtype of interface Collection $\mathtt{List} : \mathbb{CLASS}$ and $\mathit{Inherit}\,(\mathtt{List}, \mathtt{Collection})$
class AbstractList ... **implements** List ...	Class AbstractList is a subtype of interface List $\mathtt{AbstractList} : \mathbb{CLASS}$ and $\mathit{Inherit}\,(\mathtt{AbstractList}, \mathtt{List})$
class Vector ... **extends** AbstractList ...	Class Vector subclasses AbstractList $\mathtt{Vector} : \mathbb{CLASS}$ and $\mathit{Inherit}\,(\mathtt{Vector}, \mathtt{AbstractList})$

LePUS3 took hierarchies to be fundamental to object-oriented design, and therefore made them primitive terms of the language. See Figure 4.2 for an illustration of their definition as provided in LePUS3 Definition IV (Appendix B).

[51] Where it follows that if there are n classes to consider then there are at most $(n * (n-1))\,/2$ possible *Inherit* relationships

We also see hierarchies as fundamental to object-oriented design, but we do not define them as primitive terms of our theory. Rather, our subclassing mechanism facilitates the definition of hierarchies in a clean and elegant fashion. We begin by defining a new relation that articulates the requirements of a hierarchy. A hierarchy is a set of classes that must contain a *root* class and at least one other class. Additionally, all members of the hierarchy must either inherit from, or be equal to, the class *root* by virtue of the *Subclass* relation[52].

$Hierarchy$

$$h : set\,(\mathbb{CLASS})$$

$$\begin{gathered}\exists root \in h \bullet h \neq root \oplus \varnothing_{\mathbb{CLASS}} \wedge \\ \forall x \in h \bullet Subclass\,(x, root)\end{gathered}$$

All we need do now is use our subtype notation to introduce a new hierarchy type:

$$\mathbb{HIERARCHY} \triangleq \{h : set\,(\mathbb{CLASS})\,|Hierarchy\,(h)\}$$

7.3 Methods

We move our discussion on to the matter of methods, which we treat as atomic elements in the same way we did classes. That is to say, methods cannot be decomposed into instructions for example. With this assumption in mind, we introduce a type of methods into our universe of types ($\mathbf{TC}_7$). In the same way we did for classes, we do not make provisions for the distinctions between static methods, class (instance) methods, procedures or functions. This level of abstraction is most appropriate to provide for the software designer as it provides the required level of detail while abstracting from implementation level decisions. However, if such distinctions are required, then these distinctions can be added as subtypes of $\mathbb{METHOD}$.

We also introduce a relation that governs the relationship between classes and methods, i.e. method membership. $MethodMember\,(c, m)$ tells us that the class c contains the method m[53]

[52] Where it follows from this definition that if there are n classes being considered, then there can be at most $2^n - (n+1)$ hierarchies, where a single hierarchy must contain between 2 and n classes.

[53] We are careful to say *contains* here rather than *defines*, as the rules $\mathbf{TC}_5$ and $\mathbf{TC}_6$ make it inappropriate to do so. That is we want to deduce that a subclass *contains* the methods defined in a superclass, it does not define

($\mathbf{TC}_8$).

$$\mathbf{TC}_7 \;\; \Gamma \vdash \mathbb{METHOD} : \mathcal{U}$$

$$\mathbf{TC}_8 \;\; \Gamma \vdash \mathit{MethodMember} : \mathit{schema}\,(\mathbb{CLASS} \times \mathbb{METHOD})$$

We could add additional constraints to the $\mathit{MethodMember}$ relation, requiring that a method must be contained by at least one class for example. This would be fine for languages such as Java and Smalltalk, where methods must always be attached to classes. But such rules would not be applicable to languages like C++, where methods can be defined globally outside the context of a single class. Therefore, to constrain the $\mathit{MethodMember}$ relation in this way would align our theory with one set of languages over another. Therefore, we opt to not impose such rules, which also keeps our theory inline with that of LePUS3. In Table 7.5 we illustrate these relationships against lines of code from the Java SDK.

Table 7.5: Modelling method membership

Code	Description and Representation

interface List ... {...
 int indexOf(Object o);
...}
	List interface contains the method List.indexOf(Object)
	`List` : $\mathbb{CLASS}$ and `List.indexOf(Object)` : $\mathbb{METHOD}$ and
	$\mathit{MethodMember}$ (`List`, `List.indexOf(Object)`)

class Vector ... {...
 int indexOf(Object o) {...}
...}
	Class Vector contains the method Vector.indexOf(Object)
	`Vector` : $\mathbb{CLASS}$ and `Vector.indexOf(Object)` : $\mathbb{METHOD}$ and
	$\mathit{MethodMember}$ (`Vector`, `Vector.indexOf(Object)`)

Now that we have methods, and their relationship to classes, we must address how we identify methods. As with compilers, we use signatures to identify methods. As we briefly mentioned in §2.1.4, a signature is often taken to be the method's name, the type of its arguments, and its return type. However, in our experience the return type does not help to distinguish one method from another. For example, in Java it would be a compile-time error for two methods to be given identical signatures that differ only in their respective return types. For this reason we consider signatures to be only those details that actually identify a method within a given class, i.e. their name and types of its arguments. We could easily articulate the return type as a relation, if so desired, but in our experience this is often unnecessary.

To this end, we introduce a new type for signatures in rule $\mathbf{TC}_9$, and a relation that allows us to identify methods by their signature with the formation rule $\mathbf{TC}_{10}$. As opposed to $\mathit{MethodMember}$, the $\mathit{SignatureOf}$ relation is one we can constrain. We enforce that a method must have exactly

them itself. We have found that this simple understanding is good enough at our level of object-oriented design [Eden and Nicholson, 2011]. However, if we wished to make a relation to articulate the definition of a method, we could easily do so.

one signature[54] ($\mathbf{TC}_{11}$), and that no signatures can exist without being related to some method ($\mathbf{TC}_{12}$). We illustrate these relationships in Table 7.6 against Java source code from the Java SDK. For consistency, we use the same code fragments as from Table 7.5, only this time we focus on the signatures of methods rather than their membership. We also add an additional code fragment that exemplifies the ellipses notation that allows a method to be executed with a variable number of arguments[55] [Sun Microsystems Inc., 2006, §Varargs]. PHP has a similar mechanism where a method can be defined with default values so that they may be omitted later [Achour et al., 2010]. These are both instances of *polyadic methods*, a form of simple polymorphism, and motivates an argument for a method to have more than one signature [Craig, 2000]. However, at our level of abstraction, polyadic methods can be considered simple syntactic sugar that is compiled away. Put another way, it is a programming shorthand that does not affect the overall design or structure of the program. For this reason we treat polyadic methods as we would any other method.

$$\mathbf{TC}_9 \quad \Gamma \vdash \mathbb{SIGNATURE} : \mathcal{U}$$

$$\mathbf{TC}_{10} \quad \Gamma \vdash \mathit{SignatureOf} : \mathit{schema}\,(\mathbb{SIGNATURE} \times \mathbb{METHOD})$$

$$\mathbf{TC}_{11} \quad \frac{\Gamma \vdash m : \mathbb{METHOD}}{\Gamma \vdash \exists! x : \mathbb{SIGNATURE} \bullet \mathit{SignatureOf}\,(x, m)}$$

$$\mathbf{TC}_{12} \quad \frac{\Gamma \vdash s : \mathbb{SIGNATURE}}{\Gamma \vdash \exists x : \mathbb{METHOD} \bullet \mathit{SignatureOf}\,(s, x)}$$

Interestingly, rule $\mathbf{TC}_{11}$ ensures that the $\mathit{SignatureOf}$ relation is a total functional relation, the proof of which is trivial. Therefore, we will also admit a new unary function symbol SigOf, where we write $\mathit{SigOf}\,(m)$ for the unique s such that $\mathit{SigOf} : \mathbb{METHOD} \longmapsto \mathbb{SIGNATURE}$ (see p.62) and $\mathit{SignatureOf}\,(s, m)$ both hold.

Finally, we introduce rule $\mathbf{TC}_{13}$ that formalizes one of the core axioms of LePUS3[57]. This rule is based on the intuitive assumption that for any signature and class, no more than one method may be identified. When there is more than one possible method available, such as in the case of method overriding, then the definition of the programming language (enforced by the compiler,

[54] This is the third axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII in Appendix B.

[55] Taken from the javax.swing.plaf.basic package.

[57] This is the first axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII in Appendix B.

Table 7.6: Modelling method signatures

Code	Description and Representation

interface List ... {...
int indexOf(Object o);
...}
	Method List .indexOf(Object) has the signature indexOf(Object)
	`Vector.indexOf(Object)` : $\mathbb{METHOD}$ and
	`indexOf(Object)` : $\mathbb{SIGNATURE}$ and
	$SignatureOf$ (`List.indexOf(Object)`, `indexOf(Object)`)

class Vector ... {...
int indexOf(Object o) {...}
...}
	Method Vector.indexOf(Object) has the signature indexOf(Object)
	`Vector.indexOf(Object)` : $\mathbb{METHOD}$ and
	`indexOf(Object)` : $\mathbb{SIGNATURE}$ and
	$SignatureOf$ (`Vector.indexOf(Object)`, `indexOf(Object)`)

class BasicMenuItemUI ... {...
int max(**int** ... values) {...}
...}
	The method BasicMenuItemUI.max(**int**...) is polyadic[56]
	`BasicMenuItemUI.max(int...)` : $\mathbb{METHOD}$ and
	`max(int...)` : $\mathbb{SIGNATURE}$ and
	$SignatureOf$ (`BasicMenuItemUI.max(int...)`, `max(int...)`)

virtual machine, etc.) identifies exactly which one of those methods is identified.

$$\mathbf{TC}_{13}\quad \frac{\begin{array}{c}\Gamma \vdash c : \mathbb{CLASS} \quad \Gamma \vdash m_1, m_2 : \mathbb{METHOD} \\ \Gamma \vdash MethodMember\,(c, m_1) \quad \Gamma \vdash MethodMember\,(c, m_2) \\ \Gamma \vdash SigOf\,(m_1) = SigOf\,(m_2)\end{array}}{\Gamma \vdash m_1 = m_2}$$

However, this rule is problematic in the case of method overriding as it contradicts rule $\mathbf{TC}_5$ (p.81). Method overriding is a very common form of polymorphism implemented in most object-oriented programming languages[58]. Overriding allows more specialized behaviour to replace (hide) more general behaviour. Craig defines overriding (or redefinition) as "a method, m_1, redefines another method, m_2, when m_2 is defined in a superclass of the class in which m_1 is defined and both m_1 and m_2 have the same signature or related, but their behaviour is different" [Craig, 2000, p.153]. Given this, we may articulate the knowledge in the case of method overriding, thereby illustrating the inconsistency of the two rules:

Example 15 We assume the following knowledge in our context, which exemplifies method overriding:

- $c_1, c_2 : \mathbb{CLASS}$
- $m_1, m_2 : \mathbb{METHOD}$
- $m_1 \neq m_2$
- $SigOf\,(m_1) = SigOf\,(m_2)$

[58] We know of no object-oriented language that does not implement method overriding, but that does not mean that they don't exist.

- $Inherit\,(c_2, c_1)$
- $MethodMember\,(c_1, m_1) \wedge MethodMember\,(c_2, m_2)$

That is, we have two methods that share the same signature, and are each members of different classes that are related by inheritance. By rule $\mathbf{TC}_5$ (p.81), we may conclude that:

- $MethodMember\,(c_2, m_1)$

and by $\mathbf{TC}_{13}$ (p.87) we may therefore conclude that $m_1 = m_2$. This is a contradiction as we already know that $m_1 \neq m_2$ holds.

This issue is a consequence of the explicit identification of methods by their signatures, which introduces issues of scope. This is an unresolved problem in LePUS3 that has only come to our attention through conducting this research. The only reason we discovered it now is that the theory and intuitions on which LePUS3 is based is made explicit by their definition in **TPL**. However, identifying the problem is only half the battle: we must also resolve the issue. To solve the problem we must ensure that methods can be overridden, i.e. that in not all cases can we conclude that a class inherits a method from its superclass. Therefore, we replace rule $\mathbf{TC}_5$ with two, more specific, cases: $\mathbf{TC}_{5a}$ and $\mathbf{TC}_{5b}$. Rule $\mathbf{TC}_{5a}$ is identical to $\mathbf{TC}_5$, with the exception that it ignores the problematic case of the *MethodMember* relation.

$$\mathbf{TC}_{5a}\quad \frac{\begin{array}{c}\Gamma \vdash T : \mathcal{U} \quad \Gamma \vdash R : schema\,(\mathbb{CLASS} \times T) \quad \Gamma \vdash R \neq MethodMember \\ \Gamma \vdash t : T \quad \Gamma \vdash sub, super : \mathbb{CLASS} \\ \Gamma \vdash Inherit\,(sub, super) \quad \Gamma \vdash R\,(super, t)\end{array}}{\Gamma \vdash R\,(sub, t)}$$

Now we need to deal with the case where $R = MethodMember$. Rule $\mathbf{TC}_{5b}$ does just this. The first and second lines of the premises are familiar from $\mathbf{TC}_{5a}$ where $T = \mathbb{CLASS}$ and $R = MethodMember$. The third line is the interesting one, which states the premise that there must not exist a method that:

1. is different to the one contained in the superclass;
2. shares the same signature as the one contained in the superclass; or
3. is a member of the subclass

If no such method exists, then we can conclude that the subclass inherits the method from its superclass:

$$\mathbf{TC}_{5b}\;\frac{\begin{array}{c}\Gamma \vdash \mathit{sub}, \mathit{super} : \mathbb{CLASS} \quad \Gamma \vdash m : \mathbb{METHOD}\\ \Gamma \vdash \mathit{Inherit}\,(\mathit{sub}, \mathit{super}) \quad \Gamma \vdash \mathit{MethodMember}\,(\mathit{super}, m)\\ \Gamma \vdash \neg\exists x : \mathbb{METHOD} \bullet m \neq x \wedge \mathit{SigOf}\,(m) = \mathit{SigOf}\,(x) \wedge \mathit{MethodMember}\,(\mathit{sub}, x)\end{array}}{\Gamma \vdash \mathit{MethodMember}\,(\mathit{sub}, m)}$$

By replacing rule $\mathbf{TC}_5$ with the above two rules we are no longer able to prove the same contradiction. We have therefore removed this inconsistency from our theory. Consequently, we are able to represent accessing the behaviour of a subclass by using the interface of its superclass, otherwise called *inclusion polymorphism* [Craig, 2000, p.152] as mentioned in §2.1.5. We are also now able to define method overriding very easily as a specification in our theory:

Overrides

$m_1, m_2 : \mathbb{METHOD}$

$$\begin{gathered} m_1 \neq m_2 \wedge \mathit{SigOf}\,(m_1) = \mathit{SigOf}\,(m_2) \wedge \\ \exists x, y : \mathbb{CLASS} \bullet \mathit{MethodMember}\,(x, m_1) \wedge \mathit{MethodMember}\,(y, m_2) \wedge \mathit{Inherit}\,(x, y) \end{gathered}$$

The fact that we are able to define new relations like this so easily in **TC** demonstrates how much more elegant our theory is over LePUS3. Because LePUS3 specifications are meta-level constructs, they do not introduce new object-level relations. Although this facility could be added to LePUS3, this would require extending the language and its underlying theory.

Another interesting outcome of the rules so far is our ability to express the *diamond problem* [Martin, 1998]. The diamond problem is faced by languages that implement multiple inheritance, and occurs loosely when there are several possible methods with the same signature that a subclass can inherit. We may are able to articulate the diamond problem as follows[59]:

Problem 1 (The diamond problem) Given the knowledge below, which method is selected when there is a request for the method with signature s in class *sub*?

[59] m_1 is usually also given as abstract, which we are yet to discuss, and omitting this fact does not affect the issue at hand

- $super, c_1, c_2, sub : \mathbb{CLASS}$
- $m_{super}, m_1, m_2 : \mathbb{METHOD}$
- $s : \mathbb{SIGNATURE}$
- $m_{super} \neq m_1$, $m_{super} \neq m_2$, and $m_1 \neq m_2$
- $SigOf\,(m_{super}) = s$, $SigOf\,(m_1) = s$, and $SigOf\,(m_2) = s$
- $Inherit\,(c_1, super)$, $Inherit\,(c_2, super)$, $Inherit\,(sub, c_1)$, and $Inherit\,(sub, c_2)$
- $MethodMember\,(super, m_{super})$, $MethodMember\,(c_1, m_1)$, and $MethodMember\,(c_2, m_2)$

We are able to articulate the diamond problem, but not solve it. That is, exactly which method is selected is a matter for the algorithms in the implementation language's definition, and therefore outside the scope of our theory and that of our level of object-oriented design. However, rules $\mathbf{TC}_{5b}$ (p.89) and $\mathbf{TC}_{13}$ (p.87) do allow us to describe the result of any such algorithm. For example, we may firstly conclude that m_1 must be excluded from consideration as it has already been overridden by m_2 and m_3. Secondly, whichever method is selected (if one is) must override any other candidate methods.

With this we conclude the basic rules of methods. In the next section we introduce a function that constructs sets of methods according to their signatures and scope (the classes of which they are members).

7.4 Clans and Tribes

The previous section articulates all our primitive intuitions of methods, except for one of the most fundamental abstraction mechanisms in object-oriented languages: *dynamic binding.* "Dynamic binding means that the operation that is executed when objects are requested to perform an operation is the operation associated with the object itself and not with one of its ancestors" [Craig, 2000]. However, we do not represent objects, and nor do we not attempt to identify exactly which method is executed at runtime. What we can represent are sets of dynamically bound methods, i.e. a set of methods that all share the same signature such that one of them may be dynamically selected and executed at runtime. We call these sets of methods *clans*[60], where

[60] We use the term *clan* for its abstract similarity to its use in subjects such as genealogy, i.e. people in a Scottish clan all share the same surname

single methods are also referred to as clans (a clan of only one member). Similarly, we call sets of many clans a *tribe*[61].

Clans are made by design rather than by coincidence, i.e. two methods are given the same name because their intent is similar, even if their implementations differ. For example, a class might implement a default algorithm, which a subclass might override with specialized version (or change it entirely). Clans therefore tend to have similar functionality and share common relationships with other parts of the program.

From this we conclude that clans and tribes are important structures of object-oriented design that we must preserve. To this end, we wish to introduce a new function that identifies a method by a class (its scope) and its signature (its name). This function must extend naturally over nested sets such that it preserves the desirable structure of clans by enforcing that methods are grouped by their signatures before their class. However, although our theory allows us to reason over finite sets, it is not currently expressive enough to accommodate reasoning over hierarchies of nested finite sets. To add this functionality we must extend our theory with a new type constructor for nested finite set terms. We call this type contructor $SetOf$, which takes a single type argument $B : \mathcal{U}$ (the base type). $SetOf$ carves up our universe of types, i.e. $SetOf\,(B)$ is the type of which type B, and all types that result from some series of applications of *set* on B, are members[62].

[Turner, 2009] provides just the mechanism for accomplishing this in the form of *inductive types*, the rules for which we present below. $\mathbf{I}_1$ is a formation rule that dictates how a new inductive type is constructed. That is, on the assumption that $\phi\,[T,t]$ *prop* holds for any $t : T$, a new inductive type $\mathbf{I}\,[T,\phi] : \mathcal{U}$ may be formed. $\mathbf{I}_2$ is the closure condition that tells us what the members of the inductive type are. Finally, $\mathbf{I}_3$ is the induction principle that allows us to reason over inductive types:

$$\mathbf{I}_1 \quad \frac{\Gamma, t : T \vdash \phi\,[T,t]\ \ prop}{\Gamma \vdash \mathbf{I}\,[T,\phi] : \mathcal{U}}$$

$$\mathbf{I}_2 \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash \phi\,[\mathbf{I}\,[T,\phi]\,,t]}{\Gamma \vdash t : \mathbf{I}\,[T,\phi]}$$

$$\mathbf{I}_3 \quad \frac{\Gamma, t : T \vdash \theta\,[t]\ \ prop \quad \Gamma \vdash \forall x : T \bullet \phi\,[\theta,x] \implies \theta\,[x]}{\Gamma \vdash \forall x : \mathbf{I}\,[T,\phi] \bullet \theta\,[x]}$$

Importantly, as with specification (see Chapter 6), $\phi\,[T,t]$ is restricted to Σ propositions (see Definition 1). Note that where T occurs in a proposition, it occurs in a subterm of the form $\exists y : T \bullet \sigma\,[x,y]$, and that the proposition $\phi\,[\theta,x]$ is obtained by replacing every occurrence of

[61] We use the term *tribe* for its abstract similarity to its use in subjects such as anthropology, i.e. an organization based on smaller groups (clans in this case).

[62] The members of $SetOf\,(B)$ are therefore B, $set\,(B)$, $set\,(set\,(B))$, $set\,(set\,(set\,(B)))$, and so on.

$\exists y : T \bullet \sigma [x, y]$ with $\exists y : T \bullet \theta [x] \wedge \sigma [x, y]$.

Let us examine this new type in more detail. Firstly we observe that there is no direct rule that allows us to treat a term $t : \mathbf{I} [T, \phi]$ as $t : T$. This is an important result that is derivable via the following lemma:

Lemma 1 $t : \mathbf{I} [T, \phi] \vdash \exists x : T \bullet t = x$

Proof. By induction on $\mathbf{I} [T, \phi]$ using the induction principle $\mathbf{I}_3$. For brevity, let $\theta [t]$ be the proposition $\exists x : T \bullet t = x$.

We are required to prove that each premise of $\mathbf{I}_3$ holds:

1. $t : T \vdash \theta [t]$ *prop*

   This holds immediately by the equality ($\mathbf{L}_{23}$, p.46) and existential quantification ($\mathbf{L}_{26}$, p.47) formation rules.

2. $\Gamma \vdash \forall x : T \bullet \phi [\theta, x] \implies \theta [x]$

   This also holds immediately by observing that the proposition $\phi [\theta, x]$ must contain the subterm $\exists y : T \bullet \theta [x] \wedge \sigma [x, y]$ by definition, and that $\theta [x] \implies \theta [x]$ holds directly.

Given the above we conclude that:

$$t : \mathbf{I} [T, \phi] \vdash \exists x : T \bullet t = x$$

■

Given the above lemma and proof, we are able to derive the following useful rule:

$$\mathbf{I}_{basic} \frac{\Gamma \vdash t : \mathbf{I} [T, \phi]}{\Gamma \vdash t : T}$$

Let us now consider a simple example of inductive types. We paraphrase the paradigm example of natural numbers from [Turner, 2009]. Consider the following rules for a new simple type Num. $\mathbf{Num}_1$ introduces the type, $\mathbf{Num}_2$ defines $\mathsf{0}$ as a constant, and $\mathbf{Num}_3$ introduces the successor function.

$$\mathbf{Num}_1 \ \Gamma \vdash Num : \mathcal{U} \qquad \mathbf{Num}_3 \ \frac{\Gamma \vdash a : Num}{\Gamma \vdash a^+ : Num}$$

$$\mathbf{Num}_2 \ \Gamma \vdash \mathsf{0} : Num$$

These rules will be familiar from our definition of $\mathbb{N}$ in §5.2.1. Indeed, using inductive types we are able obtain the standard induction principle of $\mathbb{N}$. First, we define a new inductive type $\mathbf{I} [Num, \phi]$, where:

$$\phi\left[Num, n\right] \triangleq n = \mathsf{0} \vee \exists x : Num \bullet n = x^{+}$$

which when substituted into $\mathbf{I}_3$ yields the following induction principle:

$$\frac{\Gamma, n : Num \vdash \theta\left[n\right]\ prop \quad \Gamma \vdash \forall x : Num \bullet x = \mathsf{0} \vee \exists y : Num \bullet \theta\left[y\right] \wedge x = y^{+} \implies \theta\left[x\right]}{\Gamma \vdash \forall x : \mathbf{I}\left[Num, \phi\right] \bullet \theta\left[x\right]}$$

This may then be simplified to obtain a rule in a more familiar form[63]:

$$\frac{\Gamma, n : Num \vdash \theta\left[n\right]\ prop \quad \Gamma \vdash \theta\left[\mathsf{0}\right] \quad \Gamma \vdash \forall x : Num \bullet \theta\left[x\right] \implies \theta\left[x^{+}\right]}{\Gamma \vdash \forall x : \mathbf{I}\left[Num, \phi\right] \bullet \theta\left[x\right]}$$

The inductive type we are to define for hiearchies of nested finite sets greatly resembles that of $\mathbf{I}\left[Num, \phi\right]$. In the above we facilitate reasoning over the structure of applications of the successor function to the constant $\mathsf{0}$. Our inductive type $SetOf$ will facilitate reasoning over the structure of applications of the *set* type constructor on some variable type B, defined as follows:

Definition 7 Let $B : \mathcal{U}$. We define the unary type constructor $SetOf\left(B\right) : \mathcal{U}$ as an inductive type such that $SetOf\left(B\right) \triangleq \mathbf{I}\left[\mathcal{U}, \phi\right]$ where $\phi\left[\mathcal{U}, B, T\right] \triangleq T = B \vee \exists X : \mathcal{U} \bullet T = set\left(X\right)$

Notice that in the example of the Num type we explicitly identified only two subterms of ϕ, i.e. $\phi\left[Num, n\right]$, and that the constant $\mathsf{0}$ was an implicit third subterm. We could have equally identified the occurrence of $\mathsf{0}$ explicitly with $\phi\left[Num, \mathsf{0}, n\right]$. In the case of $SetOf$, the argument $B : \mathcal{U}$ is not a constant so we must make its appearance as a subterm of ϕ explicit, i.e. $\phi\left[\mathcal{U}, B, T\right]$.

Let us now examine the induction principle for $SetOf\left(B\right)$. By substituting $\phi\left[\mathcal{U}, B, T\right]$ into rule $\mathbf{I}_3$, in the context of some $B : \mathcal{U}$, we obtain the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash B : \mathcal{U} \quad \Gamma, T : \mathcal{U} \vdash \theta\left[T\right]\ prop \\ \Gamma \vdash \forall X : \mathcal{U} \bullet \left(X = B \vee \left(\exists Y : \mathcal{U} \bullet \theta\left[Y\right] \wedge X = set\left(Y\right)\right)\right) \implies \theta\left[X\right]\end{array}}{\Gamma \vdash \forall X : SetOf\left(B\right) \bullet \theta\left[X\right]}$$

Again, we simplify the above to a more familiar form that is easier to work with. This yields the

[63] We could reduce the rule further to obtain a rule in the exact same form as we gave for $\mathbb{N}$, i.e.:

$$\frac{\Gamma, n : Num \vdash \theta\left[n\right]\ prop \quad \Gamma \vdash \theta\left[\mathsf{0}\right] \quad \Gamma, n : Num, \theta\left[n\right] \vdash \theta\left[n^{+}\right]}{\Gamma, n : \mathbf{I}\left[Num, \phi\right] \vdash \theta\left[n\right]}$$

but doing so is unnecessary.

following general induction principle $SetOf\,(B)_{ind}$ for some $B : \mathcal{U}$:

$$SetOf\,(B)_{ind}\ \frac{\begin{array}{c}\Gamma \vdash B : \mathcal{U} \quad \Gamma, T : \mathcal{U} \vdash \theta\,[T]\ \ prop \\ \Gamma \vdash \theta\,[B] \quad \Gamma \vdash \forall X : \mathcal{U} \bullet \theta\,[X] \implies \theta\,[set\,(X)]\end{array}}{\Gamma \vdash \forall X : SetOf\,(B) \bullet \theta\,[X]}$$

We obtain specialized instances of this induction principle when we apply the $SetOf$ type constructor to some specific type. For example, consider applying $SetOf$ to type $\mathbb{CLASS}$: we obtain the type $SetOf\,(\mathbb{CLASS})$ and its induction principle $SetOf\,(\mathbb{CLASS})_{ind}$:

$$SetOf\,(\mathbb{CLASS})_{ind}\ \frac{\begin{array}{c}\Gamma \vdash \mathbb{CLASS} : \mathcal{U} \quad \Gamma, T : \mathcal{U} \vdash \theta\,[T]\ \ prop \\ \Gamma \vdash \theta\,[\mathbb{CLASS}] \quad \Gamma \vdash \forall X : \mathcal{U} \bullet \theta\,[X] \implies \theta\,[set\,(X)]\end{array}}{\Gamma \vdash \forall X : SetOf\,(\mathbb{CLASS}) \bullet \theta\,[X]}$$

Let us present an example of using the general $SetOf$ induction principle ($SetOf\,(B)_{ind}$). Suppose we wish to show that for any $B : \mathcal{U}$ and $T : SetOf\,(B)$, T is either equal to B or to an application of the *set* type constructor on some other type in $SetOf\,(B)$:

Lemma 2 $B : \mathcal{U}, T : SetOf\,(B) \vdash T = B \vee \exists X : SetOf\,(B) \bullet T = set\,(X)$

Proof. By induction on $SetOf\,(B)$ using the induction principle $SetOf\,(B)_{ind}$. For brevity, let $\theta\,[B,T]$ be the proposition $T = B \vee \exists X : SetOf\,(B) \bullet T = set\,(X)$. We are required to prove that each of the following hold:

1. $B : \mathcal{U}$

   This holds immediately from the premises of the lemma.

2. $B : \mathcal{U}, T : \mathcal{U} \vdash \theta\,[B,T]\ \ prop$

   This holds, as shown in the following simple derivation[64]:

1)	$B : \mathcal{U}$	premise
2)	$T : \mathcal{U}$	premise
3)	$T = B\ prop$	$\mathbf{L}_{23}$ 1,2
4)	$X : SetOf\,(B)$	assumption
5)	$X : \mathcal{U}$	$\mathbf{I}_{basic}$ 4
6)	$set\,(X) : \mathcal{U}$	$\mathbf{S}_1$ 5
7)	$T = set\,(X)\ \ prop$	$\mathbf{L}_{23}$ 2,6
8)	$\exists X : SetOf\,(B) \bullet T = set\,(X)\ \ prop$	$\mathbf{L}_{26}$ 4,7
9)	$T = B \vee \exists X : SetOf\,(B) \bullet T = set\,(X)\ \ prop$	$\mathbf{L}_{12}$ 3,8

[64] $\mathbf{L}_{12}$, p.46; $\mathbf{L}_{23}$, p.46; $\mathbf{L}_{26}$, p.47; $\mathbf{S}_1$, p.54; $\mathbf{I}_{basic}$, p.92

3. $B : \mathcal{U} \vdash \theta\left[B, B\right]$

   This follows immediately from the lemma. By substituting B into our lemma we obtain:

$$B : \mathcal{U} \vdash B = B \vee \exists X : \mathit{SetOf}\left(B\right) \bullet B = \mathit{set}\left(X\right)$$

   where, by $\mathbf{L}_{24}$ (p.46), it is a tautology that:

$$B : \mathcal{U} \vdash B = B$$

4. $B : \mathcal{U} \vdash \forall X : \mathcal{U} \bullet \theta\left[B, X\right] \implies \theta\left[B, \mathit{set}\left(X\right)\right]$

   This holds by the following derivation, where we make the inductive assumption that $\theta\left[B, X\right]$ holds (step 3)[65]:

1)	$B : \mathcal{U}$	premise
2)	$X : \mathcal{U}$	assumption
3)	$X = B \vee \exists Y : \mathit{SetOf}\left(B\right) \bullet X = \mathit{set}\left(Y\right)$	inductive assumption
4)	$\mathit{set}\left(X\right) : \mathcal{U}$	$\mathbf{S}_1$ 2
5)	$\mathit{set}\left(X\right) = B\ \mathit{prop}$	$\mathbf{L}_{23}$ 4,1
6)	$X = B$	assumption
7)	$\exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	immediate where $Z = B$
8)	$\exists Y : \mathcal{U} \bullet X = \mathit{set}\left(Y\right)$	assumption
9)	$\exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	immediate where $Z = \mathit{set}\left(Y\right)$
10)	$\exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	$\mathbf{L}_{16}$ 3,6-7,8-9
11)	$\mathit{set}\left(X\right) = B \vee \exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	$\mathbf{L}_{13}$ 5,10
12)	$X = B \vee \exists Y : \mathit{SetOf}\left(B\right) \bullet X = \mathit{set}\left(Y\right) \implies \mathit{set}\left(X\right) = B \vee \exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	$\mathbf{L}_{17}$ 3,11
13)	$\forall X : \mathcal{U} \bullet X = B \vee \exists Y : \mathit{SetOf}\left(B\right) \bullet X = \mathit{set}\left(Y\right) \implies \mathit{set}\left(X\right) = B \vee \exists Z : \mathit{SetOf}\left(B\right) \bullet \mathit{set}\left(X\right) = \mathit{set}\left(Z\right)$	$\mathbf{L}_{30}$ 2,12

   Importantly, examine steps 6-7, which was immediate as from $X = B$ (step 6) it follows that $\mathit{set}\left(X\right) = \mathit{set}\left(B\right)$ and that B can be abstracted by existential quantification (step 7). Steps 8-9 follow the same logic, i.e. if $X = \mathit{set}\left(Y\right)$ then it follows that $\mathit{set}\left(X\right) = \mathit{set}\left(\mathit{set}\left(Y\right)\right)$, and that by abstracting $\mathit{set}\left(Y\right)$ with existential quantification we obtain the result at step 9.

Given the above, we conclude that:

$$B : \mathcal{U}, T : \mathit{SetOf}\left(B\right) \vdash T = B \vee \exists X : \mathit{SetOf}\left(B\right) \bullet T = \mathit{set}\left(X\right)$$

[65] $\mathbf{L}_{13}$ and $\mathbf{L}_{16}$, p.46; $\mathbf{L}_{17}$, p.46; $\mathbf{L}_{23}$, p.46; $\mathbf{L}_{30}$, p.48; $\mathbf{S}_1$, p.54

■

Given the above lemma we are able to derive the following rule, which tells us that if some $T : \mathit{SetOf}\,(B)$ is not equal to the type B then T must be equal to the application of the *set* type constructor on som other member of $\mathit{SetOf}\,(B)$:

$$\mathit{SetOf}\,(B)_{set}\,\frac{\Gamma \vdash B : \mathcal{U} \quad \Gamma \vdash T : \mathit{SetOf}\,(B) \quad \Gamma \vdash T \neq B}{\Gamma \vdash \exists X : \mathit{SetOf}\,(B) \bullet T = \mathit{set}\,(X)}$$

This concludes our introduction of the SetOf type constructor, with which we are now able to define a new function that identifies a method by a class (its scope) and its signature (its name) that extends naturally to sets thereof while preserving the desired clan structure. We call this the superimposition ($\otimes$) function[66], which we derive from the following functional relation:

$\otimes$

$$S : \mathit{SetOf}\,(\mathbb{SIGNATURE})\,,\ C : \mathit{SetOf}\,(\mathbb{CLASS})\,,\ M : \mathit{SetOf}\,(\mathbb{METHOD})$$
$$s : S,\ c : C,\ m : M$$

$$\begin{gathered}
\left(\begin{gathered} S = \mathbb{SIGNATURE} \wedge C = \mathbb{CLASS} \wedge M = \mathbb{METHOD} \implies \\ \mathit{SignatureOf}\,(s, m) \wedge \mathit{MethodMember}\,(c, m) \end{gathered}\right) \\
\wedge \\
\left(\begin{gathered} S = \mathbb{SIGNATURE} \wedge C \neq \mathbb{CLASS} \wedge M \neq \mathbb{METHOD} \implies \\ \exists Y : \mathit{SetOf}\,(\mathbb{CLASS}) \bullet \exists Z : \mathit{SetOf}\,(\mathbb{METHOD}) \bullet C = \mathit{set}\,(Y) \wedge M = \mathit{set}\,(Z) \wedge \\ (\forall y \in c \bullet \exists z \in m \bullet \otimes (S, Y, Z, s, y, z)) \wedge (\forall z \in m \bullet \exists y \in c \bullet \otimes (S, Y, Z, s, y, z)) \end{gathered}\right) \\
\wedge \\
\left(\begin{gathered} S \neq \mathbb{SIGNATURE} \wedge M \neq \mathbb{METHOD} \implies \\ \exists X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \bullet \exists Z : \mathit{SetOf}\,(\mathbb{METHOD}) \bullet S = \mathit{set}\,(X) \wedge M = \mathit{set}\,(Z) \wedge \\ (\forall x \in s \bullet \exists z \in m \bullet \otimes (X, C, Z, x, c, z)) \wedge (\forall z \in m \bullet \exists x \in s \bullet \otimes (X, C, Z, x, c, z)) \end{gathered}\right) \\
\wedge \\
(S = \mathbb{SIGNATURE} \wedge C = \mathbb{CLASS} \wedge M \neq \mathbb{METHOD} \implies \Omega) \wedge \\
(S = \mathbb{SIGNATURE} \wedge C \neq \mathbb{CLASS} \wedge M = \mathbb{METHOD} \implies \Omega) \wedge \\
(S \neq \mathbb{SIGNATURE} \wedge M = \mathbb{METHOD} \implies \Omega)
\end{gathered}$$

[66] Terminology borrowed from LePUS3, the language of Codecharts (see LePUS3 Definition V, Appendix B)

This is the most complex specification of our theory, and it is therefore appropriate to take the time to examine it in more detail. We will show that this specification type checks, i.e. that it is always a proposition under appropriately typed arguments. As the structure of this specification exemplifies several less complex specifications that we discuss later (§7.6.1, §7.6.2, and §7.6.3), similar proofs of these later examples follow the same pattern as presented below. We conclude that little would be added by discussing each specification in such detail.

To show that the superimposition ($\otimes$) relation type checks we must prove that it is always a proposition when given any possible well-typed arguments, that is:

Proposition 2 $$\frac{\begin{array}{c} S : SetOf\,(\mathbb{SIGNATURE}) \quad s : S \\ C : SetOf\,(\mathbb{CLASS}) \quad c : C \\ M : SetOf\,(\mathbb{METHOD}) \quad m : M \end{array}}{\otimes\,(S, C, M, s, c, m)\ \ prop}$$

Proof. By simultaneous induction on the number of applications of the *set* type constructor using the induction principles of the types $SetOf\,(\mathbb{SIGNATURE})$, $SetOf\,(\mathbb{CLASS})$, and $SetOf\,(\mathbb{METHOD})$.

Firstly, we observe the structure of $\otimes$ is of the form:

$$(\phi_1 \implies \varphi_1) \wedge \ldots \wedge (\phi_n \implies \varphi_n)$$

By the formation rule for conjunction ($\mathbf{L}_8$, p.46), $\otimes\,(S, C, M, s, c, m)\ \ prop$ holds if every constituent $\phi_k \implies \varphi_k$ *prop* holds. Observe that each ϕ_k $(1 \leq k \leq n)$ is a series of (in)equality propositions joined by conjunction and that:

1. it is always the case that ϕ_k *prop* holds, and
2. for any given arguments exactly one ϕ_k holds. That is, they are mutually exclusive and they cover all cases.

Additionally, by the formation rule for implication ($\mathbf{L}_{17}$, p.46), we know $\phi_k \implies \varphi_k$ *prop* when

1. ϕ_k holds and we prove that φ_k *prop* holds, or
2. ϕ_k does not hold, in which case we are not required to prove φ_k *prop*.

Therefore, to show $\otimes\,(S, C, M, s, c, m)\ \ prop$ holds we are required to explicitly show that φ_k *prop* holds where ϕ_k is true, and we may implicitly appeal to these observations to know that all other $\phi_i \implies \varphi_i$ *prop* hold $(1 \leq i \leq n$ and $i \neq k)$.

Given these observations, we continue our proof with the base case as follows:

Base Case: Here we are required to prove that the following holds:

$$\frac{\begin{array}{c}\mathbb{SIGNATURE} : SetOf\left(\mathbb{SIGNATURE}\right) \quad x : \mathbb{SIGNATURE} \\ \mathbb{CLASS} : SetOf\left(\mathbb{CLASS}\right) \quad y : \mathbb{CLASS} \\ \mathbb{METHOD} : SetOf\left(\mathbb{METHOD}\right) \quad z : \mathbb{METHOD}\end{array}}{\otimes\left(\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right) \ prop}$$

By our earlier discussion, the proposition of $\otimes\left(\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right)$ that we must prove to be a *prop* in this context is:

$$SignatureOf\left(x, z\right) \wedge MethodMember\left(z, y\right) \ prop$$

which is easily shown to hold in the following derivation[67]:

1)	$x : \mathbb{SIGNATURE}$	premise
2)	$y : \mathbb{CLASS}$	premise
3)	$z : \mathbb{METHOD}$	premise
4)	$SignatureOf\left(x, z\right) \ prop$	$\mathbf{TC}_{10}$ 1,3
5)	$MethodMember\left(z, y\right) \ prop$	$\mathbf{TC}_{8}$ 2,3
6)	$SignatureOf\left(x, z\right) \wedge MethodMember\left(z, y\right) \ prop$	$\mathbf{L}_{8}$ 4,5

Inductive Step: We begin the inductive step by making the inductive assumption that the following holds:

$$\frac{\begin{array}{c}X : SetOf\left(\mathbb{SIGNATURE}\right) \quad x : X \\ Y : SetOf\left(\mathbb{CLASS}\right) \quad y : Y \\ Z : SetOf\left(\mathbb{METHOD}\right) \quad z : Z\end{array}}{\otimes\left(X, Y, Z, x, y, z\right) \ prop}$$

We refer to this assumption in the remaining derivations as *ind.* Given this we are required to prove each of the following:

1. We are required to prove:

$$\frac{\begin{array}{c}set\left(X\right) : SetOf\left(\mathbb{SIGNATURE}\right) \quad u : set\left(X\right) \\ Y : SetOf\left(\mathbb{CLASS}\right) \quad y : Y \\ Z : SetOf\left(\mathbb{METHOD}\right) \quad z : Z\end{array}}{\otimes\left(set\left(X\right), Y, Z, u, y, z\right) \ prop}$$

[67] $\mathbf{L}_{8}$, p.46; $\mathbf{TC}_{8}$, p.85; $\mathbf{TC}_{10}$, p.86

By our earlier discussion, we must prove Ω *prop* in this context, which is immediate by the formation rule of contradiction (**L**$_4$, p.45).

2. We are required to prove:

$$\frac{\begin{array}{c} X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad x : X \\ \mathit{set}\,(Y) : \mathit{SetOf}\,(\mathbb{CLASS}) \quad v : \mathit{set}\,(Y) \\ Z : \mathit{SetOf}\,(\mathbb{METHOD}) \quad z : Z \end{array}}{\otimes\,(X, \mathit{set}\,(Y)\,, Z, x, v, z)\ \ \mathit{prop}}$$

By our earlier discussion, we must prove Ω *prop* in this context, which is immediate by the formation rule of contradiction (**L**$_4$, p.45).

3. We are required to prove:

$$\frac{\begin{array}{c} X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad x : X \\ Y : \mathit{SetOf}\,(\mathbb{CLASS}) \quad y : Y \\ \mathit{set}\,(Z) : \mathit{SetOf}\,(\mathbb{METHOD}) \quad w : \mathit{set}\,(Z) \end{array}}{\otimes\,(X, Y, \mathit{set}\,(Z)\,, x, y, w)\ \ \mathit{prop}}$$

By our earlier discussion, we must prove Ω *prop* in this context, which is immediate by the formation rule of contradiction (**L**$_4$, p.45).

4. We are required to prove:

$$\frac{\begin{array}{c} \mathit{set}\,(X) : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad u : \mathit{set}\,(X) \\ \mathit{set}\,(Y) : \mathit{SetOf}\,(\mathbb{CLASS}) \quad v : \mathit{set}\,(Y) \\ Z : \mathit{SetOf}\,(\mathbb{METHOD}) \quad z : Z \end{array}}{\otimes\,(\mathit{set}\,(X)\,, \mathit{set}\,(Y)\,, Z, u, v, z)\ \ \mathit{prop}}$$

By our earlier discussion, we must prove Ω *prop* in this context, which is immediate by the formation rule of contradiction (**L**$_4$, p.45).

5. We are required to prove:

$$\frac{\begin{array}{c} X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad x : X \\ \mathit{set}\,(Y) : \mathit{SetOf}\,(\mathbb{CLASS}) \quad v : \mathit{set}\,(Y) \\ \mathit{set}\,(Z) : \mathit{SetOf}\,(\mathbb{METHOD}) \quad w : \mathit{set}\,(Z) \end{array}}{\otimes\,(X, \mathit{set}\,(Y)\,, \mathit{set}\,(Z)\,, x, v, w)\ \ \mathit{prop}}$$

By our earlier discussion, we must prove the following in this context:

$$\exists Y' : SetOf\left(\mathbb{CLASS}\right) \bullet \exists Z' : SetOf\left(\mathbb{METHOD}\right) \bullet set\left(Y\right) = set\left(Y'\right) \wedge set\left(Z\right) = set\left(Z'\right) \wedge$$
$$\left(\forall y \in v \bullet \exists z \in w \bullet \otimes\left(X, Y', Z', x, y, z\right)\right) \wedge \left(\forall z \in w \bullet \exists y \in v \bullet \otimes\left(X, Y', Z', x, y, z\right)\right)\ \ prop$$

and follows from the derivation below[68]:

1)	$X : SetOf\left(\mathbb{SIGNATURE}\right)$	premise
2)	$set\left(Y\right) : SetOf\left(\mathbb{CLASS}\right)$	premise
3)	$set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right)$	premise
4)	$x : X$	premise
5)	$v : set\left(Y\right)$	premise
6)	$w : set\left(Z\right)$	premise
7)	$\mid\ \underline{y : Y}$	assumption
8)	$\mid\ \mid\ \underline{z : Z}$	assumption
9)	$\mid\ \mid\ \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	ind
10)	$\mid\ \exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	**S**$_{13}$, 6,8,9
11)	$\forall y \in v \bullet \exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	**S**$_{16}$, 5,7,10
12)	$\mid\ \underline{z : Z}$	assumption
13)	$\mid\ \mid\ \underline{y : Y}$	assumption
14)	$\mid\ \mid\ \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	ind
15)	$\mid\ \exists y \in v \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	**S**$_{13}$, 5,13,14
16)	$\forall z \in w \bullet \exists y \in v \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	**S**$_{16}$, 6,12,15
17)	$set\left(Y\right) = set\left(Y\right)\ \ prop$	**L**$_{23}$, 2
18)	$set\left(Z\right) = set\left(Z\right)\ \ prop$	**L**$_{23}$, 3
19)	$set\left(Y\right) = set\left(Y\right) \wedge set\left(Z\right) = set\left(Z\right) \wedge$ $\left(\forall y \in v \bullet \exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\right) \wedge$ $\left(\forall z \in w \bullet \exists y \in v \bullet \otimes\left(X, Y, Z, x, y, z\right)\right)\ \ prop$	**L**$_8$, 11,16,17,18
20)	$\exists Y' : SetOf\left(\mathbb{CLASS}\right) \bullet \exists Z' : SetOf\left(\mathbb{METHOD}\right) \bullet$ $set\left(Y\right) = set\left(Y'\right) \wedge set\left(Z\right) = set\left(Z'\right) \wedge$ $\left(\forall y \in v \bullet \exists z \in w \bullet \otimes\left(X, Y', Z', x, y, z\right)\right) \wedge$ $\left(\forall z \in w \bullet \exists y \in v \bullet \otimes\left(X, Y', Z', x, y, z\right)\right)\ \ prop$	immediate where $Y = Y'$ and $Z = Z'$

6. We are required to prove:

$$\frac{\begin{array}{c} set\left(X\right) : SetOf\left(\mathbb{SIGNATURE}\right) \quad u : set\left(X\right) \\ Y : SetOf\left(\mathbb{CLASS}\right) \quad y : Y \\ set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right) \quad w : set\left(Z\right) \end{array}}{\otimes\left(set\left(X\right), Y, set\left(Z\right), u, y, w\right)\ \ prop}$$

[68] **L**$_8$, p.46; **L**$_{23}$, p.46; **S**$_{13}$, p.56; **S**$_{16}$, p.56

By our earlier discussion, we must prove the following in this context:

$$\exists X' : SetOf\left(\mathbb{CLASS}\right) \bullet \exists Z' : SetOf\left(\mathbb{METHOD}\right) \bullet set\left(X\right) = set\left(X'\right) \wedge set\left(Z\right) = set\left(Z'\right) \wedge$$
$$\left(\forall x \in u \bullet \exists z \in w \bullet \otimes\left(X', Y, Z', x, y, z\right)\right) \wedge \left(\forall z \in w \bullet \exists x \in u \bullet \otimes\left(X', Y, Z', x, y, z\right)\right)\ \ prop$$

and follows from the derivation below[69]:

1)	$set\left(X\right) : SetOf\left(\mathbb{SIGNATURE}\right)$	premise
2)	$Y : SetOf\left(\mathbb{CLASS}\right)$	premise
3)	$set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right)$	premise
4)	$u : set\left(X\right)$	premise
5)	$y : Y$	premise
6)	$w : set\left(Z\right)$	premise
7)	$x : X$	assumption
8)	$z : Z$	assumption
9)	$\otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	ind
10)	$\exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	$\mathbf{S}_{13}$, 6,8,9
11)	$\forall x \in u \bullet \exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	$\mathbf{S}_{16}$, 5,7,10
12)	$z : Z$	assumption
13)	$x : X$	assumption
14)	$\otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	ind
15)	$\exists x \in u \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	$\mathbf{S}_{13}$, 5,13,14
16)	$\forall z \in w \bullet \exists x \in u \bullet \otimes\left(X, Y, Z, x, y, z\right)\ \ prop$	$\mathbf{S}_{16}$, 6,12,15
17)	$set\left(X\right) = set\left(X\right)\ \ prop$	$\mathbf{L}_{23}$, 1
18)	$set\left(Z\right) = set\left(Z\right)\ \ prop$	$\mathbf{L}_{23}$, 3
19)	$set\left(X\right) = set\left(X\right) \wedge set\left(Z\right) = set\left(Z\right) \wedge$ $\left(\forall x \in u \bullet \exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)\right) \wedge$ $\left(\forall z \in w \bullet \exists x \in u \bullet \otimes\left(X, Y, Z, x, y, z\right)\right)\ \ prop$	$\mathbf{L}_{8}$, 11,16,17,18
20)	$\exists X' : SetOf\left(\mathbb{CLASS}\right) \bullet \exists Z' : SetOf\left(\mathbb{METHOD}\right) \bullet$ $set\left(X\right) = set\left(X'\right) \wedge set\left(Z\right) = set\left(Z'\right) \wedge$ $\left(\forall x \in u \bullet \exists z \in w \bullet \otimes\left(X', Y, Z', x, y, z\right)\right) \wedge$ $\left(\forall z \in w \bullet \exists x \in u \bullet \otimes\left(X', Y, Z', x, y, z\right)\right)\ \ prop$	immediate where $X = X'$ and $Z = Z'$

7. We are required to prove:

$$\frac{\begin{array}{cc} set\left(X\right) : SetOf\left(\mathbb{SIGNATURE}\right) & u : set\left(X\right) \\ set\left(Y\right) : SetOf\left(\mathbb{CLASS}\right) & v : set\left(Y\right) \\ set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right) & w : set\left(Z\right) \end{array}}{\otimes\left(set\left(X\right), set\left(Y\right), set\left(Z\right), u, v, w\right)\ \ prop}$$

[69] $\mathbf{L}_{8}$, p.46; $\mathbf{L}_{23}$, p.46; $\mathbf{S}_{13}$, p.56; $\mathbf{S}_{16}$, p.56

By our earlier discussion, we must prove the following in this context:

$$\exists X' : \mathit{SetOf}\,(\mathbb{CLASS}) \bullet \exists Z' : \mathit{SetOf}\,(\mathbb{METHOD}) \bullet \mathit{set}\,(X) = \mathit{set}\,(X') \wedge \mathit{set}\,(Z) = \mathit{set}\,(Z') \wedge$$
$$(\forall x \in u \bullet \exists z \in w \bullet \otimes (X', Y, Z', x, y, z)) \wedge (\forall z \in w \bullet \exists x \in u \bullet \otimes (X', Y, Z', x, y, z))\ \ \mathit{prop}$$

which holds by an almost identical derivation as presented in point six, differing only in premises 2 and 5 which in this case are $\mathit{set}\,(Y) : \mathit{SetOf}\,(\mathbb{CLASS})$ and $v : \mathit{set}\,(Y)$ respectively. As these premises do not affect the derivation, and the same result is sought after, we refer the reader to the previous derivation.

Given the steps presented above we conclude that:

$$\frac{\begin{array}{c} S : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad s : S \\ C : \mathit{SetOf}\,(\mathbb{CLASS}) \quad c : C \\ M : \mathit{SetOf}\,(\mathbb{METHOD}) \quad m : M \end{array}}{\otimes (S, C, M, s, c, m)\ \ \mathit{prop}}$$

■

The above proves that the superimposition ($\otimes$) relation will always form a *prop* when provided appropriate arguments.

We are also required to prove our statement that the superimposition ($\otimes$) relation is functional (p.96). To accomplish this we are required to prove that the following holds (see p.62):

$$\otimes : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \times \mathit{SetOf}\,(\mathbb{CLASS}) \times \mathit{SetOf}\,(\mathbb{METHOD}) \times S \times C \longmapsto M$$

That is, for any terms $S : \mathit{SetOf}\,(\mathbb{SIGNATURE})$, $C : \mathit{SetOf}\,(\mathbb{CLASS})$, $M : \mathit{SetOf}\,(\mathbb{METHOD})$, $s : S$, and $c : C$ that are in the domain of $\otimes$ we can show that there exists exactly one $m : M$ such that $\otimes (S, C, M, s, c, m)$ holds. We formalize this in the following proposition and prove it below:

Proposition 3 $$\frac{\begin{array}{c} S : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad s : S \\ C : \mathit{SetOf}\,(\mathbb{CLASS}) \quad c : C \\ M : \mathit{SetOf}\,(\mathbb{METHOD}) \\ \mathit{Dom}\,(\otimes, (S, C, M, s, c)) \end{array}}{\exists! m : M \bullet \otimes (S, C, M, s, c, m)}$$

Proof. By simultaneous induction on the number of applications of the *set* type constructor using the induction principles of the types $\mathit{SetOf}\,(\mathbb{SIGNATURE})$, $\mathit{SetOf}\,(\mathbb{CLASS})$, and $\mathit{SetOf}\,(\mathbb{METHOD})$.

Firstly, we observe the structure of $\otimes$ is of the form:

$$(\phi_1 \implies \varphi_1) \wedge \ldots \wedge (\phi_n \implies \varphi_n)$$

Observe that each ϕ_k $(1 \leq k \leq n)$ is a series of (in)equality propositions joined by conjunction such that for any given arguments exactly one ϕ_k holds. That is, they are mutually exclusive and they cover all cases. Therefore, to show $\exists ! m : M \bullet \otimes (S, C, M, s, c, m)$ holds we are required to explicitly show that $\exists ! m : M \bullet \varphi_k [S, C, M, s, c, m]$ where ϕ_k is true, and we may implicitly appeal to these observations and the rules of implication to know that all other $\phi_i \implies \varphi_i$ hold $(1 \leq i \leq n$ and $i \neq k)$, but as they do not effect the value of m they automatically preserve functionality.

Given these observations, we continue our proof with the base case as follows:

Base Case: Here we are required to prove that the following holds:

$$\frac{\begin{array}{c} \mathbb{SIGNATURE} : SetOf\left(\mathbb{SIGNATURE}\right) \quad x : \mathbb{SIGNATURE} \\ \mathbb{CLASS} : SetOf\left(\mathbb{CLASS}\right) \quad y : \mathbb{CLASS} \\ \mathbb{METHOD} : SetOf\left(\mathbb{METHOD}\right) \\ Dom\left(\otimes, \left(\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, s, c\right)\right) \end{array}}{\exists ! z : \mathbb{METHOD} \bullet \otimes \left(\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right)}$$

Firstly, by our earlier discussion, the proposition of $\otimes\left(\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right)$ that we must prove to be functional in this context is:

$$SignatureOf\,(x, z) \wedge MethodMember\,(z, y)$$

Let this proposition be referred to as θ, meaning that we are required to prove:

$$\exists ! z : \mathbb{METHOD} \bullet \theta\left[\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right]$$

which is a *prop* by virtue of Proposition 2 and the formation rule of uniqueness quantification ($\mathbf{L}_{32}$, p.48). The same can be said for the propositions we are required to prove during the inductive step. To prove this holds we must first prove that:

$$\frac{z : \mathbb{METHOD} \quad \theta\left[\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right]}{\forall z' : \mathbb{METHOD} \bullet \theta\left[\mathbb{SIGNATURE}, \mathbb{CLASS}, \mathbb{METHOD}, x, y, z\right] \implies z' = z}$$

which is shown to hold in the following derivation[70]:

1)	$\mathbb{SIGNATURE}: \mathit{SetOf}\,(\mathbb{SIGNATURE})$	premise
2)	$\mathbb{CLASS}: \mathit{SetOf}\,(\mathbb{CLASS})$	premise
3)	$\mathbb{METHOD}: \mathit{SetOf}\,(\mathbb{METHOD})$	premise
4)	$x : \mathbb{SIGNATURE}$	premise
5)	$y : \mathbb{CLASS}$	premise
6)	$\mid\ z : \mathbb{METHOD}$	assumption
7)	$\mid\ \mathit{SignatureOf}\,(x, z) \wedge \mathit{MethodMember}\,(z, y)$	assumption
8)	$\mid\ \mid\ z' : \mathbb{METHOD}$	assumption
9)	$\mid\ \mid\ \mid\ \mathit{SignatureOf}\,(x, z') \wedge \mathit{MethodMember}\,(z', y)$	assumption
10)	$\mid\ \mid\ \mid\ z' = z$	$\mathbf{TC}_{13}$ 7,9
11)	$\mid\ \mid\ \mathit{SignatureOf}\,(x, z') \wedge \mathit{MethodMember}\,(z', y) \implies z' = z$	$\mathbf{L}_{18}$ 9,10
	$\mid\ \forall z' : \mathbb{METHOD} \bullet \mathit{SignatureOf}\,(x, z') \wedge \mathit{MethodMember}\,(z', y) \implies z' = z$	$\mathbf{L}_{30}$ 8,11
12)	$\exists! z : \mathbb{METHOD} \bullet \mathit{SignatureOf}\,(x, z) \wedge \mathit{MethodMember}\,(z, y)$	$\mathbf{L}_{33}$ 6,7,11

The above derivation follows a simple pattern: We assume that our θ holds for two different results $z, z' : \mathit{set}\,(Z)$, and then by showing that it is always the case that $w = w'$ we may conclude there exists a unique w where θ holds.

Inductive Step: We begin the inductive step by making the inductive assumption that the following holds:

$$\frac{\begin{array}{c} X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad x : X \\ Y : \mathit{SetOf}\,(\mathbb{CLASS}) \quad y : Y \\ Z : \mathit{SetOf}\,(\mathbb{METHOD}) \\ \mathit{Dom}\,(\otimes, (X, Y, Z, x, y)) \end{array}}{\exists! z : Z \bullet \otimes (X, Y, Z, x, y, z)}$$

We refer to this assumption in the remaining derivations as *ind.* Given this:

1. We are required to prove:

$$\frac{\begin{array}{c} \mathit{set}\,(X) : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad u : \mathit{set}\,(X) \\ Y : \mathit{SetOf}\,(\mathbb{CLASS}) \quad y : Y \\ Z : \mathit{SetOf}\,(\mathbb{METHOD}) \\ \mathit{Dom}\,(\otimes, (\mathit{set}\,(X), Y, Z, u, y)) \end{array}}{\exists! z : Z \bullet \otimes (\mathit{set}\,(X), Y, Z, u, y, z)}$$

   which, by the definition of $\otimes$, the proposition $\mathit{Dom}\,(\otimes, (\mathit{set}\,(X), Y, Z, u, y))$ cannot hold as it results in a contradiction; therefore this case holds.

[70] $\mathbf{L}_{18}$, p.46; $\mathbf{L}_{30}$, p.48; $\mathbf{L}_{33}$, p.48; $\mathbf{TC}_{13}$, p.87

2. We are required to prove:

$$\frac{\begin{array}{c} X : SetOf\left(\mathbb{SIGNATURE}\right) \quad x : X \\ set\left(Y\right) : SetOf\left(\mathbb{CLASS}\right) \quad v : set\left(Y\right) \\ Z : SetOf\left(\mathbb{METHOD}\right) \\ Dom\left(\otimes, \left(X, set\left(Y\right), Z, x, v\right)\right) \end{array}}{\exists! z : Z \bullet \otimes\left(X, set\left(Y\right), Z, x, v, z\right)}$$

which, by the definition of $\otimes$, the proposition $Dom\left(\otimes, \left(X, set\left(Y\right), Z, x, v\right)\right)$ cannot hold as it results in a contradiction; therefore this case holds.

3. We are required to prove:

$$\frac{\begin{array}{c} X : SetOf\left(\mathbb{SIGNATURE}\right) \quad x : X \\ Y : SetOf\left(\mathbb{CLASS}\right) \quad y : Y \\ set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right) \\ Dom\left(\otimes, \left(X, Y, set\left(Z\right), x, y\right)\right) \end{array}}{\exists! w : set\left(Z\right) \bullet \otimes\left(X, Y, set\left(Z\right), x, y, w\right)}$$

which, by the definition of $\otimes$, the proposition $Dom\left(\otimes, \left(X, Y, set\left(Z\right), x, y\right)\right)$ cannot hold as it results in a contradiction; therefore this case holds.

4. We are required to prove:

$$\frac{\begin{array}{c} set\left(X\right) : SetOf\left(\mathbb{SIGNATURE}\right) \quad u : set\left(X\right) \\ set\left(Y\right) : SetOf\left(\mathbb{CLASS}\right) \quad v : set\left(Y\right) \\ Z : SetOf\left(\mathbb{METHOD}\right) \\ Dom\left(\otimes, \left(set\left(X\right), set\left(Y\right), Z, u, v\right)\right) \end{array}}{\exists! z : Z \bullet \otimes\left(set\left(X\right), set\left(Y\right), Z, u, v, z\right)}$$

which, by the definition of $\otimes$, the proposition $Dom\left(\otimes, \left(set\left(X\right), set\left(Y\right), Z, u, v\right)\right)$ cannot hold as it results in a contradiction; therefore this case holds.

5. We are required to prove:

$$\frac{\begin{array}{c} X : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad x : X \\ \mathit{set}\,(Y) : \mathit{SetOf}\,(\mathbb{CLASS}) \quad v : \mathit{set}\,(Y) \\ \mathit{set}\,(Z) : \mathit{SetOf}\,(\mathbb{METHOD}) \\ \mathit{Dom}\,(\otimes, (X, \mathit{set}\,(Y), \mathit{set}\,(Z), x, v)) \end{array}}{\exists! w : \mathit{set}\,(Z) \bullet \otimes (X, \mathit{set}\,(Y), \mathit{set}\,(Z), x, v, w)}$$

Firstly, by referring to our earlier discussion, let θ be the proposition that is to be proved in this context: $\otimes (X, \mathit{set}\,(Y), \mathit{set}\,(Z), x, v, w)$[71]. Therefore, we are required to prove:

$$\ldots \vdash \exists! w : \mathit{set}\,(Z) \bullet \theta\,[X, \mathit{set}\,(Y), \mathit{set}\,(Z), x, v, w]$$

which we show by the derivation on the next page[72].

[71] Inspection of $\otimes$ tells us this is the case in question:

$$\begin{array}{c} \exists Y' : \mathit{SetOf}\,(\mathbb{CLASS}) \bullet \exists Z' : \mathit{SetOf}\,(\mathbb{METHOD}) \bullet \mathit{set}\,(Y) = \mathit{set}\,(Y') \wedge \mathit{set}\,(Z) = \mathit{set}\,(Z') \wedge \\ (\forall y \in v \bullet \exists z \in w \bullet \otimes (X, Y', Z', x, y, z)) \wedge (\forall z \in w \bullet \exists y \in v \bullet \otimes (X, Y', Z', x, y, z)) \end{array}$$

[72] $\mathbf{L}_{18}$, p.46; $\mathbf{L}_{30}$, p.48; $\mathbf{L}_{33}$, p.48; $\mathbf{S}_{17}$, p.56

1)	$X : SetOf\left(\mathbb{SIGNATURE}\right)$	premise
2)	$set\left(Y\right) : SetOf\left(\mathbb{CLASS}\right)$	premise
3)	$set\left(Z\right) : SetOf\left(\mathbb{METHOD}\right)$	premise
4)	$x : X$	premise
5)	$v : set\left(Y\right)$	premise
6)	$w : set\left(Z\right)$	assumption
7)	$\theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w\right]$	assumption
8)	$w' : set\left(Z\right)$	assumption
9)	$\theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w'\right]$	assumption
10)	$z \in w$	assumption
11)	$y \in v$	assumption
12)	$\otimes\left(X, Y, Z, x, y, z\right)$	assumption
13)	$\exists z' \in w' \bullet \otimes\left(X, Y, Z, x, y, z'\right)$	immediate from 9
14)	$z \in w'$	ind 13
15)	$\forall z \in w \bullet z \in w'$	$\mathbf{S}_{17}$ 6,10, 14
16)	$w \subseteq w'$	$\subseteq$ 15
17)	$z' \in w'$	assumption
18)	$y \in v$	assumption
19)	$\otimes\left(X, Y, Z, x, y, z'\right)$	assumption
20)	$\exists z \in w \bullet \otimes\left(X, Y, Z, x, y, z\right)$	immediate from 7
21)	$z' \in w$	ind 20
22)	$\forall z' \in w' \bullet z \in w$	$\mathbf{S}_{17}$ 8,17, 21
23)	$w' \subseteq w$	$\subseteq$ 22
24)	$w' = w$	$=_{set\left(SetOf\left(\mathbb{METHOD}\right)\right)}$ 16,23
25)	$\theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w'\right] \implies w' = w$	$\mathbf{L}_{18}$ 9,24
26)	$\forall w' : set\left(Z\right) \bullet \theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w'\right] \implies w' = w$	$\mathbf{L}_{30}$ 8,25
27)	$\exists! w : set\left(Z\right) \bullet \theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w\right]$	$\mathbf{L}_{33}$ 6,7,36

The above derivation follows the same pattern as we presented for the base case, where steps 10–23 deserve some additional explanation. These steps can be divided into two cases, one that proves $w \subseteq w'$ (steps 10–16) and its mirror image that proves $w' \subseteq w$ (steps 17–23). We discuss the first in more detail, the second follows the exact same method. Firstly, assume that there is a $y \in v$ and a $z \in w$ such that $\otimes\left(X, Y, Z, x, y, z\right)$ holds (steps 10–12). As $y \in v$ and $\theta\left[X, set\left(Y\right), set\left(Z\right), x, v, w'\right]$ (step 9), then there must be some $z' \in w'$ such that $\otimes\left(X, Y, Z, x, y, z'\right)$ holds (step 13). By our inductive assumption (*ind*) we know $\exists! z : Z \bullet \otimes\left(X, Y, Z, x, y, z\right)$, so z and z' must be equal, hence $z \in w'$ (step 14). By introduction of the universal quantifier (step 15), and rewriting with $\subseteq$, we obtain $w \subseteq w'$ (step 16). Finally, as steps 17–23 show that $w' \subseteq w$, we conclude that $w = w'$ and finish the proof in the expected way.

6. We are required to prove:

$$\frac{\begin{array}{c} set\left(X\right):SetOf\left(\mathbb{SIGNATURE}\right) \quad u:set\left(X\right) \\ Y:SetOf\left(\mathbb{CLASS}\right) \quad y:Y \\ set\left(Z\right):SetOf\left(\mathbb{METHOD}\right) \\ Dom\left(\otimes,\left(set\left(X\right),Y,set\left(Z\right),u,y\right)\right) \end{array}}{\exists!w:set\left(Z\right)\bullet\otimes\left(set\left(X\right),Y,set\left(Z\right),u,y,w\right)}$$

Firstly, by referring to our earlier discussion, let θ be the proposition that is to be proved in this context: $\otimes\left(X,set\left(Y\right),set\left(Z\right),x,v,w\right)$[73]. By the introduction Therefore we are required to prove:

$$\ldots\vdash\exists!w:set\left(Z\right)\bullet\theta\left[set\left(X\right),Y,set\left(Z\right),u,y,w\right]$$

which we show by the derivation on the next page[74].

[73] Inspection of $\otimes$ tells us this is the case in question:

$$\exists X':SetOf\left(\mathbb{SIGNATURE}\right)\bullet\exists Z':SetOf\left(\mathbb{METHOD}\right)\bullet set\left(X\right)=set\left(X'\right)\wedge set\left(Z\right)=set\left(Z'\right)\wedge$$
$$\left(\forall x\in u\bullet\exists z\in w\bullet\otimes\left(X',Y,Z',x,y,z\right)\right)\wedge\left(\forall z\in w\bullet\exists x\in u\bullet\otimes\left(X',Y,Z',x,y,z\right)\right)$$

[74] $\mathbf{L}_{18}$, p.46; $\mathbf{L}_{30}$, p.48; $\mathbf{L}_{33}$, p.48; $\mathbf{S}_{17}$, p.56

1)	$set\,(X) : SetOf\,(\mathbb{SIGNATURE})$	premise
2)	$Y : SetOf\,(\mathbb{CLASS})$	premise
3)	$set\,(Z) : SetOf\,(\mathbb{METHOD})$	premise
4)	$u : set\,(X)$	premise
5)	$y : Y$	premise
6)	$w : set\,(X)$	assumption
7)	$\theta\,[set\,(X)\,, Y, set\,(Z)\,, u, y, w]$	assumption
8)	$w' : set\,(Z)$	assumption
9)	$\theta\,[set\,(X)\,, Y, set\,(Z)\,, u, y, w']$	assumption
10)	$z \in w$	assumption
11)	$x \in u$	assumption
12)	$\otimes\,(X, Y, Z, x, y, z)$	assumption
13)	$\exists z' \in w' \bullet \otimes\,(X, Y, Z, x, y, z')$	immediate from 9
14)	$z \in w'$	ind 13
15)	$\forall z \in w \bullet z \in w'$	$\mathbf{S}_{17}$ 6,10, 14
16)	$w \subseteq w'$	$\subseteq$ 15
17)	$z' \in w'$	assumption
18)	$x \in u$	assumption
19)	$\otimes\,(X, Y, Z, x, y, z')$	assumption
20)	$\exists z \in w \bullet \otimes\,(X, Y, Z, x, y, z)$	immediate from 7
21)	$z' \in w$	ind 20
22)	$\forall z' \in w' \bullet z \in w$	$\mathbf{S}_{17}$ 8,17, 21
23)	$w' \subseteq w$	$\subseteq$ 22
24)	$w' = w$	$=_{set(SetOf(\mathbb{METHOD}))}$ 16,23
25)	$\theta\,[set\,(X)\,, Y, set\,(Z)\,, u, y, w'] \implies w' = w$	$\mathbf{L}_{18}$ 9,24
26)	$\forall w' : set\,(Z) \bullet \theta\,[set\,(X)\,, Y, set\,(Z)\,, u, y, w'] \implies w' = w$	$\mathbf{L}_{30}$ 8,25
27)	$\exists! w : set\,(Z) \bullet \theta\,[set\,(X)\,, Y, set\,(Z)\,, u, y, w]$	$\mathbf{L}_{33}$ 6,7,36

7. We are required to prove:

$$\frac{\begin{array}{c} set\,(X) : SetOf\,(\mathbb{SIGNATURE}) \qquad u : set\,(X) \\ set\,(Y) : SetOf\,(\mathbb{CLASS}) \qquad v : set\,(Y) \\ set\,(Z) : SetOf\,(\mathbb{METHOD}) \\ Dom\,(\otimes, (set\,(X)\,, set\,(Y)\,, set\,(Z)\,, u, v)) \end{array}}{\exists! w : set\,(Z) \bullet \otimes\,(set\,(X)\,, set\,(Y)\,, set\,(Z)\,, u, v, w)}$$

The proposition we are required to prove in this case is the same as that at point six, and therefore the proof is nearly identical. The only difference occurs in the inconsequential premises 2 and 5, which would be $set\,(Y) : SetOf\,(\mathbb{CLASS})$ and $v : set\,(Y)$ respectively in this case. Therefore, we refer the reader to point six for the proof of this case.

Given the steps presented above we conclude that:

$$\frac{\begin{array}{c} S : \mathit{SetOf}\,(\mathbb{SIGNATURE}) \quad s : S \\ C : \mathit{SetOf}\,(\mathbb{CLASS}) \quad c : C \\ M : \mathit{SetOf}\,(\mathbb{METHOD}) \\ \mathit{Dom}\,(\otimes, (S, C, M, s, c)) \end{array}}{\exists! m : M \bullet \otimes (S, C, M, s, c, m)}$$

■

The above proves that superimposition relation is indeed functional, allowing us to use it as the basis of our superimposition function. We shall overload the $\otimes$ symbol as our function symbol as which we are refering to is clear from the context. We also abstract from the type arguments S, C, and M where they are recoverable from the context. Our $\otimes$ function is therefore a binary one that we shall write in the infix notation. Specifically, we write $s \otimes c$ for the unique m where both $\otimes (s, c, m)$ and $\otimes : S \times C \longmapsto M$ (see p.62) hold, in the context of $S : \mathit{SetOf}\,(\mathbb{SIGNATURE})$, $C : \mathit{SetOf}\,(\mathbb{CLASS})$, and $M : \mathit{SetOf}\,(\mathbb{METHOD})$. This function enforces the desired clan structure[75] of the identified sets of methods by consistently unpacking signatures before classes. To explain, consider Figure 7.1: a grid of methods identifying all methods that must appear in some form in the term $\{s_1, \ldots, s_n\} \otimes \{c_1, \ldots, c_m\}$, where $s_1, \ldots, s_n : \mathbb{SIGNATURE}$ and $c_1, \ldots, c_n : \mathbb{CLASS}$.

$$\begin{array}{cccc} \otimes & c_1 & \cdots & c_m \\ s_1 & s_1 \otimes c_1 & \cdots & s_1 \otimes c_m \\ \vdots & \vdots & \ddots & \vdots \\ s_n & s_n \otimes c_1 & \cdots & s_n \otimes c_m \end{array}$$

Figure 7.1: Unpacking superimposition terms

Reading this grid vertically is the class-first approach to unpacking signatures and classes. That is, each column represents a set of methods contained in our set of sets of methods:

$$\{\{s_1 \otimes c_1, \ldots, s_n \otimes c_1\}, \ldots, \{s_1 \otimes c_m, \ldots, s_n \otimes c_m\}\}$$

However, this results in an undesirable set of tribes: sets of methods that do not share a common signature. Contrast this to reading the grid horizontally, the signature-first approach where each

[75] The depth of the nesting of the method set must be equal to the sum of the depths of the signature and class sets. Consider introducing a superscript shorthand for the *set* type constructor, where $\mathit{set}^{n+1}\,(T)$ is defined to be $\mathit{set}\,(\mathit{set}^{n}\,(T))$, and $\mathit{set}^{0}\,(T)$ is defined to be T. Given this, with two terms $s : \mathit{set}^{a}\,(\mathbb{SIGNATURE})$ and $c : \mathit{set}^{b}\,(\mathbb{CLASS})$, it stands that $s \otimes c : \mathit{set}^{a+b}\,(\mathbb{METHOD})$. This is a result that is derivable in our theory from the above proofs. LePUS3 does not provide any such proofs, but used a similar meta notation to define its superimposition function (LePUS3 Definition VI, Appendix B).

row represents a set of methods in the set of sets of methods:

$$\{\{s_1 \otimes c_1, \ldots, s_1 \otimes c_m\}, \ldots, \{s_n \otimes c_1, \ldots, s_n \otimes c_m\}\}$$

In this way we identify a desirable set of clans: sets of methods that share a common signature. It is for this reason we defined the superimposition function to unpack signature terms first.

Finally, there are a few very useful corollaries we can deduce from the superimposition relation:

Corollary 1 If any two superimposition expressions are equal, the signature arguments must also be equal:

$$\frac{\Gamma \vdash s_1, s_2 : \mathbb{SIGNATURE} \quad \Gamma \vdash c_1, c_2 : \mathbb{CLASS} \qquad \Gamma \vdash \otimes(s_1, c_1) = \otimes(s_2, c_2)}{\Gamma \vdash s_1 = s_2}$$

By $\mathbf{TC}_{11}$ (p.86) every method has exactly one signature, so it must follow that if two methods are equal so too must their signatures.

Corollary 2 If any two superimposition expressions are equal, one class must be a subclasses of the other:

$$\frac{\Gamma \vdash s : \mathbb{SIGNATURE} \quad \Gamma \vdash c_1, c_2 : \mathbb{CLASS} \qquad \Gamma \vdash \otimes(s, c_1) = \otimes(s, c_2)}{\Gamma \vdash \mathit{Subclass}(c_1, c_2) \vee \mathit{Subclass}(c_2, c_1)}$$

We cannot conclude that c_1 and c_2 are equal as methods can be inherited ($\mathbf{TC}_{5b}$, p.89), but we can conclude that either they are equal or there must be an inheritance relationship between them (*Subclass*).

This concludes our general discussion on methods, with which we have introduced all our basic types of our theory. In the next section we populate our theory with several important relationships that articulate other decidable properties of object-oriented design.

7.5 Simple Relations

Thus far we have laid the foundation of our theory of classes. We introduced a type for classes and discussed inheritance relationships between them (§7.1). From this we defined inheritance hierarchies as sets of classes that are all related by inheritance (§7.2). We introduced methods and their signatures, and discussed the rules which govern their relationships with each other and with classes (§7.3). And lastly (§7.4), we discussed the importance of the structure of sets of methods with regards to dynamic binding. However, what we can say about object-oriented design with this is limited. We need additional relations that capture more detailed decidable properties and relationships in an object-oriented program. This is the matter of this section: to enhance our theory with those relations that constitute the vocabulary of LePUS3 [Eden and Nicholson, 2011]. As we introduce each relation we will formalize our intuitions about them, and their arguments. In doing so we increase the expressive power of our theory.

7.5.1 *Method*

We begin with the LePUS3 unary relation *Method* [Eden and Nicholson, 2011]. The purpose of this relation was to say that a method (or set thereof) exists for a given signature and class (or sets thereof), but not constraining it further in any other relation. For example, we want to say that a class has a certain interface (methods), but not to specify any actions that may be taken by that class's methods. This is a necessity in Java for the cases of interfaces and abstract classes with abstract methods.

In practice, the supplied argument to this LePUS3 relation was always a superimposition term. Therefore, the *Method* relation essentially performed a typing operation on the result of the superimposition function. This approach is completely redundant given the strong typing mechanisms in **TC**, as we always know that the result of the superimposition function is a method (or nested set thereof). Instead we define a relation that has the same intent as its LePUS3 counterpart, but is more rigorously formulated. We will also call this relation *Method*, and define it by hiding the method argument and its type in the superimposition relation with existential quantifiers:

Method

$$C : SetOf\left(\mathbb{CLASS}\right), S : SetOf\left(\mathbb{SIGNATURE}\right)$$
$$c : C, s : S$$

$$\exists M : SetOf\left(\mathbb{METHOD}\right) \bullet \exists m : M \bullet \otimes\left(s, c, m\right)$$

where we abstract from the type arguments $C : SetOf\left(\mathbb{CLASS}\right)$, and $S : SetOf\left(\mathbb{SIGNATURE}\right)$ when these are recoverable from the context, therefore we use the shorthand relation $Method : schema\left(C \times S\right)$. The proposition $Method\left(c, s\right)$ should be read "there exists a method (or nested set thereof) in the scope of c, and identified by s". We are now able to state the existence of methods very easily without specifying other constraints on that method, for example consider Table 7.7. Of course, when considering single methods it would be just as simple to use the existential quantifier directly. But, as we discussed in the previous section, we need to ensure sets of methods are structured in a specific way. This relation allows this with the minimum of effort on the part of the designer.

Table 7.7: Modelling the existence of methods

Code	Description and Representation

interface List ... {...
int indexOf(Object o);
...}
	Interface List has the method List.indexOf(Object)
	$Method\,(\texttt{List}, \texttt{List.indexOf(Object)})$

class Vector ... {...
int indexOf(Object o) {...}
...}
	Class Vector has the method Vector.indexOf(Object)
	$Method\,(\texttt{Vector}, \texttt{Vector.indexOf(Object)})$

7.5.2 *DataMember*

In §7.4 we discussed a relation, *MethodMember*, which articulated the relationship between a class and the methods it contains. In this subsection we discuss the mirror image of this relation for data members. Therefore, in rule $\mathbf{TC}_{14}$, we introduce a relation called *DataMember* that is a binary relation on classes. We demonstrate how this relationship captures statements from Java source code in Table 7.8.

$$\mathbf{TC}_{14}\;\; \Gamma \vdash DataMember : schema\,(\mathbb{CLASS} \times \mathbb{CLASS})$$

We opt to not capture the names of data members, as this is considered to be an irrelevance at this level of design [Eden and Nicholson, 2011]. That is, at our level of abstraction we wish to indicate that a class has an instance of another class, but how it should be referenced is left to the implementation. If names of fields are required in some extension of our theory, then it is possible to include them with a mechanism similar to that used for methods, i.e. with some sort of signature/identifier. However, if such an extension were created, then it will encounter the same issues of scope that we encountered with methods, and require modifying rule $\mathbf{TC}_{5b}$ (p.89) to accommodate for another exception.

A consequence of this is that we do not capture the number of data memberships a class may have. That is, a $DataMember\,(a, b)$ relation states that the class a contains one or more data members (fields) of class b. In our experience this is a minimal sacrifice, and that at this level of abstraction it is more important to articulate that such dependencies exist rather than how they may be implemented.

Table 7.8: Modelling data members (fields)

Code	Description and Representation

class Vector ... {...
protected int elementCount;
...}
	Class Vector has a data member of type **int**
	$DataMember\,(\texttt{Vector}, \texttt{int})$

class Vector ... {...
protected Object[] elementData;
...}
	Class Vector has a data member of type Object[]
	$DataMember\,(\texttt{Vector}, \texttt{Object}[])$

We now have two relations for things that are members of a class; $DataMember$ for class members, and $MethodMember$ for method members. In LePUS3 there was a single $Member$ relation that performed the roles of both relations [Eden and Nicholson, 2011]. By using a polymorphic specification, we can define this unifying $Member$ relation as follows:

$$\begin{array}{|l}
\textit{Member} \\
\hline
T : \mathcal{U} \\
a : \mathbb{CLASS},\ b : T \\
\hline
(T = \mathbb{CLASS} \lor T = \mathbb{METHOD}) \land \\
(T = \mathbb{CLASS} \implies DataMember\,(a, b)) \land \\
(T = \mathbb{METHOD} \implies MethodMember\,(a, b)) \\
\hline
\end{array}$$

where the first clause ensures that T is either equal to the $\mathbb{CLASS}$ or $\mathbb{METHOD}$ type; The relation is implicitly false in all other cases. The rest of the relation dictates what relation should be used in accordance with the right type, i.e. $DataMember$ in the case of $T = \mathbb{CLASS}$ and $MethodMember$ in the case of $T = \mathbb{METHOD}$. We abstract from the type argument T when it is recoverable from the context, therefore we use the shorthand relation $Member : schema\,(\mathbb{CLASS} \times T)$.

This illustrates an important difference between our theory and LePUS3, which we alluded to in §4.3. LePUS3 has types, but they are virtually inconsequential. For example, the $Member$ relation in LePUS3 is not well-typed, as its contents are dictated by the how a program can be abstracted to a design model. Therefore, the LePUS3 $Member$ relation may hold for arguments of any type.

But what would it mean for a method to be a member of another method? Similar questions can be asked with regards to any LePUS3 relation. In practice, however, LePUS3 relations always have a fixed type that is implied by their use. We formalize this implicit type information in our theory. This is one of many reasons our theory is more explicit, expressive, and elegant than the current definition of LePUS3.

7.5.3 *Aggregate*

The $DataMember$ relation is used to represent a class that has one or more data members (fields) of another class. LePUS3 also has a relation called *Aggregate*, which represents a class that is an aggregation (or collection) of instances of another class. That is, the formula $Aggregate\,(a, b)$ in LePUS3 means that class a *has* or *is* a class defined as some sort of collection that may contain any number of instances of class b. Although these two relations capture different aspects of a program, there is some obvious overlap; if a class a is an aggregation of another class b, then a has at least one data member of class b. This fits with our intuitions that specifying a data membership may be implemented as an aggregation of some sort. This interaction between the two relations is recognized in LePUS3 by part of the fourth axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII (Appendix B).

We will introduce a similar relation into our theory, but we decompose it into more specific components. In other words, we define the *is* and *has* an aggregate separately as the unary *Collection* and binary *Aggregate* relations respectively. This makes our theory cleaner and more elegant than LePUS3, as each of our relations represent one interesting property about a program.

We begin with introducing the *Collection* relation in rule $\mathbf{TC}_{15}$, where if a class is in the relation *Collection* then it is an aggregate class. But what exactly constitutes a collection? Arrays are always collections, but everything else is open to interpretation in a given programming language. For Java we also take the interface java . util . Collection to define collections, i.e. everything that inherits from java . util . Collection is a collection. We articulate this in rule $\mathbf{TC}_{16}$, where anything that inherits from a class that is in *Collection* is itself in *Collection*. For example, consider Table 7.9 which illustrates this against some Java code.

$$\mathbf{TC}_{15}\quad \Gamma \vdash Collection : schema\,(\mathbb{CLASS})$$

$$\mathbf{TC}_{16}\quad \frac{\begin{array}{c}\Gamma \vdash sub, col : \mathbb{CLASS}\\ \Gamma \vdash Collection\,(col) \quad \Gamma \vdash Inherit\,(sub, col)\end{array}}{\Gamma \vdash Collection\,(sub)}$$

Table 7.9: Modelling collections

Code	Description and Representation
interface List ... **extends** Collection ...	List is a collection because it extends Collection $Collection\,(\texttt{List})$
class Vector ... **implements** List ...	Vector is a collection by indirectly implementing Collection $Collection\,(\texttt{Vector})$

class Calendar ... {...
protected int fields [];
...}
	An array (**int** []) is a collection
	$Collection\,(\texttt{int}[])$

With this we can define classes that *have* a collection, and the class of the instances they aggregate. For this we introduce the *Aggregate* relation in rule $\mathbf{TC}_{17}$. *Aggregate* is unusual as we are able to provide a partial introduction rule, $\mathbf{TC}_{18}$. We may introduce an aggregate relationship between classes a and b if there is an appropriate collection class that is a member of a and is an aggregate of b. This is illustrated in the third example provided in Table 7.10. $\mathbf{TC}_{19}$ is an elimination counterpart to rule $\mathbf{TC}_{18}$. This tells us that an aggregation between classes a and b means that either a must be a collection class, or there exists an appropriate collection that is a data member of a and contains instances of class b. Finally, rule $\mathbf{TC}_{20}$ is another elimination rule, which articulates the interaction between aggregates and data members that we discussed at the beginning of this subsection. That is, an aggregate relationship allows us to conclude data membership.

$$\mathbf{TC}_{17}\quad \Gamma \vdash Aggregate : schema\,(\mathbb{CLASS} \times \mathbb{CLASS})$$

$$\mathbf{TC}_{18}\quad \frac{\begin{array}{c}\Gamma \vdash a, b, c : \mathbb{CLASS} \\ \Gamma \vdash DataMember\,(a, b) \quad \Gamma \vdash Collection\,(b) \quad \Gamma \vdash Aggregate\,(b, c)\end{array}}{\Gamma \vdash Aggregate\,(a, c)}$$

$$\mathbf{TC}_{19}\quad \frac{\begin{array}{c}\Gamma \vdash a, b : \mathbb{CLASS} \\ \Gamma \vdash Aggregate\,(a, b)\end{array}}{\begin{array}{c}\Gamma \vdash Collection\,(a) \vee \\ \exists x : \mathbb{CLASS} \bullet DataMember\,(a, x) \wedge Collection\,(x) \wedge Aggregate\,(x, b)\end{array}}$$

$$\mathbf{TC}_{20}\quad \frac{\begin{array}{c}\Gamma \vdash a, b : \mathbb{CLASS} \\ \Gamma \vdash Aggregate\,(a, b)\end{array}}{\Gamma \vdash DataMember\,(a, b)}$$

Table 7.10: Modelling aggregates of instances

Code	Description and Representation
interface List ... **extends** Collection ...	List is an aggregate of Object because it extends Collection $Aggregate\,(\texttt{List}, \texttt{Object})$
class Vector ... **implements** List ...	Vector is an aggregate of Object because it implements List $Aggregate\,(\texttt{Vector}, \texttt{Object})$

class Calendar ... {...
protected int [] fields ;
...}
	int [] is an array (aggregate) of **int**
	$Aggregate\,(\texttt{int}[], \texttt{int})$
	By $\mathbf{TC}_{18}$, therefore Calendar is also an aggregate of **int**
	$Aggregate\,(\texttt{Calendar}, \texttt{int})$

7.5.4 *Call*

We do not specify the most behavioural properties of object-oriented design as, in the general case, these are undecidable properties. However, we are able to capture a certain degree of behaviour at an abstract level. For example, LePUS3 has a relation called *Call*, which specifies that a method may call (execute) another method at some point during the execution of its body. We say *may* as we do not capture the conditions under which a method is executed, i.e. the values past to a method call or any control statements preceding it.

We capture this same simple execution relationship between methods. We also name the relation *Call* as introduced in rule $\mathbf{TC}_{21}$, which we will elaborate on as we introduce other simple relations through the remainder of this section. The proposition $Call\,(a, b)$ should be read as "method a may call method b at some point during its execution". Table 7.11 demonstrates this relation with some example source code from the Java SDK.

$$\mathbf{TC}_{21}\ \ \Gamma \vdash Call : schema\,(\mathbb{METHOD} \times \mathbb{METHOD})$$

Table 7.11: Modelling method calls

Code	Description and Representation

class Vector ... {...
int indexOf(Object o, **int** index) {...}
int indexOf(Object o) {
return indexOf(o, 0);
} ...}
	Method Vector.indexOf(Object, **int**) calls (executes)
	method Vector.indexOf(Object)
	$Call\,(\texttt{Vector.indexOf(Obj)}, \texttt{Vector.indexOf(Obj}, \texttt{int}))$

7.5.5 *Forward*

Building on our discussion in the previous subsection, some method calls hold a special meaning in object-oriented design. In this subsection we focus on method forwarding. A method forwards its method call to another method if they share the same signature, and the method is called with the same arguments as those passed to the original method. In Java this has to be accomplished manually, for example see Table 7.12. however, in languages such as Smalltalk, forwarding can accomplished automatically using its `doesNotUnderstand` mechanism [Gamma et al., 1994].

To this end, we introduce the relation $Forward$ with rule $\mathbf{TC}_{22}$. As we previously described, if a forwarding method exists between two methods, then they must have the same signature, as articulated in $\mathbf{TC}_{23}$. Finally, a forwarding relation is a specialization of a method call, as articulated in rule $\mathbf{TC}_{24}$[76].

$$\mathbf{TC}_{22}\quad \Gamma \vdash Forward : schema\,(\mathbb{METHOD} \times \mathbb{METHOD})$$

$$\mathbf{TC}_{23}\quad \frac{\begin{array}{c}\Gamma \vdash a, b : \mathbb{METHOD}\\ \Gamma \vdash Forward\,(a, b)\end{array}}{\Gamma \vdash SigOf\,(a) = SigOf\,(b)}$$

$$\mathbf{TC}_{24}\quad \frac{\begin{array}{c}\Gamma \vdash a, b : \mathbb{METHOD}\\ \Gamma \vdash Forward\,(a, b)\end{array}}{\Gamma \vdash Call\,(a, b)}$$

Table 7.12: Modelling method forwarding

Code	Description and Representation

`class AbstractList ... {...`
` List subList(int from, int to) {...}`
`...}`
 	
`class Vector extends AbstractList ... {...`
` List subList(int from, int to) {...`
`  super.subList(from, to);`
` ...} ...}`
	Method `Vector.subList(int,int)` calls (executes) method `AbstractList.subList(int,int)`, with the same arguments that it was provided. Although it breaks our conventions, to save space we refer to `Vector.subList(int,int)` with `VSubList`, and `AbstractList.subList(int,int)` with `ALSubList`, therefore: $Forward\,($`VSubList`, `ALSubList`$)$

7.5.6 *Return*

Another type of method behaviour we intend to capture is interaction of methods with instances, or rather the class of the instances with which they interact. We start with capturing return statements, i.e. in Java these are instructions inside a methods body that start with the `return` keyword. We accomplish this by introducing our *Return* relation, rule $\mathbf{TC}_{25}$, and as previously we illustrate the *Return* relation against some example Java source code in Table 7.13.

Following suit from LePUS3, we do not capture the return type of a method, as we see such

[76] Part of the fourth axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII in Appendix B.

information as primarily an implementation detail. Although we may add such a relation if we so wish, in our experience using LePUS3 it is enough to specify that a method should be able to return an instance of some class. The return type would therefore have to be that class, or one of its superclasses, but exactly which is usually best left to the point of implementation. We also formulate this notion in rule $\mathbf{TC}_{26}$, i.e. if there are two return relations, then they must share a common root class. This is one of the many rules that we are able to capture in our theory of classes that does not exist in LePUS3.

$$\mathbf{TC}_{25}\quad \Gamma \vdash \mathit{Return} : \mathit{schema}\,(\mathbb{METHOD} \times \mathbb{CLASS})$$

$$\mathbf{TC}_{26}\quad \frac{\begin{array}{cc}\Gamma \vdash m : \mathbb{METHOD} & \Gamma \vdash c_1, c_2 : \mathbb{CLASS} \\ \Gamma \vdash \mathit{Return}\,(m, c_1) & \Gamma \vdash \mathit{Return}\,(m, c_2)\end{array}}{\Gamma \vdash \exists \mathit{root} : \mathbb{CLASS} \bullet \mathit{Subclass}\,(c_1, \mathit{root}) \wedge \mathit{Subclass}\,(c_2, \mathit{root})}$$

Table 7.13: Modelling returning statements

Code	Description and Representation

class Vector ... {...
int elementCount;
int size () { **return** elementCount; }
...}
	Vector. size () returns an instance of **int**
	$\mathit{Return}\,(\texttt{Vector.size}(), \texttt{int})$

class Vector ... {...
int elementCount;
boolean isEmpty() { **return** elementCount==0; }
...}
	Vector. size () returns an instance of **boolean**
	$\mathit{Return}\,(\texttt{Vector.isEmpty}(), \texttt{boolean})$

class AbstractList ... {...
Iterator iterator () { **return** **new** Itr(); }
...}
	AbstractList . iterator () returns an instance of Itr
	$\mathit{Return}\,(\texttt{AbstractList.iterator}(), \texttt{Itr})$

7.5.7 *Create*

Continuing the theme of methods and their interaction with instances of classes, we introduce a relation to capture the instantiation of classes. As in previous cases, such as method calls, we do not capture under which conditions a class is instantiated, or with what parameters. In Java this most commonly equates to the **new** keyword, i.e. a method call to a class's constructor, as illustrated in Table 7.14. We call this relation *Create*, which we introduce in rule $\mathbf{TC}_{27}$. If we decided to distinguish constructors (or destructors) from other methods then we could say a little more about creating instances. For example, a method m creating a class c implies that m calls a constructor in c, which calls a constructor in a direct superclass of c, and so on up the inheritance tree. However, although we could capture such information, it is not useful for the purposes of design at our level of abstraction. Therefore, we continue to stick close to the vocabulary of LePUS3, and choose not to articulate this implementation detail.

$$\mathbf{TC}_{27}\ \ \Gamma \vdash \mathit{Create} : \mathit{schema}\,(\mathbb{METHOD} \times \mathbb{CLASS})$$

Table 7.14: Modelling return statements

Code	Description and Representation

class Vector ... {...
Object clone() {...
throw new InternalError();
...} ...}
	Vector.clone() creates an instance of InternalError
	$\mathit{Create}\,(\texttt{Vector.clone()}, \texttt{InternalError})$

class AbstractList ... {...
Iterator iterator() { **return new** Itr(); }
...}
	AbstractList.iterator() returns an instance of Itr
	$\mathit{Create}\,(\texttt{AbstractList.iterator()}, \texttt{Itr})$

7.5.8 *Produce*

Our final relation, on the theme of methods and their interaction with instances of classes, is one that captures the production of instances. That is, a method that creates and returns the same instance of a class c is said to *produce* an instance of c. We introduce this *Produce* relation in rule $\mathbf{TC}_{28}$. Simple examples of this relation are illustrated in Table 7.15, all taken from the Java SDK.

From our informal description of the relation, it is evident that *Produce* is a special sort of *Create* and *Return* relationship. However, as we do not capture objects, we are only able to say that a *Produce* relationship can be decomposed into *Create* and *Return* relationships. This is articulated in rule $\mathbf{TC}_{29}$[77].

$$\mathbf{TC}_{28}\ \ \Gamma \vdash \mathit{Produce} : \mathit{schema}\,(\mathbb{METHOD} \times \mathbb{CLASS})$$

$$\mathbf{TC}_{29}\ \ \frac{\begin{array}{c}\Gamma \vdash m : \mathbb{METHOD} \quad \Gamma \vdash c : \mathbb{CLASS} \\ \Gamma \vdash \mathit{Produce}\,(m, c)\end{array}}{\Gamma \vdash \mathit{Create}\,(m, c) \wedge \mathit{Return}\,(m, c)}$$

7.5.9 *Abstract*

Abstract classes and interfaces are one of the fundamental mechanisms of abstraction in object-oriented design. These classes are, by definition, not permitted to be instantiated. For example, in

[77] Part of the fourth axiom of class-based programming [Eden and Nicholson, 2011], LePUS3 Definition VIII in Appendix B.

Table 7.15: Modelling the production of instances

Code	Description and Representation

class AbstractList ... {...
Iterator iterator () { **return** **new** Itr(); }
...}
	AbstractList . iterator () creates and returns the same instance of Itr
	$\mathit{Produce}\,(\texttt{AbstractList.iterator()}, \texttt{Itr})$

class LinkedList ... {...
int indexOf(Object o) {...
int index = 0; ...
return index;
...} ...}
	LinkedList .indexOf(Object) creates and returns the same instance of **int**
	$\mathit{Produce}\,(\texttt{LinkedList.indexOf(Object)}, \texttt{int})$

Java an **interface** is always abstract, as is any class defined with the **abstract** keyword, as illustrated in Table 7.16. We articulate this simple relationship by introducing the *AbstractClass* relation with rule $\mathbf{TC}_{30}$.

$$\mathbf{TC}_{30}\ \ \Gamma \vdash \mathit{AbstractClass} : \mathit{schema}\,(\mathbb{CLASS})$$

Table 7.16: Modelling abstract classes

Code	Description and Representation
interface List ... {...}	List is an interface, and therefore abstract $\mathit{AbstractClass}\,(\texttt{List})$
abstract **class** AbstractList ... {...}	AbstractList is an abstract class $\mathit{AbstractClass}\,(\texttt{AbstractList})$

It is equally possible for methods to be abstract. An abstract method declares an interface for accessing a class, but does not declare any associated default behaviour. It is expected that implementing classes implement specific behaviour as appropriate. In Java, all methods declared in an interface are abstract as they are not allowed to define the method's body. Alternatively, abstract classes may have no abstract methods, where the **abstract** keyword indicates if a method is abstract. A few example selections of Java source code, presented in Table 7.17, illustrate this. Therefore, as with *AbstractClass*, we introduce the relation *AbstractMethod* in rule $\mathbf{TC}_{31}$.

$$\mathbf{TC}_{31}\ \ \Gamma \vdash \mathit{AbstractMethod} : \mathit{schema}\,(\mathbb{METHOD})$$

Table 7.17: Modelling abstract methods

Code	Description and Representation

interface List ... {...
boolean remove(Object o);
...}
	List .remove(Object) is a method in an interface
	$\mathit{AbstractMethod}\,(\texttt{List.remove(Object)})$

abstract **class** AbstractList ... {...
abstract **public** Object get(**int** index);
...}
	AbstractList .get(**int**) is an abstract method
	$\mathit{AbstractMethod}\,(\texttt{AbstractList.get(int)})$

We also articulate a few of our intuitions about abstract classes and methods. In doing so, we afford a greater level of reasoning in our theory in comparison to LePUS3. Our first intuition of abstract methods is that if they are a members of a class, then that class must also be abstract—articulated in rule $\mathbf{TC}_{32}$. The mirror of this is rule $\mathbf{TC}_{33}$, where if a method is a member of a non-abstract class then that method cannot be abstract.

Our third intuition is that an abstract method has no body, i.e. it contains no instructions. Therefore, if a method is abstract then it cannot perform any actions such as calling another method, or creating an instance of a class. This is articulated in rule $\mathbf{TC}_{34}$, with the conditions that the type T must either be $\mathbb{CLASS}$ or $\mathbb{METHOD}$, and that R is restricted to one of the following appropriate relations: *Call*, *Forward*, *Return*, *Create*, or *Produce*. Restricting R in this way ensures that rule $\mathbf{TC}_{34}$ is not too sweeping. For example, consider the case where $=_{\mathbb{METHOD}}$ is R, we would be able to show that a method is not equal to itself because it is abstract.

Conversely, another of our intuitions is that if a method is involved in one of the aforementioned relations, then it cannot be abstract. That is, if we can prove that a method has a body, then it must not be abstract. This is articulated in rule $\mathbf{TC}_{35}$, on which we place the same restrictions for the values of T and R as we did in $\mathbf{TC}_{34}$.

$$\mathbf{TC}_{32}\ \frac{\begin{array}{c}\Gamma \vdash m : \mathbb{METHOD} \quad \Gamma \vdash c : \mathbb{CLASS} \\ \Gamma \vdash \mathit{AbstractMethod}\,(m) \quad \Gamma \vdash \mathit{MethodMember}\,(c, m)\end{array}}{\Gamma \vdash \mathit{AbstractClass}\,(c)}$$

$$\mathbf{TC}_{33}\ \frac{\begin{array}{c}\Gamma \vdash m : \mathbb{METHOD} \quad \Gamma \vdash c : \mathbb{CLASS} \\ \Gamma \vdash \neg \mathit{AbstractClass}\,(c) \quad \Gamma \vdash \mathit{MethodMember}\,(c, m)\end{array}}{\Gamma \vdash \neg \mathit{AbstractMethod}\,(m)}$$

$$\mathbf{TC}_{34}\ \frac{\begin{array}{c}\Gamma \vdash R : \mathit{schema}\,(\mathbb{METHOD} \times T) \quad \Gamma \vdash m : \mathbb{METHOD} \quad \Gamma \vdash t : T \\ \Gamma \vdash \mathit{AbstractMethod}\,(m)\end{array}}{\Gamma \vdash \neg R\,(m, t)}$$

$$\mathbf{TC}_{35}\ \frac{\begin{array}{c}\Gamma \vdash R : \mathit{schema}\,(\mathbb{METHOD} \times T) \quad \Gamma \vdash m : \mathbb{METHOD} \quad \Gamma \vdash t : T \\ \Gamma \vdash R\,(m, t)\end{array}}{\Gamma \vdash \neg \mathit{AbstractMethod}\,(m)}$$

Finally, we mimic the LePUS3 vocabulary again by creating a single polymorphic *Abstract* relation. This mirrors the *Member* relation we defined earlier (§7.5.2) for method and data membership.

Abstract

$T : \mathcal{U}$

$t : T$

$$\begin{gathered}(T = \mathbb{CLASS} \lor T = \mathbb{METHOD}) \land \\ (T = \mathbb{CLASS} \implies \mathit{AbstractClass}\,(t)) \land \\ (T = \mathbb{METHOD} \implies \mathit{AbstractMethod}\,(t))\end{gathered}$$

where the first clause ensures that T is either equal to the $\mathbb{CLASS}$ or $\mathbb{METHOD}$ type; the relation is implicitly false in all other cases. The rest of the relation dictates what relation should be used in accordance with the right type, i.e. *AbstractClass* in the case of $T = \mathbb{CLASS}$ and *AbstractMethod* in the case of $T = \mathbb{METHOD}$.

7.5.10 More Relations

This section introduces all the simple relations that make up LePUS3 [Eden and Nicholson, 2011], and even adds a few more, for example, *Collection*, *Overrides* and *Subclass*. For our purposes, this theory is now expressive enough to articulate the basic properties of object-oriented design that we set out to capture (see §2.1), and abstracts from a great deal of implementation minutiae. However, our theory is not limited to only these relationships. We may continue to extend our theory with as many more detailed relationships as is seen fit. But with each new relation we must pay careful attention to its impact on the rest of the theory, i.e. is the addition of a rule a conservative or proper extension, and does it allow us to prove any propositions that conflict with our understanding of object-oriented design.

Here is a small list of just a few features that we do not cover here, but could be introduced in the future if desired:

- Structural subtyping [Abadi and Cardelli, 1998]
- Method specialization [Abadi and Cardelli, 1998] (as opposed to method overriding)
- Class/method modifiers, for example in Java: **static**, **final**, **transient**, **private**, etc.[78]

[78] There are interesting issues of *locality* surrounding accessibility of methods and data that go beyond simply

- Class constructors and destructors
- Exception handling
- Annotations and other metadata

In the remainder of this chapter we discuss how our simple relations, i.e. those we have introduced into our theory of classes, work at higher levels of abstraction. That is, how they apply to (nested) sets.

7.6 Complex Relations

In the previous sections of this chapter we constructed the bulk of our theory of classes. We defined types for the appropriate building blocks of object-oriented design at our level of abstraction: classes, hierarchies, clans and tribes. We proceeded to extend this theory with simple relations between individual classes and methods, and articulated several of our intuitions regarding them. The result of this is a theory of classes that is much more elegant, rigorous, and expressive than the current definition of LePUS3.

However, we have not yet addressed one of the central abstraction mechanisms in LePUS3, the *predicates* [Eden and Nicholson, 2011]. The LePUS3 predicates are complex relations on or between (nested) sets of terms with respect to one of the LePUS3 relations. LePUS3 predicates therefore take a LePUS3 relation[79] as an argument and hide complex quantification.

There are three predicates in LePUS3, All, Total, and Isomorphic (see Appendix B). However, as we mentioned in §4.3, these names have caused some confusion. The problem here lies with the use of the words total and isomorphic. Although the relations take inspiration from total functional relations and isomorphisms respectively, they are not standard definitions of these terms. Rather, they are modified to account for specific exceptions in the case of abstract methods. We will explain more about why we do this as we progress through this section. However, as the predicates are non-standard and domain specific, it is not appropriate to continue using this terminology. Instead we opt for Iso and Tot respectively. These shorter names still indicate their origins, but disambiguate them from the standard terms.

specifying the use of a keyword. For further information on locality see [Eden, 2005] or to a lesser extent [Eden and Nicholson, 2011].

[79] As the equivalent relations in our Theory of Classes (**TC**) are all of the type $schema\,(T)$ for some type T, it follows that our definitions of these predicates will be of type $schema\,(schema\,(T) \times A)$.

7.6.1 All

The All predicate is essentially a hidden universal quantification that extends naturally for nested sets. To explain its purpose, let us examine an example taken from the Java SDK [Sun Microsystems Inc., 2006]. Consider the following abstract classes and interfaces that form part of the list collection hierarchy:

$$\texttt{Collection}, \texttt{List}, \texttt{AbstractList} : \mathbb{CLASS}$$

$$\mathit{Abstract}\,(\texttt{Collection})$$

$$\mathit{Abstract}\,(\texttt{List})$$

$$\mathit{Abstract}\,(\texttt{AbstractList})$$

What if we want to express this information as a single proposition once these classes have been combined into a single set? That is, we want to be able to articulate formally statements like:

$$\{\texttt{Collection}, \texttt{List}, \texttt{AbstractList}\} : \mathit{set}\,(\mathbb{CLASS})$$

All elements of the set $\{\texttt{Collection}, \texttt{List}, \texttt{AbstractList}\}$ are *abstract*

This is easy to formalize as we can use a simple universal quantification, such as:

$$\forall x \in \{\texttt{Collection}, \texttt{List}, \texttt{AbstractList}\} \bullet \mathit{Abstract}\,(x)$$

This is a perfectly acceptable solution in this specific case, but it does not scale in the case of nested sets. For example, consider building a nested set of classes from our list hierarchy:

$$\{\{\texttt{Collection}\}, \{\texttt{List}\}, \{\texttt{AbstractList}\}\} : \mathit{set}\,(\mathit{set}\,(\mathbb{CLASS}))$$

Therefore, we have to use two universal quantifiers to get down to the point at which the *Abstract* relation can be applied, i.e.:

$$\forall x \in \{\{\texttt{Collection}\}, \{\texttt{List}\}, \{\texttt{AbstractList}\}\} \bullet \forall y \in x \bullet \mathit{Abstract}\,(x)$$

Obviously this gets progressively more complex the greater the nesting of sets. Neither have we dealt with the issues of vacuous truth introduced with empty sets. To simplify this, we define a polymorphic relation using recursive schema:

$$\begin{array}{|c|}
\multicolumn{1}{l}{\textsc{All}} \\
\hline
\begin{array}{l} B : \mathcal{U},\ T : \mathit{SetOf}\,(B) \\ R : \mathit{schema}\,(B)\,,\ t : T \end{array} \\
\hline
\begin{array}{c}
(T = B \implies R\,(t)) \\
\wedge \\
\left(\begin{array}{c} T \neq B \implies \exists X : \mathit{SetOf}\,(B) \bullet \\ T = \mathit{set}\,(X) \wedge t \neq \varnothing_X \wedge \forall x \in t \bullet \textsc{All}\,(B, X, R, x) \end{array} \right)
\end{array} \\
\hline
\end{array}$$

with the side condition that R must be one of the relations we have introduced in this chapter, such as *Abstract* and *Collection*. We abstract from the type arguments $B : \mathcal{U}$ and $T : \mathit{SetOf}\,(B)$ when these are recoverable from the context, and as such we treat All to be of the type $\mathit{schema}\,(\mathit{schema}\,(B) \times T)$. Note that this specification is similarly structured to the superimposition ($\otimes$) relation presented in §7.4. The proof that the All specification type checks (i.e. it is always a proposition under appropriately typed arguments) therefore follows the same pattern as the type checking proof for $\otimes$. Therefore we conclude that little would be added by repeating the proof.

The structure of the All predicate forms two mutually exclusive cases. The first deals with simple arguments where no set terms are involved, and the second accommodates nested set terms. As such, the predicate appropriately captures the relationship we discussed at the start of this subsection by hiding complex quantifications and extending naturally for nested sets. Additionally, the definition of our All predicate is identical in intent to that of the All predicate in LePUS3 [Eden and Nicholson, 2011] (see LePUS3 Definition XI in Appendix B). However, aside from the conversion into the Typed Predicate Logic (**TPL**) framework, their definitions are not quite identical as we added the constraint that sets cannot be empty. This has always been an implicit assumption in LePUS3, which we simply make explicit in our definition.

We are now able to articulate the abstract quality of the aforementioned classes as a single, or nested, set. That is, both of the following will hold:

$$\textsc{All}\left(\mathit{Abstract}, \{\texttt{Collection}, \texttt{List}, \texttt{AbstractList}\}\right)$$

$$\textsc{All}\left(\mathit{Abstract}, \{\{\texttt{Collection}\}, \{\texttt{List}\}, \{\texttt{AbstractList}\}\}\right)$$

7.6.2 Tot

The Tot predicate articulates a form of total binary relation that extends naturally for nested sets. To explain its purpose, let us examine an example taken from the java.util package of the Java SDK [Sun Microsystems Inc., 2006]. We represent two methods that share the same signature, remove elements from a Java collection, and return an indication of its success:

$$\texttt{RemoveAll(Collection)} : \mathbb{SIGNATURE}$$

$$\texttt{ArrayList}, \texttt{HashSet}, \texttt{boolean} : \mathbb{CLASS}$$

$$\mathit{Return}\left(\texttt{RemoveAll(Collection)} \otimes \texttt{ArrayList}, \texttt{boolean}\right)$$

$$\mathit{Return}\left(\texttt{RemoveAll(Collection)} \otimes \texttt{HashSet}, \texttt{boolean}\right)$$

It is logical to group these two methods together to form a clan—one of our powerful abstraction mechanisms. How then do we specify the two appropriate return relationships for this clan? That is, how can we formalize statements such as:

All elements of the clan $\texttt{RemoveAll(Collection)} \otimes \{\texttt{ArrayList}, \texttt{HashSet}\}$
return objects that are instances of class $\texttt{boolean}$

The simplest approach to this is to use a universal quantifier, for example:

$$\forall x \in \texttt{RemoveAll(Collection)} \otimes \{\texttt{ArrayList}, \texttt{HashSet}\} \bullet \mathit{Return}\left(x, \texttt{boolean}\right)$$

However, our research has shown that we need to make exceptions in the case of abstract methods. To illustrate this, let us extend our example with the root interface to all Java collections. This class also declares a method of the signature RemoveAll(Collection), but it does not define a body for the method—it is abstract. This constitutes the obvious hierarchy:

$$\texttt{Collection} : \mathbb{CLASS}$$

$$\mathit{Inherit}\,(\texttt{HashSet}, \texttt{Collection})$$

$$\mathit{Inherit}\,(\texttt{ArrayList}, \texttt{Collection})$$

$$\{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\} : \mathbb{HIERARCHY}$$

$$\mathit{Abstract}\,(\texttt{RemoveAll}(\texttt{Collection}) \otimes \texttt{Collection})$$

In this case, it is no longer as simple as doing a universal quantification over all elements in the set of methods. This is because the method $\texttt{RemoveAll}(\texttt{Collection}) \otimes \texttt{Collection}$ is abstract. It provides an interface for accessing other instances of that class hierarchy. The method must be contained in our clan by virtue of the superimposition function, indeed it is only logical that it is included. However, by rule $\mathbf{TC}_{33}$ (p.122), we know that propositions, such as the following, will not hold:

$$\forall x \in \texttt{RemoveAll}(\texttt{Collection}) \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\} \bullet \mathit{Return}\,(x, \texttt{boolean})$$

Hierarchies that contains clans of this format are not only common, but is also indicative of good object-oriented design. Therefore, we need to exclude the abstract method from being evaluated in the way the others are. We must also ensure that not all of the methods are abstract, as if this is the case then it is only logical that the proposition fails to hold. To accomplish this we introduce a simple relation:

$AbsMthOf$

$$a : \mathbb{METHOD}$$
$$b : \mathit{set}\,(\mathbb{METHOD})$$

$$\mathit{Abstract}\,(a) \land a \in b \land \exists x \in b \bullet \neg \mathit{Abstract}\,(x)$$

The $AbsMthOf$ relation articulates exactly the conditions under which we will exclude methods. That is, a is an abstract method that is a member of b, a non-empty set of methods that must have at least one non-abstract member. With this we are able to exclude abstract methods, without fiddling with the clean and logical structure of clans. For simplicity, let $\{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\}$ be referred to as `hrc`:

$$\begin{aligned} \forall x \quad &\in \quad \texttt{RemoveAll(Collection)} \otimes \texttt{hrc} \bullet \\ &\quad \mathit{AbsMthOf}\,(x, \texttt{RemoveAll(Collection)} \otimes \texttt{hrc}) \vee \mathit{Return}\,(x, \texttt{boolean}) \end{aligned}$$

This proposition now states that for all elements of our clan, the method is either abstract or it returns an instance of class `boolean` at some point during its execution. However, as with the All predicate, this solution does not extend naturally for (nested) sets. For example, consider putting the class `boolean` into a singleton set. To accommodate this we would have to extend our proposition to include an existential quantification on that set:

$$\begin{aligned} \forall x \quad &\in \quad \texttt{RemoveAll(Collection)} \otimes \texttt{hrc} \bullet \\ \exists y \quad &\in \quad \{\texttt{boolean}\} \bullet \mathit{AbsMthOf}\,(x, \texttt{RemoveAll(Collection)} \otimes \texttt{hrc}) \vee \mathit{Return}\,(x, y) \end{aligned}$$

To simplify this, we define a polymorphic relation using recursive schema:

$$\textsc{Tot}$$

$$\begin{array}{l} A, B : \mathcal{U},\ A' : \mathit{SetOf}\,(A)\,,\ B' : \mathit{SetOf}\,(B) \\ R : \mathit{schema}\,(A \times B)\,,\ a : A',\ b : B' \end{array}$$

$$\begin{array}{c}
(A = A' \wedge B = B' \implies R\,(a, b)) \\
\wedge \\
\left(\begin{array}{c} A = A' \wedge B \neq B' \implies \exists Y : \mathit{SetOf}\,(B) \bullet \\ B' = \mathit{set}\,(Y) \wedge \exists y \in b \bullet \textsc{Tot}\,(A, B, A', Y, R, a, y) \end{array} \right) \\
\wedge \\
\left(\begin{array}{c} A \neq A' \wedge B = B' \implies \exists X : \mathit{SetOf}\,(A) \bullet \\ A' = \mathit{set}\,(X) \wedge a \neq \varnothing_X \wedge \\ \forall x \in a \bullet (X = \mathbb{METHOD} \implies \mathit{AbsMthOf}\,(x, a)) \vee \textsc{Tot}\,(A, B, X, B', R, x, b) \end{array} \right) \\
\wedge \\
\left(\begin{array}{c} A \neq A' \wedge B \neq B' \implies \exists X : \mathit{SetOf}\,(A) \bullet \exists Y : \mathit{SetOf}\,(B) \bullet \\ A' = \mathit{set}\,(X) \wedge B' = \mathit{set}\,(Y) \wedge a \neq \varnothing_X \wedge \\ \forall x \in a \bullet (X = \mathbb{METHOD} \implies \mathit{AbsMthOf}\,(x, a)) \vee \exists y \in b \bullet \textsc{Tot}\,(A, B, X, Y, R, x, y) \end{array} \right)
\end{array}$$

with the side condition that R must be one of the relations we have introduced in this chapter, such as *Inherit* and *Aggregate*. We abstract from the type arguments $A, B : \mathcal{U}$, $A' : \mathit{SetOf}\,(A)$, and $B' : \mathit{SetOf}\,(B)$ when these are recoverable from the context, therefore we use the shorthand relation Tot: $\mathit{schema}\,(\mathit{schema}\,(A \times B) \times A' \times B')$. Note that this specification is similarly structured to the superimposition ($\otimes$) relation presented in §7.4. The proof that the Tot specification type checks (i.e. it is always a proposition under appropriately typed arguments) therefore follows the same pattern as the type checking proof for $\otimes$. Therefore we conclude that little would be added by repeating the proof.

The structure of the Tot predicate forms four mutually exclusive cases. The first deals with simple arguments where no set terms are involved, whereas the remaining three cases accommodate nested sets. Each case appropriately deals with empty sets, abstract methods, and captures the relationship we discussed at the start of this subsection. Additionally, the definition of our Tot predicate is identical in intent to that of the Total predicate in LePUS3 [Eden and Nicholson, 2011] (see LePUS3 Definition XII in Appendix B). However, aside from the conversion to Typed Predicate Logic (**TPL**), their definitions are not quite identical. As with All, we have added the constraint that sets cannot be empty. This has always been an implicit assumption in LePUS3, which we simply make explicit in our definition. We also ensure that at least one valid non-abstract method is exists in relevant sets. This requirement also does not exist in LePUS3, but without it we could prove that any abstract set of methods invoke other methods, create instances, etc.

We are now able to succinctly articulate the relationship between the aforementioned sets, i.e.:

$$\textsc{Tot}\left(\mathit{Return}, \texttt{RemoveAll(Collection)} \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\}, \texttt{boolean}\right)$$

7.6.3 Iso

Our last predicate is the Iso predicate. This articulates a form of isomorphic relation that extends naturally for nested sets. That is, we want to specify that there is a *one-to-one* and *onto* (bijective) correspondence between two sets of terms, but as with the Tot predicate we must be careful of abstract methods. To help illustrate the purpose of this predicate, let us examine an example taken from the java.util package of the Java SDK [Sun Microsystems Inc., 2006]. We represent two methods that share the same signature and both produce an appropriate iterator for their respective collection class:

$$\texttt{iterator()} : \mathbb{SIGNATURE}$$

$$\texttt{ArrayList}, \texttt{HashSet}, \texttt{ArrayListIterator}, \texttt{HashSetIterator} : \mathbb{CLASS}$$

$$\mathit{Produce}\left(\texttt{iterator()} \otimes \texttt{ArrayList}, \texttt{ArrayListIterator}\right)$$

$$\mathit{Produce}\left(\texttt{iterator()} \otimes \texttt{HashSet}, \texttt{HashSetIterator}\right)$$

This is a strong indication of an isomorphism, i.e. exactly one method produces exactly one kind of iterator class. Indeed, we want to be able to formally articulate statements, such as:

All elements of the clan $\texttt{iterator()} \otimes \{\texttt{ArrayList}, \texttt{HashSet}\}$

produces an instance of a unique class from $\{\texttt{ArrayListIterator}, \texttt{HashSetIterator}\}$

We can do this by applying the universal and uniqueness quantifiers, for example:

$$\left(\begin{array}{c}\forall x \in \texttt{iterator()} \otimes \{\texttt{ArrayList}, \texttt{HashSet}\} \bullet \\ \exists ! y \in \{\texttt{ArrayListIterator}, \texttt{HashSetIterator}\} \bullet \\ \mathit{Produce}\,(x, y)\end{array}\right)$$

$$\wedge$$

$$\left(\begin{array}{c}\forall x \in \{\texttt{ArrayListIterator}, \texttt{HashSetIterator}\} \bullet \\ \exists ! y \in \texttt{iterator()} \otimes \{\texttt{ArrayList}, \texttt{HashSet}\} \bullet \\ \mathit{Produce}\,(x, y)\end{array}\right)$$

This is already quite complex, and as it is a proposition that would appear regularly it is a candidate for abstracting into a relation. However, before we do so, we should also consider abstract methods in the same way we did for the Tot predicate. Let us introduce the interfaces at the root of the collection and iterator hierarchies in Java:

$$\texttt{Collection}, \texttt{Iterator} : \mathbb{CLASS}$$

$$\mathit{Inherit}\,(\texttt{ArrayList}, \texttt{Collection})$$

$$\mathit{Inherit}\,(\texttt{HashSet}, \texttt{Collection})$$

$$\mathit{Inherit}\,(\texttt{ArrayListIterator}, \texttt{Iterator})$$

$$\mathit{Inherit}\,(\texttt{HashSetIterator}, \texttt{Iterator})$$

$$\mathit{Abstract}\,(\texttt{Collection})$$

$$\mathit{Abstract}\,(\texttt{Iterator})$$

$$\mathit{Abstract}\,(\texttt{iterator()} \otimes \texttt{Collection})$$

With this we introduce the appropriate hierarchies:

$$\begin{array}{rcl}\{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\} & : & \mathbb{HIERARCHY} \\ \{\texttt{Iterator}, \texttt{ArrayListIterator}, \texttt{HashSetIterator}\} & : & \mathbb{HIERARCHY}\end{array}$$

If we submit these hierarchies in place of their respective counterparts in our proposition it will no longer hold. This is the case for two reasons. The first reason is the same as experienced for the Tot predicate. That is, the method `iterator()` $\otimes$ `Collection` is abstract, and therefore cannot produce any instances. We can resolve this in the same way as we did for the Tot predicate, by using the *AbsMthOf* relation we previously defined. Therefore, we can reformulate our formalism as:

$$\left(\begin{array}{c} \forall x \in \texttt{iterator()} \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\} \bullet \\ \exists! y \in \{\texttt{Iterator}, \texttt{ArrayListIterator}, \texttt{HashSetIterator}\} \bullet \\ \mathit{AbsMthOf}\,(x, \texttt{iterator()} \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\}) \vee \mathit{Produce}\,(x, y) \end{array}\right)$$

$$\wedge$$

$$\left(\begin{array}{c} \forall x \in \{\texttt{Iterator}, \texttt{ArrayListIterator}, \texttt{HashSetIterator}\} \bullet \\ \exists! y \in \texttt{iterator()} \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\} \bullet \\ \mathit{Produce}\,(x, y) \end{array}\right)$$

Our second problem is similar to our first. The iterator hierarchy contains the abstract `Iterator` class, the root of the hierarchy and the mirror image of the abstract `Collection` class. In the second case of our proposition, we state that for *all* of these iterator classes there must exist exactly one method that produces it, but this is not the case. As in Tot predicate, dependencies between clans and hierarchies of this sort are not only common, but are also indicative of good object-oriented design. Therefore, to accommodate for this we must allow for the exclusion of abstract classes in this case. To do so we introduce a mirror image of the *AbsMthOf* relation for classes[80]:

AbsClsOf

$$\begin{array}{|l} a : \mathbb{CLASS} \\ b : \mathit{set}\,(\mathbb{CLASS}) \\ \hline \mathit{Abstract}\,(a) \wedge a \in b \wedge \exists x \in b \bullet \neg \mathit{Abstract}\,(x) \end{array}$$

[80] We could introduce a polymorphic abstraction on both the *AbsMthOf* and *AbsClsOf* relations, but it is not really that interesting to do so at this point.

And with this relation, it is easy to allow the exclusion of abstract classes as necessary:

$$\left(\begin{array}{c} \forall x \in \mathtt{iterator()} \otimes \{\mathtt{Collection}, \mathtt{ArrayList}, \mathtt{HashSet}\} \bullet \\ \exists! y \in \{\mathtt{Iterator}, \mathtt{ArrayListIterator}, \mathtt{HashSetIterator}\} \bullet \\ \mathit{AbsMthOf}\left(x, \mathtt{iterator()} \otimes \{\mathtt{Collection}, \mathtt{ArrayList}, \mathtt{HashSet}\}\right) \vee \mathit{Produce}\left(x, y\right) \end{array}\right)$$

$$\wedge$$

$$\left(\begin{array}{c} \forall y \in \{\mathtt{Iterator}, \mathtt{ArrayListIterator}, \mathtt{HashSetIterator}\} \bullet \\ \exists! x \in \mathtt{iterator()} \otimes \{\mathtt{Collection}, \mathtt{ArrayList}, \mathtt{HashSet}\} \bullet \\ \mathit{AbsClsOf}\left(y, \{\mathtt{Iterator}, \mathtt{ArrayListIterator}, \mathtt{HashSetIterator}\}\right) \vee \mathit{Produce}\left(x, y\right) \end{array}\right)$$

But this proposition is now even more complex than the one we started with. What is worse is it will only get more complex in cases of nested sets. To simplify this, we define a polymorphic relation using recursive schema:

Iso

$$A, B : \mathcal{U},\ A' : \mathit{SetOf}\left(A\right),\ B' : \mathit{SetOf}\left(B\right)$$
$$R : \mathit{schema}\left(A \times B\right),\ a : A',\ b : B'$$

$$\left(A = A' \wedge B = B' \implies R\left(a, b\right)\right)$$

$$\wedge$$

$$\left(\begin{array}{c} A \neq A' \wedge B \neq B' \implies \exists X : \mathit{SetOf}\left(A\right) \bullet \exists Y : \mathit{SetOf}\left(B\right) \bullet \\ A' = \mathit{Set}\left(X\right) \wedge B' = \mathit{set}\left(Y\right) \wedge a \neq \varnothing_X \wedge b \neq \varnothing_Y \wedge \\ \left(\begin{array}{c} \forall x \in a \bullet \left(X = \mathbb{METHOD} \implies \mathit{AbsMthOf}\left(x, a\right)\right) \vee \\ \exists! y \in b \bullet \textsc{Iso}\left(A, B, X, Y, R, x, y\right) \end{array}\right) \wedge \\ \left(\begin{array}{c} \forall y \in b \bullet \left(Y = \mathbb{CLASS} \implies \mathit{AbsClsOf}\left(y, b\right)\right) \vee \\ \exists! x \in a \bullet \textsc{Iso}\left(A, B, X, Y, R, x, y\right) \end{array}\right) \end{array}\right)$$

$$\wedge$$

$$\left(A \neq A' \wedge B = B' \implies \Omega\right)$$

$$\wedge$$

$$\left(A = A' \wedge B \neq B' \implies \Omega\right)$$

with the side condition that R must be one of the relations we have introduced in this chapter, such as *Inherit* and *Aggregate*. We abstract from the type arguments $A, B : \mathcal{U}$,

$A' : \mathit{SetOf}\,(A)$, and $B' : \mathit{SetOf}\,(B)$ when these are recoverable from the context, therefore we use the shorthand relation Iso: $\mathit{schema}\,(\mathit{schema}\,(A \times B) \times A' \times B')$. Note that this specification is similarly structured to the superimposition ($\otimes$) relation presented in §7.4. The proof that the Iso specification type checks (i.e. it is always a proposition under appropriately typed arguments) therefore follows the same pattern as the type checking proof for $\otimes$. Therefore we conclude that little would be added by repeating the proof.

The structure of the Iso predicate forms four mutually exclusive cases. The first deals with simple arguments where no set terms are involved, and the second accommodates nested sets. The remaining two ensure all cases are covered: they are error cases for invalid arguments that simply result in contradiction. Each case appropriately deals with empty sets, abstract methods and classes, and captures the relationship we discussed at the start of this subsection. Additionally, the definition of our Iso predicate is identical in intent to that of the Isomorphic predicate in LePUS3 [Eden and Nicholson, 2011] (see LePUS3 Definition XIII in Appendix B). However, aside from the conversion to Typed Predicate Logic (**TPL**), the definitions are not quite identical. As with All and Tot, we have added the constraint that sets cannot be empty. This has always been an implicit assumption in LePUS3, which we simply make explicit in our definition. We also ensure that at least one valid non-abstract method, or at least one valid non-abstract class, exists in the relevant sets. This requirement exists to a degree in LePUS3, but is not made explicit.

We are now able to succinctly articulate this relationship between the aforementioned sets, i.e.:

$$\textsc{Iso}\left(\begin{array}{c} \mathit{Produce}, \\ \texttt{iterator()} \otimes \{\texttt{Collection}, \texttt{ArrayList}, \texttt{HashSet}\}, \\ \{\texttt{Iterator}, \texttt{ArrayListIterator}, \texttt{HashSetIterator}\} \end{array}\right)$$

In LePUS3 [Eden and Nicholson, 2011], it has always been stated that a formula involving the Isomorphic predicate can always be weakened to one that involve the Total predicate. In LePUS3, the proof of this is said to follow directly from the definitions of each predicate [Eden and Nicholson, 2011]. We will be a little more explicit in our proof:

$$\textbf{Proposition 4}\quad \frac{\begin{array}{c} A, B : \mathcal{U} \quad A' : \mathit{SetOf}\,(A) \quad B' : \mathit{SetOf}\,(B) \\ R : \mathit{Schema}\,(A \times B) \quad a : A' \quad b : B' \end{array}}{\textsc{Iso}\,(A, B, A', B', R, a, b) \implies \textsc{Tot}\,(A, B, A', B', R, a, b)}$$

Proof. By simultaneous induction on the number of applications of the *set* type constructor using the induction principles of the types $\mathit{SetOf}\,(A)$, and $\mathit{SetOf}\,(B)$.

Firstly, we observe the structure of Iso and Tot are both of the form:

$$(\phi_1 \implies \varphi_1) \wedge \ldots \wedge (\phi_n \implies \varphi_n)$$

Observe that each ϕ_k $(1 \leq k \leq n)$ is a series of (in)equality propositions joined by conjunction such that for any given arguments exactly one ϕ_k holds. That is, they are mutually exclusive and they cover all cases. Therefore, to show $\textsc{Iso}(A, B, A', B', R, a, b) \implies \textsc{Tot}(A, B, A', B', R, a, b)$ holds we are required to explicitly show that φ_k holds where ϕ_k is true in both Iso and Tot, and we may implicitly appeal to these observations and the rules of implication to know that all other $\phi_i \implies \varphi_i$ hold $(1 \leq i \leq n$ and $i \neq k)$.

Given these observations, we continue our proof with the base case as follows:

Base Case: Here we are required to prove that the following holds:

$$\frac{\begin{array}{c} A, B : \mathcal{U} \quad A : \mathit{SetOf}\,(A) \quad B : \mathit{SetOf}\,(B) \\ R : \mathit{Schema}\,(A \times B) \quad x : A \quad y : B \end{array}}{\textsc{Iso}\,(A, B, A, B, R, x, y) \implies \textsc{Tot}\,(A, B, A, B, x, y)}$$

By our discussion above, we need to prove that the following judgment holds in this context:

$$R\,(x, y) \implies R\,(x, y)$$

which holds immediately.

Inductive Step: We begin the inductive step by making the inductive assumption that the following holds:

$$\frac{\begin{array}{c} A, B : \mathcal{U} \quad X : \mathit{SetOf}\,(A) \quad Y : \mathit{SetOf}\,(B) \\ R : \mathit{Schema}\,(A \times B) \quad x : X \quad y : Y \end{array}}{\textsc{Iso}\,(A, B, X, Y, R, x, y) \implies \textsc{Tot}\,(A, B, X, Y, R, x, y)}$$

We refer to this assumption in the remaining derivations as *ind.* Given this we are required to prove each of the following:

1. We are required to prove:

$$\frac{\begin{array}{c} A, B : \mathcal{U} \quad \mathit{set}\,(X) : \mathit{SetOf}\,(A) \quad Y : \mathit{SetOf}\,(B) \\ R : \mathit{Schema}\,(A \times B) \quad v : \mathit{set}\,(X) \quad y : Y \end{array}}{\textsc{Iso}\,(A, B, \mathit{set}\,(X)\,, Y, R, v, y) \implies \textsc{Tot}\,(A, B, \mathit{set}\,(X)\,, Y, R, v, y)}$$

By our earlier discussion, we must prove the following in this context[81]:

$$\Omega \implies \textsc{Tot}\left(A, B, \mathit{set}\left(X\right), Y, R, v, y\right)$$

which is immediate by the rules of implication.

2. We are required to prove:

$$\frac{\begin{array}{c} A, B : \mathcal{U} \quad X : \mathit{SetOf}\left(A\right) \quad \mathit{set}\left(Y\right) : \mathit{SetOf}\left(B\right) \\ R : \mathit{Schema}\left(A \times B\right) \quad x : X \quad w : \mathit{set}\left(Y\right) \end{array}}{\textsc{Iso}\left(A, B, X, \mathit{set}\left(Y\right), R, x, w\right) \implies \textsc{Tot}\left(A, B, X, \mathit{set}\left(Y\right), R, x, w\right)}$$

By our earlier discussion, we must prove the following in this context:

$$\Omega \implies \textsc{Tot}\left(A, B, X, \mathit{set}\left(Y\right), R, x, w\right)$$

which is immediate by the rules of implication.

3. We are required to prove:

$$\frac{\begin{array}{c} A, B : \mathcal{U} \quad \mathit{set}\left(X\right) : \mathit{SetOf}\left(A\right) \quad \mathit{set}\left(Y\right) : \mathit{SetOf}\left(B\right) \\ R : \mathit{Schema}\left(A \times B\right) \quad v : \mathit{set}\left(X\right) \quad w : \mathit{set}\left(Y\right) \end{array}}{\textsc{Iso}\left(A, B, \mathit{set}\left(X\right), \mathit{set}\left(Y\right), R, v, w\right) \implies \textsc{Tot}\left(A, B, \mathit{set}\left(X\right), \mathit{set}\left(Y\right), R, v, w\right)}$$

where, by our earlier discussion, we must show that the appropriate proposition in the Iso predicate implies the associated one in Tot. To present our argument clearly, succinctly, and within the confines of the page, we choose to explain our reasoning rather than providing a derivation proof. We begin with the proposition from $\textsc{Iso}(A, B, \mathit{set}\left(X\right), \mathit{set}\left(Y\right), R, v, w)$:

$$\begin{array}{c} \exists X' : \mathit{SetOf}\left(A\right) \bullet \exists Y' : \mathit{SetOf}\left(B\right) \bullet \\ \mathit{Set}\left(X\right) = \mathit{Set}\left(X'\right) \wedge \mathit{Set}\left(Y\right) = \mathit{set}\left(Y'\right) \wedge v \neq \varnothing_{X'} \wedge w \neq \varnothing_{Y'} \wedge \\ \left(\begin{array}{c} \forall x \in v \bullet \left(X' = \mathbb{METHOD} \implies \mathit{AbsMthOf}\left(x, v\right)\right) \vee \\ \exists! y \in w \bullet \textsc{Iso}\left(A, B, X', Y', R, x, y\right) \end{array}\right) \wedge \\ \left(\begin{array}{c} \forall y \in w \bullet \left(Y' = \mathbb{CLASS} \implies \mathit{AbsClsOf}\left(y, w\right)\right) \vee \\ \exists! x \in v \bullet \textsc{Iso}\left(A, B, X', Y', R, x, y\right) \end{array}\right) \end{array}$$

[81] We do not expand on the proposition of Tot as it irrelevant to do so given the nature of what is to be proved.

to which we use the elimination rules for conjunction ($\mathbf{L}_{10}$ and $\mathbf{L}_{11}$, p.46) to obtain:

$$\begin{gathered}\exists X' : \mathit{SetOf}\,(A) \bullet \exists Y' : \mathit{SetOf}\,(B) \bullet \mathit{Set}\,(X) = \mathit{Set}\,(X') \wedge \mathit{Set}\,(Y) = \mathit{set}\,(Y') \wedge v \neq \varnothing_{X'} \wedge \\ (\forall x \in v \bullet (X' = \mathbb{METHOD} \implies \mathit{AbsMthOf}\,(x, v)) \vee \exists! y \in w \bullet \textsc{Iso}\,(A, B, X', Y', R, x, y))\end{gathered}$$

The last quantification in this proposition is a uniqueness quantification, which we may weaken to an existential quantification, to obtain:

$$\begin{gathered}\exists X' : \mathit{SetOf}\,(A) \bullet \exists Y' : \mathit{SetOf}\,(B) \bullet \mathit{Set}\,(X) = \mathit{Set}\,(X') \wedge \mathit{Set}\,(Y) = \mathit{set}\,(Y') \wedge v \neq \varnothing_{X'} \wedge \\ (\forall x \in v \bullet (X' = \mathbb{METHOD} \implies \mathit{AbsMthOf}\,(x, v)) \vee \exists y \in w \bullet \textsc{Iso}\,(A, B, X', Y', R, x, y))\end{gathered}$$

And finally, by our inductive assumption *ind*, we may replace $\textsc{Iso}(A, B, X', Y', R, x, y)$ by $\textsc{Tot}(A, B, X', Y', R, x, y)$ to obtain:

$$\begin{gathered}\exists X' : \mathit{SetOf}\,(A) \bullet \exists Y' : \mathit{SetOf}\,(B) \bullet \mathit{Set}\,(X) = \mathit{set}\,(X') \wedge \mathit{Set}\,(Y) = \mathit{set}\,(Y') \wedge v \neq \varnothing_{X'} \wedge \\ \forall x \in v \bullet (X' = \mathbb{METHOD} \implies \mathit{AbsMthOf}\,(x, v)) \vee \exists y \in w \bullet \textsc{Tot}\,(A, B, X', Y', R, x, y)\end{gathered}$$

which is the proposition we are required to derive from Tot, therefore we conclude that this case holds.

Given the steps presented above we conclude that:

$$\frac{\begin{gathered}A, B : \mathcal{U} \quad A' : \mathit{SetOf}\,(A) \quad B' : \mathit{SetOf}\,(B) \\ R : \mathit{Schema}\,(A \times B) \quad a : A' \quad b : B'\end{gathered}}{\textsc{Iso}\,(A, B, A', B', R, a, b) \implies \textsc{Tot}\,(A, B, A', B', R, a, b)}$$

■

This concludes our construction of our theory of classes based on the theoretical foundations of LePUS3 (Appendix B). Our theory is more rigorous and expressive than lePUS3, having formalized more interesting properties of object-oriented design than was contained in the definitions, axioms, and intuitions of LePUS3. In the next chapter we investigate how we may use this theory to specify the motifs of design patterns.

Chapter 8

Patterns

8.1 A Selection of Motifs from the "Gang of Four" Catalogue

In §2.2 we discussed design patterns with specific attention to their commonly formalized component: the design motif. It is the formalization of the design motif that we focus on here. We demonstrate how our Theory of Classes (**TC**) can be applied to the formal specification of motifs through examples drawn from the most prominent collection of object-oriented patterns [Gamma et al., 1994]. For each motif we provide an informal summary of the given motif, paraphrased from [Gamma et al., 1994], briefly discuss how the motif should be formalized, and a formal definition written in **TC**.

Indeed, LePUS3 had already been used to specify these motifs, which are originally published in [Eden et al., 2007a] and then reiterated in [Eden and Nicholson, 2011]. The formalizations of the motifs presented here extensions and refinements of those previously published. **TC** is a much more expressive theory than that of LePUS3, which was limited to only set membership and implicit conjunction. We now have the full range of propositions provided to us by **TC**. For example, we have propositions for equality, negation, disjunction, union, intersection, and quantification.

As we reviewed the existing motifs and their LePUS3 specifications, we found that the additional propositions allowed us to articulate more about each motif than was previously possible. For example, in the Object Adapter motif (§8.1.4) it is important to specify that the adaptor, and the class being adapted, are not the same class. If the adapter and adaptee were the same class then the motif specified is that of an anti-pattern. We are also able to do more with the specifications we define. For example, in §8.1.2 we demonstrate what is called *schema hiding* in the Z specification language. That is, we are able to hide variables by moving them from the declaration part of the specification, to the proposition part using an existential quantifier. Lastly, we also completely redefined the State motif, and in doing so noticed several similarities with the Strategy (§8.1.3) motif. This lead us to the hypothesis that the Strategy motif is as specialization of the State motif. In the next section (§8.2) we formalize and prove this hypothesis.

8.1.1 Formalizing the Composite motif

The first motif we consider is the Composite motif. This is a logical place to begin as it is the motif that has received quite some attention in LePUS3 [Nicholson et al., 2009, Eden and Nicholson, 2011]. The intent of the Composite is to allow composition of objects into tree structures that represent part-whole hierarchies. The participants, responsibilities and collaborations can be summarized as:

Component Declares a basic interface for all interaction. Implements default behaviour for all components.

Leaves Classes with no children, implements/extends superclass behaviour as appropriate, and handles requests directly.

Composite Declares basic branch interface. Implements default behaviour specific to components with children. Composite objects usually forward requests to each of its children, possibly performing additional operations before and/or after forwarding.

One approach to modelling this motif is to put all the participant classes into a single hierarchy with Component at the root. However, although this does capture the fact that all the classes are related by inheritance, and we can capture that they share a common interface, it is too abstract a representation. Instead we take more concrete view of the motif. We represent the Component, Composite, and Leaves by a class, another class, and a set of classes respectively. We also add the appropriate inheritance relationships, and an aggregation between Composite and Component. This is a good abstraction as we do not need to specify what data structure should be used for this collection (arrays, sets, maps, etc.) or how it should be implemented (data membership, inheritance, etc.).

There are other important relationships between these participant classes that we were previously unable to specify in LePUS3. For example, it stands to reason that the component class is not a leaf itself, as it is an abstraction from any given leaf or composite class. Neither can the component and composite classes be equal, as this would make all leaves composite classes. Similarly, the interface defined for all components is not the same as the interface defined for branches in the part-whole hierarchy. Although these requirements appear to be only minor additions, they are very important to help capture the motif of the pattern, and not the motif of an anti-pattern.

Focusing now on the methods, we are not interested on exactly what the implemented behaviour of the Component interface is for the Leaves, nor for the branch interface in Composite. All we do want to say about them is that they exist, and leave their definition as open to interpretation as possible. We do, however, want to say something about the relationship between the component interface on the Composite and Component classes. As stated, the Composite class forwards method invocations to its children, instances of the Component class. This is nicely captured by our Iso predicate, which captures the one-to-one and onto nature of the forwarding relationship, without restricting these methods from performing other actions. That is, this tribe in the Composite class can call any other method in the program, as long as it does not forward its method invocation to them and thereby break the required isomorphism.

We can articulate these participants and their relations in the following specification:

Composite

$$
\begin{aligned}
&component, composite : \mathbb{CLASS}\\
&Leaves : set\,(\mathbb{CLASS})\\
&ComponentOPs, CompositeOPs : set\,(\mathbb{SIGNATURE})
\end{aligned}
$$

$$
\begin{gathered}
composite \notin Leaves \wedge\\
composite \neq component\\
ComponentOPs \cap CompositeOPs = \varnothing \wedge\\
Method\,(composite, ComposOPs) \wedge\\
Method\,(Leaves, ComponentOPs) \wedge\\
Inherit\,(composite, component) \wedge\\
\textsc{Tot}\,(Inherit, Leaves, component) \wedge\\
Aggregate\,(composite, component) \wedge\\
\textsc{Iso}\,(Forward, ComponentOPs \otimes composite, ComponentOPs \otimes component)
\end{gathered}
$$

As in our other publications, such as [Nicholson et al., 2009] and [Eden and Nicholson, 2011], we hypothesize that this specification holds for specific classes and methods in the `java.awt` package. We return to this hypothesis in §12.1 where we formalize and show that it holds.

8.1.2 Formalizing the Factory Method and Abstract Factory motifs

The Factory Method and the Abstract Factory are very similar motifs. Both decouple specific implementations from their method of instantiation, but at different levels of abstraction. We will discuss each of these motifs in turn, beginning with the Factory Method motif. The intent of this motif is to define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. The participants, responsibilities and collaborations can be summarized as:

AbstractFactory Declares the factory method, which returns an object of type Product. Factory may also define a default implementation of the factory method that returns a default ConcreteProduct object, but generally relies on its subclasses to define a factory method that returns an instance of the appropriate ConcreteProduct.

ConcreteFactory Overrides the factory method to return an instance of a ConcreteProduct.

AbstractProduct Defines the interface of objects the factory method creates.

ConcreteProduct Implements the Product interface. There is one ConcreteProduct for each ConcreteFactory class.

It is logical that the abstract and concrete factory and product classes constitute inheritance hierarchies. The factory hierarchy has a common interface of at least one factory method, which is defined in the root class (AbstractFactory), and implemented in each ConcreteFactory class. For a specific ConcreteFactory, the factory method *produces* an instance of the appropriate ConcreteProduct class. The Iso predicate is the most appropriate mechanism of representing this one-to-one relationship between factory methods and products. Indeed, this is the very situation that this predicate is designed to capture.

We can articulate these participants and their relations in the following simple and elegant specification:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FactoryMethod}} \\
\hline
\textit{Factories} : \mathbb{HIERARCHY} \\
\textit{Products} : \mathbb{HIERARCHY} \\
\textit{FactoryMethod} : \mathbb{SIGNATURE} \\
\hline
\textsc{Iso}\left(\textit{Produce}, \textit{FactoryMethod} \otimes \textit{Factories}, \textit{Products}\right) \\
\hline
\end{array}
$$

As an interesting additional point, consider that we wanted to introduce a subtype of $\mathbb{HIERARCHY}$ where all terms of which are factories. This is very easy in our theory, as we can hide the unwanted declarations with existential quantification. As such this is a form of *schema hiding* in the *Z* specification language. As an example of this, let us define a new specification that abstracts from the *Factories* and *Products* arguments[82]:

$$
\begin{array}{|c}
\multicolumn{1}{l}{\textit{Factory}} \\
\hline
\textit{Factories} : \mathbb{HIERARCHY} \\
\hline
\exists \textit{Products} : \mathbb{HIERARCHY} \bullet \\
\exists \textit{FactoryMethod} : \mathbb{SIGNATURE} \bullet \\
\textit{FactoryMethod}\left(\textit{Factories}, \textit{Products}, \textit{FactoryMethod}\right) \\
\hline
\end{array}
$$

which we can turn into a type with the following simple statement:

$$\mathbb{FACTORY} \triangleq \{f : \mathbb{HIERARCHY} | \textit{Factory}\,(f)\}$$

Using the relations that specifications introduce in this way is unachievable in LePUS3. It simply does not have the expressive power and flexibility that our theory of classes has.

[82] We could define this directly as a subtype, but its more interesting to take this route as it also illustrates how we may abstract further from our motif specifications.

Similar to the Factory Method motif is the Abstract Factory. Informally, we reason that the Abstract Factory motif is an abstraction of the Factory Method [Eden and Nicholson, 2011]. We will return to this in the next section when we formalize relationships between relations (§8.2).

The intent of the Abstract Factory is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. The participants, responsibilities and collaborations are very similar to the Factory Method, and can be summarized as follows:

AbstractFactory Declares an interface for operations that create abstract product objects, defers (via dynamic binding) creation of product objects to its ConcreteFactory subclass

ConcreteFactory Implements the operations to create concrete product objects.

AbstractProduct Declares an interface for a type of product object.

ConcreteProduct Defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface. There is one ConcreteFactory for each kind of AbstractProduct.

It is logical that the abstract and concrete factory classes constitute an inheritance hierarchy. This hierarchy has a common interface defined in the root class (AbstractFactory), and implemented in each ConcreteFactory class. For a specific ConcreteFactory, each method in its tribe (the interface) *produces* instances of appropriate ConcreteProduct class. Therefore, there is not just a single hierarchy of products, but a set of product hierarchies; one for each ConcreteFactory class. Again, the Iso predicate is the most appropriate mechanism of representing these requirements.

Therefore, we can articulate these participants and their relations in the following specification:

$AbstractFactory$

$$Factories : \mathbb{HIERARCHY}$$
$$PRODUCTS : set\,(\mathbb{HIERARCHY})$$
$$FactoryMethods : set\,(\mathbb{SIGNATURE})$$

$$\textsc{Iso}\,(Produce, FactoryMethods \otimes Factories, PRODUCTS)$$

8.1.3 Formalizing the State and Strategy motifs

The next motif to examine is the State motif, the intent of which is to allow an object to alter its behaviour when its internal state changes. The object will appear to change its class. The participants, responsibilities and collaborations can be summarized as:

Context Defines the interface of interest to clients, and maintains an instance of a ConcreteState subclass that defines the current state. Context delegates state-specific requests to the current concrete State object. A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary. Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients do not have to deal with the State objects directly. Either Context or the concrete State subclasses can decide which state succeeds another and under what circumstances.

State Defines an interface for encapsulating the behaviour associated with a particular state of the Context.

ConcreteState Each subclass implements a behaviour associated with a state of the Context.

As with the factories, it stands to reason that there is a hierarchy of states. The state classes have a common interface, a tribe of methods, with which each state implements its unique behaviours. We do not know what those behaviours may be, so we are only able to specify that the interface exists. The Context class is not a state, but has a data member of an instance of one of the state classes. It is logical that this data member is of the class State itself, but we are making an abstraction and treating all states as a single hierarchy term to give an overall impression of the motifs structure.

The context has its own interface that in some way interacts with the interface of the states. This interaction does not necessarily have any structure to it, a method in the context may interact with several of the methods in the state's interface. Therefore, the Tot predicate is the most appropriate in this instance.

We can articulate these participants and their relations in the following specification:

State

$context : \mathbb{CLASS}$

$States : \mathbb{HIERARCHY}$

$Requests, Interface : set\,(\mathbb{SIGNATURE})$

$$\begin{gathered} context \notin States \wedge \\ \textsc{Tot}\,(Member, context, States) \wedge \\ \textsc{Tot}\,(Call, Requests \otimes context, Interface \otimes States) \end{gathered}$$

This simple specification tells us how an implementation of the State motif should be structured. Our theory does not include behavioural aspects of the motif, such as how it is decided which state follows another. Despite our inability to capture some aspects of the motif, what we do capture is interesting enough at our level of abstraction. Indeed, once we defined this specification, we noticed striking similarities between it and the Strategy motif. We contend that the Strategy motif is a special case of the State motif, at least at the structural level. To this end, we formalize the Strategy motif so that we may formalize and prove this hypothesis in the next section (§8.2).

We move on now to the Strategy motif, which is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. The participants, responsibilities and collaborations can be summarized as:

Context Maintains a reference to a Strategy object, and may define an interface that lets a Strategy access its data. Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.

Strategy Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy Implements the algorithm using the Strategy interface. There is often a family of ConcreteStrategy classes for a client to choose from.

As in the State motif, it is reasonable to capture these participants as a hierarchy of strategies, and a context class. The context, which should not be a strategy itself, has a data member of one of the strategy classes. The Context has an interface through which clients can communicate with it. These methods interact with the interface, a tribe of methods, in the strategies. The important difference here is that the Strategy motif requires that the ConcreteStrategy classes have a mechanism of accessing the interface of the Context.

We can articulate these participants and their relations in the following specification:

Strategy

$$
\begin{array}{l}
\mathit{context} : \mathbb{CLASS} \\
\mathit{Strategies} : \mathbb{HIERARCHY} \\
\mathit{Requests}, \mathit{Interface}, \mathit{CallBacks}, \mathit{GetData} : \mathit{set}\,(\mathbb{SIGNATURE})
\end{array}
$$

$$
\begin{array}{c}
\mathit{context} \notin \mathit{Strategies} \wedge \\
\textsc{Tot}\,(\mathit{Member}, \mathit{context}, \mathit{Strategies}) \wedge \\
\textsc{Tot}\,(\mathit{Call}, \mathit{Requests} \otimes \mathit{context}, \mathit{Interface} \otimes \mathit{Strategies}) \wedge \\
\textsc{Tot}\,(\mathit{Call}, \mathit{CallBacks} \otimes \mathit{Strategies}, \mathit{GetData} \otimes \mathit{context})
\end{array}
$$

To say that the Strategy motif is a special case of the State should be quite clear from this specification. Indeed, it makes sense from their intents too. In both motifs, the behaviour of the context changes depending on the current state/strategy being employed. The main difference here is that the strategies must have a mechanism for callbacks to its context, which is not essential states.

8.1.4 Formalizing the Object Adapter and Proxy motifs

Our final examples are the Object Adapter and Proxy motifs. Starting with the Object Adapter, its intent is to convert the interface of a class into another interface that the clients expect. Object Adapter lets classes work together that could not otherwise because of incompatible interfaces. The participants, responsibilities and collaborations can be summarized as:

Target Defines the domain-specific interface that Client uses. Adapter inherits from Target. When clients calls operations on an Adapter instance, these in turn calls the Adaptee operations that carry out the request.

Adapter Adapts the interface of Adaptee to the Target interface.

Adaptee Defines an existing interface that needs adapting.

In this motif, the Target and the Adapter could form a hierarchy, but that is too abstract a picture for our purposes. Instead, we will capture these participants as single classes, with the relevant relationships. Namely, the Target is abstract, it only defines a request interface for the client. The Adapter implements (inherits from) the Target class. Additionally, the Adapter and Adaptee classes cannot be the same class as otherwise there is no need for the Object Adaptor motif. However, although implied in [Gamma et al., 1994] that it is generally the case, we do not believe there is any requirement for the Target and Adapter classes to be disctinct. That is, we do not require their inequality as the Target class itself might adapt another class's interface.

The Adapter must have a data member of class Adaptee[83]. The Adaptor's interface is then implemented by calling methods in the interface of the Adaptee, i.e. the Adapter *wraps* the Adaptee. It does not matter how these calls are structured, only that all of the indicated methods relies in some way on the Adaptee's interface. Therefore, this requirement is best captured with the Tot predicate.

[83] In the Class Adapter, this would be an inheritance relationship.

We can articulate these participants and their relations in the following specification:

ObjectAdapter

$$target, adapter, adaptee : \mathbb{CLASS}$$
$$Requests, AdaptedRequests : set\left(\mathbb{SIGNATURE}\right)$$

$$\begin{gathered} adapter \neq adaptee \wedge \\ Abstract\left(target\right) \wedge \\ Method\left(target, Requests\right) \wedge \\ Member\left(adapter, adaptee\right) \wedge \\ Inherit\left(adapter, target\right) \wedge \\ \textsc{Tot}\left(Call, Requests \otimes adapter, AdaptedRequests \otimes adaptee\right) \end{gathered}$$

As we redefined these specifications from LePUS3 into our theory of classes, we noticed that there are quite some similarities between the Object Adapter and the Proxy motifs. More concretely, we believe that the Proxy is a special case of the Object Adapter as articulated within our theory. We will return to this notion in the next section (§8.2), where we formally express and prove this hypothesis.

Before we do so, we must discuss and formulate the Proxy motif, the intent of which is to provide a surrogate or placeholder for another object to control access to it. The participants, responsibilities and collaborations can be summarized as:

Proxy Maintains a reference that lets the proxy access the RealSubject. Provides an interface identical to Subject's so that a proxy can by substituted for the real subject. Proxy forwards requests to RealSubject when appropriate, and controls access to the real subject. It may also be responsible for creating and deleting it. Responsibilities of the proxy depend on the kind of proxy it is (abstracted)

Subject Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject Defines the real object that the proxy represents.

In this case, the Proxy, Subject, and RealSubject classes could form a hierarchy. But again this is abstract a representation, therefore we capture each of these participants as classes with the appropriate inheritance relationships. As in the Object Adaptor, the Proxy and RealSubject classes must not be equal, as a proxy cannot proxy itself. The Subject is abstract, and declares a common interface for the Proxy and RealSubject classes. A Proxy also stores an instance (a data member) of the RealSubject class. Instantiating the RealSubject might be expensive, so this action may be left only until absolutely necessary. They may also perform additional operations in support of the RealSubject instance, such caching information that may be expensive to get directly from the RealSubject. As the way in which the RealSubject is used is so varied, we do not specify any specific details about it. However, we do require that the Proxy has the same interface, and that each of those methods should interact (call) the interface methods in the RealSubject. They may not necessarily forward their method calls, and neither must they only call just their respective method in the RealSubject's interface. For example, consider a method m in the RealSubject, which implements an expensive algorithm of some sort that could take hours to complete. We do not want to execute this method unless we have to. We would therefore perform less expensive operations on the RealSubject, maybe calling other methods in its interface, to see if executing method m is really necessary. The Iso is not appropriate in this case, as it is too strong a condition, and does not allow for the sort of flexibility we have just discussed. However, the Tot predicate suits our purposes perfectly.

We can articulate these participants and their relations in the following specification:

Proxy

$$\begin{aligned} &subject, proxy, realSubject : \mathbb{CLASS} \\ &Requests : set\left(\mathbb{SIGNATURE}\right) \end{aligned}$$

$$\begin{gathered} proxy \neq realSubject \wedge \\ Abstract\left(subject\right) \wedge \\ Method\left(subject, Requests\right) \wedge \\ Member\left(proxy, realSubject\right) \wedge \\ Inherit\left(proxy, subject\right) \wedge \\ Inherit\left(realSubject, subject\right) \wedge \\ \textsc{Tot}\left(Call, Requests \otimes proxy, Requests \otimes realSubject\right) \end{gathered}$$

We believe that the Proxy motif, as captured above, is a special case of the Object Adapter because their structures are so similar. For example, the Object Adapter does not require that the interface being adapted, and the target interface be the same, but this could be the case. An Adapter might take exactly the same interface as the Adaptee in cases where the Adaptee is provided by some external class library, and there is a need to mimic its functionality inside a local class hierarchy.

8.2 Reasoning about Motifs

Reasoning over LePUS3 specifications, represented by Codecharts, is done at an abstract level [Eden and Nicholson, 2011]. Reasoning is accomplished primarily through informal discussion, and any real detail is only ever provided at the meta-level. As such LePUS3 is very limited in its ability to reason on the relationship between specifications. None the less, it defines a *semantic entailment* relation ($\models$) between specifications. The definition of this relies on the relationship between specifications and programs. That is, given two specifications Ψ and Φ, $\Psi \models \Phi$ holds if and only if for every program that satisfies Ψ, that same program also satisfies Φ. The conclusion from this is that Φ is a more abstract specification than Ψ, or in other words that Ψ is a specialization of Φ. LePUS3 classifies four broad abstraction mechanisms that allow this relationship to be deduced.

Therefore, we can deduce that $\Psi \models \Phi$ holds if we can show Φ is the result of applying one or more of the following abstraction mechanisms to Ψ:

1. *Abstraction viz. Information Neglect*—The consistent removal of terms and/or relations.
2. *Abstraction viz. Weakening*[84]—Replacing one relation with a weaker, more general, one.
3. *Abstraction viz. Generalization*—The consistent replacement of a constant term with a new declared variable in the specification.
4. *Abstraction viz. Exponentiation*—The consistent replacement of a term t with another term that contains t as an element with appropriate modification to associated propositions.

However, each of these abstraction mechanisms are defined more informally than we would like. That is, they are defined as operations on the Codechart, the diagrammatic representation, rather than on the underlying specification. This does not mean that the abstraction mechanisms are wrong, far from it. But the way in which they are achieved has to be done at such an abstract level because the definition of LePUS3 does not allow them to be more formally defined elsewhere.

Our theory of classes, however, allows for each of these abstraction mechanisms to be defined directly within the theory. In fact, most are already derivable from the existing theory:

1. Since formulas in LePUS3 are only constructed through use of the conjunction, this is derivable from the conjunction elimination rules ($\mathbf{L}_{10-11}$, p.46).
2. This is derivable from many of the rules, proofs, etc. throughout the previous chapter.
3. This mechanism is derivable from the rule of substitution (**Sub**).
4. In our theory, this is replacing a term $t : T$ with $x : \mathit{set}\,(T)$ where $t \in x$, and replacing any $\phi\,[t]$ with an appropriate $\varphi\,[x]$ where $\varphi\,[x] \implies \phi\,[t]$. This is not always a simple case, and has to be proved in each individual case.

Therefore, the abstraction mechanisms really only classify which rules are used to prove the semantic entailment relationship holds, rather than being rules themselves. And as each of these abstraction mechanisms are expressible directly in our theory, we define our own semantic entailment relation within our theory:

[84] In [Eden and Nicholson, 2011] this only allows weakening of predicates, but that's a little restrictive since other relations an be weakened such as *Forward* can be weakened to *Call*. We modify our intuition of this mechanism to reflect this.

Definition 8 Let $R_1 : schema\,(A)$ and $R_2 : schema\,(B)$ be two relations. R_1 *semantically entails* R_2, written $R_1 \models R_2$, if and only if $\forall x : A \bullet R_1\,[x] \implies \exists y : B \bullet R_2\,[x, y]$

Defining semantic entailment in this way means that it is decoupled from programs and any definition of design verification (see Part III), unlike LePUS3. As such, our semantic entailment relation is more powerful and versatile, capable of articulating relationships that were always implied but otherwise inexpressible in LePUS3. For example, in rule $\mathbf{TC}_{24}$ (p.118) we stated that whenever a $Forward$ relation holds, then so to does a $Call$ relation with the same arguments. We can now rewrite this rule as the following semantic entailment proposition:

$$Forward \models Call$$

Another example is rule $\mathbf{TC}_{29}$ (p.120), which states that whenever the *Produce* relation holds, then so too do *Create* and *Produce* relations with the same arguments:

$$\begin{aligned} \textit{Produce} &\models \textit{Create} \\ \textit{Produce} &\models \textit{Return} \end{aligned}$$

Our final example comes from the proof presented at the end of §7.6.3, which tells us that the Iso predicate semantically entails the Tot predicate. That is, whenever a proposition involving the Iso predicate holds, it can always be weakened to one that uses the Tot predicate:

$$\textsc{Iso} \models \textsc{Tot}$$

This relationship also allows us to formalize the relationships between motifs that we hypothesized in the previous section. To reiterate, these are:

1. the Abstract Factory motif is an abstraction of the Factory Method motif;
2. the Strategy motif is a specialization of the State motif; and
3. the Proxy motif is a specialization of the Object Adapter motif.

Taking each in turn, we formulate the hypothesis as a proposition and present its proof:

Proposition 5 $FactoryMethod \models AbstractFactory$

Proof. By deduction on the definition of the relations $FactoryMethod$ and $AbstractFactory$. We are required to prove:

$$\frac{f : \mathbb{HIERARCHY} \quad p : \mathbb{HIERARCHY} \quad fm : \mathbb{SIGNATURE}}{FactoryMethod\,(f, p, fm) \implies AbstractFactory\,(f, \{p\}\,, \{fm\})}$$

which holds by the following derivation[85]:

$$\begin{array}{rll}
1) & f : \mathbb{HIERARCHY} & \text{premise} \\
2) & p : \mathbb{HIERARCHY} & \text{premise} \\
3) & fm : \mathbb{SIGNATURE} & \text{premise} \\
4) & \mid FactoryMethod\,(f, p, fm) & \text{assumption} \\
\hline
5) & \mid \textsc{Iso}(Produce, fm \otimes f, p) & \mathbf{R}_3\ 4 \\
6) & \mid \textsc{Iso}(Produce, \{fm \otimes f\}\,, \{p\}) & \textsc{Iso}\ \text{def}\ 5 \\
7) & \mid \textsc{Iso}(Produce, \{fm\} \otimes f, \{p\}) & \otimes\ \text{def}\ 6 \\
8) & \mid AbstractFactory\,(f, \{p\}\,, \{fm\}) & \mathbf{R}_2\ 1\text{–}3,7 \\
9) & FactoryMethod\,(f, p, fm) \implies AbstractFactory\,(f, \{p\}\,, \{fm\}) & \mathbf{L}_{18}\ 4,8
\end{array}$$

A few steps in this derivation deserve some commentary. Firstly, step 6 holds by observation that if the Iso predicate holds for some arguments, it will hold when those same arguments compose singleton sets. Step 7 is by observation on the way in which superimposition terms are unpacked (§7.4), that is $\{fm \otimes f\}$ and $\{fm\} \otimes f$ both yield the same set of set of methods. Finally, what we are required to prove follows immediately from step 9 where $x = \{p\}$ and $y = \{fm\}$. ■

The second hypothesis is that the Strategy motif is a specialization of the State motif, which we formalize as follows with another trivial sketched proof:

Proposition 6 $Strategy \models State$

Proof. By deduction on the definition of the relations $Strategy$ and $State$. We are required to prove:

$$\frac{\begin{array}{c} c : \mathbb{CLASS} \quad s : \mathbb{HIERARCHY} \quad r : set\,(\mathbb{SIGNATURE}) \quad i : set\,(\mathbb{SIGNATURE}) \\ cb : set\,(\mathbb{SIGNATURE}) \quad gd : set\,(\mathbb{SIGNATURE}) \end{array}}{Strategy\,(c, s, r, i, cb, gd) \implies State\,(c, s, r, i)}$$

[85] $\mathbf{L}_{18}$, p.46; $\mathbf{R}_{2-3}$, p.60

which holds by the following derivation[86]:

$$\begin{array}{rll}
1) & c : \mathbb{CLASS} & \text{premise} \\
2) & s : \mathbb{HIERARCHY} & \text{premise} \\
3) & r : \mathit{set}\,(\mathbb{SIGNATURE}) & \text{premise} \\
4) & i : \mathit{set}\,(\mathbb{SIGNATURE}) & \text{premise} \\
5) & cb : \mathit{set}\,(\mathbb{SIGNATURE}) & \text{premise} \\
6) & gd : \mathit{set}\,(\mathbb{SIGNATURE}) & \text{premise} \\
7) & \mid \underline{\mathit{Strategy}\,(c, s, r, i, cb, gd)} & \text{assumption} \\
8) & \mid c \notin s \wedge \textsc{Tot}\,(\mathit{Member}, c, s) \wedge \textsc{Tot}\,(\mathit{Call}, r \otimes c, i \otimes s) \wedge \textsc{Tot}\,(\mathit{Call}, cb \otimes s, gd \otimes c) & \mathbf{R}_3\ 7 \\
9) & \mid c \notin s \wedge \textsc{Tot}\,(\mathit{Member}, c, s) \wedge \textsc{Tot}\,(\mathit{Call}, r \otimes c, i \otimes s) & \mathbf{L}_{10}\ 8 \\
10) & \mid \mathit{State}\,(c, s, r, i) & \mathbf{R}_2\ 1\text{–}4{,}9 \\
11) & \mathit{Strategy}\,(c, s, r, i, cb, gd) \implies \mathit{State}\,(c, s, r, i) & \mathbf{L}_{18}\ 7{,}10
\end{array}$$

■

Moving onto our third and final hypothesis: that the Proxy motif is a specialization of the Object Adapter motif. We formalize this in the same way we have the others, and sketch its proof:

Proposition 7 $\mathit{Proxy} \models \mathit{ObjectAdapter}$

Proof. By deduction on the definition of the relations Proxy and $\mathit{ObjectAdaptor}$. We are required to prove:

$$\frac{s : \mathbb{CLASS} \quad p : \mathbb{CLASS} \quad rs : \mathbb{CLASS} \quad r : \mathit{set}\,(\mathbb{SIGNATURE})}{\mathit{Proxy}\,(s, p, rs, r) \implies \mathit{ObjectAdaptor}\,(s, p, rs, r, r)}$$

which holds by the following derivation[87]:

$$\begin{array}{rll}
1) & s : \mathbb{CLASS} & \text{premise} \\
2) & p : \mathbb{CLASS} & \text{premise} \\
3) & rs : \mathbb{CLASS} & \text{premise} \\
4) & r : \mathit{set}\,(\mathbb{SIGNATURE}) & \text{premise} \\
5) & \mid \underline{\mathit{Proxy}\,(s, p, rs, r)} & \text{assumption} \\
6) & \mid p \neq rs \wedge \mathit{Abstract}\,(s) \wedge \mathit{Method}\,(s, r) \wedge \mathit{Member}\,(p, rs) \wedge & \mathbf{R}_3\ 5 \\
 & \mid \quad \mathit{Inherit}\,(p, s) \wedge \mathit{Inherit}\,(rs, s) \wedge \textsc{Tot}\,(\mathit{Call}, r \otimes p, r \otimes rs) & \\
7) & \mid p \neq rs \wedge \mathit{Abstract}\,(s) \wedge \mathit{Method}\,(s, r) \wedge \mathit{Member}\,(p, rs) \wedge & \mathbf{L}_{9-11}\ 6 \\
 & \mid \quad \mathit{Inherit}\,(p, s) \wedge \textsc{Tot}\,(\mathit{Call}, r \otimes p, r \otimes rs) & \\
8) & \mid \mathit{ObjectAdaptor}\,(s, p, rs, r, r) & \mathbf{R}_2\ 1\text{–}4{,}7 \\
9) & \mathit{Proxy}\,(s, p, rs, r) \implies \mathit{ObjectAdaptor}\,(s, p, rs, r, r) & \mathbf{L}_{18}\ 5{,}8
\end{array}$$

where at step 7 we use conjunction elimination/introduction rules to drop the proposition $\mathit{Inherit}\,(rs, s)$. ■

[86] $\mathbf{L}_{10}$, p.46; $\mathbf{L}_{18}$, p.46; $\mathbf{R}_{2-3}$, p.60

[87] $\mathbf{L}_{9-11}$, p.46; $\mathbf{L}_{18}$, p.46; $\mathbf{R}_{2-3}$, p.60

This chapter by no means is an exhaustive discussion on all motifs and their relationships. It does, however, demonstrate how our theory of classes can be used to formalize enough interesting detail about each motif, and to formally reason about their relationships.

Chapter 9

Methods Refined

In Chapter 7 we defined our theory of classes with direct inspiration from LePUS3 [Eden and Nicholson, 2011]. Then, in Chapter 8, we demonstrated how this new theory is able to capture more details about design motifs than were previously expressible in LePUS3. In this chapter we begin to extend, or rather refine, our theory to allow us to capture an even greater level of object-oriented design detail. Specifically, we will refine our notion of method signatures.

A method signature is the name (identifier) and argument types (classes) of a method, which have so far been abstracted as atomic terms. This abstraction has served us well as in many motifs we had no need to specify either the identifier or arguments explicitly. However, there are cases where it is desirable to specify the exact method signature, or at least part of it. For example, let us consider the following specification for a very simple list class that is to contain instances of `Integer`[88]:

IntegerList

$\texttt{List}, \texttt{Integer}, \texttt{boolean} : \mathbb{CLASS}$
$\texttt{add(Integer)} : \mathbb{SIGNATURE}$

$$\mathit{Aggregate}\,(\texttt{List}, \texttt{Integer}) \wedge$$
$$\mathit{Return}\,(\texttt{add(Integer)} \otimes \texttt{List}, \texttt{boolean})$$

In this specification the signature `add(Integer)` is an atomic term. The fact that the string `Integer` appears within it does not mean that the class represented by `Integer` is actually one of the signature's argument types. We could equally represent this signature with `addMthSig` instead, and still specify the same level of detail. That is, the text used for the signature constant provides

[88] The constants here do not actually exist in our theory, but the reader should assume they do for the sake of the discussion. We will discuss how we can introduce constants in the context of a program in Part III. Additionally, we do not actually declare these constants, despite their appearance in the declaration part of the specification. This is a redundancy that helps us to be explicit, and mimics the approach taken in LePUS3.

implicit information about that signature's identifier and argument types, but these details are not actually being specified.

Consider then how we could abstract the above specification to one that specifies a list class of any class. We are easily able to abstract on the classes represented by `Integer` and `List`, using the rule of substitution (**Sub**), resulting in the following specification:

brokenList

$list, e, \texttt{boolean} : \mathbb{CLASS}$

$\texttt{add(Integer)} : \mathbb{SIGNATURE}$

$$\begin{aligned} &Aggregate\,(list, e) \wedge \\ &Return\,(\texttt{add(Integer)} \otimes list, \texttt{boolean}) \end{aligned}$$

Notice that we substituted `List` with variable *list*, and `Integer` with variable e, but we still have `add(Integer)`. It stands to reason that only a list of integers would have a method to add an integer. More abstractly, a list of x should have a method to add instances of class x. Therefore we need to appropriately abstract this signature, but due to the atomic nature of signature terms there is no way of abstracting only on the signature's argument types. Instead we will have to substitute the entire signature with a variable, and in doing so we would lose two important pieces of implicit information:

1. Every list class should have a method whose identifier is "add"
2. Every such method should take an argument of the same type as is being aggregated in the list class, i.e. e

Therefore, we refine our notion of signature so that their identifiers and argument types can be appropriately specified. To accomplish this we need to introduce a few new types with their appropriate relations and operations. The most important type we introduce is the finite list type constructor. We choose lists because we reason that a method's arguments is most appropriately captured as a list of the existing $\mathbb{CLASS}$ type. Additionally we consider an identifier to be a special sort of string (a list of characters) in a program that identifies a component therein. The conditions as to what identifiers are acceptable vary between programming languages. For example, Java's identifiers are case sensitive strings that start with a letter and cannot be a keyword, a boolean literal, or the **null** literal [Sun Microsystems Inc., 2006]. C++ has similar rules, but has different keywords for example. We will therefore define identifiers as special kinds of strings, which are lists of characters. We could leave identifiers as atomic terms, but by making them a subtype of a list of characters we will be able to perform some interesting reasoning over them. For example, signatures that begin with specific words, such as with *get*ter and *set*ter methods.

In the next section we introduce the finite list type constructor into our theory, with its associated relations and operations. In §9.2 we will then use lists to redefine signatures, and demonstrate the improvements it makes to our theory.

9.1 Finite Lists

Our rules for the list type constructor almost exactly mirrors the rules for the set type constructor. In $\mathbf{E}_1$ we introduce the list type constructor, and in $\mathbf{E}_2$ we introduce the empty list constant.

$$\mathbf{E}_1\ \frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash \mathit{list}\,(T) : \mathcal{U}} \qquad \mathbf{E}_2\ \frac{\Gamma \vdash T : \mathcal{U}}{\Gamma \vdash []_T : \mathit{list}\,(T)}$$

A list is then constructed by a finite series of *list append* operations, i.e. we can always make a bigger list by appending another element to the front of an existing list. Indeed, in this way our treatment of lists is identical to our treatment of sets. The formation rule for appending elements to a list is given in $\mathbf{E}_3$, where $\mathbf{E}_4$ states that appending an element to a list always results in a non-empty list. $\mathbf{E}_5$ is an induction rule that allows us to reason over lists.

$$\mathbf{E}_3\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{list}\,(T)}{\Gamma \vdash a \boxplus b : \mathit{list}\,(T)}$$

$$\mathbf{E}_4\ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{list}\,(T)}{\Gamma \vdash a \boxplus b \neq []_T}$$

$$\mathbf{E}_5\ \frac{\begin{array}{c}\Gamma, l : \mathit{list}\,(T) \vdash \varphi\,[l]\ \ \mathit{prop} \\ \Gamma \vdash \varphi\,[[]_T] \quad \Gamma, l : \mathit{list}\,(T)\,, t : T, \varphi\,[l] \vdash \varphi\,[t \boxplus l]\end{array}}{\Gamma, l : \mathit{list}\,(T) \vdash \varphi\,[l]}$$

We then add to this the standard *head* and *tail* operations for lists. $\mathbf{E}_6$ is the formation rule for the *head* operation, where we are only able to take the *head* of a non-empty list. Similarly, $\mathbf{E}_7$ is the formation rule for the *tail* operation, where again we are only able to get the tail of a non-empty list. $\mathbf{E}_8$ tells us that the *head* operation behaves as expected. That is, taking the head of a list gives us the most recently appended element of that list. Mirroring this, $\mathbf{E}_9$ ensures that the *tail* operation behaves as expected. That is, taking the *tail* of a list gives us the list without the most recently appended element. Finally, $\mathbf{E}_{10}$ ensures that equality of lists is preserved by the *head* and *tail* operations.

$$\mathbf{E}_6 \ \frac{\Gamma \vdash l : \mathit{list}\,(T) \quad \Gamma \vdash l \neq []_T}{\Gamma \vdash \mathit{head}\,(l) : T}$$

$$\mathbf{E}_7 \ \frac{\Gamma \vdash l : \mathit{list}\,(T) \quad \Gamma \vdash l \neq []_T}{\Gamma \vdash \mathit{tail}\,(l) : \mathit{list}\,(T)}$$

$$\mathbf{E}_8 \ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{list}\,(T)}{\Gamma \vdash a = \mathit{head}\,(a \boxplus b)}$$

$$\mathbf{E}_9 \ \frac{\Gamma \vdash a : T \quad \Gamma \vdash b : \mathit{list}\,(T)}{\Gamma \vdash b = \mathit{tail}\,(a \boxplus b)}$$

$$\mathbf{E}_{10} \ \frac{\Gamma \vdash l : \mathit{list}\,(T) \quad \Gamma \vdash l \neq []_T}{\Gamma \vdash l = \mathit{head}\,(l) \boxplus \mathit{tail}\,(l)}$$

With this simple definition of finite lists, we define a few other standard operations and relations on sets. The first of which is concatenating two lists together to form a third list. We begin to define this with the following *Concat* relation:

$$\mathit{Concat} \triangleq [a, b, c : \mathit{list}\,(T) \mid (a = []_T \wedge b = c) \vee (\mathit{head}\,(a) = \mathit{head}\,(c) \wedge \mathit{Concat}\,(\mathit{tail}\,(a)\,, b, \mathit{tail}\,(c)))]$$

which should be clear that this relation is functional. Therefore we define the appropriate *Concat* function. Therefore, we write $\mathit{Concat}\,(a, b)$ for the unique c, where *Concat* is a binary function such that $\mathit{Concat} : \mathit{list}\,(T) \times \mathit{list}\,(T) \longmapsto \mathit{list}\,(T)$ (see p.62) holds. With the *Concat* function we are able to define the other common relationships on lists: prefix, suffix, sublists and the element of relationships respectively:

$$\begin{aligned}
Prefix &\triangleq [p, l : \mathit{list}\,(T)\,|\exists x : \mathit{list}\,(T) \bullet l = \mathit{Concat}\,(p, x)] \\
Suffix &\triangleq [s, l : \mathit{list}\,(T)\,|\exists x : \mathit{list}\,(T) \bullet l = \mathit{Concat}\,(x, s)] \\
\mathit{Sublist} &\triangleq [\mathit{sub}, l : \mathit{list}\,(T)\,|\exists x, y : \mathit{list}\,(T) \bullet l = \mathit{Concat}\,(x, \mathit{Concat}\,(\mathit{sub}, y))] \\
\in_{\mathit{list}(T)} &\triangleq [e : T, l : \mathit{list}\,(T)\,|\mathit{Sublist}\,(e \boxplus []_T\,, l)]
\end{aligned}$$

To narrate their purpose, each of these relations are described below, where each list $[t_1, \ldots, t_n] : \mathit{list}\,(T)$, and each term $t_x : T$ where $1 \leq x \leq n$:

- $Prefix$ allows us to assert that a list begins with another list[89], for example $Prefix\left([t_1, t_2], [t_1, t_2, t_3, t_4]\right);$

- $Suffix$ allows us to assert that a list ends with another list[90], for example $Suffix\left([t_3, t_4], [t_1, t_2, t_3, t_4]\right);$

- *Sublist* a partial order relation akin to the subset operation, for example $\mathit{Sublist}\left([t_2, t_3], [t_1, t_2, t_3, t_4]\right);$

- $\in_{\mathit{list}(T)}$ a relation akin to set membership, for example $t_1 \in_{\mathit{list}(T)} [t_1, t_2, t_3, t_4]$.

Our final set of rules on lists introduce the existential, universal and uniqueness quantifiers for lists into our theory. However, the rules for each of these quantifiers mirror those for sets. Therefore, in the interests of brevity, we only present the rules for the existential quantifier as an example and opt to not present the others. The formation rule for the existential quantifier is provided in $\mathbf{E}_{11}$, where we may conclude that the existential quantification on lists is well formed when ϕ is a proposition for all terms of the appropriate type. $\mathbf{E}_{12}$ is the introduction rule, where we may introduce the existential quantifier if it is grammatically acceptable to do so, and we have a term of the right type in our list that can replace a variable in ϕ. $\mathbf{E}_{13}$ is the elimination rule, where if for any term in our list where ϕ holds we may deduce η, then we may conclude η. The usual assumptions are made here, i.e. that x must not be free in Γ, T, or η.

$$\mathbf{E}_{11}\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop} \quad \Gamma \vdash l : \mathit{list}\,(T)}{\Gamma \vdash \exists x \in_{\mathit{list}(T)} l \bullet \phi\ \mathit{prop}}$$

$$\mathbf{E}_{12}\ \frac{\Gamma, x : T \vdash \phi\ \mathit{prop} \quad \Gamma \vdash l : \mathit{list}\,(T) \quad \Gamma \vdash \phi\,[t/x] \quad \Gamma \vdash t \in_{\mathit{list}(T)} l}{\Gamma \vdash \exists x \in_{\mathit{list}(T)} l \bullet \phi}$$

$$\mathbf{E}_{13}\ \frac{\Gamma \vdash \exists x \in_{\mathit{list}(T)} l \bullet \phi \quad \Gamma, x \in_{\mathit{list}(T)} l, \phi \vdash \eta}{\Gamma \vdash \eta}$$

With the introduction of the list type constructor, and all the above associated rules, relations and operators, we are now able to move on to redefining the signature type.

[89] The $Prefix$ relationship is therefore a special case of the *Sublist* relationship, that is $Prefix \models \mathit{Sublist}$ holds.

[90] The $Suffix$ relationship is therefore a special case of the *Sublist* relationship, that is $Suffix \models \mathit{Sublist}$ holds.

9.2 Signatures redefined

As we stated at the beginning of this chapter, identifiers are special strings in a programming language that identify a component of a program. It follows therefore, that identifiers are a subtype of strings (lists of characters[91]). The conditions as to what identifiers are acceptable vary between programming languages, for example Java and C++ are similar, but not identical. Therefore, we opt to not define what constitutes an identifier directly in our theory as that would align our theory with one object-oriented programming language over another. Instead we abstract from this detail as we have done in many others (Chapter 7). We provide a single decidable relation that tells us if an identifier is acceptable. Given this, we define the following rules to construct the identifier type:

$$\mathbf{E}_{14}\ \ \Gamma \vdash \mathbb{CHAR} : \mathcal{U}$$

$$\mathbb{STRING} \triangleq \mathit{list}\,(\mathbb{CHAR})$$

$$\mathbf{E}_{15}\ \ \Gamma \vdash \mathit{Identifier} : \mathit{schema}\,(\mathbb{STRING})$$

$$\mathbb{IDENTIFIER} \triangleq \{s : \mathit{list}\,(\mathbb{CHAR})\,| \mathit{Identifier}\,(s)\}$$

With the addition of the $\mathbb{IDENTIFIER}$ type into our theory, the definition of signatures follows almost immediately. What remains is to model the signature's arguments, which is easily representable in our theory as a list of classes. Therefore, we can redefine the signature type as follows:

$$\begin{aligned} \mathbb{ARGS} &\triangleq \mathit{list}\,(\mathbb{CLASS}) \\ \mathbb{SIGNATURE} &\triangleq \mathbb{IDENTIFIER} \times \mathbb{ARGS} \end{aligned}$$

This allows us to now capture more details of method signatures, and therefore makes our theory more expressive. Returning to our original example at the beginning of this chapter, the signature `add`(`Integer`) can now be formally represented as the pair consisting of the identifier `add` and the list of class arguments [`Integer`], i.e. (`add`, [`Integer`]). We improve this notation to make it more intuitive to read, i.e. we introduce a shorthand notation that is similar to how signatures are normally written in programming languages. But in introducing such a shorthand we should be careful not to introduce ambiguity. For example, the most intuitive format

[91] Where we take strings to be lists of characters, where we write plain text as shorthand for a series of append operations where it is not ambiguous to do so. For example, we may write $\mathtt{AString} : \mathbb{STRING}$ in place of $\mathtt{A} \boxplus \mathtt{S} \boxplus \mathtt{t} \boxplus \mathtt{r} \boxplus \mathtt{i} \boxplus \mathtt{n} \boxplus \mathtt{g} \boxplus [\,]_{\mathbb{CHAR}} : \mathbb{STRING}$.

for signatures is $identifier\,(arg_1, \ldots, arg_n)$, but this is very similar to functions and relations. Therefore, we use very similar brackets to keep an intuitive look, while distinguishing signatures from functions or relations; we write $identifier\,\langle arg_1, \ldots, arg_n\rangle$ as shorthand for the signature $(identifier, [arg_1, \ldots, arg_n])$.

Additionally, we define the following functional relations:

$$IdentOf \triangleq [sig : \mathbb{SIGNATURE},\ i : \mathbb{IDENTIFIER} | i = \pi_1\,(sig)]$$
$$ArgsOf \triangleq [sig : \mathbb{SIGNATURE},\ args : list\,(\mathbb{CLASS})\,| args = \pi_2\,(sig)]$$

where we define their respective functions. That is, we write $IdentOf\,(sig)$ for the unique i such that $IdentOf : \mathbb{SIGNATURE} \longmapsto \mathbb{IDENTIFIER}$ (see p.62) holds. And, we write $ArgsOf\,(sig)$ for the unique $args$ such that $ArgsOf : \mathbb{SIGNATURE} \longmapsto \mathbb{IDENTIFIER}$ holds. These just allow us to easily access the components of a signature in with intuitive operations.

But what benefits do we get by redefining the signature type in this way? Firstly, and quite importantly, it still allows us to reason over signatures as atomic terms as in Chapters 7. For example, $sig : \mathbb{SIGNATURE}$ in the relations $IdentOf$ and $ArgsOf$ above. This means that all the specifications of motifs we presented in Chapter 8 remain unaffected by this modification.

Secondly, it also allows us to reason over signatures at a greater level of detail, such as in cases like at the beginning of this chapter. For example, we are now able to appropriately substitute `Integer` for a variable to result in a more appropriate specification:

$$\begin{array}{|l|}
\multicolumn{1}{l}{genericList} \\
\hline
list, e, \texttt{boolean} : \mathbb{CLASS} \\
\texttt{add}\,\langle e\rangle : \mathbb{SIGNATURE} \\
\hline
\qquad Aggregate\,(list, e) \wedge \\
Return\,(\texttt{add}\,\langle e\rangle \otimes list, \texttt{boolean}) \\
\hline
\end{array}$$

This specification now tells us that every $list$ class is an aggregate of e, and has a method identified by `add` and takes a single instance of e as an argument. When signatures do appear partially bound, as with $\texttt{add}\,\langle e\rangle$ in the above specification, we always treat this as shorthand for a relation with the minimum and most atomic declarations with any necessary existential quantification. For example, take the following specification:

$$R \triangleq [\texttt{ident}\,\langle c\rangle : \mathbb{SIGNATURE} | \phi\,[\texttt{ident}\,\langle c\rangle]]$$

Although it may not be obvious at first, this specification is of type *schema* ($\mathbb{CLASS}$), as by providing a class term we bind the signature term. Therefore, the above specification is shorthand for:

$$R \triangleq [c : \mathbb{CLASS} | \exists s : \mathbb{SIGNATURE} \bullet s = \texttt{ident}\,\langle c\rangle \wedge \phi\,[\texttt{ident}\,\langle c\rangle]]$$

Similarly, consider the following specification where the argument of the signature is a constant:

$$S \triangleq [\mathit{ident}\,\langle \texttt{c}\rangle : \mathbb{SIGNATURE} | \phi\,[\mathit{ident}\,\langle \texttt{c}\rangle]]$$

This specification is of type *schema* ($\mathbb{IDENTIFIER}$) as it is the minimum that actually needs to be declared, therefore it is shorthand for:

$$S \triangleq [\mathit{ident} : \mathbb{IDENTIFIER} | \exists s : \mathbb{SIGNATURE} \bullet s = \mathit{ident}\,\langle \texttt{c}\rangle \wedge \phi\,[\mathit{ident}\,\langle \texttt{c}\rangle]]$$

This convention allows us to only supply those arguments required to build the desired variables. To a certain extent these sorts of specifications are slightly more difficult to understand as the type of a relation may not always be immediately obvious. We primarily highlight this issue since it is important to demonstrate that even the smallest modification can have drastic consequences, some more obvious than others. In this case we believe that breaking signatures into more logical components is worth the slight obfuscation of a specified relation's true type.

Finally, we are now able to reason over signatures in ways we were previously unable to do. The most interesting of which is in the form of method overloading. Method overloading is a form of polymorphism implemented by programming languages, which we were previously incapable of capturing in LePUS3.

> "The simplest case of overloading occurs when methods with the same name but different signatures are defined in different classes.... The method that is actually called at runtime will depend upon the type of the object supplied to it as an actual parameter; this, in a statically typed language, can be performed at compile time.

> Overloading can also occur in the same class as that in which the method being overloaded is defined; it can occur in a subclass of the defining class." [Craig, 2000, p.158]

Craig therefore affords us two possible interpretations of method overloading: methods that have the same identifier where the context is the entire program, or where the context is some class in that program. We can easily specify both, but it is the latter that fits best with our intuitions of object-oriented design. Therefore, we define the following unary relation for methods that are overloaded within the context of some class:

Overloaded

$$m : \mathbb{METHOD}$$

$$\begin{gathered}\exists x : \mathbb{CLASS} \bullet SigOf\,(m) \otimes x = m \land \\ \exists y : \mathbb{METHOD} \bullet SigOf\,(y) \otimes x = y \land \\ y \neq m \land IdentOf\,(SigOf\,(y)) = IdentOf\,(SigOf\,(m))\end{gathered}$$

We are also able to constrain signatures by what their identifier starts with, *get*ter and *set*ter methods for example. Another example of this can be found in the JUnit testing framework [Gamma and Beck, 1999]. In JUnit, a test case contains a set of assertion methods[92] that all begin with the string `assert`. To articulate this we introduce the *StartsWith* and *EndsWith* relations:

$$StartsWith \triangleq [sig : \mathbb{SIGNATURE},\ str : \mathbb{STRING} | Prefix\,(str, IdentOf\,(sig))]$$

$$EndsWith \triangleq [sig : \mathbb{SIGNATURE},\ str : \mathbb{STRING} | Suffix\,(str, IdentOf\,(sig))]$$

With these relations we can easily capture this detail of JUnit test cases, such as in the following

[92] These are inherited from a superclass, but that details is not necessary here.

specification:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{JUnit}} \\
\hline
\mathit{testCase} : \mathbb{CLASS} \\
\mathit{assertions} : \mathit{set}\,(\mathbb{SIGNATURE}) \\
\hline
\quad \mathit{Method}\,(\mathit{testCase}, \mathit{assertions}) \wedge \\
\textsc{Tot}\,(\mathit{StartsWith}, \mathit{assertions}, \texttt{assert}) \\
\hline
\end{array}$$

To summarize, we redefined our signature type to be a composite of an identifier and its arguments. We defined these as appropriate types, and demonstrated how this improves our theory by expressing more decidable and interesting details of object-oriented design. For example, the ability to properly substitute constants for variables without causing inconsistencies, capturing signatures that start with a certain prefix, and method overloading. We conclude from this that our redefinition the signature type has improved our theory of classes far beyond the boundaries of LePUS3.

Chapter 10

Genericity

This small chapter is devoted primarily to a brief discussion of genericity within our theory of classes. There is a wealth of material on genericity, but we primarily focus our attentions on those implemented in object-oriented programming languages. In this context, "genericity is considered to be a kind of polymorphism because it takes a piece of code—a class, procedure or function—which implements an algorithm or collection of algorithms in a type independent fashion (or: in terms of an abstract, most general type) and which employs a type substitution mechanism to produce instances which are specialized to particular types" [Craig, 2000, p.153].

To reiterate part of our discussion in §2.1.5, two of the most well known implementations of genericity are Java's *generics* and C++'s *templates*. Although these mechanisms exist to solve similar problems, and therefore are abstractly quite similar, they are implemented in very different ways.

> "While generics look like the C++ templates, it is important to note that they are not the same. Generics simply provide compile-time type safety and eliminate the need for casts. The main difference is encapsulation: errors are flagged where they occur and not later at some use site, and source code is not exposed to clients. Generics use a technique known as type erasure . . . , and the compiler keeps track of the generics internally, and all instances use the same class file at compile/run time." [Mahmoud, 2004]

Both Java and C++ therefore implement genericity very differently, and there is no requirement that an object-oriented language must have an implementation of some form of genericity. Therefore, capturing genericity as employed by implementation languages means adopting its assumptions and mechanisms to the exclusion of others. We wish to capture object-oriented design at a level of abstraction that does not tie us to any single implementation language. Therefore, we argue that the use (or not) of implementation specific genericity mechanisms is a matter left to either a lower level of design than we are interested in, or directly in the hands of the programmer. We are, however, able to use similar mechanisms in our theory at the level of specification. Specifically we will take most of our influence from parametrized classes.

The first, and simplest, form of genericity that we are able to capture is that of more generic design motifs. Just as classes can be grouped together and abstracted using mechanisms such as parametrized classes, we are able to do exactly the same thing to our specifications. That is, by identifying similarities between motifs we can abstract them to a more generic specification of which those motifs are both instances. For example, take the Factory Method and the Abstract Factory motifs as previously defined. Their definitions are strikingly similar as they differ only by the type of certain terms, additionally in §8.2 we state that the Factory Method semantically entails the Abstract Factory. This strongly suggests that there is a more generic factory relation to which they both comply. The universe of types and the schema type constructor (§6.2) allow us formalize this intuition as demonstrated in the *GenericFactory* specification below:

$GenericFactory$

$$F, P, FM : \mathcal{U}$$
$$Factory : schema\,(F, P, FM)$$

$$Factory = [f : F, p : P, fm : FM | \textsc{Iso}\,(Produce, fm \otimes f, p)]$$

This specification defines a relation that takes three types and a relation as arguments, which must be appropriately typed. It defines a family of factory motifs of which the Factory Method and Abstract Factory motifs are both members, i.e.:

$$GenericFactory\,(\mathbb{HIERARCHY}, \mathbb{HIERARCHY}, \mathbb{SIGNATURE}, FactoryMethod)$$
$$GenericFactory\,(\mathbb{HIERARCHY}, set\,(\mathbb{HIERARCHY})\,, set\,(\mathbb{SIGNATURE})\,, AbstractFactory)$$

This is an interesting observation that could not be articulated in LePUS3, but we can take even further influence from parameterized classes. A parameterized class is not a class itself, but its instantiations are. All such instantiations must comply with the requirements set out in the parameterized class, such as what it inherits from and what its interface is.

In our theory, any proposition can be turned into a subtype. Relations can therefore be used to introduce a range of new types into our theory. In this way relations could be considered akin to parameterized classes. They can be instantiated to form a subtype by a combination of *hiding* arguments, using the existential quantifier, or *binding* them, by assigning specific values to arguments.

For example, consider our Factory Method motif, which we reiterate here:

$FactoryMethod$

$$\begin{array}{|l}
Factories : \mathbb{HIERARCHY} \\
Products : \mathbb{HIERARCHY} \\
FactoryMethod : \mathbb{SIGNATURE} \\
\hline
\textsc{Iso}\,(Produce, FactoryMethod \otimes Factories, Products)
\end{array}$$

We then manually hid the *Products* and *FactoryMethod* arguments using existential quantification, the result of which formed the following specification:

$Factory$

$$\begin{array}{|l}
Factories : \mathbb{HIERARCHY} \\
\hline
\exists Products : \mathbb{HIERARCHY} \bullet \\
\exists FactoryMethod : \mathbb{SIGNATURE} \bullet \\
FactoryMethod\,(Factories, Products, FactoryMethod)
\end{array}$$

This was then turned into a subtype with the following simple statement:

$$\mathbb{FACTORY} \triangleq \{f : \mathbb{HIERARCHY} | Factory\,(f)\}$$

This is an example of how we may use relations to introduce subtypes into our theory. However, the advantages of parameterized classes is that they can be instantiated on demand, easily and elegantly. For this to really compare, we would need to introduce a mechanism to do the work we did manually as a simple proposition in our theory. Consider the following relaiton, which hides arguments of a relation using the existential quantifier in exactly the same way as we did in our example above.

Hide

$$i : \mathbb{N}$$
$$R : schema\,(T_1 \times \ldots \times T_{i-1} \times T_i \times T_{i+1} \times \ldots \times T_n)$$
$$S : schema\,(T_1 \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_n)$$

$$S = [x_1 : T_1, \ldots, x_{i-1} : T_{i-1}, x_{i+1} : T_{i+1}, \ldots, x_n : T_n | \exists x_i : T_i \bullet R\,(x_1, \ldots, x_n)]$$

The $Hide$ relation tells us that the relation S is a facade for relation R where the ith argument of R is hidden with the existential quantifier. It follows immediately that this is functional, therefore we write $Hide\,(i, R)$ for the unique S such that the following holds (see p.62):

$$Hide : \mathbb{N} \times schema\,(T_1 \times \ldots \times T_{i-1} \times T_i \times T_{i+1} \times \ldots \times T_n) \longmapsto$$
$$schema\,(T_1 \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_n)$$

This $Hide$ function now allows us to formalize the relationship between $FactoryMethod$ and $Factory$, specifically:

$$Factory \triangleq Hide\,(2, Hide\,(3, FactoryMethod))$$

This allows us to *hide* arguments, then to *bind* them to given values uses a very similar mechanism. Consider the following specification:

Bind

$$\begin{array}{l} i : \mathbb{N} \\ x_i : T_i \\ R : \mathit{schema}\left(T_1 \times \ldots \times T_{i-1} \times T_i \times T_{i+1} \times \ldots \times T_n\right) \\ S : \mathit{schema}\left(T_1 \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_n\right) \\ \hline S = [x_1 : T_1, \ldots, x_{i-1} : T_{i-1}, x_{i+1} : T_{i+1}, \ldots, x_n : T_n | R\left(x_1, \ldots, x_n\right)] \end{array}$$

The *Bind* relation tells us that the relation S is a facade for relation R where the ith argument of R is bound to the value x_i. It follows immediately that this is functional, therefore we write $\mathit{Bind}\left(i, x_i, R\right)$ for the unique S such that the following holds (see p.62):

$$\mathit{Bind} : \mathbb{N} \times T_i \times \mathit{schema}\left(T_1 \times \ldots \times T_{i-1} \times T_i \times T_{i+1} \times \ldots \times T_n\right) \longmapsto \\ \mathit{schema}\left(T_1 \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_n\right)$$

We do not have a direct example of using this, so let us assume that we know that all factory methods should have the signature `instantiate` $\langle\rangle$. Given this we can easily get a relation for all factory hierarchies that have this specific interface such that it complies with the *FactoryMethod* relation with some products hierarchy:

$$\mathit{InstFactory} \triangleq \mathit{Hide}\left(2, \mathit{Bind}\left(3, \texttt{instantiate}\left\langle\right\rangle, \mathit{FactoryMethod}\right)\right)$$

What remains is to introduce a subtype based on a relation directly. To achieve this we introduce the following relation, which relates a subtype to the relation that forms the entirety of its proposition[93].

$$\mathit{Typify} \triangleq [R : \mathit{schema}\left(T\right), S : \mathcal{U} | S = \{x : T | R\left(x\right)\}]$$

It follows immediately that this is also functional, therefore we write $\mathit{Typify}\left(R\right)$ for the unique S

[93] We use the term *typify* for this, as the new subtype S represents (symbolizes, typifies) all elements of the relation R.

such that $Typify : schema\,(T) \longmapsto \mathcal{U}$ holds.

We can now easily define subtypes from relations in one simple step through application of $Hide$ and $Bind$ functions as appropriate, and finally the $Typify$ function. For example, the $\mathbb{FACTORY}$ subtype is now definable as a simple proposition:

$$\mathbb{FACTORY} \triangleq Typify\,(Hide\,(2, Hide\,(3, FactoryMethod)))$$

However, it would be nice to simplify this notation a little further and make these statements a little more intuitive. We accomplish this by adding a little shorthand, so that we may use relations as types when written in the form:

$$R\,|a_1, \ldots, a_n| : \mathcal{U}$$

where $R : T_1 \times \ldots \times T_n$, and each a_i is one of the following:

1. a term of type T_i, for which we must apply the $Bind$ function
2. an argument that should be hidden using the $Hide$ function, which we represent by $\bullet$
3. an argument that is to become part of the subtype, of which there must be at least one

For example, we can rewrite our $\mathbb{FACTORY}$ subtype as:

$$\mathbb{FACTORY} \triangleq FactoryMethod\,|\circ, \bullet, \bullet|$$

That is, the second and third arguments are to be hidden with the $Hide$ function, as indicated by $\bullet$, and the first argument is the one on which we define the subtype, as indicated by $\circ$. Similarly, we can define a new subtype based on $FactoryMethod$ where the method signature is specified as `instantiate` $\langle\rangle$, just as we did in the $InstFactory$ relation:

$$\mathbb{INSTFACTORY} \triangleq FactoryMethod\,|\circ, \bullet, \texttt{instantiate}\,\langle\rangle|$$

which can be unpacked as:

$$\mathbb{INSTFACTORY} \triangleq Typify\,(Hide\,(2, Bind\,(3, \texttt{instantiate}\,\langle\rangle\,, FactoryMethod)))$$

Let us see this in another example; the $genericFactory$ from Chapter 9, which we reiterate here:

genericList

$$list, e, \texttt{boolean} : \mathbb{CLASS}$$
$$\texttt{add}\,\langle e\rangle : \mathbb{SIGNATURE}$$

$$Aggregate\,(list, e) \wedge$$
$$Return\,(\texttt{add}\,\langle e\rangle \otimes list, \texttt{boolean})$$

From our discussion on how to interpret partially bound signatures in specifications, we know that the specified relation is $genericList : schema\,(\mathbb{CLASS} \times \mathbb{CLASS})$. Therefore, we can easily define a new subtype of $\mathbb{CLASS}$ whose members are lists of `Integer`:

$$genericList\,|\circ, \texttt{Integer}| : \mathcal{U}$$

which is unpacked as:

$$Typify\,(Bind\,(2, \texttt{Integer}, genericList)) : \mathcal{U}$$

Similarly, we introduce a new subtype of $\mathbb{CLASS}$ whose members are lists of some class:

$$genericList\,|\circ, \bullet| : \mathcal{U}$$

which is unpacked as:

$$Typify\,(Hide\,(2, genericList)) : \mathcal{U}$$

To summarize, this chapter presented our reasons for not capturing genericity at the level of implementation. Instead we presented a couple of genericity mechanisms available in our theory of classes, which promote consistency and design reuse. Our theory is able to articulate such interesting abstractions where LePUS3 was unable to do so. In Part III we discuss one method of practically applying our theory to programs in the form of design verification. We will present some design verification case studies, and discuss its implementation in the Two-Tier Programming Toolkit.

Part III

Practical Investigation

Rather than define design verification for our entire theory of classes, we demonstrate how we might define it for a subset thereof. As we began our investigation with the theoretical foundations of LePUS3, it is fair to restrict our theory to one that is expressible in an simple extended version of the visual notation of LePUS3[94] (Appendix C). Restricting ourselves to a limited theory does reduce our expressive power as to what is being verified. However, our restriction is still more expressive than LePUS3, which we have shown to be useful in its practical application in some of our previous publications [Eden and Gasparis, 2009, Nicholson et al., 2009, Eden and Nicholson, 2011]. We could extend LePUS3 to represent more of our Theory of Classes (**TC**) as we have done for generics, but as this work does not focus on visual languages so we leave this as an open problem. Our definition of design verification takes the same form as in LePUS3, i.e. we use model theoretic structures to represent interesting and decidable facts about a program. However, we have paid a great deal of attention to cleaning this approach so that it is both simpler and fitting with more of the standard terminology and definitions. As such we completely redefined design verification from how it is presented in our previous publications for LePUS3 [Nicholson et al., 2009, Eden and Nicholson, 2011]. Our definition should then be easier to work with, and extend, as desired.

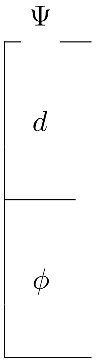
We call the restricted theory our Restricted Theory of Classes (**RTC**), which we define it using a Backus-Naur Form (BNF) grammar pruned by the semantic rules of **TC**. We also adopt the same specification notation as used in [Eden and Nicholson, 2011], which helps to distinguish **RTC** specifications from those in **TC**. This is summarized in the following definition:

[94] And as such this subset can be considered a redefinition of Class-Z.

Definition 9 Backus-Naur Form for **RTC**:

$$
\begin{array}{lcl}
\text{specification} & ::= & \Psi \triangleq \lfloor d | \phi \rfloor \\
\Psi & ::= & \text{A specification identifier} \\
d & ::= & t : T \parallel d\ d \\
t & ::= & \text{A term} \parallel t, t \parallel t \otimes t \\
a & ::= & a, a \| t \| \bullet \parallel \circ \\
T & ::= & \mathbb{CLASS} \| \mathbb{IDENTIFIER} \| \mathbb{SIGNATURE} \| \mathbb{HIERARCHY} \| \\
 & & \mathit{set}\,(T) \parallel \mathit{list}\,(T) \, \| T \times T \parallel \Psi\,|a| \\
\phi & ::= & \phi \wedge \phi \parallel R\,(t) \parallel \rho\,(R, t) \\
R & ::= & \mathit{Abstract} \parallel \mathit{Inherit} \parallel \mathit{Member} \parallel \mathit{Aggregate} \parallel \mathit{Call} \parallel \mathit{Forward} \parallel \mathit{Create} \parallel \\
 & & \mathit{Produce} \parallel \mathit{Return} \parallel \mathit{Return} \parallel \mathit{Overloaded} \parallel \mathit{Overrides} \parallel R^{+} \\
\rho & ::= & \textsc{All} \| \textsc{Tot} \| \textsc{Iso}
\end{array}
$$

We may also use the vertical notation for a *specification*, below, in which the conjunction symbol ($\wedge$) is omitted in ϕ if each atomic proposition appears on a new line.



With this practicable subset of our theory in mind, in Chapter 11 we define what a formal method is according to [Wing, 1990]. We use this to help motivate our approach, which we base on a non-standard use of model theory. This is demonstrated in Chapter 12 with three design verification case studies; verifying an instance of the Composite design motif in Java's Abstract Window Toolkit, verifying a selection of requirements of a JUnit test case, and verifying instantiations of a generic class. Finally, in Chapter 13, we discuss our contribution to tool support for LePUS3 in the form of the Two-Tier Programming Toolkit. The Two-Tier Programming Toolkit is designed for round-trip engineering, and supports both design verification and reverse engineering activities. These are implemented in the Verifier and Navigator respectively, which are both examined in a very small empirical study that yielded a marked improvement to the productivity of its users.

Chapter 11

Design Verification

There are many different approaches as to how we could define design verification. We base our approach on the broad definition of a formal method provided in [Wing, 1990], summarized in Table 11.1.

Table 11.1: A definition of formal methods [Wing, 1990]

1. A formal specification language is a triple, (Syn, Sem, Sat), where Syn and Sem are sets and $Sat \subseteq Syn \times Sem$ is a relation between them. Syn is called the languages syntactic domain; Sem its semantic domain; and Sat its satisfies relation.
2. Given a specification language, (Syn, Sem, Sat), if $sat\,(syn, sem)$ then syn is a *specification* of sem, and sem is a *specificand* of syn.
3. Given a specification language, (Syn, Sem, Sat), the specificand set of a specification syn to Syn is the set of all the specificands sem to Sem such that $Sat\,(syn, sem)$.
4. Given a specification language, (Syn, Sem, Sat), an implementation $prog$ in Sem is correct with respect to a given specification syn in Syn if and only if $Sat\,(syn, prog)$.
5. Given a semantic domain, Sem, a semantic abstraction function is a homomorphism, $A : Sem \longmapsto 2^{sem}$, that maps elements of the semantic domain into equivalence classes.
6. Given a specification language, (Syn, Sem, Sat), and a semantic abstraction function, A, defined on Sem, an abstract satisfies relation, $ASat : Syn \longmapsto 2^{sem}$, is the induced relation such that

$$\forall syn \in Syn, prog \in Sem \bullet Sat\,(syn, prog) \iff ASat\,(syn, A\,(prog))$$

7. Given a specification language, (Syn, Sem, Sat), a specification syn in Syn is unambiguous if and only if Sat maps syn to exactly one specificand set.
8. Given a specification language, (Syn, Sem, Sat), a specification syn in Syn is consistent (or satisfiable) if and only if Sat maps syn to a non-empty specificand set.

For our purposes, we take the semantic domain (Sem) to be all the programs[95] of a specific object-oriented programming language, and the syntactic domain (Syn) to be specifications articulated in **RTC**. Since our specifications represent purely decidable and primarily syntactic

[95] Where when we refer to a program, software, or an implementation, we refer to its source code.

properties of programs we could define our satisfaction relation as a pattern matching algorithm. However, such an algorithm have to be redefined for each applicable object-oriented language. Reasoning over programs in this way is an arduous task as they are inherently:

Large Even a reasonably sized program consists of many thousand (millions) lines of code[96]

Verbose Contains myriad implementation minutiae irrelevant at our level of abstraction

Decentralized Usually spanning hundreds of text files distributed across a directory structure, and any number of external libraries

Non-standard Each object-oriented language, and dialects thereof, adheres to a somewhat different (if similar) set of syntactic and semantic rules

In light of this, we choose to continue to develop on the mechanism of design verification presented in [Nicholson et al., 2009, Eden and Nicholson, 2011]. As such, we introduce an abstract semantics mechanism[97] and an abstracted satisfaction relation [Wing, 1990] that are based on a non-standard use of model theory. Abstract semantic[98] functions are pattern matching algorithms that translate programs to an intermediate model theoretic structure which we call *design models*[99]. For this we adopt as much as is possible of the standard definitions and terminology of model theory. To relate this to Table 11.1, design models are equivalence classes of programs[100]. The practical benefit of this is that the satisfaction algorithm becomes standardized across all applicable object-oriented implementation languages, where only the abstract semantics need change. The general idea of an abstract semantics is defined as follows:

Definition 10 An *abstract semantics*, $\mathcal{A}_{pl}$, is a function from programs articulated in an object-oriented programming language *pl* to design models.

As previously mentioned, design models are based on non-standard model theory [Eden and Nicholson, 2011]. Traditional model theory provides an interpretation of truth for statements in a formal language. Design models, however, provide a mechanism for reasoning

[96] Programs may be very large, but they are never infinite in size. They are always bounded by the resources of the machine and/or implementation language. This is a fundamental assumption we make in this work.

[97] This is called a *semantics abstraction* function in Table 11.1, but we adopt the same terminology as in our previous publications, such as [Eden and Nicholson, 2011].

[98] Since it is out of the scope of this work we do not define an abstract semantics. We do however allude to one for Java 6 through the course of Chapter 7, in Appendix D, and further examples can be found in [Nicholson et al., 2007].

[99] This chapter should be considered to redefine the same intention as primarily presented in LePUS3 Definitions I, VII, XIV–XVIII, and the addition of all "design model" components of the other definitions (Appendix B). LePUS3 Definition XIX is an obvious consequence of this work.

[100] We leave the definition of equivalence between design models until Appendix E.1.

over programs and to dynamically introduce constant terms into our theory. As such design models facilitate a notion of conformance rather than truth. Another point at which design models differ from standard model theory is that design models are finite by definition. This comes from the observation that programs and specifications are finite constructs.

Definition 11 A *design model* for **RTC** is a pair $(\mathfrak{M}, \mathcal{I}_{\mathfrak{M}})$ where $\mathfrak{M}$ is a model [Doets, 1996] and $\mathcal{I}_{\mathfrak{M}}$ is an interpretation function for $\mathfrak{M}$, both of which are governed by the conditions:

- $\mathfrak{M}$ contains:
    1. a non-empty finite set $\mathbf{M}$, called the universe of $\mathfrak{M}$
    2. a finite set of n-placed $(\mathtt{1} \leq n)$ relations on $\mathbf{M}$, which can be determined to be either *true* or *false*
    3. a finite set of n-placed $(\mathtt{1} \leq n)$ functions on $\mathbf{M}$, where for any appropriate $x_1, \ldots, x_n \in \mathbf{M}$ then $f(x_1, \ldots, x_n) \in \mathbf{M}$
    4. a finite (possibly empty) set of designated elements of $\mathbf{M}$
- $\mathfrak{M}$ models types as intensional sets
- The domain of $\mathcal{I}_{\mathfrak{M}}$ is the set consisting of all the constant, function and relation symbols in **RTC**, where for some x in the domain of $\mathcal{I}_{\mathfrak{M}}$ one of the following conditions hold:
    1. if x is an n-placed $(\mathtt{1} \leq n)$ function symbol then $x^{\mathfrak{M}}$ is an n-placed function on $\mathbf{M}$
    2. if x is an n-placed $(\mathtt{1} \leq n)$ relation symbol then $x^{\mathfrak{M}}$ is an n-placed relation on $\mathbf{M}$
    3. if x is a constant symbol then $x^{\mathfrak{M}} \in \mathbf{M}$ and $x^{\mathfrak{M}}$ is called a *(program) entity*, where if x is a complex term of the form $f(x_1, \ldots, x_n)$ then $\mathcal{I}_{\mathfrak{M}}(f(x_1, \ldots, x_n)) = f^{\mathfrak{M}}\left(x_1^{\mathfrak{M}}, \ldots, x_n^{\mathfrak{M}}\right)$

In addition to this definition we simplify the interpretation function notation, i.e. we write $\Theta^{\mathfrak{M}}$ as shorthand for $\mathcal{I}_{\mathfrak{M}}(\Theta)$. Traditionally, models are developed for a single formal language and contain only those language's constants. We use design models as a bridge between our restricted theory (**RTC**) and an object-oriented programming language. As such we treat constants a little differently. A design model contains all the constants from **RTC**, but also a constant for each syntactic entity derivable from the program it represents. For example, any Java program contains the java.lang.Object class, which we may represent in the design model as the constant `java.lang.Object`. Classes, methods, and method signatures are all *program entities* that are represented in this way. Design models therefore afford *contextual constants*: terms that are constant within the context of the given design model. In a similar vein, *assignments* [Doets, 1996], similar to environments/look-up tables in [Huth and Ryan, 2000, p.127–128], allow us to bind variables to any of the constants, contextual or otherwise, in the domain of the interpretation function.

Definition 12 Given a design model $\mathfrak{M}$ and an *assignment* α, the value of term t, written $t^{\mathfrak{M}}[\alpha]$, is determined by the following rules:

1. If t is a variable, then $t^{\mathfrak{M}}[\alpha] = \alpha(t)$
2. If t is a constant symbol, then $t^{\mathfrak{M}}[\alpha] = t^{\mathfrak{M}}$
3. If t has the form $f(t_1, \ldots, t_n)$, where f is an n-placed function symbol and $t_1, \ldots, t_n$ are terms, then $t^{\mathfrak{M}}[\alpha] = f^{\mathfrak{M}}\left(t_1^{\mathfrak{M}}[\alpha], \ldots, t_n^{\mathfrak{M}}[\alpha]\right)$

Assignments allow us to refer to (alias) constants. In our theory, assigned variables act in the same way as constants, and as such they are written in the same style. For example, we can alias long constants, such as `java.lang.Object`, with more memorable names, such as `Object`:

$$a(\texttt{Object}) = \texttt{java.lang.Object}$$

They also allow us to construct named sets. For example, for a design model representative of the Java SDK we could define an assignment to bind the term `Collections` to the set containing all the classes that inherit from the collection interface. That is:

$$a(\texttt{Collections}) = \{\texttt{java.util.Collection}, \ldots, \texttt{java.util.Vector}\}$$

As with other constants introduced by a design model, we also refer to assigned variables as contextual constants.

We now define a simplified version of model satisfaction, which we use as our abstract satisfaction relation (*ASat*, Table 11.1). Since we are defining this for **RTC** we only need consider the logical connective conjunction, with relation and type membership. If the whole of **TC** were being examined we would include all other required propositions in accordance with [Doets, 1996].

Definition 13 Given a design model $\mathfrak{M}$ and an assignment α, the statement s is *satisfied* by α in $\mathfrak{M}$, written $\mathfrak{M} \models s[\alpha]$ or $\mathfrak{M} \models_{\alpha} s$, if and only if one of the following statements hold:

1. if s is of the form $t : T$, then $\mathfrak{M} \models_a s$ means that $t^{\mathfrak{M}}[\alpha] \in T^{\mathfrak{M}}$ holds

2. if s is a proposition then either:

   (a) s is of the form $R(t_1, \ldots, t_n)$, where R is an n-placed $(\mathtt{1} \leq n)$ relation symbol, then $\mathfrak{M} \models s[\alpha]$ means that $\left(t_1^{\mathfrak{M}}[\alpha], \ldots, t_n^{\mathfrak{M}}[\alpha]\right) \in R^{\mathfrak{M}}$ holds

   (b) s is of the form $\phi \wedge \varphi$ then $\mathfrak{M} \models s[\alpha]$ means that $\mathfrak{M} \models \phi[\alpha]$ and $\mathfrak{M} \models \varphi[\alpha]$ hold

Where we may omit α if it is clear from the context.

Design verification of programs against **RTC** specifications is therefore definable as an abstract satisfaction relationship on the intermediate design model representation of the program. Referring back to Table 11.1, we have defined a formal method such that:

Syn the set of all **RTC** specifications

Sem the set of all programs in a given object-oriented implementation language *pl*

A the abstract semantics function $\mathcal{A}_{pl}$ that generates a design model from a program written in the object-oriented implementation language *pl*

ASat An abstract satisfaction relation $\models$ between design models and **RTC** specifications

and *Sat* being easily definable from the above in the next definition, where we overload the familiar $\models$ symbol:

Definition 14 Given a program p articulated in object-oriented programming language *pl*, an assignment α, and a **RTC** specification Ψ, *p satisfies Ψ under α* (or *p implements Ψ according to $\mathcal{A}_{pl}(p)$ and α*, or simply *p implements* Ψ), written $p \models_\alpha \Psi$, if and only if $\mathcal{A}_{pl}(p) \models \Psi(t_1, \ldots, t_n)[\alpha]$ for some series of bound terms/constants $t_1, \ldots, t_n$

Therefore we have a formal method defined as $(Syn, Sem, \models)$. In addition to this, the uses of design models are not limited to design verification. One such example of this is that design models are equivalence classes. That is, two programs are structurally equivalent if they yield the same design model, no matter which object-oriented implementation language was used. We formalize and present this idea in Appendix E.1. In the next chapter we demonstrate how to apply this to a few complete design verification case studies taken from the Java SDK and the JUnit testing Framework.

Chapter 12

Case Studies in Design Verification

In general, to verify a claim that a program p written in programming language pl implements design d requires two distinct stages of investigation. The first stage formalizes our claim into our notation, refining it to a state that can be used by an automated tool, where the second stage is the verification algorithm itself. The first stage is broken down to the following steps:

1. First, we create an **RTC** specification that captures as much of design d as is possible at that level of abstraction. We'll refer to this specification as Ψ, and allows us to formulate the claim in formal notation:
$$p \models \Psi$$
This is by no means a straight forward task, as not all requirements in a design can be captured by **RTC**. Design verification will produce a conclusive result, but that result is worthless if the specification does not appropriately capture the intended design.

2. We then use an appropriate abstract semantics function, $\mathcal{A}_{pl}$, to identify design model that is representative of program p. That is, $\mathcal{A}_{pl}(p)$. Adding this detail to our claim provides us with:
$$\mathcal{A}_{pl}(p) \models \Psi$$
This is a fully automatable step as a static analyzer, for example the Two-Tier Programming Toolkit (Chapter 13) does exactly this for Java programs.

3. Optionally we create an assignment that allows us to define aliases to constants in the domain of the interpretation function of design model $\mathcal{A}_{pl}(p)$. For example, we might be required to introduce a new constant for a specific set of method signatures in the context of $\mathcal{A}_{pl}(p)$ to establish our claim. To accomplish this we define a new assignment a, where:
$$a(\mathtt{sigSet}) = \{\mathtt{sig}_1, \ldots, \mathtt{sig}_n\}$$

Such an assignment would appear in our claim as:

$$\mathcal{A}_{pl}(p) \models_a \Psi$$

This step is currently performed manually as it is dependant on what the design model contains, what has been specified, and what the intent of the specification is.

4. The final step is to inspect program p, its documentation, and/or its design model to find candidate participants of the Ψ specification as required. That is, if Ψ is of type $Schema(T)$ then we must find a candidate $x : T$ such that we believe $\Psi(x)$ holds in the context of $\mathcal{A}_{pl}(p)$. Our claim is then rewritten one last time to accommodate this detail:

$$\mathcal{A}_{pl}(p) \models_a \Psi(x)$$

This step is currently performed manually as our work does not address participant detection mechanisms, which would be an interesting future direction.

The second stage is the design verification algorithm, in which we show that the above formalized claim either holds or does not. The algorithm is relatively straightforward being directly derived from Definition 13 where the design model is assumed to contain all possible details that are derivable from the simple facts obtained directly from the program. However, there are very real practical issues that prevent us from implementing design models in this way. For example, they would have to contain every possible term of every possible type: a practical impossibility.

Alternatively, design model implementations need contain just enough information such that all other details may be derived as required. In this way the theory remains conceptually simple, and we avoid the exponential explosion of unnecessary information, but consequently we are forced to define a more complex verification algorithm. Therefore, we present a more detailed practical algorithm for verification that might appear in an automated verification tool:

1. For each term t in $\Psi(x)$:

   (a) If t is declared to be of type T, then $\mathcal{A}_{pl}(p) \models_a t : T$ holds if T is:

      i. $\mathbb{CLASS}$ or $\mathbb{SIGNATURE}$, and $t^{\mathfrak{M}}[\alpha] \in T^{\mathfrak{M}}$

      ii. $\mathbb{IDENTIFIER}$, and both the following hold:

      - $\mathcal{A}_{pl}(p) \models_a t : list(\mathbb{CHAR})$

- $\mathcal{A}_{pl}(p) \models_a Identifier(t)$

iii. $\mathbb{HIERARCHY}$, and both the following hold:

- $\mathcal{A}_{pl}(p) \models_a t : set(\mathbb{CLASS})$
- $\mathcal{A}_{pl}(p) \models_a Hierarchy(t)$

iv. $set(X)$ or $list(X)$, and for each element x in t:

- $\mathcal{A}_{pl}(p) \models_a x : X$

v. $X \times Y$, and both the following must hold:

- $\mathcal{A}_{pl}(p) \models_a \pi_1(t) : X$
- $\mathcal{A}_{pl}(p) \models_a \pi_2(t) : Y$

vi. $\Phi|\ldots|$, and both the following must hold, where $\Delta : Schema(X)$ is that specification resulting from the indicated series of $Hide$ and $Bind$ functions on Φ:

- $\mathcal{A}_{pl}(p) \models_a t : X$
- $\mathcal{A}_{pl}(p) \models_a \Delta(t)$

(b) If t is the result of the Superimposition function, i.e. $s \otimes c$, then all the following must hold:

- $\mathcal{A}_{pl}(p) \models_a s : S$ where $\mathcal{A}_{pl}(p) \models_a S : SetOf(\mathbb{SIGNATURE})$
- $\mathcal{A}_{pl}(p) \models_a c : C$ where $\mathcal{A}_{pl}(p) \models_a C : SetOf(\mathbb{CLASS})$
- there exists an x such that the following hold:
  - $\mathcal{A}_{pl}(p) \models_a x : M$ where $\mathcal{A}_{pl}(p) \models_a M : SetOf(\mathbb{METHOD})$
  - $\mathcal{A}_{pl}(p) \models_a \otimes(s, c, x)$

2. For each proposition ϕ, such as yielded by the application of rule $\mathbf{R}_3$ to $\Psi(x)$, $\mathcal{A}_{pl}(p) \models_a \phi$ holds if ϕ is of the form:

(a) $R(t_1, \ldots, t_n)$ and $\left(t_1^{\mathfrak{M}}[\alpha], \ldots, t_n^{\mathfrak{M}}[\alpha]\right) \in R^{\mathfrak{M}}$ holds either immediately in $\mathcal{A}_{pl}(p)$ or is derivable therefrom

(b) $\varphi \wedge \psi$ and both the following hold:

- $\mathcal{A}_{pl}(p) \models_a \varphi$
- $\mathcal{A}_{pl}(p) \models_a \psi$

If this algorithm completes with a positive result then we are able to conclude our claim holds under this process. That is, the claim that a program p written in programming language pl implements design d by virtue of $\mathcal{A}_{pl}\left(p\right) \models_a \Psi\left(x\right)$.

Let us consider a simple example. Consider the following simple specification:

Ex

$s : \mathbb{SIGNATURE}$

$c : \mathbb{CLASS}$

$Create\left(s \otimes c, c\right)$

Ex specifies that some class c has a method with signature s that creates itself. Now consider we wish to verify this specification against some design model $\mathfrak{M}$, which we write as:

$$\mathfrak{M} \models Ex$$

To show this holds, by our discussion above, we must find some candidate participants in $\mathfrak{M}$ for which we believe *Ex* holds. Let us assume the candidate participants are the contextual constants s and c respectively. Now we must apply the algorithm above to obtain a series of simpler satisfaction relations that we may check independently of one another. Firstly, we consider what terms are specified in *Ex*: s: $\mathbb{SIGNATURE}$, c: $\mathbb{CLASS}$, and $\mathsf{s}\otimes\mathsf{c}$: $\mathbb{METHOD}$. Therefore we derive the following satisfaction relations by rule 1:

1. $\mathfrak{M} \models \mathsf{s}$: $\mathbb{SIGNATURE}$

2. $\mathfrak{M} \models \mathsf{c}$: $\mathbb{CLASS}$

3. $\mathfrak{M} \models \mathsf{s}\otimes\mathsf{c}$: $\mathbb{METHOD}$

and the following proposition by rule 2:

4. $\mathfrak{M} \models Create\left(\mathsf{s} \otimes \mathsf{c}, \mathsf{c}\right)$

Normally we would have to apply rule 2.b to obtain all the propositions, but this is not necessary in this case. Now we may apply the appropriate rules of the design verification algorithm to derive the required model theoretic check. To accomplish this in a readable way, we will use the symbol $\Rrightarrow$ to indicate informally an application of the design verification algorithm and unpacking of specifications:

1. $\mathfrak{M} \models \mathtt{s} \colon \mathbb{SIGNATURE}$
   $\Rrightarrow \mathtt{s}^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$

2. $\mathfrak{M} \models \mathtt{c} \colon \mathbb{CLASS}$
   $\Rrightarrow \mathtt{c}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$

3. $\mathfrak{M} \models \mathtt{s} \otimes \mathtt{c} : \mathbb{METHOD}$
   $\Rrightarrow \mathfrak{M} \models \mathtt{m} : \mathbb{METHOD}$
   $\Rrightarrow \mathtt{m}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$
   $\Rrightarrow \mathfrak{M} \models \otimes (\mathtt{s}, \mathtt{c}, \mathtt{m})$
   $\Rrightarrow \mathfrak{M} \models SignatureOf\,(\mathtt{s}, \mathtt{m})$
   $\Rrightarrow \left(\mathtt{s}^{\mathfrak{M}}, \mathtt{m}^{\mathfrak{M}}\right) \in SignatureOf^{\mathfrak{M}}$
   $\Rrightarrow \mathfrak{M} \models MethodMember\,(\mathtt{c}, \mathtt{m})$
   $\Rrightarrow \left(\mathtt{c}^{\mathfrak{M}}, \mathtt{m}^{\mathfrak{M}}\right) \in MethodMember^{\mathfrak{M}}$

4. $\mathfrak{M} \models Create\,(\mathtt{m}, \mathtt{c})$
   $\Rrightarrow \mathfrak{M} \models \left(\mathtt{m}^{\mathfrak{M}}, \mathtt{c}^{\mathfrak{M}}\right) \in Create^{\mathfrak{M}}$

where m is the method indicated by s$\otimes$c. An implementation of this algorithm would have to search the design model $\mathfrak{M}$ for such a suitable method, the details of which we do not discuss. Note that in step 3 the $\otimes$ relation was unpacked to two relations $SignatureOf$ and $MethodMember$, which satisfies the definition of $\otimes$ (see §7.4). Also note, that in step 4 the superimposition s$\otimes$c was replaced by the resulting term m as found from step 3.

In the following sections we demonstrate this verification algorithm with a few case studies taken from real world programs.

12.1 Composite in the Abstract Window Toolkit

The Abstract Window Toolkit (AWT) is a standard Java package that provides user interface widgets, such as buttons and scrollbars, that compose a wide variety of user interfaces. This is an ideal situation in which to use the Composite design motif[101], allowing hierarchies of user interface widgets to be accessed as if they were a single widget.

The hypothesis that Java's AWT package contains an instance of the Composite design motif is supported by work in fields such as design pattern based design recovery [Dong and Zhao, 2007, Seemann and von Gudenberg, 1998], in text books [Stelting and Maassen, 2002], and is a common example in our own previous publications [Nicholson et al., 2009, Eden and Nicholson, 2011]. Because of the existing evidence that this hypothesis holds, it is reasonable for us to begin with the same case study:

Claim 1 The Java 1.6 package java.awt implements the Composite design motif.

To rewrite this claim in accordance with the process of design verification (p.183) we begin by formalizing the Composite design motif. We previously presented a formalization of this motif in **TC** in Chapter 8, by simplifying it we arrive at the **RTC** variant[102]:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Composite}} \\
\hline
\\
\mathit{component}, \mathit{composite} : \mathbb{CLASS} \\
\mathit{Leaves} : \mathit{set}\,(\mathbb{CLASS}) \\
\mathit{ComponentOPs}, \mathit{CompositeOps} : \mathit{set}\,(\mathbb{SIGNATURE}) \\
\\
\hline
\\
\mathit{Method}\,(\mathit{composite}, \mathit{CompositeOPs}) \\
\mathit{Method}\,(\mathit{Leaves}, \mathit{ComponentOPs}) \\
\mathit{Inherit}\,(\mathit{composite}, \mathit{component}) \\
\textsc{Tot}\,(\mathit{Inherit}, \mathit{Leaves}, \mathit{component}) \\
\mathit{Aggregate}\,(\mathit{composite}, \mathit{component}) \\
\textsc{Iso}\,(\mathit{Forward}, \mathit{ComponentOPs} \otimes \mathit{composite}, \mathit{ComponentOPs} \otimes \mathit{component}) \\
\\
\hline
\end{array}
$$

Composite can now be used to formalize the claim in our notation as:

$$\mathsf{java.awt} \models \mathit{Composite}$$

[101] As discussed previously (§2.2), we are limited to specifying (and therefore verifying) the motifs of design patterns.

[102] Specifically, see §8.1.1 for the formalization of *Composite* in **TC**. See also §7.1 for the definition of $\mathbb{CLASS}$, §7.3 for $\mathbb{METHOD}$, §9.2 for $\mathbb{SIGNATURE}$, §7.4 for the $\otimes$ function, §7.5 for the simple relations (*Inherit*, *Aggregate*, etc.), and finally §7.6 for the complex relations (Tot and Iso).

As the claim states, the program java.awt is written in Java 1.6 so we should use the appropriate abstract semantics function $\mathcal{A}_{java1.6}$. Our design model is therefore $\mathcal{A}_{java1.6}$ (java.awt), which we shall refer to as $\mathfrak{M}$ for brevity, and whose contents are detailed in Appendix D.1. This provides us with the a more formalized claim:

$$\mathfrak{M} \models Composite$$

We do not need an assignment for this case study, so we proceed straight to identifying candidate participants. In our inspection of the java.awt source code and associated documentation gives us clues as to what might be participants. For example, we can immediately see that the Container and Component classes are ideal candidates for *composite* and *component* respectively. Similarly, the widgets Button, Canvas, Scrollbar, etc. are all obvious candidates for members of *Leaves*. For brevity in this case study we focus on just the Button and Canvas classes. However, when it comes to methods (or rather their signatures) we make the same observation as [Stelting and Maassen, 2002, p.190]: "Several methods fall short of true Composite behavior because they call different methods for Containers and Components rather than using a single method defined in the Component class and overridden in the other classes". This is interesting as methods that we would assume to be participants in the Composite motif are not implemented as we might expect. Despite this, inspection of the code suggests that at least the method signatures addNotify() and removeNotify() could be participant component operations. Similarly, the method signatures getComponents() and getComponent(int) are reasonable candidate participants for the composite operations. These candidate classes and method signatures are used to further formalize our claim[103]:

$$\mathfrak{M} \models Composite \left(\begin{array}{c} \texttt{Component}, \texttt{Container}, \\ \{\texttt{Button}, \texttt{Canvas}\}, \\ \{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\}, \\ \{\texttt{getComponents}\left\langle\right\rangle, \texttt{getComponent}\left\langle\texttt{int}\right\rangle\} \end{array} \right)$$

The above claim is now detailed enough that we can verify it fully automatically in accordance to our verification algorithm (p.183). We will go through each step of the algorithm to some degree of detail to demonstrate how it is applied and illustrate the sort of activities an implementation of the design verification algorithm would go through. The process starts by deriving a satisfaction relation for each term in the specification, each of these cases are presented in Table 12.1.

[103] We assume that the interpretation function of $\mathfrak{M}$ is such that program entities are identified by their simple names rather than their fully qualified names. For example, the constant Component is representative of the program entity java.awt.Component.

Table 12.1: Eight satisfaction relations for the terms in *Composite*

1)	$\mathfrak{M} \models \texttt{Component} : \mathbb{CLASS}$
2)	$\mathfrak{M} \models \texttt{Container} : \mathbb{CLASS}$
3)	$\mathfrak{M} \models \texttt{int} : \mathbb{CLASS}$
4)	$\mathfrak{M} \models \{\texttt{Button}, \texttt{Canvas}\} : \mathit{set}\,(\mathbb{CLASS})$
5)	$\mathfrak{M} \models \{\texttt{addNotify}\,\langle\rangle\,, \texttt{removeNotify}\,\langle\rangle\} : \mathit{set}\,(\mathbb{SIGNATURE})$
6)	$\mathfrak{M} \models \{\texttt{getComponents}\,\langle\rangle\,, \texttt{getComponent}\,\langle\texttt{int}\rangle\} : \mathit{set}\,(\mathbb{SIGNATURE})$
7)	$\mathfrak{M} \models \{\texttt{addNotify}\,\langle\rangle\,, \texttt{removeNotify}\,\langle\rangle\} \otimes \texttt{Container}$
8)	$\mathfrak{M} \models \{\texttt{addNotify}\,\langle\rangle\,, \texttt{removeNotify}\,\langle\rangle\} \otimes \texttt{Component}$

We now examine each case in turn and illustrate how they are manipulated by the design verification algorithm. To accomplish this in a readable way, we will use the symbol $\Rrightarrow$ to indicate informally an application of the design verification algorithm. We also state what line of $\mathfrak{M}$ has been used to verify each result, with the associated line from the original `java.awt` source code. To save space we do this in the form $[x, y]$, where x is the line number in the design model (§D.1.2), and y the number in the source code (§D.1.1). Consider the first satisfaction relation:

1. $\mathfrak{M} \models \texttt{Component} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Component}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[3, 1]$

Where to show $\mathfrak{M} \models \texttt{Component} : \mathbb{CLASS}$ holds, we applied the rules of the design verification algorithm (in this case rule 1.a.i.) to arrive at the statement $\texttt{Component}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$. This holds directly by inspection of the design model, specifically line 3, and the line number of the source code is given for good measure. Programmatically, this inspection can be performed by nothing more complicated than a `SELECT` query on a relational database[104], indeed this is how it is implemented in the Two-Tier Programming Toolkit. The next two cases (2 and 3) follow the same process:

2. $\mathfrak{M} \models \texttt{Container} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Container}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[4, 14]$

3. $\mathfrak{M} \models \texttt{int} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{int}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[5, 16]$

The next three cases (4–6) differ from those above as they are set terms. To show that these hold requires us to check that each of their elements are of the right type (rule 1.a.iv.):

4. $\mathfrak{M} \models \{\texttt{Button}, \texttt{Canvas}\} : \mathit{set}\,(\mathbb{CLASS})$

   $\Rrightarrow \mathfrak{M} \models \texttt{Button} : \mathbb{CLASS}$

[104] Where generating that database is accomplished by static analysis of the source code.

$\Rrightarrow \texttt{Button}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ [1, 6]

$\Rrightarrow \mathfrak{M} \models \texttt{Canvas} : \mathbb{CLASS}$

$\Rrightarrow \texttt{Canvas}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ [2, 10]

5. $\mathfrak{M} \models \{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\} : set\,(\mathbb{SIGNATURE})$

$\Rrightarrow \mathfrak{M} \models \texttt{addNotify}\left\langle\right\rangle : \mathbb{SIGNATURE}$

$\Rrightarrow \texttt{addNotify}\left\langle\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [6, 2]

$\Rrightarrow \mathfrak{M} \models \texttt{removeNotify}\left\langle\right\rangle : \mathbb{SIGNATURE}$

$\Rrightarrow \texttt{removeNotify}\left\langle\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [7, 3]

6. $\mathfrak{M} \models \{\texttt{getComponents}\left\langle\right\rangle, \texttt{getComponent}\left\langle\texttt{int}\right\rangle\} : set\,(\mathbb{SIGNATURE})$

$\Rrightarrow \mathfrak{M} \models \texttt{getComponents}\left\langle\right\rangle : \mathbb{SIGNATURE}$

$\Rrightarrow \texttt{getComponents}\left\langle\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [8, 17]

$\Rrightarrow \mathfrak{M} \models \texttt{getComponent}\left\langle\texttt{int}\right\rangle : \mathbb{SIGNATURE}$

$\Rrightarrow \texttt{getComponent}\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [9, 16]

The remaining cases deal with method terms resulting from the Superimposition function, which are terms that need to be handled a little differently than those we have seen previously (rule 1.b.). To accomplish this the design verification algorithm must identify a set of methods, and ensure that it meets the requirements imposed by the definition of the superimposition function. Firstly we present the expected result and confirm it is a set of methods. We are then required to check that the expected set of methods satisfies the ⊗ relation on which the superimposition function is defined. We do not show all details of the search that a tool might perform to confirm that the ⊗ relation holds. Instead we present those checks that are key in indicating the successful satisfaction of the proposition:

7. $\mathfrak{M} \models \{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\} \otimes \texttt{Container}$

$\Rrightarrow \mathfrak{M} \models \{\texttt{Container.addNotify()}, \texttt{Container.removeNotify()}\} : set\,(\mathbb{METHOD})$

$\Rrightarrow \mathfrak{M} \models \texttt{Container.addNotify()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{Container.addNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [13, 18]

$\Rrightarrow \mathfrak{M} \models \texttt{Container.removeNotify()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{Container.removeNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [15, 20]

$$\Rrightarrow \mathfrak{M} \models \otimes \left(\left\{ \begin{array}{c} \texttt{addNotify}\left\langle\right\rangle, \\ \texttt{removeNotify}\left\langle\right\rangle \end{array} \right\}, \texttt{Container}, \left\{ \begin{array}{c} \texttt{Container.addNotify()}, \\ \texttt{Container.removeNotify()} \end{array} \right\} \right)$$

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Container}, \texttt{Container.addNotify()}\right)$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Container.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{addNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Container.addNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [25, 18]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Container}, \texttt{Container.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.addNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [35, 18]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Container}, \texttt{Container.removeNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Container.removeNotify()}\right)$$

$$\Rrightarrow \left(\texttt{removeNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Container.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [27, 20]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Container}, \texttt{Container.removeNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [37, 20]$$

The final case follows the same reasoning as above:

8. $\mathfrak{M} \models \left\{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\right\} \otimes \texttt{Component}$

$$\Rrightarrow \mathfrak{M} \models \left\{\texttt{Component.addNotify()}, \texttt{Component.removeNotify()}\right\} : \mathit{set}\left(\mathbb{METHOD}\right)$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Component.addNotify()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Component.addNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [12, 2]$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Component.removeNotify()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Component.removeNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [14, 3]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\left\{\begin{array}{c}\texttt{addNotify}\left\langle\right\rangle, \\ \texttt{removeNotify}\left\langle\right\rangle\end{array}\right\}, \texttt{Component}, \left\{\begin{array}{c}\texttt{Component.addNotify()}, \\ \texttt{Component.removeNotify()}\end{array}\right\}\right)$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Component}, \texttt{Component.addNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Component.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{addNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [24, 2]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Component}, \texttt{Component.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Component}^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [34, 2]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Component}, \texttt{Component.removeNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Component.removeNotify()}\right)$$

$$\Rrightarrow \left(\texttt{removeNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [26, 3]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Component}, \texttt{Component.removeNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Component}^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [36, 3]$$

This completes design verification of the specified terms. Now we turn our attention to the proposition specified in *Composite*. We immediately turn this proposition into a series of smaller propositions (rule 2.b.), which we present in Table 12.2.

Table 12.2: Six satisfaction relations for each proposition in *Composite*

9)	$\mathfrak{M} \models \mathit{Method}\,(\texttt{Container}, \{\texttt{getComponents}\,\langle\rangle\,, \texttt{getComponent}\,\langle\texttt{int}\rangle\})$
10)	$\mathfrak{M} \models \mathit{Method}\,(\{\texttt{Button}, \texttt{Canvas}\}\,, \{\texttt{addNotify}\,\langle\rangle\,, \texttt{removeNotify}\,\langle\rangle\})$
11)	$\mathfrak{M} \models \mathit{Inherit}\,(\texttt{Container}, \texttt{Component})$
12)	$\mathfrak{M} \models \textsc{Tot}\,(\mathit{Inherit}, \{\texttt{Button}, \texttt{Canvas}\}\,, \texttt{Component})$
13)	$\mathfrak{M} \models \mathit{Aggregate}\,(\texttt{Container}, \texttt{Component})$
14)	$\mathfrak{M} \models \textsc{Iso}\left(\mathit{Forward}, \left\{\begin{array}{c}\texttt{Container.addNotify()},\\ \texttt{Container.removeNotify()}\end{array}\right\}, \left\{\begin{array}{c}\texttt{Component.addNotify()},\\ \texttt{Component.removeNotify()}\end{array}\right\}\right)$

As we did previously, we examine each of these cases in turn and show that they hold. The first two cases (9 and 10) are uses of the *Method* relation, which hides a use of the superimposition function and therefore follow a very similar line of investigation as the method terms (above):

9. $\mathfrak{M} \models \mathit{Method}\,(\texttt{Container}, \{\texttt{getComponents}\,\langle\rangle\,, \texttt{getComponent}\,\langle\texttt{int}\rangle\})$

$$\Rrightarrow \mathfrak{M} \models \{\texttt{Container.getComponents()}, \texttt{Container.getComponent(int)}\} : \mathit{set}\,(\mathbb{METHOD})$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Container.getComponents()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Container.getComponents()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [16, 17]$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Container.getComponent(int)} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Container.getComponent(int)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [17, 16]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\left\{\begin{array}{c}\texttt{getComponents}\,\langle\rangle\,,\\ \texttt{getComponent}\,\langle\texttt{int}\rangle\end{array}\right\}, \texttt{Container}, \left\{\begin{array}{c}\texttt{Container.getComponents()},\\ \texttt{Container.getComponent(int)}\end{array}\right\}\right)$$

$$\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{getComponents}\,\langle\rangle\,, \texttt{Container}, \texttt{Container.getComponents()})$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{getComponents}\,\langle\rangle\,, \texttt{Container.getComponents()})$$

$$\Rrightarrow \left(\texttt{getComponents}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.getComponents()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [28, 17]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{Container}, \texttt{Container.getComponents()})$$

$$\Rrightarrow \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.getComponents()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [30, 17]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{getComponent}\,\langle\texttt{int}\rangle\,, \texttt{Container}, \texttt{Container.getComponent(int)})$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{getComponent}\,\langle\texttt{int}\rangle\,, \texttt{Container.getComponent(int)})$$

$$\Rrightarrow \left(\texttt{getComponent}\,\langle\texttt{int}\rangle^{\mathfrak{M}}, \texttt{Container.getComponent(int)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [29, 16]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{Container}, \texttt{Container.getComponent(int)})$$

$$\Rrightarrow \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.getComponent(int)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [31, 16]$$

However, the second *Method* relation (10) is a little more interesting as it is the first case we have seen of a tribe of clans. See how the set identified contains two sets, one for each signature specified. Also notice that, unlike previous examples where each class directly declares a method with the specified signature, the method `Component.removeNotify()` is inherited by both `Button` and `Canvas`. It is therefore a singleton set (or a singleton clan):

10. $\mathfrak{M} \models \mathit{Method}\left(\{\texttt{Button}, \texttt{Canvas}\}, \{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\}\right)$

$$\Rrightarrow \mathfrak{M} \models \left\{ \begin{array}{c} \{\texttt{Button.addNotify()}, \texttt{Canvas.addNotify()}\}, \\ \{\texttt{Component.removeNotify()}\} \end{array} \right\} : \mathit{set}\left(\mathit{set}\left(\mathbb{METHOD}\right)\right)$$

$$\Rrightarrow \mathfrak{M} \models \{\texttt{Button.addNotify()}, \texttt{Canvas.addNotify()}\} : \mathit{set}\left(\mathbb{METHOD}\right)$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Button.addNotify()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Button.addNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}} \; [10, 7]$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Canvas.addNotify()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Canvas.addNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}} \; [11, 11]$$

$$\Rrightarrow \mathfrak{M} \models \{\texttt{Component.removeNotify()}\} : \mathit{set}\left(\mathbb{METHOD}\right)$$

$$\Rrightarrow \mathfrak{M} \models \texttt{Component.removeNotify()} : \mathbb{METHOD}$$

$$\Rrightarrow \texttt{Component.removeNotify()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}} \; [14, 3]$$

$$\Rrightarrow \mathfrak{M} \models \otimes \left(\left\{ \begin{array}{c} \texttt{addNotify}\left\langle\right\rangle, \\ \texttt{removeNotify}\left\langle\right\rangle \end{array} \right\}, \left\{ \begin{array}{c} \texttt{Button}, \\ \texttt{Canvas} \end{array} \right\}, \left\{ \begin{array}{c} \left\{ \begin{array}{c} \texttt{Button.addNotify()}, \\ \texttt{Canvas.addNotify()} \end{array} \right\}, \\ \{\texttt{Component.removeNotify()}\} \end{array} \right\} \right)$$

$$\Rrightarrow \mathfrak{M} \models \otimes \left(\texttt{addNotify}\left\langle\right\rangle, \left\{ \begin{array}{c} \texttt{Button}, \\ \texttt{Canvas} \end{array} \right\}, \left\{ \begin{array}{c} \texttt{Button.addNotify()}, \\ \texttt{Canvas.addNotify()} \end{array} \right\} \right)$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Button}, \texttt{Button.addNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Button.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{addNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Button.addNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}} \; [22, 7]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Button}, \texttt{Button.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Button}^{\mathfrak{M}}, \texttt{Button.addNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}} \; [32, 7]$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Canvas}, \texttt{Canvas.addNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{addNotify}\left\langle\right\rangle, \texttt{Canvas.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{addNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Canvas.addNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}} \; [23, 11]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Canvas}, \texttt{Canvas.addNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Canvas}^{\mathfrak{M}}, \texttt{Canvas.addNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}} \; [33, 11]$$

$$\Rrightarrow \mathfrak{M} \models \otimes \left(\texttt{removeNotify}\left\langle\right\rangle, \left\{ \begin{array}{c} \texttt{Button}, \\ \texttt{Canvas} \end{array} \right\}, \{\texttt{Component.removeNotify()}\} \right)$$

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Button}, \texttt{Component.removeNotify()}\right)$$

$$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{removeNotify}\left\langle\right\rangle, \texttt{Component.removeNotify()}\right)$$

$$\Rrightarrow \left(\texttt{removeNotify}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}} \; [26, 3]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Component}, \texttt{Component.ComponentNotify()}\right)$$

$$\Rrightarrow \left(\texttt{Component}^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}} \; [36, 3]$$

$$\Rrightarrow \mathfrak{M} \models \mathit{Inherit}\left(\texttt{Button}, \texttt{Component}\right)$$

$\Rrightarrow \left(\mathtt{Button}^{\mathfrak{M}}, \mathtt{Component}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [18, 6]

$\Rrightarrow \mathfrak{M} \models \otimes \left(\mathtt{removeNotify}\left\langle\right\rangle, \mathtt{Canvas}, \mathtt{Component.removeNotify()}\right)$

$\Rrightarrow$See above

Case 11 is a simple *Inherit* relation, which is easily shown to hold in our design model. Case 12, the Tot relation, is a little more complex to satisfy since it is shown to hold by unpacking the set of classes and checking that the resultant proposition is satisfied:

11. $\mathfrak{M} \models \mathit{Inherit}\left(\mathtt{Container}, \mathtt{Component}\right)$

    $\Rrightarrow \left(\mathtt{Container}^{\mathfrak{M}}, \mathtt{Component}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [20, 14]

12. $\mathfrak{M} \models \textsc{Tot}(\mathit{Inherit}, \left\{\mathtt{Button}, \mathtt{Canvas}\right\}, \mathtt{Component})$

    $\Rrightarrow \mathfrak{M} \models \textsc{Tot}(\mathit{Inherit}, \mathtt{Button}, \mathtt{Component})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Inherit}\left(\mathtt{Button}, \mathtt{Component}\right)$

    $\Rrightarrow \left(\mathtt{Button}^{\mathfrak{M}}, \mathtt{Component}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [18, 6]

    $\Rrightarrow \mathfrak{M} \models \textsc{Tot}(\mathit{Inherit}, \mathtt{Canvas}, \mathtt{Component})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Inherit}\left(\mathtt{Canvas}, \mathtt{Component}\right)$

    $\Rrightarrow \left(\mathtt{Canvas}^{\mathfrak{M}}, \mathtt{Component}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [19, 10]

Case 13, the *Aggregate* relation, is satisfied in the same way as the *Inherit* relation above:

13. $\mathfrak{M} \models \mathit{Aggregate}\left(\mathtt{Container}, \mathtt{Component}\right)$

    $\Rrightarrow \left(\mathtt{Container}^{\mathfrak{M}}, \mathtt{Component}^{\mathfrak{M}}\right) \in \mathit{Aggregate}^{\mathfrak{M}}$ [21, 15]

Finally, we check that the Iso relation (case 14) is satisfied. This is done in the same way as the Tot relation above, where the proposition is unpacked into smaller and simpler satisfaction propositions. As before, we only show the steps that are key to satisfying this proposition, and not every step an automated tool would have to perform.

14. $\mathfrak{M} \models \textsc{Iso}\left(\mathit{Forward}, \left\{\begin{array}{c}\mathtt{Container.addNotify()},\\ \mathtt{Container.removeNotify()}\end{array}\right\}, \left\{\begin{array}{c}\mathtt{Component.addNotify()},\\ \mathtt{Component.removeNotify()}\end{array}\right\}\right)$

    $\Rrightarrow \mathfrak{M} \models \textsc{Iso}(\mathit{Forward}, \mathtt{Container.addNotify()}, \mathtt{Component.addNotify()})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Forward}\left(\mathtt{Container.addNotify()}, \mathtt{Component.addNotify()}\right)$

    $\Rrightarrow \left(\mathtt{Container.addNotify()}^{\mathfrak{M}}, \mathtt{Component.addNotify()}^{\mathfrak{M}}\right) \in \mathit{Forward}^{\mathfrak{M}}$ [38, 19]

    $\Rrightarrow \mathfrak{M} \models \textsc{Iso}(\mathit{Forward}, \mathtt{Container.removeNotify()}, \mathtt{Component.removeNotify()})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Forward}\left(\mathtt{Container.removeNotify()}, \mathtt{Component.removeNotify()}\right)$

    $\Rrightarrow \left(\mathtt{Container.removeNotify()}^{\mathfrak{M}}, \mathtt{Component.removeNotify()}^{\mathfrak{M}}\right) \in \mathit{Forward}^{\mathfrak{M}}$ [39, 22]

With this we finish our sketch of the design verification process of this claim, from which we conclude:

Conclusion 1 The Java 1.6 package `java.awt` implements the Composite design motif by virtue of:

$$\mathcal{A}_{java1.6}\left(\texttt{java.awt}\right) \models Composite\left(\begin{array}{c}\texttt{Component}, \texttt{Container},\\ \left\{\texttt{Button}, \texttt{Canvas}\right\},\\ \left\{\texttt{addNotify}\left\langle\right\rangle, \texttt{removeNotify}\left\langle\right\rangle\right\},\\ \left\{\texttt{getComponents}\left\langle\right\rangle, \texttt{getComponent}\left\langle\texttt{int}\right\rangle\right\}\end{array}\right)$$

12.2 JUnit Testing Framework

JUnit[105] is a unit testing framework for Java that aims to simplify the development, reuse, and continual commitment to unit testing [Gamma and Beck, 1999]. It provides a simple interface that, once implemented, automates test execution and report generation.

As with describing design motifs, describing frameworks can be tricky. They must adequately convey to the user exactly which parts of a framework already exist, and which are to be implemented. In **RTC** we indicate parts of the program that are fixed with contextual constants, and those that are to be implemented with variables. We illustrate how we turn a series of design statements, paraphrased from JUnit's own documentation (Table 12.3), into a formal specification in **RTC** ($TestCase$). We then use this specification to verify one of the unit testing examples from the same source (§D.2).

Table 12.3: Summary of the requirements for a JUnit test case [Gamma and Beck, 1999]

1. Implement a subclass of TestCase, which is a subclass of class Assert and implements interface Test
2. Define instance variables that store the state of the fixture
3. Initialize the fixture state by overriding setUp()
4. Clean-up after a test by overriding tearDown()
5. Implement one method for each test to be executed, the results of which should be verified with assertion methods from the Assert class
6. Override runTest() so that it calls each desired test method

From point 1 we know that we need a variable class, let us call this $userTest$, and three classes

[105] Version 3.8.1

that are provided by JUnit framework: TestCase, Assert, Test; each of which will be contextual constants in any implementing program. Each of these classes are related by inheritance relationships, so we introduce the necessary $Inherit$ propositions in our specification.

For point 2, we define a variable set of classes, which we will call $Fixture$, to represent the classes of the data members of the $userTest$ class. We define $Fixture$ to be a set so as to give the widest interpretation of what the "fixture state" may be; if $Fixture$ were a single class then it does not capture cases where there is more than one class needing to be instantiated. Similarly, declaring $Fixture$ as a hierarchy would be too restrictive as it imposes unnecessary inheritance constraints. A set of classes therefore is the most appropriate representation of the "fixture state".

By point 3 we know that the $userTest$ class must have a method with the specific signature setUp(), which may create instances of one or more of the classes in $Fixture$.

Point 4 tells us that the $userTest$ class must have a method with the specific signature tearDown(), although we are unable to specify how this method might "clean up" the test once it has been executed.

By point 5 we know that the $userTest$ class defines a tribe of methods, but their signatures can be user defined, i.e. the JUnit framework does not require specific naming conventions for these methods. Each of these methods should call at least one of the tribe of assertion methods, the signatures of which are defined by the interface of the class Assert.

Finally, point 6 tells us that the $userTest$ class should implement a method with the specific signature runTest(). This method should call each of the tribe of user defined test methods discussed for point 5.

We can articulate all of this concisely in the following specification[106]:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TestCase}} \\
\hline
\\
\mathit{userTest}, \texttt{TestCase}, \texttt{Assert}, \texttt{Test} : \mathbb{CLASS} \\
\mathit{Fixture} : \mathit{set}\left(\mathbb{CLASS}\right) \\
\texttt{runTest}\left\langle\right\rangle, \texttt{setUp}\left\langle\right\rangle, \texttt{tearDown}\left\langle\right\rangle : \mathbb{SIGNATURE} \\
\texttt{Assertions}, \mathit{Tests} : \mathit{set}\left(\mathbb{SIGNATURE}\right) \\
\\
\hline
\\
\mathit{Method}\left(\texttt{TestCase}, \texttt{runTest}\left\langle\right\rangle\right) \\
\mathit{Method}\left(\texttt{TestCase}, \texttt{setUp}\left\langle\right\rangle\right) \\
\mathit{Method}\left(\texttt{TestCase}, \texttt{tearDown}\left\langle\right\rangle\right) \\
\mathit{Method}\left(\mathit{userTest}, \texttt{tearDown}\left\langle\right\rangle\right) \\
\mathit{Abstract}\left(\texttt{Test}\right) \\
\mathit{Abstract}\left(\texttt{TestCase}\right) \\
\mathit{Inherit}\left(\texttt{TestCase}, \texttt{Assert}\right) \\
\mathit{Inherit}\left(\texttt{TestCase}, \texttt{Test}\right) \\
\mathit{Inherit}\left(\mathit{userTest}, \texttt{TestCase}\right) \\
\textsc{Tot}\left(\mathit{Member}, \mathit{userTest}, \mathit{Fixture}\right) \\
\textsc{Tot}\left(\mathit{Create}, \texttt{setUp}\left\langle\right\rangle \otimes \mathit{userTest}, \mathit{Fixture}\right) \\
\textsc{Tot}\left(\mathit{Call}, \texttt{runTest}\left\langle\right\rangle \otimes \mathit{userTest}, \mathit{Tests} \otimes \mathit{userTest}\right) \\
\textsc{Tot}\left(\mathit{Call}, \mathit{Tests} \otimes \mathit{userTest}, \texttt{Assertions} \otimes \texttt{Assert}\right) \\
\\
\hline
\end{array}
$$

Notice that our specification of *TestCase* requires a specific set of method signatures called `Assertions`. This is the set of signatures refering to the set of assertion methods in point 5 above. This is not a variable because we know what this set of signatures is, and it does not change between implementations of a JUnit test case. To this end, we define `Assertions` with the use of an assignment. We introduce an assignment a, which maps the contextual constant

[106] See §7.1 for the definition of $\mathbb{CLASS}$, §7.3 for $\mathbb{METHOD}$, §9.2 for $\mathbb{SIGNATURE}$, §7.4 for the $\otimes$ function, §7.5 for the simple relations (*Abstract*, *Inherit*, etc.), and finally §7.6 for the complex relations (Tot, etc.).

`Assertions` to the appropriate set of method signatures, defined as follows[107]:

$$a\left(\texttt{Assertions}\right) = \left\{ \begin{array}{c} \texttt{assertTrue}\left\langle \texttt{boo}\right\rangle, \\ \texttt{assertEquals}\left\langle \texttt{Obj}, \texttt{Obj}\right\rangle, \ldots \end{array} \right\}$$

where for brevity we identify only two of the many assertion method signatures.

Now that we have constructed the motif of a JUnit test case, and defined the necessary contextual constant `Assertions`, we are able to inspect cases of its use. We examine the MoneyTest example from JUnit's documentation [Gamma and Beck, 1999] where it is claimed that:

Claim 2 The MoneyTest example is an implementation of a JUnit test case

Although this claim appears to be phrased differently from what we have seen before, we verify it in the same way. We begin by identifying the relevant source code of the test case and the JUnit framework [Gamma and Beck, 1999], which we present in §D.2.1. We refer to this source code as JUnitExample. Using all we have discussed—the JUnitExample source code, the $TestCase$ specification, and the assignment a—we construct the first step at formalizing and proving our claim:

$$\mathsf{JUnitExample} \models_a TestCase$$

As in our previous case study (§12.1), we must create an appropriate design model. The programming language we are using is Java 1.6 again, so we must use the $\mathcal{A}_{java1.6}$ abstract sematnics function. Given this we obtain the design model $\mathcal{A}_{java1.6}\left(\mathsf{JUnitExample}\right)$, which we refer to as $\mathfrak{M}$ and is presented in §D.2.2. We may inspect $\mathfrak{M}$ so as to identify the possible participants. The most crucial participant to identify is $UserTest$, for which the class `MoneyTest` as the obvious candidate. Identifying this class makes identifying the other potential participants much easier: for $Fixture$ we use $\{\texttt{Money}\}$ and for $Tests$ we use $\{\texttt{testEquals}\left\langle\right\rangle\}$. Given this we are able to further formalize our claim as follows:

$$\mathfrak{M} \models_a TestCase\left(\texttt{MoneyTest}, \left\{\texttt{Money}\right\}, \left\{\texttt{testEquals}\left\langle\right\rangle\right\}\right)$$

As in the previous case study (§12.1), we show that the above holds by checking the conformance of each proposition asserted by the specification. However, from this point on there is little difference between the two case studies, so we will not provide as much discussion .To save space,

[107] To try and keep each step of this case study within the width of the page, we refer to `boolean` as `bool`, and `Object` as `Obj`.

we indicate the appropriate lines in the design model and source code the form $[x, y]$, where x is the line number in the design model (§D.2.2), and y the number in the source code (§D.2.1).

We begin with identifying the terms specified in this instance of $TestCase$ and verifying them agains the design model $\mathfrak{M}$, which we present in Table 12.4.

Table 12.4: 16 satisfaction relations for the terms in $TestCase$

1)	$\mathfrak{M} \models \texttt{bool} : \mathbb{CLASS}$
2)	$\mathfrak{M} \models \texttt{Obj} : \mathbb{CLASS}$
3)	$\mathfrak{M} \models \texttt{MoneyTest} : \mathbb{CLASS}$
4)	$\mathfrak{M} \models \texttt{TestCase} : \mathbb{CLASS}$
5)	$\mathfrak{M} \models \texttt{Assert} : \mathbb{CLASS}$
6)	$\mathfrak{M} \models \texttt{Test} : \mathbb{CLASS}$
7)	$\mathfrak{M} \models \{\texttt{Money}\} : set\,(\mathbb{CLASS})$
8)	$\mathfrak{M} \models \texttt{runTest}\,\langle\rangle : \mathbb{SIGNATURE}$
9)	$\mathfrak{M} \models \texttt{setUp}\,\langle\rangle : \mathbb{SIGNATURE}$
10)	$\mathfrak{M} \models \texttt{tearDown}\,\langle\rangle : \mathbb{SIGNATURE}$
11)	$\mathfrak{M} \models \texttt{Assertions} : set\,(\mathbb{SIGNATURE})$
12)	$\mathfrak{M} \models \{\texttt{testEquals}\,\langle\rangle\} : set\,(\mathbb{SIGNATURE})$
13)	$\mathfrak{M} \models \texttt{setUp}\,\langle\rangle \otimes \texttt{MoneyTest}$
14)	$\mathfrak{M} \models \texttt{runTest}\,\langle\rangle \otimes \texttt{MoneyTest}$
15)	$\mathfrak{M} \models \{\texttt{testEquals}\,\langle\rangle\} \otimes \texttt{MoneyTest}$
16)	$\mathfrak{M} \models \texttt{Assertions} \otimes \texttt{Assert}$

The first terms specified are (sets of) class terms:

1. $\mathfrak{M} \models \texttt{bool} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{bool}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[1, 2]$

2. $\mathfrak{M} \models \texttt{Obj} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Obj}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[2, 1]$

3. $\mathfrak{M} \models \texttt{MoneyTest} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{MoneyTest}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[4, 12]$

4. $\mathfrak{M} \models \texttt{TestCase} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{TestCase}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[5, 6]$

5. $\mathfrak{M} \models \texttt{Assert} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Assert}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[6, 1]$

6. $\mathfrak{M} \models \texttt{Test} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Test}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ $[7, 6]$

7. $\mathfrak{M} \models \{\texttt{Money}\} : set\,(\mathbb{CLASS})$

$\Rrightarrow \mathfrak{M} \models \texttt{Money} : \mathbb{CLASS}$

$\Rrightarrow \texttt{Money}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}$ [3, 13]

Now we move on to (sets of) signature terms:

8. $\mathfrak{M} \models \texttt{runTest}\,\langle\rangle : \mathbb{SIGNATURE}$

   $\Rrightarrow \texttt{runTest}\,\langle\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [8, 9]

9. $\mathfrak{M} \models \texttt{setUp}\,\langle\rangle : \mathbb{SIGNATURE}$

   $\Rrightarrow \texttt{setUp}\,\langle\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [9, 8]

10. $\mathfrak{M} \models \texttt{tearDown}\,\langle\rangle : \mathbb{SIGNATURE}$

    $\Rrightarrow \texttt{tearDown}\,\langle\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [10, 7]

11. $\mathfrak{M} \models \texttt{Assertions} : \mathit{set}\,(\mathbb{SIGNATURE})$

    $\Rrightarrow \mathfrak{M} \models \texttt{assertTrue}\,\langle\texttt{bool}\rangle : \mathbb{SIGNATURE}$

    $\Rrightarrow \texttt{assertTrue}\,\left\langle\texttt{bool}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [12, 2]

    $\Rrightarrow \mathfrak{M} \models \texttt{assertEquals}\,\langle\texttt{Obj}, \texttt{Obj}\rangle : \mathbb{SIGNATURE}$

    $\Rrightarrow \texttt{assertEquals}\,\left\langle\texttt{Obj}^{\mathfrak{M}}, \texttt{Obj}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [13, 3]

    $\Rrightarrow \ldots$

12. $\mathfrak{M} \models \{\texttt{testEquals}\,\langle\rangle\} : \mathit{set}\,(\mathbb{SIGNATURE})$

    $\Rrightarrow \mathfrak{M} \models \texttt{testEquals}\,\langle\rangle : \mathbb{SIGNATURE}$

    $\Rrightarrow \texttt{testEquals}\,\langle\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}$ [11, 21]

What remains is to verify (sets of) method terms:

13. $\mathfrak{M} \models \texttt{setUp}\,\langle\rangle \otimes \texttt{MoneyTest}$

    $\Rrightarrow \mathfrak{M} \models \texttt{MoneyTest.setUp()} : \mathbb{METHOD}$

    $\Rrightarrow \texttt{MoneyTest.setUp()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [21, 16]

    $\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{setUp}\,\langle\rangle\,, \texttt{MoneyTest}, \texttt{MoneyTest.setUp()})$

    $\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{setUp}\,\langle\rangle\,, \texttt{MoneyTest.setUp()})$

    $\Rrightarrow \left(\texttt{setUp}\,\langle\rangle^{\mathfrak{M}}, \texttt{MoneyTest.setUp()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [35, 16]

    $\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{MoneyTest}, \texttt{MoneyTest.setUp()})$

    $\Rrightarrow \left(\texttt{MoneyTest}^{\mathfrak{M}}, \texttt{MoneyTest.setup()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [44, 16]

14. $\mathfrak{M} \models \texttt{runTest}\,\langle\rangle \otimes \texttt{MoneyTest}$

    $\Rrightarrow \mathfrak{M} \models \texttt{MoneyTest.runTest()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{MoneyTest.runTest()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [19, 28]

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{runTest}\left\langle\right\rangle, \texttt{MoneyTest}, \texttt{MoneyTest.runTest()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{runTest}\left\langle\right\rangle, \texttt{MoneyTest.runTest()}\right)$

$\Rrightarrow \left(\texttt{runTest}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{MoneyTest.runTest()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [33, 28]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{MoneyTest}, \texttt{MoneyTest.runTest()}\right)$

$\Rrightarrow \left(\texttt{MoneyTest}^{\mathfrak{M}}, \texttt{MoneyTest.runTest()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [42, 28]

15. $\mathfrak{M} \models \left\{\texttt{testEquals}\left\langle\right\rangle\right\} \otimes \texttt{MoneyTest}$

$\Rrightarrow \mathfrak{M} \models \left\{\texttt{MoneyTest.testEquals()}\right\} : \mathit{set}\left(\mathbb{METHOD}\right)$

$\Rrightarrow \mathfrak{M} \models \texttt{MoneyTest.testEquals()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{MoneyTest.testEquals()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [16, 21]

$\Rrightarrow \mathfrak{M} \models \otimes\left(\left\{\texttt{testEquals}\left\langle\right\rangle\right\}, \texttt{MoneyTest}, \left\{\texttt{MoneyTest.testEquals()}\right\}\right)$

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{testEquals}\left\langle\right\rangle, \texttt{MoneyTest}, \texttt{MoneyTest.testEquals()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{testEquals}\left\langle\right\rangle, \texttt{MoneyTest.testEquals()}\right)$

$\Rrightarrow \left(\texttt{testEquals}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{MoneyTest.testEquals()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [31, 21]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{MoneyTest}, \texttt{MoneyTest.testEquals()}\right)$

$\Rrightarrow \left(\texttt{MoneyTest}^{\mathfrak{M}}, \texttt{MoneyTest.testEquals()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [40, 21]

16. $\mathfrak{M} \models \texttt{Assertions} \otimes \texttt{Assert}$

$$\Rrightarrow \mathfrak{M} \models \left\{\begin{array}{c}\texttt{Assert.assertTrue(bool)},\\ \texttt{Assert.assertEquals(Obj, Obj)}, \ldots\end{array}\right\} : \mathit{set}\left(\mathbb{METHOD}\right)$$

$\Rrightarrow \mathfrak{M} \models \texttt{Assert.assertTrue(bool)} : \mathbb{METHOD}$

$\Rrightarrow \texttt{Assert.assertTrue(bool)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [14, 2]

$\Rrightarrow \mathfrak{M} \models \texttt{Assert.assertEquals(Obj, Obj)} : \mathbb{METHOD}$

$\Rrightarrow \texttt{Assert.assertEquals(Obj, Obj)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [15, 3]

$$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{Assertions}, \texttt{Assert}, \left\{\begin{array}{c}\texttt{Assert.assertTrue(bool)},\\ \texttt{Assert.assertEquals(Obj, Obj)}, \ldots\end{array}\right\}\right)$$

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{assertTrue}\left\langle\texttt{bool}\right\rangle, \texttt{Assert}, \texttt{Assert.assertTrue(bool)}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{assertTrue}\left\langle\texttt{bool}\right\rangle, \texttt{Assert.assertTrue(bool)}\right)$

$\Rrightarrow \left(\texttt{assertTrue}\left\langle\texttt{bool}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{Assert.assertTrue(bool)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [28, 2]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Assert}, \texttt{Assert.assertTrue(bool)}\right)$

$\Rrightarrow \left(\texttt{Assert}^{\mathfrak{M}}, \texttt{Assert.assertTrue(bool)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [37, 2]

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{assertEquals}\left\langle\texttt{Obj}, \texttt{Obj}\right\rangle, \texttt{Assert}, \texttt{Assert.assertEquals(Obj, Obj)}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{assertEquals}\left\langle\texttt{Obj}, \texttt{Obj}\right\rangle, \texttt{Assert.assertEquals(Obj, Obj)}\right)$

$\Rrightarrow \left(\texttt{assertEquals}\left\langle \texttt{Obj}^{\mathfrak{M}}, \texttt{Obj}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ $[29, 3]$

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{Assert}, \texttt{Assert.assertEquals(Obj,Obj)}\right)$

$\Rrightarrow \left(\texttt{Assert}^{\mathfrak{M}}, \texttt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ $[38, 3]$

This completes design verification of the specified terms. Now we turn our attention to the proposition specified in $\mathit{TestCase}$. We immediately turn this proposition into a series of smaller propositions (rule 2.b.), which we present in Table 12.5.

Table 12.5: 13 satisfaction relations for each proposition in $\mathit{TestCase}$

17)	$\mathfrak{M} \models \mathit{Method}\left(\texttt{TestCase}, \texttt{runTest}\left\langle\right\rangle\right)$
18)	$\mathfrak{M} \models \mathit{Method}\left(\texttt{TestCase}, \texttt{setUp}\left\langle\right\rangle\right)$
19)	$\mathfrak{M} \models \mathit{Method}\left(\texttt{TestCase}, \texttt{tearDown}\left\langle\right\rangle\right)$
20)	$\mathfrak{M} \models \mathit{Method}\left(\texttt{MoneyTest}, \texttt{tearDown}\left\langle\right\rangle\right)$
21)	$\mathfrak{M} \models \mathit{Abstract}\left(\texttt{Test}\right)$
22)	$\mathfrak{M} \models \mathit{Abstract}\left(\texttt{TestCase}\right)$
23)	$\mathfrak{M} \models \mathit{Inherit}\left(\texttt{TestCase}, \texttt{Assert}\right)$
24)	$\mathfrak{M} \models \mathit{Inherit}\left(\texttt{TestCase}, \texttt{Test}\right)$
25)	$\mathfrak{M} \models \mathit{Inherit}\left(\texttt{MoneyTest}, \texttt{TestCase}\right)$
26)	$\mathfrak{M} \models \textsc{Tot}(\mathit{Member}, \texttt{MoneyTest}, \{\texttt{Money}\})$
27)	$\mathfrak{M} \models \textsc{Tot}(\mathit{Create}, \texttt{MoneyTest.setUp()}, \{\texttt{Money}\})$
28)	$\mathfrak{M} \models \textsc{Tot}(\mathit{Call}, \texttt{MoneyTest.runTest()}, \{\texttt{MoneyTest.testEquals()}\})$
29)	$\mathfrak{M} \models \textsc{Tot}\left(\mathit{Call}, \{\texttt{MoneyTest.testEquals()}\}, \left\{\begin{array}{c}\texttt{Assert.assertTrue(bool)},\\ \texttt{Assert.assertEquals(Obj,Obj)}, \ldots\end{array}\right\}\right)$

We begin with parts 17–20, which are all Method relations. The last of which is the most interesting as it is an example of a method being inherited from a superclass:

17. $\mathfrak{M} \models \mathit{Method}\left(\texttt{TestCase}, \texttt{runTest}\left\langle\right\rangle\right)$

$\Rrightarrow \mathfrak{M} \models \texttt{TestCase.runTest()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{TestCase.runTest()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ $[18, 9]$

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{runTest}\left\langle\right\rangle, \texttt{TestCase}, \texttt{TestCase.runTest()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\texttt{runTest}\left\langle\right\rangle, \texttt{TestCase.runTest()}\right)$

$\Rrightarrow \left(\texttt{runTest}\left\langle\right\rangle^{\mathfrak{M}}, \texttt{TestCase.runTest()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ $[32, 9]$

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\texttt{TestCase}, \texttt{TestCase.runTest()}\right)$

$\Rrightarrow \left(\texttt{TestCase}^{\mathfrak{M}}, \texttt{TestCase.runTest()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ $[41, 9]$

18. $\mathfrak{M} \models \mathit{Method}\left(\texttt{TestCase}, \texttt{setUp}\left\langle\right\rangle\right)$

$\Rrightarrow \mathfrak{M} \models \texttt{TestCase.setUp()} : \mathbb{METHOD}$

$\Rrightarrow \texttt{TestCase.setUp()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ $[20, 8]$

$\Rrightarrow \mathfrak{M} \models \otimes\left(\texttt{setUp}\left\langle\right\rangle, \texttt{TestCase}, \texttt{TestCase.setUp()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\mathtt{setUp}\left\langle\right\rangle, \mathtt{TestCase.setUp()}\right)$

$\Rrightarrow \left(\mathtt{setUp}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.setUp()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [34, 8]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\mathtt{TestCase}, \mathtt{TestCase.setUp()}\right)$

$\Rrightarrow \left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.setUp()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [43, 8]

19. $\mathfrak{M} \models \mathit{Method}\left(\mathtt{TestCase}, \mathtt{tearDown}\left\langle\right\rangle\right)$

$\Rrightarrow \mathfrak{M} \models \mathtt{TestCase.tearDown()} : \mathbb{METHOD}$

$\Rrightarrow \mathtt{TestCase.tearDown()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [10, 7]

$\Rrightarrow \mathfrak{M} \models \otimes\left(\mathtt{tearDown}\left\langle\right\rangle, \mathtt{TestCase}, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\mathtt{tearDown}\left\langle\right\rangle, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \left(\mathtt{tearDown}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [30, 7]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\mathtt{TestCase}, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [39, 7]

20. $\mathfrak{M} \models \mathit{Method}\left(\mathtt{MoneyTest}, \mathtt{tearDown}\left\langle\right\rangle\right)$

$\Rrightarrow \mathfrak{M} \models \mathtt{TestCase.tearDown()} : \mathbb{METHOD}$

$\Rrightarrow \mathtt{TestCase.tearDown()}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}$ [10, 7]

$\Rrightarrow \mathfrak{M} \models \otimes\left(\mathtt{tearDown}\left\langle\right\rangle, \mathtt{MoneyTest}, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\left(\mathtt{tearDown}\left\langle\right\rangle, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \left(\mathtt{tearDown}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$ [30, 7]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\mathtt{MoneyTest}, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{Inherit}\left(\mathtt{MoneyTest}, \mathtt{TestCase}\right)$

$\Rrightarrow \left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{TestCase}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [26, 12]

$\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\left(\mathtt{TestCase}, \mathtt{TestCase.tearDown()}\right)$

$\Rrightarrow \left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}$ [39, 7]

Next are parts 21–22, which are all *Abstract* relations. By the definition of *Abstract*, and in the context of the contextual constants `Test` and `TestCase` being classes, we unpack the *Abstract* relation to that of *AbstractClass*:

21. $\mathfrak{M} \models \mathit{Abstract}\left(\mathtt{Test}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{AbstractClass}\left(\mathtt{Test}\right)$

$\Rrightarrow \mathtt{Test}^{\mathfrak{M}} \in \mathit{AbstractClass}^{\mathfrak{M}}$ [23, 6]

22. $\mathfrak{M} \models \mathit{Abstract}\left(\mathtt{TestCase}\right)$

$\Rrightarrow \mathfrak{M} \models \mathit{AbstractClass}\,(\texttt{TestCase})$

$\Rrightarrow \texttt{TestCase}^{\mathfrak{M}} \in \mathit{AbstractClass}^{\mathfrak{M}}$ [24, 6]

Parts 23–25 are all simple *Inherit* relationships:

23. $\mathfrak{M} \models \mathit{Inherit}\,(\texttt{TestCase}, \texttt{Assert})$

    $\Rrightarrow \left(\texttt{TestCase}^{\mathfrak{M}}, \texttt{Assert}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [25, 6]

24. $\mathfrak{M} \models \mathit{Inherit}\,(\texttt{TestCase}, \texttt{Test})$

    $\Rrightarrow \left(\texttt{TestCase}^{\mathfrak{M}}, \texttt{Test}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [27, 6]

25. $\mathfrak{M} \models \mathit{Inherit}\,(\texttt{MoneyTest}, \texttt{TestCase})$

    $\Rrightarrow \left(\texttt{MoneyTest}^{\mathfrak{M}}, \texttt{TestCase}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$ [26, 12]

The last four parts are satisfaction relations of predicate propositions:

26. $\mathfrak{M} \models \textsc{Tot}(\mathit{Member}, \texttt{MoneyTest}, \{\texttt{Money}\})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Member}\,(\texttt{MoneyTest}, \texttt{Money})$

    $\Rrightarrow \mathfrak{M} \models \mathit{DataMember}\,(\texttt{MoneyTest}, \texttt{Money})$

    $\Rrightarrow \left(\texttt{MoneyTest}^{\mathfrak{M}}, \texttt{Money}^{\mathfrak{M}}\right) \in \mathit{DataMember}^{\mathfrak{M}}$ [46, 13]

27. $\mathfrak{M} \models \textsc{Tot}(\mathit{Create}, \texttt{MoneyTest.setUp()}, \{\texttt{Money}\})$

    $\Rrightarrow \mathfrak{M} \models \textsc{Tot}(\mathit{Create}, \texttt{MoneyTest.setUp()}, \texttt{Money})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Create}\,(\texttt{MoneyTest.setUp()}, \texttt{Money})$

    $\Rrightarrow \left(\texttt{MoneyTest.setUp()}^{\mathfrak{M}}, \texttt{Money}^{\mathfrak{M}}\right) \in \mathit{Create}^{\mathfrak{M}}$ [47, 17]

28. $\mathfrak{M} \models \textsc{Tot}(\mathit{Call}, \texttt{MoneyTest.runTest()}, \{\texttt{MoneyTest.testEquals()}\})$

    $\Rrightarrow \mathfrak{M} \models \textsc{Tot}(\mathit{Call}, \texttt{MoneyTest.runTest()}, \texttt{MoneyTest.testEquals()})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Call}\,(\texttt{MoneyTest.runTest()}, \texttt{MoneyTest.testEquals()})$

    $\Rrightarrow \left(\texttt{MoneyTest.runTest()}^{\mathfrak{M}}, \texttt{MoneyTest.testEquals()}^{\mathfrak{M}}\right) \in \mathit{Call}^{\mathfrak{M}}$ [48, 29]

29. $\mathfrak{M} \models \textsc{Tot}\left(\mathit{Call}, \{\texttt{MoneyTest.testEquals()}\}, \left\{\begin{array}{c}\texttt{Assert.assertTrue(bool)},\\ \texttt{Assert.assertEquals(Obj, Obj)}, \ldots\end{array}\right\}\right)$

    $\Rrightarrow \mathfrak{M} \models \textsc{Tot}(\mathit{Call}, \texttt{MoneyTest.testEquals()}, \texttt{Assert.assertEquals(Obj, Obj)})$

    $\Rrightarrow \mathfrak{M} \models \mathit{Call}\,(\texttt{MoneyTest.testEquals()}, \texttt{Assert.assertEquals(Obj, Obj)})$

    $\Rrightarrow \left(\texttt{MoneyTest.testEquals()}^{\mathfrak{M}}, \texttt{Assert.assertEquals(Obj, Obj)}^{\mathfrak{M}}\right) \in \mathit{Call}^{\mathfrak{M}}$ [50, 23]

With this we finish our sketch of the design verification process of this claim, from which we conclude:

Conclusion 2 The MoneyTest example is an implementation of a JUnit test case by virtue of:

$$\mathcal{A}_{java1.6}\left(\textsf{JUnitExample}\right) \models_a TestCase\left(\texttt{MoneyTest}, \left\{\texttt{Money}\right\}, \left\{\texttt{testEquals}\left\langle\right\rangle\right\}\right)$$

where

$$a\left(\texttt{Assertions}\right) = \left\{ \begin{array}{c} \texttt{assertTrue}\left\langle\texttt{boo}\right\rangle, \\ \texttt{assertEquals}\left\langle\texttt{Obj},\texttt{Obj}\right\rangle, \ldots \end{array} \right\}$$

We conclude from this case study that **RTC** can adequately specify the structural design of software frameworks, their intended use, and verify an implementing program.

12.3 Java's Generic Lists

As we discussed genericity in Chapter 10, so it would be remiss of us to not include a case study of design verification in this area. Below is the claim that motivates our case study:

Claim 3 Java 1.6's generic class LinkedList instantiated with the class argument Integer is a list.

Our approach to verifying our claim follows the same pattern as the previous two case studies (§12.1 and §12.2). We begin with formalizing our claim in terms applicable to our design verification algorithm. To this end we must define our notion of list in this context[108], which we base on *genericList* (see Chapter 10) and extended with respects to the List interface in Java. We call our new specification *List*[109]:

List

$$list, type, \texttt{int}, \texttt{boolean} : \mathbb{CLASS}$$
$$add\left\langle type\right\rangle, remove\left\langle\texttt{int}\right\rangle, get\left\langle\texttt{int}\right\rangle : \mathbb{SIGNATURE}$$

$$Aggregate\left(list, type\right)$$
$$Return\left(add\left\langle type\right\rangle \otimes list, \texttt{boolean}\right)$$
$$Return\left(remove\left\langle\texttt{int}\right\rangle \otimes list, \texttt{boolean}\right)$$
$$Return\left(get\left\langle\texttt{int}\right\rangle \otimes list, type\right)$$

[108] Of course there are many possible formalizations of the notion of lists in this context, we focus on just the one we present here.

[109] See §7.1 for the definition of $\mathbb{CLASS}$, §7.3 for $\mathbb{METHOD}$, §9.2 for $\mathbb{SIGNATURE}$, §7.4 for the $\otimes$ function, and finally §7.5 for the simple relations (*Abstract*, *Inherit*, etc.).

According to our discussion in §9.2, the type of this specification is:

$$\mathit{Schema}\,(\mathbb{CLASS} \times \mathbb{CLASS} \times \mathbb{IDENTIFIER} \times \mathbb{IDENTIFIER} \times \mathbb{IDENTIFIER})$$

where the first argument (*list*) is the actual list class. *list* is an aggregate of instances of *type*, the second argument. The last three arguments are identifiers that name specific operations: adding an element to the list (*add*), removing an element from the list (*remove*), and retrieving an element from that list (*get*) respectively. We could directly specify what each of these identifiers must be, but doing so limits the scope of the specification. That is, there is as much reason to call the method that adds an element to the list.add as there is append[110]. The identifier used is a matter of style and context. It is for this reason that our *List* specification does not fix these identifiers to any specific constants.

The methods that add and remove instances from our list should inform the user of its success, i.e. they should return a boolean value. The method that retrieves an element of the list should take an integer argument, which acts as an index, and may return the appropriate instance if the index is acceptable. However, as our theory operates at an abstract level of design, we are unable to articulate some of these requirements in our *List* specification. For example, we cannot formalize the notion of success or how the integer arguments are to be used. These additional details could be specified in other languages, such as the *Z* specification language [Spivey, 1992], or our theory could be extended to allow specification of such details (see §14.1).

Having defined *List* we are a step closer to formalizing our claim. Additionaly, our claim states that we are working the the implementation language Java 1.6, and therefore we should use the appropriate abstract semantics function $\mathcal{A}_{java1.6}$ to generate our design model. The source code for this case study is presented in §D.3.1, and referred to as ListExample. Therefore, our design model is $\mathcal{A}_{java1.6}$ (ListExample), which we refer to as $\mathfrak{M}$ for brevityand presented in §D.3.2.

Using $\mathfrak{M}$ we identify candidate participants of our *List* specificaiton. As it is the key class of the claim, the most important participant to identify is the class resulting from the instantiation of the generic class LinkedList with class argument Integer. We represent this class with the contextual constant $\mathtt{LL} < \mathtt{Integer} >$[111], where the remaining candidates are Integer, add, remove, and get. It is important to stress that the distinction between the symbols <> and ⟨⟩ is intentional. That

[110] Neither must we be limited to English, ajouter (French) or tasu (Japanese) are also equally acceptable. Any word is acceptable as long as it makes sense in the context it is used.

[111] We would normally use LinkedList<Integer>, but we use LL<Integer> so that our later discussions fit on the width of the page.

is, $\texttt{LL} < \texttt{Integer} >$ is an atomic contextual constant of type $\mathbb{CLASS}$. $\texttt{LL} < \texttt{Integer} >$ represents that class resulting from the instantiation of generic class $\textsf{LinkedList}$, which we do not directly represent (see Chapter 10), with the class argument $\textsf{Integer}$. The constant name $\texttt{LL} < \texttt{Integer} >$ is used to convey intuitively the origin of the class being represented. The <> symbols (in the typewriter font) should therefore be treated in the same way as any other character in the name of a constant. However, $add\,\langle type\rangle$ is a $\mathbb{SIGNATURE}$ with the variable identifier *add* and a list of arguments containing only the class variable *type*. In this case the $\langle\rangle$ symbols have a clear and specific meaning in our theory, as defined in §9.2.

There is one more detail to formalize before we may rephrase and verify our claim. We claimed that the class resulting from the instantiation of the generic class $\textsf{LinkedList}$ with class argument $\textsf{Integer}$ is a list, i.e. $\texttt{LL} < \texttt{Integer} >$ *is a List*. We discussed the *is a* relationship in Chapter 10, which requires us to use the *List* specification as a type. To accomplish this we use our *Typify* and *Bind* functions with their respective shorthands (see Chapter 10).That is, we define the type:

$$Typify\,(Bind\,(2, \texttt{Integer}, Bind\,(3, \texttt{add}, Bind\,(4, \texttt{remove}, Bind\,(5, \texttt{get}, List)))))$$

which is the relation *List*, bound to contextual constants for arguments two through 5 as indicated, and turned into a type. We may rewrite with our shorthand in the more concise form:

$$List\,|\circ, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get}|$$

This means that our claim may now be formalized as follows:

$$\mathfrak{M} \models \texttt{LL} < \texttt{Integer} > : List\,|\circ, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get}|$$

The above formalizes our initial claim, which is no more complicated to verify than our previous two case studies (§12.1 and §12.2). That is, we verify this claim by verifying:

$$\mathfrak{M} \models List\,(\texttt{LL} < \texttt{Integer} >, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get})$$

Indeed, verifying the above also verifies many other possible claims[112]. Therefore, we have managed to rework a claim that was phrased very differently compared to those we have previously encountered, into a familiar and easily verifiable statement. Following our design verification algorithm, Table 12.6 presents the terms specified in this us of *List*.

Table 12.6: Ten satisfaction relations for the terms in *List*

1)	$\mathfrak{M} \models \texttt{int} : \mathbb{CLASS}$
2)	$\mathfrak{M} \models \texttt{boolean} : \mathbb{CLASS}$
3)	$\mathfrak{M} \models \texttt{Integer} : \mathbb{CLASS}$
4)	$\mathfrak{M} \models \texttt{LL} < \texttt{Integer} > : \mathbb{CLASS}$
5)	$\mathfrak{M} \models \texttt{add}\,\langle\texttt{Integer}\rangle : \mathbb{SIGNATURE}$
6)	$\mathfrak{M} \models \texttt{remove}\,\langle\texttt{int}\rangle : \mathbb{SIGNATURE}$
7)	$\mathfrak{M} \models \texttt{get}\,\langle\texttt{int}\rangle : \mathbb{SIGNATURE}$
8)	$\mathfrak{M} \models \texttt{add}\,\langle\texttt{Integer}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$
9)	$\mathfrak{M} \models \texttt{remove}\,\langle\texttt{int}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$
10)	$\mathfrak{M} \models \texttt{get}\,\langle\texttt{int}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$

We now verify each of these in turn. Where to save space, we indicate the appropriate lines in the design model and source code the form $[x, y]$, where x is the line number in the design model (§D.2.2), and y the number in the source code (§D.2.1). As is required in this case study, some entries in our design model come from pieces of generic code that have been instantiated elsewhere. We indicate this in the form $[x, y\,(z)]$, where x is the line number in the design model, y is the line number in source code that is to be instantiated (generic), and z is the line number in the source code at which line y is instantiated. The first terms specified are all class terms:

1. $\mathfrak{M} \models \texttt{int} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{int}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}\ [1, 4]$

2. $\mathfrak{M} \models \texttt{boolean} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{boolean}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}\ [2, 2]$

3. $\mathfrak{M} \models \texttt{Integer} : \mathbb{CLASS}$

   $\Rrightarrow \texttt{Integer}^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}\ [3, 8]$

4. $\mathfrak{M} \models \texttt{LL} < \texttt{Integer} > : \mathbb{CLASS}$

   $\Rrightarrow \texttt{LL} < \texttt{Integer} >^{\mathfrak{M}} \in \mathbb{CLASS}^{\mathfrak{M}}\ [5, 8]$

[112] By verifying that $\mathit{List}\,(\texttt{LL} < \texttt{Integer} >, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get})$ holds allows us to prove statements such as:

$$\begin{array}{rcl} \texttt{LL <Integer >} & : & \mathit{List}\,|\circ, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get}| \\ \texttt{LL <Integer >} & : & \mathit{List}\,|\circ, \texttt{Integer}, \bullet, \bullet, \bullet| \\ \texttt{LL <Integer >} & : & \mathit{List}\,|\circ, \bullet, \bullet, \bullet, \bullet| \\ \texttt{Integer} & : & \mathit{List}\,|\bullet, \circ, \bullet, \bullet, \bullet| \end{array}$$

or any other permutations thereon.

5. $\mathfrak{M} \models \texttt{add}\,\langle\texttt{Integer}\rangle : \mathbb{SIGNATURE}$

   $\Rrightarrow \texttt{add}\left\langle\texttt{Integer}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}\ [6, 2\,(8)]$

6. $\mathfrak{M} \models \texttt{remove}\,\langle\texttt{int}\rangle : \mathbb{SIGNATURE}$

   $\Rrightarrow \texttt{remove}\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}\ [8, 3\,(8)]$

7. $\mathfrak{M} \models \texttt{get}\,\langle\texttt{int}\rangle : \mathbb{SIGNATURE}$

   $\Rrightarrow \texttt{get}\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} \in \mathbb{SIGNATURE}^{\mathfrak{M}}\ [7, 4\,(8)]$

8. $\mathfrak{M} \models \texttt{add}\,\langle\texttt{Integer}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$

   $\Rrightarrow \mathfrak{M} \models \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)} : \mathbb{METHOD}$

   $\Rrightarrow \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [9, 2\,(8)]$

   $\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{add}\,\langle\texttt{Integer}\rangle\,, \texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)})$

   $\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{add}\,\langle\texttt{Integer}\rangle\,, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)})$

   $\Rrightarrow \left(\texttt{add}\left\langle\texttt{Integer}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [12, 2\,(8)]$

   $\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)})$

   $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [15, 2\,(8)]$

9. $\mathfrak{M} \models \texttt{remove}\,\langle\texttt{int}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$

   $\Rrightarrow \mathfrak{M} \models \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)} : \mathbb{METHOD}$

   $\Rrightarrow \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [11, 3\,(8)]$

   $\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{remove}\,\langle\texttt{int}\rangle\,, \texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)})$

   $\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{remove}\,\langle\texttt{int}\rangle\,, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)})$

   $\Rrightarrow \left(\texttt{remove}\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [14, 3\,(8)]$

   $\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)})$

   $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [17, 3\,(8)]$

10. $\mathfrak{M} \models \texttt{get}\,\langle\texttt{int}\rangle \otimes \texttt{LL} < \texttt{Integer} > : \mathbb{METHOD}$

    $\Rrightarrow \mathfrak{M} \models \texttt{LL} < \texttt{Integer} > \texttt{.get(int)} : \mathbb{METHOD}$

    $\Rrightarrow \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}} \in \mathbb{METHOD}^{\mathfrak{M}}\ [10, 4\,(8)]$

    $\Rrightarrow \mathfrak{M} \models \otimes\,(\texttt{get}\,\langle\texttt{int}\rangle\,, \texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)})$

    $\Rrightarrow \mathfrak{M} \models \mathit{SignatureOf}\,(\texttt{get}\,\langle\texttt{int}\rangle\,, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)})$

    $\Rrightarrow \left(\texttt{get}\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}\ [13, 4\,(8)]$

    $\Rrightarrow \mathfrak{M} \models \mathit{MethodMember}\,(\texttt{LL} < \texttt{Integer} >, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)})$

    $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}\right) \in \mathit{MethodMember}^{\mathfrak{M}}\ [16, 4\,(8)]$

This completes design verification of the specified terms. Now we turn our attention to the proposition specified in *List*. We immediately turn this proposition into a series of smaller propositions (rule 2.b.), which we present in Table 12.7.

Table 12.7: four satisfaction relations for each proposition in *List*

11)	$\mathfrak{M} \models \mathit{Aggregate}\,(\texttt{LL} < \texttt{Integer} >, \texttt{Integer})$
12)	$\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}, \texttt{boolean})$
13)	$\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}, \texttt{boolean})$
14)	$\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.get(int)}, \texttt{Integer})$

11. $\mathfrak{M} \models \mathit{Aggregate}\,(\texttt{LL} < \texttt{Integer} >, \texttt{Integer})$

    $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{Integer}^{\mathfrak{M}}\right) \in \mathit{Aggregate}^{\mathfrak{M}}\ [20, 1\,(8)]$

12. $\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}, \texttt{boolean})$

    $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}, \texttt{boolean}^{\mathfrak{M}}\right) \in \mathit{Return}^{\mathfrak{M}}\ [21, 2\,(8)]$

13. $\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}, \texttt{boolean})$

    $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}, \texttt{boolean}^{\mathfrak{M}}\right) \in \mathit{Return}^{\mathfrak{M}}\ [23, 3\,(8)]$

14. $\mathfrak{M} \models \mathit{Return}\,(\texttt{LL} < \texttt{Integer} > \texttt{.get(int)}, \texttt{Integer})$

    $\Rrightarrow \left(\texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}, \texttt{Integer}^{\mathfrak{M}}\right) \in \mathit{Return}^{\mathfrak{M}}\ [22, 4\,(8)]$

With this we finish our sketch of the design verification process of this claim, from which we conclude:

Conclusion 3 Java 1.6's generic class LinkedList instantiated with the class argument Integer is a list by virtue of:

$$\mathfrak{M} \models \texttt{LL} < \texttt{Integer} > : \mathit{List}\,|\circ, \texttt{Integer}, \texttt{add}, \texttt{remove}, \texttt{get}|$$

We conclude from this case study that **RTC** can adequately specify and verify the structural properties of instances of Java's generic classes, and that to do so follows the same process as in our previous two case studies (§12.1 and §12.2).

Chapter 13

The Two-Tier Programming Toolkit

In Chapter 7 we introduced our theory of classes. We then demonstrated in Chapter 8 how this theory can be applied to specifying motifs of design patterns. In Chapter 11 we applied this theory to the practical task of design verification. We did this with a non-standard application of model theory as abstractions of programs, which we are able to verify against specification articulated in a restricted version of our theory. The practical application of our theory promotes its suitability for tasks such as round-trip engineering. Round-trip engineering is the combination of:

Forward engineering authoring (or generating) programs from a priori design specifications;

Reverse engineering creating visualizations (specifications) from a program's source code;

Design verification the ability to synchronize an implementation with its design documentation allowing for concurrent development.

Ideally, round-trip engineering tools can create and manage visual abstractions that represent arbitrarily large programs. These must be kept accurate and up to date throughout the development and evolution phases of the software's life cycle. Effective use of such tools can reduce development and evolution costs, and improve quality and productivity significantly. The two-tier programming (TTP) paradigm is a conceptual framework to the ends of round-trip engineering [Eden et al., 2003]. In it, a two-tier program is represented by two broad *tiers*:

Implementation Tier traditionally referred to as the *program*: the source code, libraries, compilation parameters, operational environment, etc.

Design Tier traditionally referred to as the *documentation*, articulated in a precise and formal language. That is, all the metadata regarding the program: any sort of precise specifications, diagrams, etc.

A two-tier program therefore consists of both these tiers as distinct components, an idea that is not uncommon. For example, model-driven architecture (MDA) separates a program into separate implementation (code) and design (models) tiers. MDA, which is strongly tied to the Unified Modelling Language (UML), is “an ambitious effort to build programs from models using model transformation” [Thomas, 2004]. The focus of MDA is therefore on code generation. However, in our experience code generation techniques often hijack a project. They tend to produce code that is very fragile to modification and difficult to maintain. For this reason we believe that such techniques have a limited scope to those projects (or self contained components thereof) where most of the code is to be generated directly, and are less applicable to existing and evolving systems.

The TTP paradigm is a more general approach to software development[113]. It places strong emphasis on design verification[114], which allows for concurrent evolution of both design and implementation, i.e. *coevolution*. Additionally, the TTP paradigm is not restricted to any particular languages in either tier. An implementation tier may contain a program written using both Java and C#, and the design tier may contain specifications written in LePUS3 and Spine for example. The only requirement is that elements of each tier can be kept consistent with the others. Such consistency and conformance checks must be fast, reliable, and repeatable. As such there is a tendency toward formal languages that have fully decidable methods of design verification, such as LePUS3 and our own Theory of Classes/Restricted Theory of Classes (**TC**/**RTC**)

Additional to design verification, the TTP paradigm also classifies other relationships between the design and implementation tiers. These are summarized as the following four broad operations:

Specify Write a priori design documents that specify a new program.

Implement Forward engineer (manually or otherwise) an implementation from a specification in the design tier.

Verify Ensure consistency between the design and implementation tier. This must not depend on fragile code instrumentation, i.e. littering the source with awkward comments that must not be removed or altered, or on extending the implementation language, for example ArchJava [Aldrich et al., 2002].

Visualize Reverse engineer design documentation from an implementation.

Where the tiers and their operations are best illustrated with the simple data-flow diagram in Figure 13.1.

[113] MDA could be considered to be a specialisation of the TTP paradigm.

[114] By which we mean the general act of checking consistency/conformance between the design and implementation tiers, using any appropriate notion of correctness.

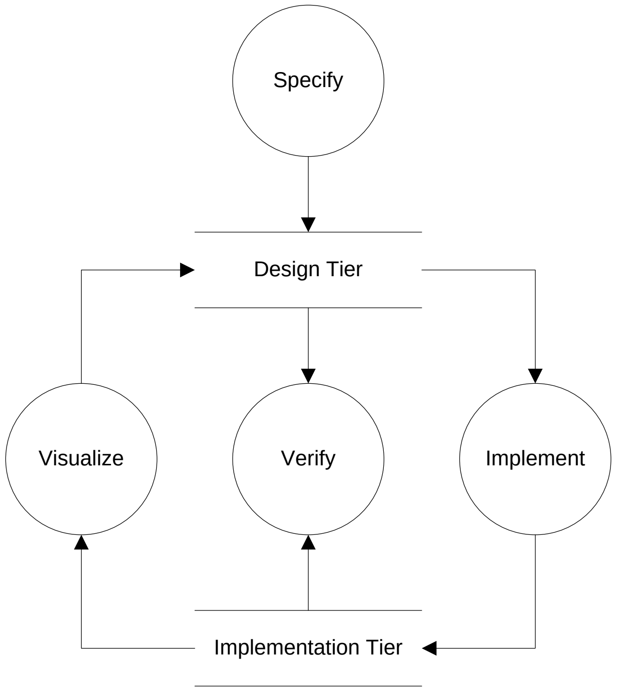


Figure 13.1: A data-flow diagram of two-tier programs

The Two-Tier Programming Toolkit [Nicholson et al., 2010], henceforth simply the *Toolkit*, is a proof of concept collection of tools that have been developed specifically for the TTP paradigm. The first version of the Toolkit (0.3) was developed by E.Gasparis and had simple functionality. It was a complete rewrite of previous tools developed in earlier research projects [Iyaniwura, 2003, Bo, 2004, Liang, 2004]. This served as a good platform for the development of further tools [Nicholson, 2006, Fragkos, 2006, Ayodeji, 2006]. These extensions highlighted several issues and limitations with version 0.3, so it was rewritten by J. Nicholson[115] and E. Gasparis[116] as version 0.5.1. The current version (0.5.4) contains several bug fixes and minor improvements to version 0.5.1 of the Toolkit. It supports analysing Java source code, visualizing (reverse engineering) and verifying it using LePUS3 as defined in [Eden and Nicholson, 2011]. More specifically, it provides the following features:

Pattern Library Download and deploy predefined pattern motifs, which currently consists of those specifications in [Eden et al., 2007a].

Editor Visually create Codecharts (representing LePUS3 specifications, Appendix C) with an intuitive drag-and-drop interface.

[115] J. Nicholson implemented the general user interface, the pattern library (including specification import/export), the abstract semantics (analyser) for Java 1.5, and the verifier modules.

[116] E. Gasparis implemented the design model, Codechart editor, and Design Navigator modules.

Analyser Generate abstract semantics of unmodified existing Java 1.5 programs, excluding certain aspects of Java's generics mechanism.

Verifier Ensures consistency of a programs abstract semantics with specifications represented by Codecharts in accordance to [Eden and Nicholson, 2011].

Design Navigator A step-wise, user guided, reverse engineering tool to produce useful formal specifications (Codecharts) from a programs abstract semantics [Gasparis, 2010].

As stated above, the Toolkit currently supports design verification as defined for LePUS3, i.e. in accordance with the definitions in Appendix B [Eden and Nicholson, 2011]. At this time it does not support **TC**/**RTC** or its method of design verification that we presented in Chapter 11. However, as these approaches are so similar, and as **RTC** follows the theoretical foundations of LePUS3 very closely, it takes no stretch of the imagination that the Toolkit could be modified to work with these new definitions. Therefore, what benefits and abilities the Toolkit currently demonstrates for LePUS3 can also be demonstrated for our theory of classes.

In the following sections we discuss two of the most interesting components of the Toolkit in more detail: the Design Navigator (§13.1), and the Verifier (§13.2). Lastly, we discuss a small empirical study of the Toolkit in §13.3. As this experiment had a very small sample size we are only able to drawn very limited conclusions from it. However, its results are promising and indicate that the Toolkit could make a big impact on the productivity of software developers.

13.1 The Design Navigator

[Gasparis, 2010] surveys the current state of design recovery tools, and develops a set of requirements that all such tools should comply with. To put our discussion of the Design Navigator in context, we paraphrase these requirements here:

User-guided The exploratory nature of reverse engineering requires a user's judgment as to what is useful from their perspective, and to discard irrelevant visualizations.

Automated discovery of design abstractions Design recovery methods should automatically detect related (sets of) program entities, and the correlations between them regardless of the physical structure and organization of the source code. For example, in Java the program entities should at least be (sets of) classes and methods, abstracted from a collection of decentralized source code files spread across a directory structure.

Programming Language-Independent Design recovery tools should not be overly generic. They need to be tailored to the paradigm/context in which they are to be used, but not to a specific implementation language. For example, a tool tailored to the object-oriented paradigm should be applicable to any object-oriented programming language.

Informative The opportunistic nature of design recovery, in combination with the vast size and complexity of programs, dictate that much of the intricate details in an implementation must be ignored. However, in any given context there is a minimal set of details to be preserved.

Verifiable A reverse engineered design specification must be as verifiable as an a priori one.

Non-intrusive A design recovery tool should complement the developer's existing practices, without adding much overhead or conflicting with existing tools, compilers and programming environments.

Scalable Programs usually span many thousands or even millions of lines of code. To address real world problems tools need to scale well. This means that they should be able to analyze and detect design information in programs of any size.

Visual Design recovery tools should ideally use a visual language or notation to express the extracted design information and should also visually support the navigation and exploration of the generated design representations.

The Design Navigator of course conforms to the above requirements [Gasparis, 2010]. Taking a top-down approach, the process starts with the most abstract depiction of a program[117] (a set of classes). The user selects a symbol in the Codechart that they want to inspect further (to *zoom-in*), or several symbols that they want to abstract away from (to *zoom-out*). Once an appropriate concretization or abstraction operation is applied to the selected symbol(s), a new Codechart is generated. This process ends at the point that the user has reached what they deem to be a sufficiently interesting and informative visualization of the program.

Consider a medium to large-sized software project, which is lacking adequate documentation. The documentation is either out of sync with the latest implementation, fails to address the concerns of particular stakeholders, or is simply nonexistent. What documentation does exist is therefore incomplete or misleading. These problems are common to a very wide range of free and

[117] The user can take a bottom-up approach, however our experience shows that this is undesirable due to the sheer volume of information that would be presented.

proprietary software and in both new and legacy systems. Visualization techniques help to fill in the gaps of the documentation by reverse engineering useful facts about an implementation's design, thereby helping the user to understand the system.

For example, we present a use case (Figure 13.2) that shows how the Toolkit can be used to produce design documentation from Java's Abstract Window Toolkit (AWT, package `java.awt`). The following specific sequence of steps illustrates the role of the Toolkit in this task:

1. Add all source code from the `java.awt` package to the project. This can be done by selecting individual files, or entire directories to be examined recursively. In this case we add the entire `java/awt` directory, which contains over sixty four thousand lines of code[118].

2. Click the "Analyze" button to generate an appropriate design model for the selected implementation.

3. Start a top-down visualization process in the Design Navigator tool, which presents the most generic representation of the entire program: a set of classes called `AllClasses`. For an overview of LePUS3, the language of Codecharts, see Appendix C.

4. Concretize `AllClasses` by partitioning it into a series of class hierarchies[119], most importantly note the hierarchy `ComponentHrc`. This reveals information about the relationships between the hierarchies, and between the method signatures that are shared in the hierarchies. Note the set of signatures `ComponentHrcOps` on the `ComponentHrc` hierarchy.

5. Focusing on the hierarchy `ComponentHrc` and the set of signatures `ComponentHrcOps`[120], expand the hierarchy to reveal more of its structure. In this case the *Abstract*, *Inherit*, *Aggregate* and *Forward* relationships.

6. Finally all the sets of methods and the set of classes are further expanded.

We notice that the Codechart resulting from step 5 looks similar to the Composite motif as presented in [Eden and Nicholson, 2011]. Indeed we have already shown that Java's AWT implements the Composite design motif in §12.1, but we may do this at the click of a button within the Toolkit as discussed in the next section.

[118] In Java 1.6 update 12, the `java.awt` package contains 64546 lines of executable code, excluding blank lines (13656) and comments (87210) across 386 files. Data collected using the program *CLOC* (*C*ount *L*ines *O*f *C*ode) [Danial, 2010] on 31/07/2010.

[119] Some symbols removed from the figure in the interests of simplicity.

[120] Achieved by abstracting the Codechart by removing all symbols except for the hierarchy `ComponentHrc` and the set of signatures `ComponentHrcOps`.

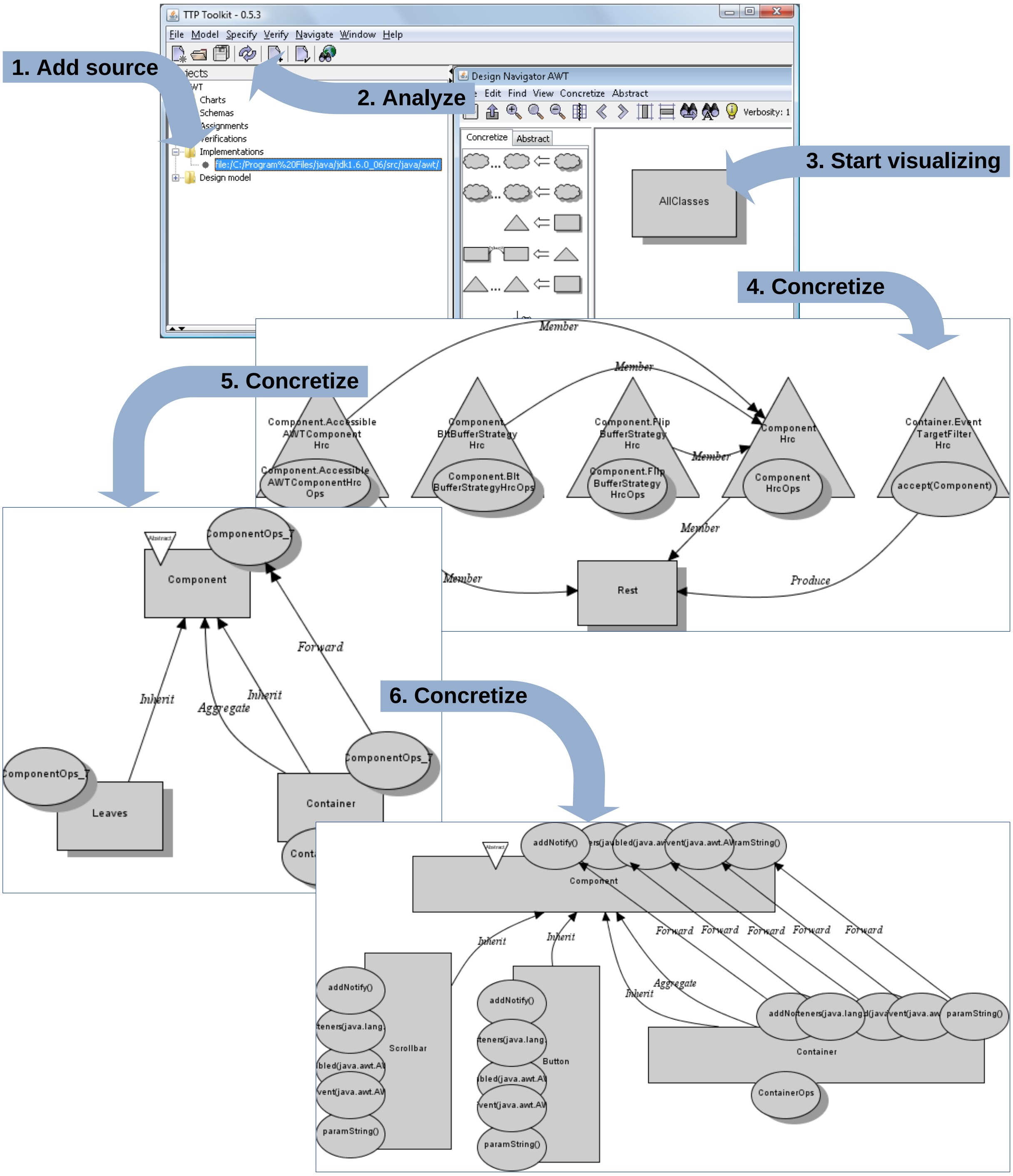


Figure 13.2: A use case for visualizing Java's AWT in the Toolkit

Let us briefly compare these results with those from other tools. For example, analyzing the `java.awt` package in NetBeans version 6.1 yielded Figure 13.3. NetBeans reverse engineers UML Class diagrams from a programs source code, but these diagrams do not scale very well [Gasparis, 2010]. Therefore, NetBeans violates our requirements for a design recovery tool that we summarized at the beginning of this section. NetBeans also requires the insertion of special tags into the implementation; these *merge markers* also violate our requirements.

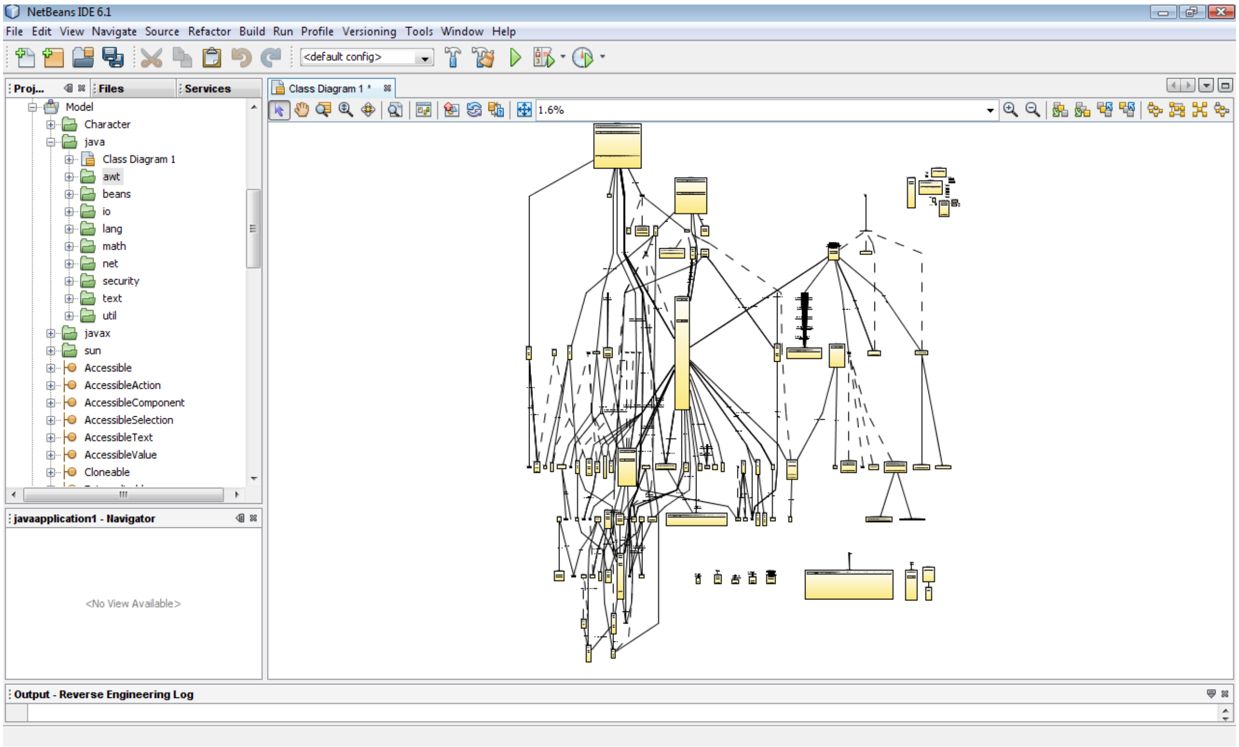


Figure 13.3: Visualizing Java's AWT in NetBeans 6.1

A more in depth discussion on NetBeans and other reverse engineering tools in light of the Design Navigator is out of the scope of this thesis, but can be found in [Gasparis, 2010].

13.2 The Verifier

The Toolkit implements the design verification algorithm as defined in Appendix C for LePUS3 specifications against design models abstracted from a program's source code[121]. This is a fully automated process that ensures consistency between the decidable properties of object-oriented programs with LePUS3 specifications. What this means in the Toolkit is that once the Verifier is passed the relevant configuration information, it can be executed at the click of a button, always

[121] As previously mentioned, in Chapter 11 we cleaned up and extended this method of design verification for our theory of classes.

completes within reasonable time, and produces consistent and meaningful results.

For example, we present a use case (Figure 13.4) that shows how the Toolkit can be used to verify the hypothesis that Java's Abstract Window Toolkit (AWT, package java.awt) implements the Composite design motif. We have already manually shown that this is the case in §12.1. We then evolve the implementation in a way which violates this specification so that the Toolkit then reports the inconsistency. The developer then decides if it is the design or implementation that should be modified to correct the problem. The following specific sequence of steps illustrates the role of the Toolkit in this task, and assumes that the implementation has already been added to the project and analyzed to create an appropriate design model:

1. Download and import into the project the LePUS3 specification of the Composite design motif from the Toolkit's pattern library.

2. State which parts of the program are participants of the design motif by assigning the variables in the specification to constants interpretable in the design model. In particular, we assign the variable $Composite$ to class `Container`, add the signature `addNotify()` to the set of component operations ($ComponentOPs$).

3. Click the "Verify" button to execute the design verification algorithm.

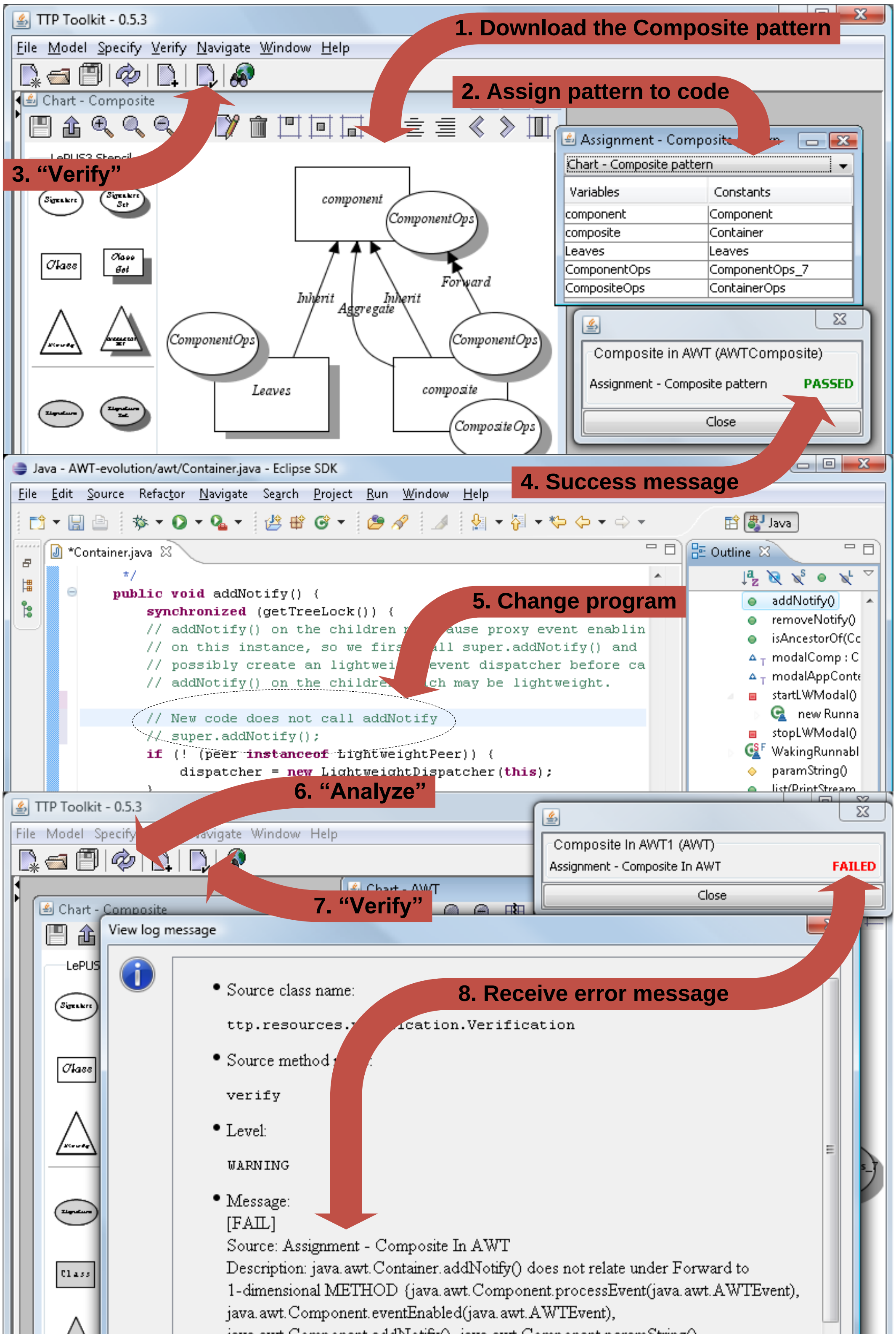
4. Receive a report that the implementation conforms to the Composite motif as assigned in step 1. Note that the current version of Container.addNotify() forwards the method call to Component.addNotify(), that is $Forward\,(\texttt{addNotify}() \otimes \texttt{Container}, \texttt{addNotify}() \otimes \texttt{Component})$ holds.

5. Evolve the program by editing the method Container.addNotify() and remove the statement that forwards the method call. We used Eclipse to do this, but any editor can be used.

6. Update the design model of these changes by clicking the "Analyze" button.

7. Re-execute the design verification algorithm by clicking the "Verify" button again.

8. Receive a report that the source code no longer conforms to the Composite motif as assigned in step 1. The precise nature of the violation is detailed to ensure the user understands the inconsistency, that $Forward\,(\texttt{addNotify}() \otimes \texttt{Container}, \texttt{addNotify}() \otimes \texttt{Component})$ no longer holds, so that they may reconcile it.



1. Download the Composite pattern

2. Assign pattern to code

3. "Verify"

Variables	Constants
component	Component
composite	Container
Leaves	Leaves
ComponentOps	ComponentOps_7
CompositeOps	ContainerOps

Composite in AWT (AWTComposite)
Assignment - Composite pattern PASSED
Close

4. Success message

```
    */
    public void addNotify() {
        synchronized (getTreeLock()) {
        // addNotify() on the children m   ause proxy event enablin
        // on this instance, so we firs   all super.addNotify() and
        // possibly create an lightwei   event dispatcher before ca
        // addNotify() on the childre   ich may be lightweight.

        // New code does not call addNotify
        // super.addNotify();
        if (! (peer instanceof LightweightPeer)) {
            dispatcher = new LightweightDispatcher(this);
        }
```

5. Change program

6. "Analyze"

Composite In AWT1 (AWT)
Assignment - Composite In AWT FAILED
Close

7. "Verify"

8. Receive error message

- Source class name:

  ttp.resources.v   ication.Verification
- Source method n   :

  verify
- Level:

  WARNING
- Message:
  [FAIL]
  Source: Assignment - Composite In AWT
  Description: java.awt.Container.addNotify() does not relate under Forward to 1-dimensional METHOD {java.awt.Component.processEvent(java.awt.AWTEvent), java.awt.Component.eventEnabled(java.awt.AWTEvent), java.awt.Component.addNotify(), java.awt.Component.paramString(),

Figure 13.4: A use case for evolving Java's AWT to violate the Composite motif in the toolkit

13.3 Empirical Evaluation

The merits of a new technology can only be measured with respect to the state of the art, and a software development tool should be no different. Claims about the benefits of a tool should be supported by reliable and repeatable experimental results. To this end, we conducted several small controlled experiments [Eden and Gasparis, 2009] to support the claim that the Toolkit improves the process of program understanding, program conformance and evolving programs. We break this claim into three distinct hypotheses, each of which formed the basis of an experiment. We hypothesized that the Toolkit:

H_1 reduces the time required to become sufficiently familiar with a program (comprehension);

H_2 improves the (structural) correctness of a program (conformance);

H_3 reduces the time required to evolve a program in accordance with new requirements (evolution).

With the appropriate respective null hypotheses, where the Toolkit:

H'_1 does not reduce the time required to become sufficiently familiar with a program;

H'_2 does not improve the (structural) correctness of a program;

H'_3 does not reduce the time required to evolve a program in accordance with new requirements.

At the beginning of each experiment subjects received one hour training in carrying out the tasks with each tool: the experimental software, the Toolkit, and the control software, Sun's integrated development environment (IDE) NetBeans version 6.1. To minimize bias each experiment was split into two equivalent tasks, and each set of subjects into two groups (A and B). During the first task of the experiment one of the groups acted as a control group (A), and the other was the experimental group (B). Once all subjects had completed the first task, each group took the opposite role in the second task. That is, group A became the experimental group and group B the control. The subjects, all final year undergraduate or graduate students, were paid for their time. To minimize possible bias they were paid for a fixed period[122] even if they took less time to finish each experiment. The specific tasks performed, and the data for our analysis, are both documented in Appendix F.

[122] Each participant was paid for a five hour period at a rate of £10 per hour, per experiment. Totalling £150 if they completed all three experiments.

Our method of analysing the data differs from the method originally used in the previous publications [Eden and Gasparis, 2009, Eden and Nicholson, 2011]. The previous results were obtained by analysing each task as an experiment in its own right; those figures were then combined into per experiment results. But as each pair of tasks were designed to be equivalent, so as to minimize bias, our approach was to combine results from each task. In doing so we were able to directly analyze the accumulated data and maximize our sample sizes. A side effect of this is that we had to exclude data of those subjects who did not complete both constituent tasks of an experiment[123]. This method of analysis is both a better fit with the definition of the experiment and yields better results, summarized in Table 13.1.

Table 13.1: Mean results of three experiments evaluating the Toolkit

Experiment	**Time (seconds)**		**Accuracy**	
	Toolkit	*NetBeans*	*Toolkit*	*NetBeans*
Comprehension	447	1945	100%	93%
Conformance	1291	1264	100%	57%
Evolution	233	177	100%	100%

In the first experiment, program comprehension, subjects were provided with excerpts from the Java SDK [Sun Microsystems Inc., 2006] and asked to find methods that met certain criteria (Table F.1). The subject's accuracy and time taken to complete each task were recorded (Table F.2). The results show that the Toolkit took only 23% of the time for the seven valid subjects to complete the experiment in comparison to the control software. This result is a statistically significant (at the 5% level) productivity gain, as illustrated in Figure 13.5, that allows us to reject the experiment's null hypothesis. However, with such a small sample size we cannot draw any conclusions about the wider community outside of the subjects that participated on our study. It is also worth mentioning that the results also show a minor decrease in accuracy when using the control software (Figure 13.6), but this is within acceptable tolerances for the experiment (±10%).

In the second experiment, program conformance, subjects were provided with a description of a design motif and excerpts of the Java SDK [Sun Microsystems Inc., 2006] and asked to confirm the consistency of the two (Table F.3). The subject's accuracy and time taken to complete each task were recorded (Table F.4). The results show that using the Toolkit nearly doubled the accuracy of the seven valid subjects in the performed tasks (1.75 times). This result is a statistically significant (at the 5% level) productivity gain, as illustrated in Figure 13.6, that allows us to reject the experiment's null hypothesis. However, with such a small sample size we cannot draw any

[123] 21% of all data had to be excluded.

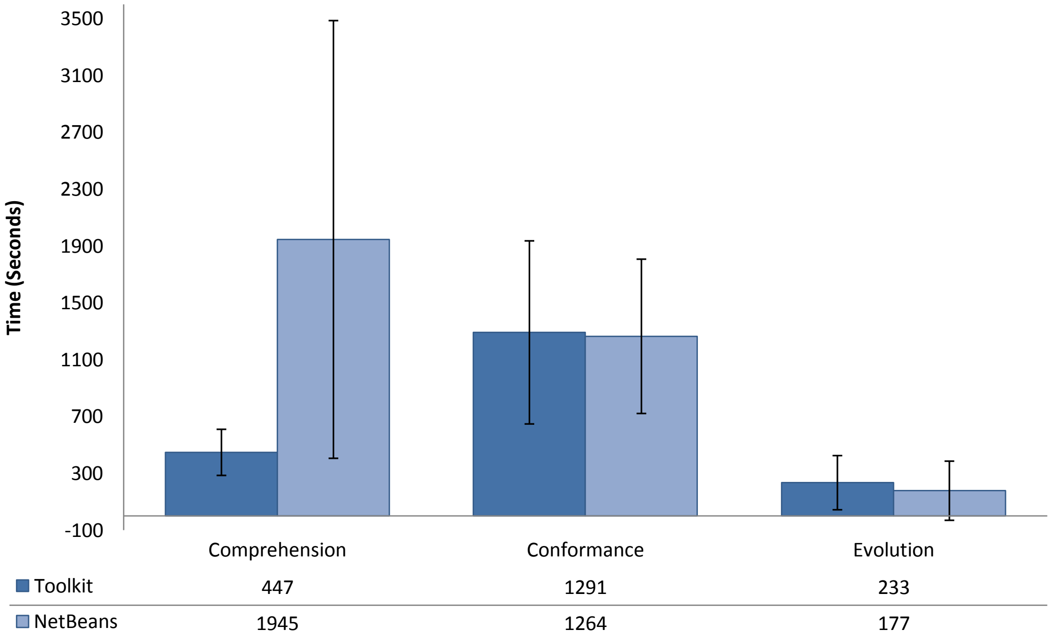


Figure 13.5: A graph depicting the mean time taken and the standard deviation each experiment

conclusions about the wider community outside of the subjects that participated on our study. The difference in the time taken for subjects to complete the experiment using either tool was negligible (Figure 13.5), although we would expect that a larger sample size would show a bigger difference.

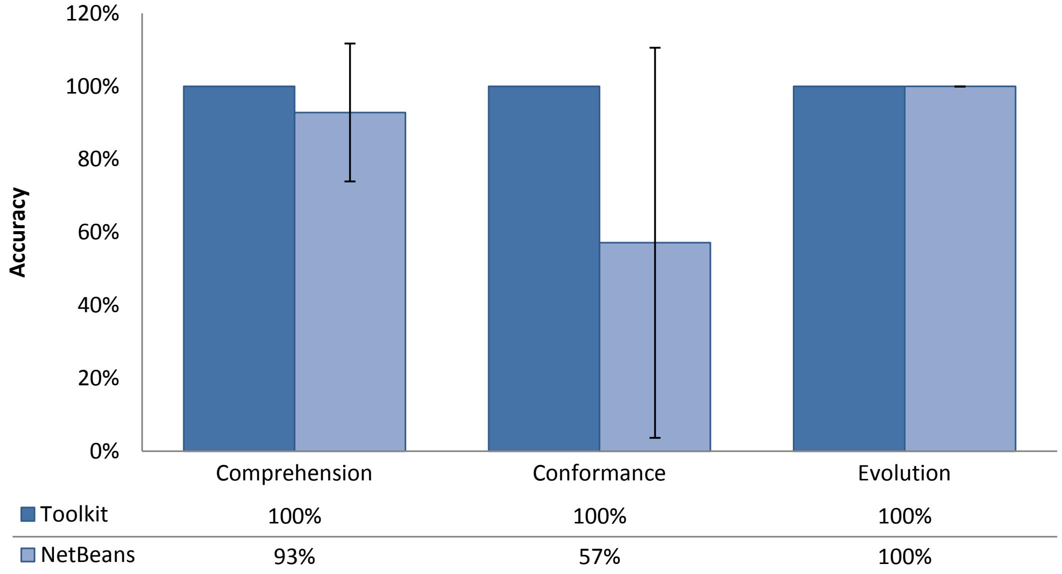


Figure 13.6: A graph depicting the mean accuracy and the standard deviation each experiment

In the final experiment, program evolution, subjects were provided with a description and excerpts of the Java SDK [Sun Microsystems Inc., 2006], and asked to evolve them to accommodate for a new method with a specified body (Table F.5). The five valid subjects accuracy and time taken to complete each task were recorded (Table F.6). The results show that these results are statistically insignificant, meaning that we cannot reject the experiment's null hypothesis. However, both the time and accuracy data indicate to us that this experiment was flawed. For example, half the tasks were unexpectedly completed in under two minutes. We suspect that the tasks were too simple to yield appropriate and meaningful results.

Despite the unexpected results of the software evolution experiment, overall these results are highly encouraging having shown evidence of consistent productivity gains. Although the sample size greatly limits the scope of these results, they do suggest that further studies should be conducted. Such a study would certainly need to examine a larger sample size[124], preferably selecting a majority of subjects from industrial positions rather than undergraduate and postgraduate students. Additionally, the evolution experiment showed that the complexity (or rather simplicity) of each component task could have a large impact on the results. With this in mind, we recommend further investigation into what tasks are set[125] so that they have more depth to them, thereby minimizing possible anomalous readings. Finally, NetBeans is only one of many possible industrial tools against which the Toolkit should be compared, a further study should compare the Toolkit to a wider range of industrial tools to show more conclusive benefits against current competition.

[124] Ideally a full industrial experiment should consist of at least 96 participants. Calculated with [Raosoft Inc., 2004] at a 10% margin of error, 95% confidence level (5% significance), a population size of $20,000$ and 50% response distribution.

[125] A greater number of tasks per experiment should also be set so as to further minimize bias, e.g. a participants existing familiarity with the supplied source code.

Chapter 14

Conclusion

Our motivation and aim of this thesis has been the problem of architectural erosion [Perry and Wolf, 1992], where over time software devolves into a continuous process of introducing violations to the design, ultimately making the system more brittle and difficult to maintain. This is one of the core challenges being faced by software, and a contributory factor to software's chronic crisis [Gibbs, 1994]. There is no single solution to software's chronic crisis, but a combination of good representations and methods can improve the situation.

Our work tries to close the gap between a program's design and its implementation through the fully automatable and decidable process of design verification. To accomplish this, our work is both strongly theoretical and practical. We therefore decomposed our aim into two goals (§1.1), one which focused on the theoretical challenges (**G1**), and one for the practical ones (**G2**). We structured our thesis to reflect these goals, in that Part II contained our theoretical investigation (**G1**), and Part III contained our practical investigation (**G2**). We will now discuss each of these parts in turn and identify how our research met the three objects we defined for each goal.

We began our theoretical investigation (**G1**) in Part I with an examination and critique of LePUS3, a formal and visual design description language for the structural and creational aspects of object-oriented design. LePUS3 [Eden and Nicholson, 2011], which evolved from the original oft-cited LePUS [Eden, 1999], has shown great promise in areas such as design verification [Nicholson et al., 2009] and program visualization [Gasparis, 2010]. However, its complicated and brittle definition, inadequate type system, and lack of reasoning capability within the language are all criticisms of LePUS3. We therefore devoted Part II of this thesis to teasing out, and improving on, the theoretical foundations of LePUS3 (Chapter 7).

The result of this was a more expressive theory of classes, which used the same fundamental building-blocks of object-oriented design as LePUS3 (**G1.1.1**). That is, our basic type system consisted of classes, methods, and sets thereof, which we then enriched with a range of interesting and decidable relationships. Our theory captures more details about object-oriented design (**G1.2**), for example many of the rules we introduced in §7.5 that govern relationships between our relations were not provided in LePUS3. Our theory is also more expressive than LePUS3 as we are able to define relationships such as *Overrides* and *Overloaded*, which were simply inexpressible in LePUS3. In light of these extensions, we demonstrated our theory's capability to represent design motifs (Chapter 8) in a greater level of detail than their respective LePUS3 counterparts.

Our theory is also much more rigorous than LePUS3 (**G1.1.2**). Our first reason for saying this is that we defined our theory with the Typed Predicate Logic (**TPL**), a very powerful framework for the definition and inference on type theories. In addition to this, we also based the rules of our theory on the same intuitions as LePUS3, although somewhat cleaned up. For example, in defining our theory in **TPL** we identified and fixed several problems and issues with the definition of LePUS3 that were only became apparent through our investigation. That is, we discovered and fixed several inherent flaws in the definition of LePUS3, such as the problem with method overriding, §7.3. Indeed, the complexity of the solution we provided to the method overloading problem suggests that there needs to be further investigation in this area such that a more optimal solution can be found. Our theory easily facilitates this, as opposed to LePUS3 in which we were unable to even identify the problem. Another example comes from previous reviewers of LePUS3 who had identified certain aspects of the language to be ambiguous, such as the Total and Isomorphic predicates (§7.6). The names of these predicates were described as misleading as their definition was not the standard interpretation of the terms. We have responded to this criticism by renaming them (Tot and Iso respectively) and clarifying their definition.

The scalability and genericity of our theory is increased in comparison to LePUS3 (**G1.1.3** and **G1.1.4**). In our theory we are able to specify an unbounded number of possible implementations of any size in the same way we can in LePUS3. However, we are able to specify these implementations both in more detail, and at a greater level of abstraction. For example, in Chapter 9 we refined our basic building blocks so that we captured more information about signatures: their identifier and arguments. In doing this we are able to capture object-oriented design in more detail. Similarly, we are able to specify implementations in more abstract ways, mainly as a result of using **TPL**. For example, the universe of types allow us to define polymorphic specifications. We demonstrated this in Chapter 10 were we abstracted both the Abstract Factory and Factory Method motifs into a more abstract (generic) specification for a family of factory motifs. This chapter also extends our theory to allow us to dynamically add more subtypes to our theory (**G1.3**). That is, we introduced a simple mechanism that, in a single step, turns relations into subtypes that enrich our theory.

In Part III we applied our Theory of Classes to design verification (**G2**). We defined design verification for a subset of our theory of classes (**G2.1**, Chapter 11), based on the same non-standard use of model theory. We could have accomplished this with pattern matching algorithms for example, but we decided to keep our definition close to that of LePUS3. However, we ensured that our use of model theory was, although still non-standard, closer to the definitions and terminology of standard model theory. We proceeded to demonstrate design verification in three more detailed case studies (Chapter 12): verifying an instance of the Composite motif in Java's Abstract Window Toolkit; verifying an example JUnit test case against the requirements of the JUnit testing framework; and verifying instances of the Java LinkedList generic class against our generic *List* specification. Finally we discussed an implementation of design verification for LePUS3 (the Verifier), and an associated reverse engineering tool (the Navigator), in the Two-Tier Programming Toolkit (**G2.2**, Chapter 13). Despite being an implementation of the older definition in [Eden and Nicholson, 2011], the Toolkit demonstrates that design verification mechanisms like ours are implementable, conclusive, and complete within reasonable time. But further work is needed to update the Toolkit to support our theory rather than LePUS3 and our design verification mechanism. We also described, and reanalyzed, the results from a small empirical study (**G2.3**, §13.3) of the Toolkit for typical software re-engineering tasks. The study showed, among other results, a statistically significant (at the 5% confidence interval) and substantive 175% improvement in software accuracy. However, our sample size was too small to draw any conclusions outside of the experimental group; a much larger study, preferably conducted with subjects from industry, would have to be conducted to establish that these findings hold in the wider world of software development.

The overall result of this research is a rigorous theory of classes that is based on, and a distinct improvement to, LePUS3. We have also shown that, like LePUS3, our theory can also be applied to the task of design verification. We therefore conclude that our research has met our original objectives. In the next section we briefly outline several future investigations that can be conducted based on our work.

14.1 Future Investigation

This body of work encompasses both theoretical and practical investigations, and as such there are many avenues for future research. We present some of these in the following short list:

1. **Relations**. There are lots of other decidable properties of object-oriented programs that we could capture, for a short summary of these see §7.5.10. Additionally, our investigation into this subject, and our intuitions regarding point 4 below, suggests that the *Abstract* relation may be a confusion originating in the syntax of implementation languages. In many implementation languages a class is considered to be concrete (instantiable) if it lacks the abstract (or similar) keyword. This is misleading as it indicates a class is concrete by default, but the semantics suggest the opposite.

2. **Inheritance structure**. Inheritance is generally considered to be a tree, but most object-oriented programming languages have a root class from which all other classes inherit, and a null object (keyword) that could be considered to have a class that inherits from every other class. If this is the case is inheritance not a lattice?

3. **Class signatures**. Classes are often given signatures in the same way methods are, they name classes and even defining argument types in the case of parameterized classes [Sun Microsystems Inc., 2006]. This is an interesting avenue of research, but should be pursued in conjunction with point 4 below.

4. **Objects**. We believe that this could best be achieved by turning classes into types. Doing so would simplify the theory in several places, for example we can remove the problematic rules $\mathbf{TC}_{5a}$ (p.88) and $\mathbf{TC}_{5b}$ (p.89). Conversely it would cause several issues elsewhere, for example if for $c : \mathbb{CLASS}$ the relation $\mathit{Abstract}\,(c)$ holds then no subclass of c can ever not be abstract without causing a contradiction. Introducing objects would allow for more detailed specification; relationships between, and the roles of, objects could be specified. However, this would come at the sacrifice of fully automated design verification.

5. **Design recovery**. Design navigation [Gasparis, 2010] is a step-wise user-driven design recovery tool mechanism on [Eden and Nicholson, 2011], allowing users to *zoom-in* and *zoom-out* of the design of their programs (see §13.1). As we have redefined the language on which design navigation was based, it is logical to also redefine and perhaps extend design navigation in terms of our new theory. One possible extension to design navigation could be the addition of dependency graphs, discussed in a little more detail in §E.2. Another could be using genetic programming techniques, in combination with software metrics (see point 5), to automatically calculate interesting visualizations of a program as alternative starting points for design navigation.

6. **Statistical analysis** and **Metrics**. The use of our theory to discover statistics about a program being analyzed. To name only a few:

   (a) The size of sets, specifically hierarchies and sets of dynamically bound methods (clans).

   (b) Calculating the depth of inheritance hierarchies. This would help to identify areas of a program that are structurally flat and might be candidates for restructuring.

   (c) Hints toward the identification of refactoring candidates through the detection of *bad smells* and *anti-patterns*. A simple example of this would be identification of God classes and methods by counting relationships.

7. **Pattern detection**. Define an algorithm to search for instances of appropriately specified pattern motifs.

8. **Abstract semantics**. Our publications thus far only allude to the definition of an abstract semantics, and one task could be to formally define the pattern matching algorithm for the languages we are interested in.

9. **Relationships between programs**. Design models allow us to compare programs at the structural level (discussed in §E.1), which could play an important role in areas such as program translation and compilation. This suggests that our theory could have applications in the area of refactoring [Mens et al., 2002]. Our theory could be used to specify the external interfaces of programs that must remain unchanged (the invariants) between each application of a refactoring technique.

10. **Patterns**. There are lots of other object-oriented pattern motifs that we could capture, such as those in [Gil and Maman, 2005]. Capturing pattern motifs allow us to further study their relationships and similarities as we have begun to do in this thesis. Beyond this, our theory could be further extended to capture more aspects of patterns in the same vein as [Raje and Chinnasamy, 2001].

11. **Implementation** and **Experimentation**. As stated in Chapter 13, the Toolkit is no longer up to date with our current knowledge. It should be re-implemented to facilitate further experimental investigation with the most up-to-date definitions.

Appendices

Appendix A

Proof Notation

We adopt a Fitch-style notation for detailed proof derivations, similar to the box notation used in [Huth and Ryan, 2000]. We find this notation easier to construct and understand than proof trees as it allows us to "concentrate only on finding the proof, not on how to fit a growing tree onto a sheet of paper" [Huth and Ryan, 2000], and maintains the a hierarchical structure of proof as advocated by Lamport [Lamport, 1995]. The rest of this section provides a brief overview to our notation.

The simplest sequent we might need to prove would be of the form:

$$\Gamma \vdash \Phi$$

where the context Γ is a series of judgments that constitute our *premises*, from which we conclude judgment Φ. Suppose that to prove this sequent we need only apply some rule r. This is represented in the following proof tree:

$$\frac{\Gamma}{\Phi}\, r$$

Compare this to the flattened approach presented below:

$$\begin{array}{lll} 1) & \Gamma & \text{premise} \\ 2) & \Phi_b & r\ 1 \end{array}$$

Notice that we number each line of the proof. This ensures clarity as to exactly what steps in the proof are used when applying a rule. In this way we are able to trace results.

Example 16 Let us examine a case from [Huth and Ryan, 2000, Example 1.4, p.6], where the sequent to be proved is:

$$\phi \wedge \varphi, \psi \vdash \varphi \wedge \psi$$

for which we would construct the following proof tree:

$$\frac{\dfrac{\phi \wedge \varphi}{\varphi}\,\mathbf{L}_{11} \qquad \psi}{\varphi \wedge \psi}\,\mathbf{L}_9$$

This can be flattened and represented in our notation very easily, as shown below[126]:

1)	$\phi \wedge \varphi$	premise
2)	ψ	premise
3)	φ	$\mathbf{L}_{11}$ 1
4)	$\varphi \wedge \psi$	$\mathbf{L}_{9}$ 2,3

Often, proof requires making one or more temporary *assumptions.* Consider the following step in a proof tree:

$$\frac{\Phi_p \qquad \Phi_a \vdash \Phi_s}{\Phi_c}\, r$$

Judgment Φ_p is a premise, as is judgment Φ_s.under the assumption of judgment Φ_a, and by application of some other rule r we conclude judgment Φ_c. Compare this to the same written in our flattened notation:

1)	Φ_p	premise
2)	$\mid \Phi_a$	assumption
3)	$\mid \Phi_s$	
4)	Φ_c	r 1,2,3

Notice the horizontal line that helps us to visually identify the step at which an assumption is made, and the horizontal line that identifies the scope of the assumption.

Example 17 Let us examine another case from [Huth and Ryan, 2000, Example 1.13, p.15], where the sequent to be proved is:

$$\phi \wedge \varphi \implies \psi \vdash \phi \implies (\varphi \implies \psi)$$

To prove this we would construct the following proof tree:

$$\cfrac{\cfrac{\cfrac{\phi \wedge \varphi \implies \psi \qquad \phi, \varphi \vdash \phi \wedge \varphi}{\phi, \varphi \vdash \psi}\,\mathbf{L}_{19}}{\phi \vdash \varphi \implies \psi}\,\mathbf{L}_{18}}{\phi \implies (\varphi \implies \psi)}\,\mathbf{L}_{18}$$

which can be flattened and represented in our notation very easily, as shown below[127]:

1)	$\phi \wedge \varphi \implies \psi$	premise
2)	$\mid \phi$	assumption
3)	$\mid \mid \varphi$	assumption
4)	$\mid \mid \phi \wedge \varphi$	$\mathbf{L}_{9}$ 2,3
5)	$\mid \mid \psi$	$\mathbf{L}_{19}$ 1,4
6)	$\mid \varphi \implies \psi$	$\mathbf{L}_{18}$ 3,5
7)	$\phi \implies (\varphi \implies \psi)$	$\mathbf{L}_{18}$ 2,6

[126] $\mathbf{L}_9$ and $\mathbf{L}_{11}$, p.46

[127] $\mathbf{L}_9$, p.46; $\mathbf{L}_{18}$ and $\mathbf{L}_{19}$, p.46

However, such detailed proofs are not always required or desirable. We keep our proofs as detailed as we believe to be reasonable. In the above examples, each step is the result of a premise, assumption, or an application of an individual rule. At times it is desirable to less verbose by referencing previous proofs, or stating that the result is immediate without going into greater detail. For example, suppose we have previously proved that $a, b : T$ and $a = b$, then it follows immediately that $\exists x : T \bullet a = x$ where $x = b$. We opted to skip the intermediate steps as the result is so direct.

Appendix B

Summary of LePUS3 Definitions

The following definitions are taken from [Eden and Nicholson, 2011][128], and are provided strictly as a reference to the style of definition of LePUS3. Note that these definitions are written in the First-Order Predicate Logic (**FOPL**) and use a *completely* different set of notational conventions, which we detail below:

- Iff is shorthand for "if and only if"
- The notation $\{\underline{\mathtt{a}}, \underline{\mathtt{b}}\}$ denotes a set with two elements $\underline{\mathtt{a}}$ and $\underline{\mathtt{b}}$
- Given a set T, we use $\mathcal{P}(T)$ to denote the set of all subsets of T (the power set of T), and $\mathcal{P}^n(T)$ to denote $\mathcal{P}\left(\mathcal{P}^{n-1}(T)\right)$, where $\mathcal{P}^0(T)$ is T
- The notation $\uplus$ stands for the disjoint union
- Text styled as abc is a source code extract, $\mathtt{abc}$ is a constant, $\underline{\mathtt{abc}}$ is an entity, and *abc* is a variable
- Upper case Greek letters Φ, Ψ, Δ, ... are reserved for specifications, i.e. Codecharts and schemas
- Unless otherwise specified, $\underline{\mathit{Relation}}$ is a finite set represented by the relation symbol *Relation* and the interpretation of $\mathtt{constant}$ is $\underline{\mathtt{constant}}$

LePUS3 Definition I A **finite structure** is a pair $\mathfrak{F} = \langle \mathbb{U}_0, \mathbb{R} \rangle$ such that:

1. $\mathbb{U}_0$ is a set of primitive elements or **entities of dimension** *0* called the **universe** of $\mathfrak{F}$.
2. $\mathbb{R}$ is a finite set of relations.

LePUS3 Definition II A **unary** [**binary**] **relation** is a set of 1-tuples [2-tuples, or *pairs*] of entities of dimension 0. A **class** [**method**, **signature**] of dimension 0 is an entity in the

[128] Some very minor modifications have been made to the definitions in the conversion from the original Microsoft Word format to LaTeX.

relation $\underline{\mathit{Class}}$ [$\underline{\mathit{Method}}$, $\underline{\mathit{Signature}}$]. A **class** [**method**, **signature**] **of dimension** d is a set of classes [methods, signatures] of dimension $d-1$.

LePUS3 Definition III Given a binary relation $\underline{\mathit{BinaryRelation}}$, the transitive closure of $\underline{\mathit{BinaryRelation}}$, written $\underline{\mathit{BinaryRelation}^{+}}$, is that set of pairs $\langle x, y\rangle$ such that at least one of the following conditions hold:

1. $\langle x, y\rangle \in \underline{\mathit{BinaryRelation}}$
2. there exists an element z in $\mathbb{U}$ such that $\langle x, z\rangle \in \underline{\mathit{BinaryRelation}}$ and $\langle z, y\rangle \in \underline{\mathit{BinaryRelation}^{+}}$.

LePUS3 Definition IV A class of dimension 1, $\underline{\texttt{Hrc}}$, is also called a **hierarchy of dimension** 1, or simply a **hierarchy** iff both the following conditions hold:

1. $\underline{\texttt{Hrc}}$ contains at least two classes of dimension 0, and
2. $\underline{\texttt{Hrc}}$ contains a class of dimension 0 $\underline{\texttt{root}}$ such that for all other classes x in $\underline{\texttt{Hrc}}$: $\langle x, \underline{\texttt{root}}\rangle \in \underline{\mathit{Inherit}^{+}}$.

LePUS3 Definition V The **superimposition operator** $\otimes$ is a binary partial-functional relation. Let $\underline{\texttt{sig}}$ designate a signature of dimension 0, $\underline{\texttt{cls}}$ a class of dimension 0, then:

1. If there exists $\underline{\texttt{mth}}$ a method of dimension 0 such that $\left\langle \underline{\texttt{sig}}, \underline{\texttt{mth}}\right\rangle \in \underline{\mathit{SignatureOf}}$ and $\langle \underline{\texttt{cls}}, \underline{\texttt{mth}}\rangle \in \underline{\mathit{Member}}$ then $\underline{\texttt{sig}} \otimes \underline{\texttt{cls}} \triangleq \underline{\texttt{mth}}$;
2. Otherwise, if there exists a $\underline{\texttt{supercls}}$ class of dimension 0 where $\left\langle \underline{\texttt{cls}}, \underline{\texttt{supercls}}\right\rangle \in \underline{\mathit{Inherit}}$ such that $\underline{\texttt{sig}} \otimes \underline{\texttt{supercls}}$ is defined then $\underline{\texttt{sig}} \otimes \underline{\texttt{cls}} \triangleq \underline{\texttt{sig}} \otimes \underline{\texttt{supercls}}$;
3. Otherwise, $\underline{\texttt{sig}} \otimes \underline{\texttt{cls}}$ is undefined.

Let $\underline{\texttt{Signatures}} = \{\underline{\texttt{s}}_1, \ldots, \underline{\texttt{s}}_n\}$ be signature of dimension 1, and $\underline{\texttt{Classes}} = \{\underline{\texttt{c}}_1, \ldots, \underline{\texttt{c}}_k\}$ a class of dimension d. Then we also define:

1. $\underline{\texttt{sig}} \otimes \underline{\texttt{Classes}} \triangleq \left\{\underline{\texttt{sig}} \otimes \underline{\texttt{c}}_1, \ldots, \underline{\texttt{sig}} \otimes \underline{\texttt{c}}_k\right\}$ ($\underline{\texttt{sig}} \otimes \underline{\texttt{Classes}}$ is a **clan**).
2. $\underline{\texttt{Signatures}} \otimes \underline{\texttt{cls}} \triangleq \{\underline{\texttt{s}}_1 \otimes \underline{\texttt{cls}}, \ldots, \underline{\texttt{s}}_n \otimes \underline{\texttt{cls}}\}$ ($\underline{\texttt{Signatures}} \otimes \underline{\texttt{cls}}$ is a **tribe**).
3. $\underline{\texttt{Signatures}} \otimes \underline{\texttt{Classes}} \triangleq \{\underline{\texttt{s}}_1 \otimes \underline{\texttt{c}}_1, \ldots, \underline{\texttt{s}}_n \otimes \underline{\texttt{c}}_k\}$ ($\underline{\texttt{Signatures}} \otimes \underline{\texttt{Classes}}$ is a **tribe of clans**).

LePUS3 Definition VI Constants ($\mathtt{x}$) and variables (y) are terms in the language such that:

1. A d-dimension class is a term of type $\mathcal{P}^d\mathbb{CLASS}$.
2. A d-dimension signature is a term of type $\mathcal{P}^d\mathbb{SIGNATURE}$.
3. A d-dimension hierarchy is a term of type $\mathcal{P}^d\mathbb{HIERARCHY}$ and also of type $\mathcal{P}^{d+1}\mathbb{CLASS}$.
4. If cls is a c-dimension class, and sig is an s-dimension signature, then $sig \otimes cls$ is a term of type $\mathcal{P}^{c+s}\mathbb{METHOD}$. If either cls or sig are variables, then so too is $sig \otimes cls$.
5. Finally, a term of type $\mathcal{P}^d\mathbb{T}$ is called a **d-dimensional term**. $\mathcal{P}\mathbb{T}$ is short-hand for type $\mathcal{P}^1\mathbb{T}$ and $\mathbb{T}$ is shorthand for type $\mathcal{P}^0\mathbb{T}$.

LePUS3 Definition VII A **design model** is a triple $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ such that:

1. $\mathbb{U}_* \triangleq \mathbb{U}_0 \uplus \mathbb{U}_1 \uplus \ldots \uplus \mathbb{U}_d$ is the **universe** of $\mathfrak{M}$ where each $\mathbb{U}_k$ is a finite set of entities of dimension k and d is some small natural number (usually no greater than 3).
2. $\mathbb{R}$ is a set of relations including the unary relations $\underline{Class}$, $\underline{Method}$, $\underline{Signature}$, and $\underline{Abstract}$, and the binary relations $\underline{Inherit}$, $\underline{Member}$, $\underline{Produce}$, $\underline{Call}$, $\underline{Return}$, $\underline{Forward}$, and $\underline{SignatureOf}$.
3. $\mathcal{I}$ is an interpretation function which maps some constant terms to entities in $\mathbb{U}_*$. If the superimposition $\mathcal{I}(t_1) \otimes \mathcal{I}(t_2)$ is defined then $\mathcal{I}(t_1 \otimes t_2) \triangleq \mathcal{I}(t_1) \otimes \mathcal{I}(t_2)$. $\mathcal{I}(\tau)$ is also called the interpretation of τ.
4. $\mathfrak{M}$ satisfies the *axioms of class-based programs.*

LePUS3 Definition VIII The **axioms of class-based programs** are the following:

Axiom 1 No two methods with the same signature are members of the same class.

Axiom 2 There are no cycles in the inheritance graph.

Axiom 3 Every method has exactly one signature.

Axiom 4 If a method produces instances of a class it also creates it and returns it; if one method forwards the call to another it can be said to call it; and if one class holds an aggregate of another, it can also be said to hold a member of it.

LePUS3 Definition IX A unary relation symbol marked $UnaryRelation$ placed over a term t stands for the **ground formula** $UnaryRelation\,(t)$. A binary relation symbol marked

$\mathit{BinaryRelation}$ connecting t_1 to t_2 stands for the ground formula $\mathit{BinaryRelation}\,(t_1, t_2)$. An All predicate symbol marked $\mathit{UnaryRelation}$ placed over τ stands for the **predicate formula** $\textsc{All}(\mathit{UnaryRelation}, \tau)$. A Total predicate symbol marked $\mathit{BinaryRelation}$ connecting τ_1 with τ_2 stands for the predicate formula $\textsc{Total}(\mathit{BinaryRelation}, \tau_1, \tau_2)$. An Isomorphic predicate symbol marked $\mathit{BinaryRelation}$ connecting τ_1 and τ_2 stands for the formula $\textsc{Isomorphic}(\mathit{BinaryRelation}, \tau_1, \tau_2)$. A **well-formed formula** (in short: **formula**) is either a *ground formula* or a *predicate formula.*

LePUS3 Definition X A design model $\mathfrak{M}$ **satisfies the ground formula** $\mathit{UnaryRelation}\,(t)$ iff

$\mathcal{I}\,(t_1) \in \underline{\mathit{UnaryRelation}}$. It satisfies the ground formula $\mathit{BinaryRelation}\,(t_1, t_2)$ if either:

1. $\langle \mathcal{I}\,(t_1)\,, \mathcal{I}\,(t_2)\rangle \in \underline{\mathit{BinaryRelation}}$, or
2. there exists a class $\underline{\texttt{sprcls}} \in \mathbb{U}_0$ such that $\mathit{BinaryRelation}\,(\texttt{sprcls}, t_2)$ and $\mathit{Inherit}\,(t_1, \texttt{sprcls})$, or
3. there exists a class $\underline{\texttt{subcls}} \in \mathbb{U}_0$ such that $\mathit{BinaryRelation}\,(t_1, \texttt{subcls})$ and $\mathit{Inherit}\,(\texttt{subcls}, t_2)$.

LePUS3 Definition XI A design model $\mathfrak{M}$ **satisfies an All predicate formula**

$\textsc{All}(\mathit{UnaryRelation}, \tau)$ iff for each entity $\underline{\mathsf{e}} \in \mathcal{I}\,(\tau)$ [if τ is a 0-dimensional term, $\underline{\mathsf{e}} = \mathcal{I}\,(\tau)$] $\mathfrak{M} \models \mathit{UnaryRelation}\,(\mathsf{e})$.

LePUS3 Definition XII A design model $\mathfrak{M}$ **satisfies a Total predicate formula**

$\textsc{Total}(\mathit{BinaryRelation}, \tau_1, \tau_2)$ iff for each entity $\underline{\mathsf{e}}_1 \in \mathcal{I}\,(\tau_1)$ [if $\underline{\mathsf{e}}_1$ is a 0-dimensional term, $\underline{\mathsf{e}}_1 = \mathcal{I}\,(\tau_1)$] that is not an abstract method there exists some entity $\underline{\mathsf{e}}_2 \in \mathcal{I}\,(\tau_2)$ [if $\underline{\mathsf{e}}_2$ is a 0-dimensional term,

$\underline{\mathsf{e}}_2 \in \mathcal{I}\,(\tau_2)$] such that $\mathfrak{M} \models \mathit{BinaryRelation}\,(\mathsf{e}_1, \mathsf{e}_2)$.

LePUS3 Definition XIII A design model $\mathfrak{M}$ **satisfies an Isomorphic predicate formula**

$\textsc{Isomorphic}(\mathit{BinaryRelation}, \tau_1, \tau_2)$ iff there exists a pair of entities $\langle \underline{\mathsf{e}}_1, \underline{\mathsf{e}}_2 \rangle$ where $\underline{\mathsf{e}}_1 \in \mathcal{I}\,(\tau_1)$ [if $\underline{\mathsf{e}}_1$ is a 0-dimensional term, $\underline{\mathsf{e}}_1 = \mathcal{I}\,(\tau_1)$] and $\underline{\mathsf{e}}_2 \in \mathcal{I}\,(\tau_2)$ [if $\underline{\mathsf{e}}_2$ is a 0-dimensional term, $\underline{\mathsf{e}}_2 \in \mathcal{I}\,(\tau_2)$] such that both conditions hold:

1. $\mathfrak{M} \models \mathit{BinaryRelation}\,(\mathsf{e}_1, \mathsf{e}_2)$, unless both $\underline{\mathsf{e}}_1$ and $\underline{\mathsf{e}}_2$ are abstract, and

2. $\textsc{Isomorphic}(\mathit{BinaryRelation}, \tau_1 - \underline{\mathsf{e}}_1, \tau_2 - \underline{\mathsf{e}}_2)$, unless both $\tau_1 - \underline{\mathsf{e}}_1$ and $\tau_2 - \underline{\mathsf{e}}_2$ are empty,

where $\mathcal{I}(\tau - \mathsf{e}) = \mathcal{I}(\tau) - \mathcal{I}(\mathsf{e})$.

LePUS3 Definition XIV An **assignment** from specification Ψ into a design model $\mathfrak{M}$ is a function g mapping each variable in Ψ to a constant in the domain of $\mathcal{I}$. The notation $\Psi[g(x)/x]$ stands for the specification that results from replacing all occurrences of variable x with the constant $g(x)$ in Ψ. The notation $\Psi[g(x_1)/x_1, \ldots, g(x_n)/x_n]$ stands for the specification that results from the consistent replacement of all occurrences of each variable x_i with the constant $g(x_i)$ in Ψ.

LePUS3 Definition XV We say that design model $\mathfrak{M}$ **satisfies** closed specification Ψ, written $\mathfrak{M} \models \Psi$, iff each one of the terms of Ψ has an interpretation and each one of the formulas is *satisfied*. We say that a design model $\mathfrak{M}$ **satisfies** open specification Φ **under** assignment g, written $\mathfrak{M} \models_g \Phi$, iff g maps each variable in Φ to a constant in the domain of $\mathcal{I}$ such that $\mathfrak{M}$ satisfies the closed specification $\Phi[g(x_1)/x_1, \ldots, g(x_n)/x_n]$. We say that $\mathfrak{M}$ satisfies open specification Φ, written $\mathfrak{M} \models \Phi$, iff there exists some mapping g such that $\mathfrak{M} \models_g \Phi$.

LePUS3 Definition XVI We say that a finite structure $\mathfrak{F} = \langle \mathbb{U}_0, \mathbb{R} \rangle$ is a submodel of design model $\mathfrak{M}$ if
$\mathfrak{M} = \langle \mathbb{U}_0 \uplus \ldots, \mathbb{R}, \mathcal{I} \rangle$.

LePUS3 Definition XVII Given an abstract semantics function $\mathcal{A}$ and a program p in the domain of $\mathcal{A}$, we say that *design model* $\mathfrak{M}$ **appropriately represents** p if $\mathcal{A}(p)$ is a submodel of $\mathfrak{M}$.

LePUS3 Definition XVIII Given a program p written in programming language $\mathbb{L}$, abstract semantics function $\mathcal{A} : \mathbb{L} \longmapsto \mathfrak{F}^*$, a design model $\mathfrak{M}$ and chart Ψ, we say that p **implements** Ψ **according** to $\mathfrak{M}$ iff:

1. $\mathfrak{M}$ *appropriate represents* p, and
2. $\mathfrak{M}$ *satisfies* Ψ.

Given program p and specification Ψ, p implements Ψ iff there exists some design model $\mathfrak{M}$ such that p implements Ψ according to $\mathfrak{M}$.

LePUS3 Definition XIX Given specifications Φ and Ψ, we say that Φ semantically entails Ψ, written $\Phi \models \Psi$, iff every design model that *satisfies* Φ also *satisfies* Ψ.

Appendix C

Codecharts Extended

This appendix covers the basics to translating LePUS3 Codecharts to specifications in the verifiable subset of our Theory of Classes (**TC**).

We also extend Codecharts a little to accommodate the signature type, and genericity, extensions. However, as this thesis is not on the topic of visual languages our discussion of Codecharts is very general. We do not attempt to define a visual grammar, and the extensions we make to the visual notation is understandably quick, dirty, and leaves much room for improvement.

We start with the primitive symbols, ellipses for signatures, rectangles for classes. We also add a variation of rectangles for generic types which are limited to both the arguments and instance being classes. All of these are called *0-dimensional* terms to indicate the parallel to geometric point. Sets of these are represented with a drop shadow, and are called *1-dimensional* terms to indicate their parallel to geometric lines (sets of points). The exception to the drop shadow rule is evident for hierarchies, represented by triangles, which are already sets by definition so decorating them with a drop shadow results in a *2-dimensional* term akin to a geometric plane (sets of sets of points). Another exception are identifiers and the class arguments of a method, which are indicated by the same text style and notation as in **TC** inside the signature visual symbol.

Table C.1: Variable symbols in Codecharts

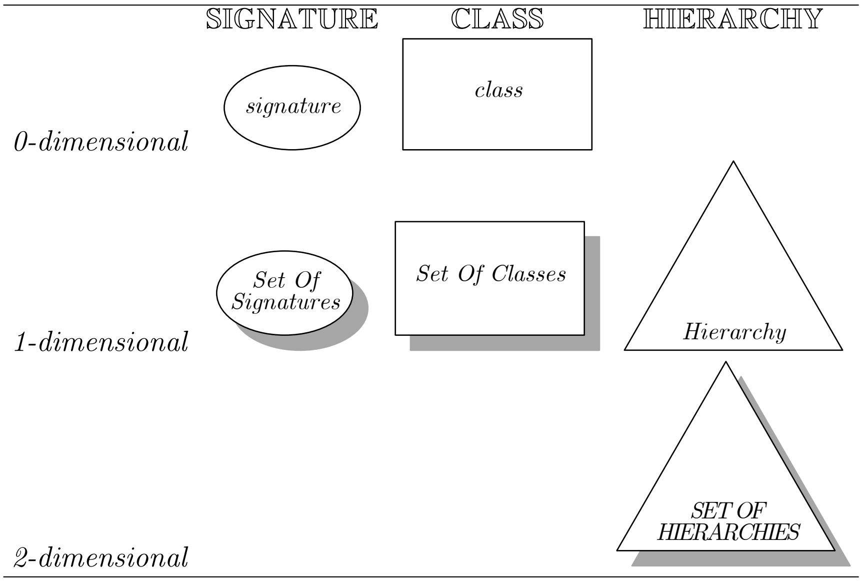
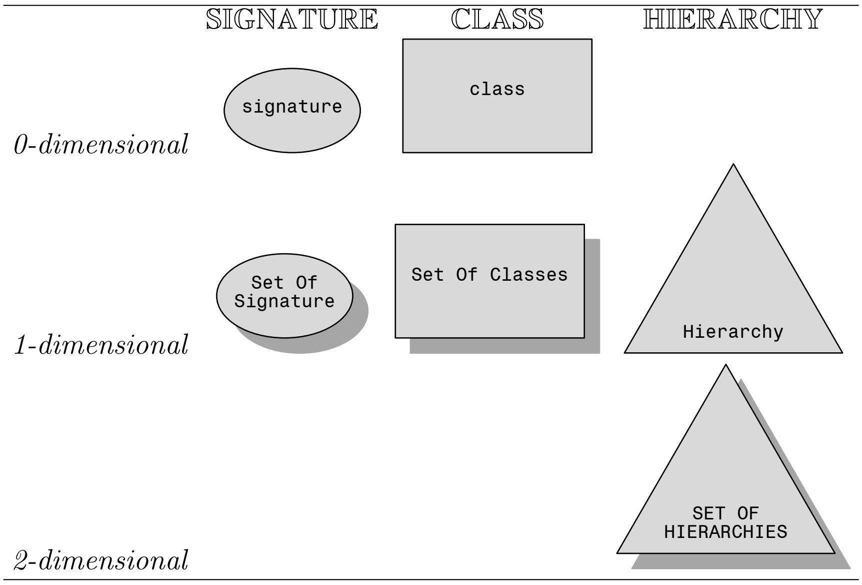

SIGNATURE
CLASS
HIERARCHY
0-dimensional
signature
class
1-dimensional
Set Of Signatures
Set Of Classes
Hierarchy
2-dimensional
SET OF HIERARCHIES

Table C.2: Constant symbols in Codecharts


SIGNATURE
CLASS
HIERARCHY
0-dimensional
signature
class
1-dimensional
Set Of Signature
Set Of Classes
Hierarchy
2-dimensional
SET OF HIERARCHIES

Subtype terms, those that result from the $t : spec\,|a_1, \ldots, a_n|$ notation, are represented by the following general rules:

- t is represented with the visual symbol of its supertype, and the appropriate decoration to indicate if it is a variable or a constant
- $spec\,|a_1, \ldots, a_n|$ is represented with a hexagon, the text of which is the symbolic notation for the type
- An undecorated edge connects the symbols for t and $spec\,|a_1, \ldots, a_n|$

Therefore, the term and subtype example `AList` : *genericList* |`Integer`| from §9.2 would be represented as in the figure below. This allows us to continue to use the original Codechart notation as much as is possible.

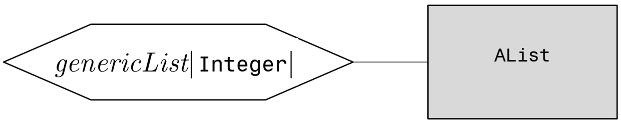


Figure C.1: An example of representing genericity in Codecharts

Methods in Codecharts are represented by overlaying[129] (superimposing) a signature on a class, as the definition of ⊗ allows. We can easily represent simple *Clans*, which are sets of dynamically bound methods (more generally a set of methods with the same signature), and *Tribes*, which are sets of clans. Notice that by overlaying certain symbols we may achieve *3-dimensional* method[130] terms, akin to 3D space (sets of sets of sets of points).

Table C.3: Clan symbols

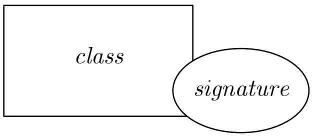
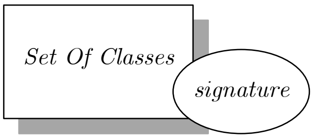
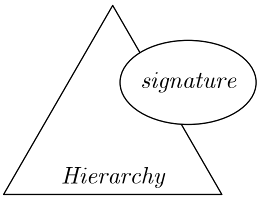
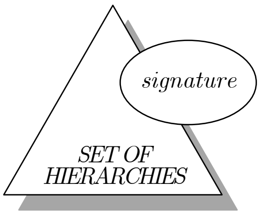
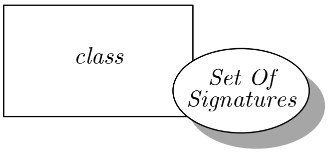
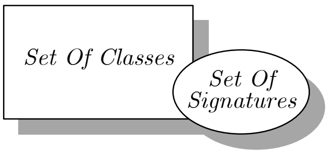
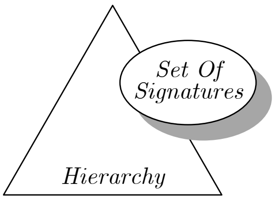
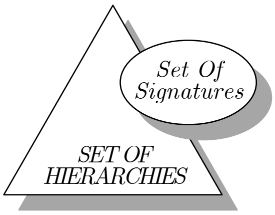
0-dimensional	*1-dimensional*	*2-dimensional*
		
		

Table C.4: Tribe symbols

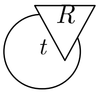
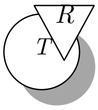
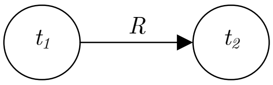
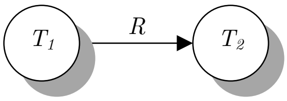
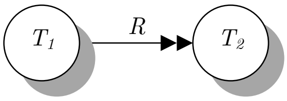
1-dimensional	*2-dimensional*	*3-dimensional*
		
		

Now we show how to represent simple propositions (relations) using circles as placeholders for the aforementioned allowable terms.

[129] We take overlaying to mean that one shape intersects, or is contained within, another shapes bounding box. We do not consider overlaying to include shapes that 'touch'.

[130] Which are rare, but arguably useful

Table C.5: Formula symbols

Ground Formulas		*Predicate Formulas*	
$R\,(t)$		$\text{A}\textsc{ll}\,(R,T)$	
$R\,(t_1,t_2)$		$\text{T}\textsc{ot}\,(R,T_1,T_2)$	
		$\text{I}\textsc{so}\,(R,T_1,T_2)$	

When a term is a method, or set thereof, then the relation is attached to/overlaid on the relevant signature term. Additionally, the overloaded relation symbols are intentional as to make the notation as simple as possible. However this introduces an ambiguity when converting from the visual to the symbolic. To resolve this, whenever there is a choice between two symbolic relations for the same visual relation, the least complex relation is always chosen For example, we select $R\,(t)$ over $\text{A}\textsc{ll}(R,T)$, and $R\,(t_1,t_2)$ over $\text{T}\textsc{ot}(R,T_1,T_2)$.

Appendix D

Case Studies in Design Verification

D.1 Java's Abstract Window Toolkit (java.awt)

D.1.1 Summarized Source Code

```
public abstract class Component ... {
  public void addNotify() { ... }
  public void removeNotify() { ... }
... }

public class Button extends Component ... {
  public void addNotify() { ... }
... }

public class Canvas extends Component ... {
  public void addNotify() { ... }
... }

public class Container extends Component {
  private List<Component> component = new ArrayList<Component>();
  public Component getComponent(int n) { ... }
  public Component[] getComponents() { ... }
  public void addNotify() { ...
    component.get(i).addNotify(); }
  public void removeNotify() { ...
    Component comp = component.get(i);
    comp.removeNotify(); }
... }
```

D.1.2 Design Model

$$\begin{array}{rrl}
1 & \texttt{Button}^{\mathfrak{M}}, & \\
2 & \texttt{Canvas}^{\mathfrak{M}}, & \\
3 & \texttt{Component}^{\mathfrak{M}}, & \\
4 & \texttt{Container}^{\mathfrak{M}}, & \\
5 & \texttt{int}^{\mathfrak{M}} & \in \mathbb{CLASS}^{\mathfrak{M}} \\
6 & \texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, & \\
7 & \texttt{removeNotify}\,\langle\rangle^{\mathfrak{M}}, & \\
8 & \texttt{getComponents}\,\langle\rangle^{\mathfrak{M}}, & \\
9 & \texttt{getComponent}\,\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} & \in \mathbb{SIGNATURE}^{\mathfrak{M}} \\
10 & \texttt{Button.addNotify()}^{\mathfrak{M}}, & \\
11 & \texttt{Canvas.addNotify()}^{\mathfrak{M}}, & \\
12 & \texttt{Component.addNotify()}^{\mathfrak{M}}, & \\
13 & \texttt{Container.addNotify()}^{\mathfrak{M}}, & \\
14 & \texttt{Component.removeNotify()}^{\mathfrak{M}}, & \\
15 & \texttt{Container.removeNotify()}^{\mathfrak{M}}, & \\
16 & \texttt{Container.getComponents()}^{\mathfrak{M}}, & \\
17 & \texttt{Container.getComponent(int)}^{\mathfrak{M}} & \in \mathbb{METHOD}^{\mathfrak{M}} \\
18 & \left(\texttt{Button}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right), & \\
19 & \left(\texttt{Canvas}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right), & \\
20 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right) & \in Inherit^{\mathfrak{M}} \\
21 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right) & \in Aggregate^{\mathfrak{M}} \\
22 & \left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Button.addNotify()}^{\mathfrak{M}}\right), & \\
23 & \left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Canvas.addNotify()}^{\mathfrak{M}}\right), & \\
24 & \left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right), & \\
25 & \left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.addNotify()}^{\mathfrak{M}}\right), & \\
26 & \left(\texttt{removeNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right), & \\
27 & \left(\texttt{removeNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.removeNotify()}^{\mathfrak{M}}\right), & \\
28 & \left(\texttt{getComponents}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.getComponents()}^{\mathfrak{M}}\right), & \\
29 & \left(\texttt{getComponent}\,\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{Container.getComponent(int)}^{\mathfrak{M}}\right) & \in SignatureOf^{\mathfrak{M}} \\
30 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.getComponents()}^{\mathfrak{M}}\right), & \\
31 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.getComponent(int)}^{\mathfrak{M}}\right), & \\
32 & \left(\texttt{Button}^{\mathfrak{M}}, \texttt{Button.addNotify()}^{\mathfrak{M}}\right), & \\
33 & \left(\texttt{Canvas}^{\mathfrak{M}}, \texttt{Canvas.addNotify()}^{\mathfrak{M}}\right), & \\
34 & \left(\texttt{Component}^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right), & \\
35 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.addNotify()}^{\mathfrak{M}}\right), & \\
36 & \left(\texttt{Component}^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right), & \\
37 & \left(\texttt{Container}^{\mathfrak{M}}, \texttt{Container.removeNotify()}^{\mathfrak{M}}\right) & \in MethodMember^{\mathfrak{M}} \\
38 & \left(\texttt{Container.addNotify()}^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right), & \\
39 & \left(\texttt{Container.removeNotify()}^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right) & \in Forward^{\mathfrak{M}}
\end{array}$$

23 $\left(\texttt{Button}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right),$

24 $\left(\texttt{Canvas}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right),$

25 $\left(\texttt{Container}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right) \in \mathit{Inherit}^{\mathfrak{M}}$

26 $\left(\texttt{Container}^{\mathfrak{M}}, \texttt{Component}^{\mathfrak{M}}\right) \in \mathit{Aggregate}^{\mathfrak{M}}$

27 $\left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Button.addNotify()}^{\mathfrak{M}}\right),$

28 $\left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Canvas.addNotify()}^{\mathfrak{M}}\right),$

29 $\left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Component.addNotify()}^{\mathfrak{M}}\right),$

30 $\left(\texttt{addNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.addNotify()}^{\mathfrak{M}}\right),$

31 $\left(\texttt{removeNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Component.removeNotify()}^{\mathfrak{M}}\right),$

32 $\left(\texttt{removeNotify}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.removeNotify()}^{\mathfrak{M}}\right),$

33 $\left(\texttt{paramString}\,\langle\rangle^{\mathfrak{M}}, \texttt{Button.paramString()}^{\mathfrak{M}}\right),$

34 $\left(\texttt{paramString}\,\langle\rangle^{\mathfrak{M}}, \texttt{Canvas.paramString()}^{\mathfrak{M}}\right),$

35 $\left(\texttt{paramString}\,\langle\rangle^{\mathfrak{M}}, \texttt{Component.paramString()}^{\mathfrak{M}}\right),$

36 $\left(\texttt{paramString}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.paramString()}^{\mathfrak{M}}\right),$

37 $\left(\texttt{getComponents}\,\langle\rangle^{\mathfrak{M}}, \texttt{Container.getComponents()}^{\mathfrak{M}}\right),$

38 $\left(\texttt{getComponent}\,\left\langle\texttt{int}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \texttt{Container.getComponent(int)}^{\mathfrak{M}}\right) \in \mathit{SignatureOf}^{\mathfrak{M}}$

D.2 JUnit's MoneyTest Example

D.2.1 Summarized Source Code

```
public class Assert { ...
  public static void assertTrue(boolean condition) { ... }
  public static void assertEquals(Object expected, Object actual) { ... }
... }

public abstract class TestCase extends Assert implements Test { ...
  protected void tearDown() { ... }
  protected void setUp() { ... }
  protected void runTest() { ... }
... }

public class MoneyTest extends TestCase {
  private Money f12CHF;
  private Money f14CHF;

  protected void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
  }

  public void testEquals() {
    Assert.assertTrue(!f12CHF.equals(null));
    Assert.assertEquals(f12CHF, f12CHF);
    Assert.assertEquals(f12CHF, new Money(12, "CHF"));
    Assert.assertTrue(!f12CHF.equals(f14CHF));
  }

  public void runTest() {
    testEquals();
  }
}
```

D.2.2 Design Model

We have simplified this design model so it fits the width of the page; specifically we write `bool` rather than `boolean` and `Obj` rather than `Object`.

$$
\begin{array}{rrl}
1 & \mathtt{bool}^{\mathfrak{M}}, & \\
2 & \mathtt{Obj}^{\mathfrak{M}}, & \\
3 & \mathtt{Money}^{\mathfrak{M}}, & \\
4 & \mathtt{MoneyTest}^{\mathfrak{M}}, & \\
5 & \mathtt{TestCase}^{\mathfrak{M}}, & \\
6 & \mathtt{Assert}^{\mathfrak{M}}, & \\
7 & \mathtt{Test}^{\mathfrak{M}} & \in \mathbb{CLASS}^{\mathfrak{M}} \\
8 & \mathtt{runTest}\left\langle\right\rangle^{\mathfrak{M}}, & \\
9 & \mathtt{setUp}\left\langle\right\rangle^{\mathfrak{M}}, & \\
10 & \mathtt{tearDown}\left\langle\right\rangle^{\mathfrak{M}}, & \\
11 & \mathtt{testEquals}\left\langle\right\rangle^{\mathfrak{M}}, & \\
12 & \mathtt{assertTrue}\left\langle\mathtt{bool}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, & \\
13 & \mathtt{assertEquals}\left\langle\mathtt{Obj}^{\mathfrak{M}},\mathtt{Obj}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}} & \in \mathbb{SIGNATURE}^{\mathfrak{M}} \\
14 & \mathtt{Assert.assertTrue(bool)}^{\mathfrak{M}}, & \\
15 & \mathtt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}, & \\
16 & \mathtt{MoneyTest.testEquals()}^{\mathfrak{M}}, & \\
17 & \mathtt{TestCase.tearDown()}^{\mathfrak{M}}, & \\
18 & \mathtt{TestCase.runTest()}^{\mathfrak{M}}, & \\
19 & \mathtt{MoneyTest.runTest()}^{\mathfrak{M}}, & \\
20 & \mathtt{TestCase.setUp()}^{\mathfrak{M}}, & \\
21 & \mathtt{MoneyTest.setUp()}^{\mathfrak{M}}, & \\
22 & \ldots & \in \mathbb{METHOD}^{\mathfrak{M}}
\end{array}
$$

$$
\begin{aligned}
\mathtt{Test}^{\mathfrak{M}}, & \\
\mathtt{TestCase}^{\mathfrak{M}} & \in \mathit{AbstractClass}^{\mathfrak{M}} \\
\left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{Assert}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{TestCase}^{\mathfrak{M}}\right), & \\
\left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{Test}^{\mathfrak{M}}\right) & \in \mathit{Inherit}^{\mathfrak{M}} \\
\left(\mathtt{assertTrue}\left\langle \mathtt{bool}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \mathtt{Assert.assertTrue(bool)}^{\mathfrak{M}}\right), & \\
\left(\mathtt{assertEquals}\left\langle \mathtt{Obj}^{\mathfrak{M}}, \mathtt{Obj}^{\mathfrak{M}}\right\rangle^{\mathfrak{M}}, \mathtt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}\right), & \\
\left(\mathtt{tearDown}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{testEquals}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{MoneyTest.testEquals()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{runTest}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.runTest()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{runTest}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{MoneyTest.runTest()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{setUp}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{TestCase.setUp()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{setUp}\left\langle\right\rangle^{\mathfrak{M}}, \mathtt{MoneyTest.setUp()}^{\mathfrak{M}}\right), & \\
\ldots & \in \mathit{SignatureOf}^{\mathfrak{M}} \\
\left(\mathtt{Assert}^{\mathfrak{M}}, \mathtt{Assert.assertTrue(bool)}^{\mathfrak{M}}\right), & \\
\left(\mathtt{Assert}^{\mathfrak{M}}, \mathtt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}\right), & \\
\left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.tearDown()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{MoneyTest.testEquals()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.runTest()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{MoneyTest.runTest()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{TestCase}^{\mathfrak{M}}, \mathtt{TestCase.setUp()}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{MoneyTest.setUp()}^{\mathfrak{M}}\right), & \\
\ldots & \in \mathit{MethodMember}^{\mathfrak{M}} \\
\left(\mathtt{MoneyTest}^{\mathfrak{M}}, \mathtt{Money}^{\mathfrak{M}}\right) & \in \mathit{DataMember}^{\mathfrak{M}} \\
\left(\mathtt{MoneyTest.setUp()}^{\mathfrak{M}}, \mathtt{Money}^{\mathfrak{M}}\right) & \in \mathit{Create}^{\mathfrak{M}} \\
\left(\mathtt{MoneyTest.runTest()}^{\mathfrak{M}}, \mathtt{MoneyTest.testEquals}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest.testEquals()}^{\mathfrak{M}}, \mathtt{Assert.assertTrue(bool)}^{\mathfrak{M}}\right), & \\
\left(\mathtt{MoneyTest.testEquals()}^{\mathfrak{M}}, \mathtt{Assert.assertEquals(Obj,Obj)}^{\mathfrak{M}}\right) & \in \mathit{Call}^{\mathfrak{M}}
\end{aligned}
$$

D.3 Java's Utility Classes (java.util)

D.3.1 Summarized Source Code

```
public class LinkedList<E> ... {
    public boolean add(E e) { ... return true; }
    public boolean remove(int index) { ... return false; }
    public E get(int index) { return entry(index).element; }
... }

public class MyClass {
  LinkedList<Integer> ints = new LinkedList<Integer>();
}
```

D.3.2 Design Model

For brevity we use LL<Integer> for the class resulting from instantiation the generic class LinkedList with the class argument Integer.

$$
\begin{array}{rrl}
1 & \texttt{int}^{\mathfrak{M}}, & \\
2 & \texttt{boolean}^{\mathfrak{M}}, & \\
3 & \texttt{Integer}^{\mathfrak{M}}, & \\
4 & \texttt{MyClass}^{\mathfrak{M}}, & \\
5 & \texttt{LL} < \texttt{Integer} >^{\mathfrak{M}} & \in \mathbb{CLASS}^{\mathfrak{M}} \\
6 & \texttt{add}\left\langle \texttt{Integer}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}}, & \\
7 & \texttt{get}\left\langle \texttt{int}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}}, & \\
8 & \texttt{remove}\left\langle \texttt{int}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}} & \in \mathbb{SIGNATURE}^{\mathfrak{M}} \\
9 & \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}, & \\
10 & \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}, & \\
11 & \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}} & \in \mathbb{METHOD}^{\mathfrak{M}} \\
12 & \left(\texttt{add}\left\langle \texttt{Integer}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}\right), & \\
13 & \left(\texttt{get}\left\langle \texttt{int}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}\right), & \\
14 & \left(\texttt{remove}\left\langle \texttt{int}^{\mathfrak{M}} \right\rangle^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}\right) & \in \mathit{SignatureOf}^{\mathfrak{M}} \\
15 & \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}\right), & \\
16 & \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}\right), & \\
17 & \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}\right) & \in \mathit{MethodMember}^{\mathfrak{M}} \\
18 & \left(\texttt{MyClass}, \texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}\right) & \in \mathit{DataMember}^{\mathfrak{M}} \\
19 & \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}\right) & \in \mathit{Collection}^{\mathfrak{M}} \\
20 & \left(\texttt{LL} < \texttt{Integer} >^{\mathfrak{M}}, \texttt{Integer}^{\mathfrak{M}}\right) & \in \mathit{Aggregate}^{\mathfrak{M}} \\
21 & \left(\texttt{LL} < \texttt{Integer} > \texttt{.add(Integer)}^{\mathfrak{M}}, \texttt{boolean}^{\mathfrak{M}}\right), & \\
22 & \left(\texttt{LL} < \texttt{Integer} > \texttt{.get(int)}^{\mathfrak{M}}, \texttt{Integer}^{\mathfrak{M}}\right), & \\
23 & \left(\texttt{LL} < \texttt{Integer} > \texttt{.remove(int)}^{\mathfrak{M}}, \texttt{boolean}^{\mathfrak{M}}\right) & \in \mathit{Return}^{\mathfrak{M}}
\end{array}
$$

Appendix E

Further Investigation

Herein are a few extraneous discussions on areas above and beyond of the focus of this body of work, yet interesting enough to be included as an addenda.

E.1 Subprograms and Structural Equivalence

As an addenda to our discussion of design verification we include a couple of standard relationships between design models, based on the respective relationships in standard model theory [Doets, 1996]. With this we will then be able to define simple relationships between programs. For these definitions we adopt the same notation as in [Doets, 1996], that is we may write S^n for the n^{th} Cartesian power of set S, and $f|S$ for the function f restricted to set S:

Definition 15 Let $\mathfrak{A}$ and $\mathfrak{B}$ be design models, $\mathfrak{A}$ is a *submodel* of $\mathfrak{B}$ (or $\mathfrak{B}$ is an *extension* or *supermodel* of $\mathfrak{A}$), written $\mathfrak{A} \subset \mathfrak{B}$, if and only if the following conditions hold:

1. $\mathbf{A} \subset \mathbf{B}$
2. And for any x in the domain of $\mathcal{I}_{\mathfrak{A}}$
   (a) If x is an n-place relation symbol then $x^{\mathfrak{A}} = x^{\mathfrak{B}} \cap \mathbf{A}^n$
   (b) If x is an n-place function symbol then $x^{\mathfrak{A}} = x^{\mathfrak{B}}|\mathbf{A}^n$
   (c) If x is a constant symbol then $x^{\mathfrak{A}} = x^{\mathfrak{B}}$

Definition 16 Let $\mathfrak{A}$ and $\mathfrak{B}$ be design models, $\mathfrak{A}$ is *equivalent* to $\mathfrak{B}$, written $\mathfrak{A} \equiv \mathfrak{B}$, if both satisfy the same set of sentences. I.e. given any ϕ, $\mathfrak{A} \models \phi$ if and only if $\mathfrak{B} \models \phi$

Design models and the above relationships allow us to lift structural subprogram and structural equivalence relationships. Of course these relationships are limited by the expressiveness of design models, but they can always be strengthened by capturing more design detail. They do, however, allow us to reason about the similarities between two programs articulated in possibly quite distinct languages, such as Smalltalk and Java:

Definition 17 Let p_1 and p_2 be programs articulated in object-oriented programming languages pl_1 and pl_2 respectively, p_1 is a *structural subprogram* of p_2 (or p_2 is a *structural extension* of p_1), written $p_1 \subset p_2$, if and only if $\mathcal{A}_{pl_1}(p_1) \subset \mathcal{A}_{pl_2}(p_2)$ holds.

Definition 18 Let p_1 and p_2 be programs articulated in object-oriented programming languages pl and pl_2 respectively, p_1 is a *structurally equivalent* to p_2, written $p_1 \equiv p_2$, if and only if $\mathcal{A}_{pl_1}(p_1) \equiv \mathcal{A}_{pl_2}(p_2)$ holds.

We speculate that further investigation into the use of design models in this fashion could yield benefits in areas other than simply design verification. For example, it could be of great benefit to the formal methods of program translation, such as ensuring the Java compiler translates Java source code to Java bytecode correctly or more broadly in the translation from one programming language to another.

E.2 Dependency Graphs

A dependency graph G is defined as a pair (P, R), where P is a set of points, and R is a transitive relation. We could programmatically define $P : set(set(\mathbb{CLASS}))$ such that P is the set of all sets of classes that one may wish to define in a program (how this can be done programmatically would be a matter of investigation), and $R = Depends^{+}$ where *Depends* is defined as follows:

Depends

$$a, b : set(\mathbb{CLASS})$$

$$\begin{gathered}
\exists x \in a \bullet \\
\textsc{Tot}(Inherit, x, b) \vee \\
\textsc{Tot}(Member, x, b) \vee \\
\exists s_1, s_2 : \mathbb{SIGNATURE} \bullet \\
\textsc{Tot}(Call, s_1 \otimes x, s_2 \otimes b) \vee \\
\textsc{Tot}(Create, s_1 \otimes x, b) \vee \\
\textsc{Tot}(Return, s_1 \otimes x, b)
\end{gathered}$$

For example, the dependencies between the Java packages java.io, java.util and java.lang can be visualized as directed graph. We demonstrate this using the Codechart notation. We would expect such a graph to show the dependencies from java.io to java.util and java.lang, java.util to java.lang, and self loops on each set. However, the reality is that all packages depend on each other[131]. We demonstrate this in Figure E.1 where for simplicity all edges are *Depends* relations.

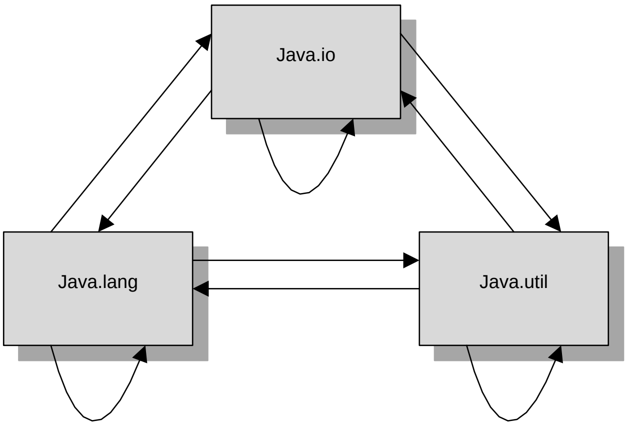


Figure E.1: Dependencies between the Java packages java.io, java.util and java.lang

[131] If it is unbelievable that these packages all depend on each other, simply look at the classes java.lang.Class and java.util.GregorianCalendar.

Appendix F

Experimental Tasks and Data

This chapter contains all the anonymized data collected from three controlled experiments on the benefits of the Two-Tier Programming Toolkit [Eden and Gasparis, 2009]. Subjects who did not complete both tasks in a given experiment, or considered to have violated the terms of the experiment, have been excluded from our analysis. All data is statically normally distributed at a confidence interval of 5% based on the Jarque-Bera test. In each experiment we predicted improvements on the part of the Toolkit, therefore to calculate the p-value we used a one-tailed unpaired Student's t test with preference to less time or greater accuracy on the part of the Toolkit. With this in mind we present each task, paraphrased from [Eden and Gasparis, 2009], and their raw results in the following sections.

F.1 Experiment 1: Comprehension

Conducted on the 14th of March 2009 with 10 subjects, all of which received one hour training in the use of NetBeans for software comprehension tasks, and similarly an hour of training was provided for the use of the Toolkit. A subject would then be given a software comprehension task using one of these tools. Once they believed they had the correct answer, they notified one of the experimenters to have their work checked. If their work was incorrect they were informed of this, given an indication of where the problem was, and asked to try again. Once the correct answer was obtained the time taken by the subject was recorded. Once all subjects had completed the first task, they were then asked to perform a second equivalent task in the other tool to the one they had previously used. The specific tasks that the subjects were asked to perform are presented in Table F.1, and their anonymized results in Table F.2. 3 subjects' data was invalid as they did not complete both tasks, or were deemed to have violated the conditions of the experiment.

Table F.1: Tasks of the Comprehension experiment with 10 test subjects

1. **Source:** Java's Abstract Window Toolkit (AWT).

   Materials: four files from the java.awt package and their respective javadoc files.

   Task: list four methods in class Container that satisfied two specific conditions.

2. **Source:** Java's InputStream classes.

   Materials: four files form java.io package and the respective javadoc files.

   Task: list four methods in class BufferedInputStream that satisfied two specific conditions.

Table F.2: Results of the Comprehension experiment with 7 valid sets of data

Participant	**Time (seconds)**		**Accuracy**	
	Toolkit	*NetBeans*	*Toolkit*	*NetBeans*
1	441	445	100%	100%
3	443	3106	100%	100%
4	595	690	100%	100%
6	420	1140	100%	100%
15	250	4260	100%	50%
18	705	753	100%	100%
22	274	3224	100%	100%
Mean:	447	1945	100%	93%
Median:	441	1140	100.00%	100.00%
Standard Deviation:	162	1540	0%	19%
Jarque-Bera Test:	0.383			
Student's t Test:	2.1%			
Ratio:	0.23			

F.2 Experiment 2: Conformance

Conducted on the 28th of March 2009 with 8 subjects, all of which received one hour training in the use of NetBeans for software conformance tasks, and similarly an hour of training was provided for the use of the Toolkit. Subjects were provided an informal summary of a design pattern taken from [Gamma et al., 1994], and an implementation that may or may not conform to it. Users of the Toolkit were also provided a specification of the pattern written in LePUS3. Subjects would then have to use their assigned tools to verify if their code sample conforms to their given design pattern. Once they had come to their conclusion, they were asked to record their decision (pass/fail), the time it took them, and to rate their confidence in their answer. Once all subjects had completed the first task, they were then asked to perform a second equivalent task in the other tool to the one they had previously used. The specific tasks that the subjects were asked to perform are presented in Table F.3, and their anonymized results in Table F.4. 1

subject's data was invalid as they did not complete both tasks, or were deemed to have violated the conditions of the experiment.

Table F.3: Tasks of the Conformance experiment with 8 test subjects

1. **Source:** Java's Abstract Window Toolkit (AWT) and the Composite design pattern.

   Materials: source code from the `java.awt` package and an informal description of the Composite design pattern (with a LePUS3 specification as appropriate).

   Task: decide if a named subset of these classes and a named subset of their methods constitute an implementation that conforms to the design pattern. The correct answer was that the implementation (named classes and methods) does conform to the Composite design pattern.

2. **Source:** Java's `java.io` package and the Decorator design pattern.

   Materials: source code form `java.io` package and an informal description of the Decorator design pattern (with a LePUS3 specification as appropriate).

   Task: decide if a named subset of these classes and a named subset of their methods constitute an implementation that conforms to the design pattern. The correct answer was that the implementation (named classes and methods) does not conform to the Decorator design pattern.

Table F.4: Results of the Conformance experiment with 7 valid sets of data

Participant	**Time (seconds)**		**Accuracy**	
	Toolkit	*NetBeans*	*Toolkit*	*NetBeans*
3	1497	1324	100%	0%
4	1125	1059	100%	100%
6	1260	1080	100%	100%
8	1001	993	100%	0%
9	180	720	100%	100%
13	1731	1256	100%	0%
18	2244	2413	100%	100%
Mean:	1291	1264	100.00%	57.14%
Median:	1260	1080	100.00%	100.00%
Standard Deviation:	644	543	0.00%	53.45%
Jarque-Bera Test:			1.169	
Student's t Test:			3.9%	
Ratio:			1.75	

F.3 Experiment 3: Evolution

Conducted on the 25th of April 2009 with 6 subjects, all of which received one hour training in the use of NetBeans for software evolution tasks, and similarly an hour of training was provided for the use of the Toolkit. Subjects were provided with some source code and asked to modify it so that it conforms with certain requirements. Once they had identified what they would do, they were asked to record the time it took them to come to their conclusion. Once all subjects had completed the first task, they were then asked to perform a second equivalent task in the other tool to the one they had previously used. The specific tasks that the subjects were asked to perform are presented in Table F.5, and their anonymized results in Table F.6. 1 subject's data was invalid as they did not complete both tasks, or were deemed to have violated the conditions of the experiment.

Table F.5: Tasks of the Conformance experiment with 8 test subjects

1. **Source:** Java's InputStream classes.

   Materials: source code from the `java.io` package, and a description of the new method to be inserted.

   Task: identify how a method with a specific body may be added so as to add a specific behaviour that is common in two named classes.

2. **Source:** Java's Writers classes.

   Materials: source code form `java.io` package, and a description of the new method to be inserted.

   Task: identify how a method with a specific body may be added so as to add a specific behaviour that is common to two named classes.

Table F.6: Results of the Evolution experiment with 5 valid sets of data

Participant	**Time (seconds)**		**Accuracy**	
	Toolkit	NetBeans	Toolkit	NetBeans
4	78	46	100%	100%
6	75	50	100%	100%
10	168	537	100%	100%
13	523	176	100%	100%
18	323	75	100%	100%
Mean:	233	177	100%	100%
Median:	168	75	100%	100%
Standard Deviation:	191	208	0%	0%
Jarque-Bera Test:	0.613			
Student's t Test:	33.3%			
Ratio:	1.32			

References

[Abadi and Cardelli, 1998] Abadi, M. and Cardelli, L. (1998). *A Theory of Objects*. Springer.

[Achour et al., 2010] Achour, M., Betz, F., Dovgal, A., Lopes, N., Magnusson, H., Richter, G., Seguy, D., and Vrana, J. (2010). PHP5 manual. Available from: http://php.net/manual/en/index.php.

[Aldrich et al., 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, Florida. ACM.

[ArgoUML Open Source Community, 2010] ArgoUML Open Source Community (2010). ArgoUML, version 0.30. Available from: http://argouml.tigris.org/.

[Ayodeji, 2006] Ayodeji, O. A. (2006). *Graphical Editor for the Class-Z Specification Language*. MSc, University of Essex, UK.

[Beck and Cunningham, 1987] Beck, K. and Cunningham, W. (1987). Using pattern languages for Object-Oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, Orlando, Florida, USA.

[Bell, 2004] Bell, A. E. (2004). Death by UML fever: Self-diagnosis and early treatment are crucial in the fight against UML fever. *Queue*, 2(1):72–80.

[Blewitt, 2006] Blewitt, A. (2006). *Hedgehog: Automatic verification of Design patterns in Java*. PhD, University of Edinburgh, UK.

[Blewitt, 2007] Blewitt, A. (2007). Spine: Language for pattern verification. In *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA.

[Blewitt et al., 2001] Blewitt, A., Bundy, A., and Stark, I. (2001). Automatic verification of Java design patterns. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 324—327. IEEE Computer Society.

[Bo, 2004] Bo, G. (2004). *An Analysis Tool for Java Programs.* MSc, University of Essex, UK.

[Brooks, 1987] Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19.

[Craig, 2000] Craig, I. (2000). *The Interpretation of Object-Oriented Programming Languages.* Springer, second edition.

[Cutland, 1980] Cutland, N. (1980). *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press.

[Danial, 2010] Danial, A. (2010). CLOC, version 1.09. Available from: http://cloc.sourceforge.net/.

[Doets, 1996] Doets, K. (1996). *Basic Model Theory.* Center for the Study of Language and Information.

[Dong et al., 2007] Dong, J., Alencar, P., and Cowan, D. (2007). Formal specification and verification of design patterns. In *Design Patterns Formalization Techniques.* IGI Global, Hershey, USA.

[Dong and Zhao, 2007] Dong, J. and Zhao, Y. (2007). Experiments on design pattern discovery. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering.* IEEE Computer Society.

[Eden, 1999] Eden, A. H. (1999). *Precise specification of design patterns and tool support in their application.* PhD, The Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

[Eden, 2005] Eden, A. H. (2005). Strategic versus tactical design. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, volume 9, Honolulu, HI, USA. IEEE Computer Society.

[Eden and Gasparis, 2009] Eden, A. H. and Gasparis, E. (2009). Three controlled experiments in software engineering with the Two-Tier Programming Toolkit: Final report. Technical Report CES-496, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex.

[Eden et al., 2007a] Eden, A. H., Gasparis, E., and Nicholson, J. (2007a). The 'Gang of Four' companion: Formal specification of design patterns in LePUS3 and Class-Z. Technical Report

CSM-472, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex.

[Eden et al., 2007b] Eden, A. H., Gasparis, E., and Nicholson, J. (2007b). LePUS3 and Class-Z reference manual. Technical Report CSM-474, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex.

[Eden et al., 2006] Eden, A. H., Hirshfeld, Y., and Kazman, R. (2006). Abstraction classes in software design. *IEE Software*, 153(4):163–182.

[Eden et al., 2003] Eden, A. H., Kazman, R., and Fox, C. J. (2003). Two-Tier Programming. Technical Report CSM-387, ISSN 1744-8050, University of Essex, School of Computer Science and Electronic Engineering.

[Eden and Nicholson, 2011] Eden, A. H. and Nicholson, J. (2011). *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs.* Wiley-Blackwell.

[Eden et al., 2008] Eden, A. H., Nicholson, J., and Gasparis, E. (2008). LePUS3 and Class-Z home page. http://www.lepus.org.uk/ [Last accessed: 1 of apr 2010].

[Eden et al., 1997] Eden, A. H., Yehudai, A., and Gil, J. (1997). Precise specification and automatic application of design patterns. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, pages 143–152, Lake Tahoe, CA. IEEE Computer Society.

[Fowler, 2003] Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison Wesley, third edition.

[Fragkos, 2006] Fragkos, D. (2006). *Class-Z Editor: Application for Editing Class-Z Visual Specifications.* MSc, University of Essex, UK.

[France et al., 2004] France, R. B., Kim, D., Ghosh, S., and Song, E. (2004). A UML-Based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206.

[Fujaba Development Group, 2009] Fujaba Development Group (2009). Fujaba, version 5.0.4.1. Available from: http://www.fujaba.de/.

[Gamma and Beck, 1999] Gamma, E. and Beck, K. (1999). JUnit: a cook's tour. *Java Report*, 4(5):27–38.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

[Gasparis, 2005] Gasparis, E. (2005). LePUS2 user guide. Technical Report CSM-436, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex.

[Gasparis, 2010] Gasparis, E. (2010). *Design Navigation: Recovering Design Charts From Object-Oriented Programs.* PhD, University of Essex.

[Gasparis et al., 2008a] Gasparis, E., Eden, A. H., Nicholson, J., and Kazman, R. (2008a). The Design Navigator: Charting Java programs. In *Proceedings of the 30th international conference on Software engineering*, pages 945–946, Leipzig, Germany. ACM.

[Gasparis et al., 2008b] Gasparis, E., Nicholson, J., and Eden, A. (2008b). LePUS3: an Object-Oriented design description language. In *Diagrammatic Representation and Inference*, volume 5223 of *Lecture Notes in Computer Science*, pages 364–367. Springer Berlin.

[Gibbs, 1994] Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American*, 271(3):72–81.

[Gil and Maman, 2005] Gil, J. Y. and Maman, I. (2005). Micro patterns in Java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 97–116, San Diego, CA, USA. ACM.

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

[Hoare, 1975] Hoare, C. A. R. (1975). Software design: a parable. *Software World*, 5(9):53–56.

[Hoare, 1989] Hoare, C. A. R. (1989). Programming is an engineering profession. In Jones, C. B., editor, *Essays in computing science.* Prentice-Hall, Inc.

[Hoare, 2000] Hoare, C. A. R. (2000). A hard act to follow. *Higher-Order and Symbolic Computation*, 13(1-2):71–72.

[Huth and Ryan, 2000] Huth, M. R. A. and Ryan, M. D. (2000). *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, England, second edition.

[Iyaniwura, 2003] Iyaniwura, O. A. (2003). *A Verification Tool for Object-Oriented Programs.* MSc, University of Essex, UK.

[Jurafsky and Martin, 2000] Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall, US edition.

[Lamport, 1995] Lamport, L. (1995). How to write a proof. *The American Mathematical Monthly*, 102(7):600–608.

[Lehman et al., 1997] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 20–32. IEEE Computer Society.

[Liang, 2004] Liang, M. T. (2004). *Specification Module of TTP Toolkit.* MSc, Chalmers University of Technology, Sweden.

[Liang, 2006] Liang, Y. D. (2006). *Introduction to Java Programming: Comprehensive Version.* Prentice Hall, sixth edition.

[Lightfoot, 2001] Lightfoot, D. (2001). *Formal Specification Using Z.* Palgrave, Basingstoke, second edition.

[Mahmoud, 2004] Mahmoud, Q. H. (2004). Using and programming generics in J2SE 5.0. *Sun Developer Network.*

[Mak et al., 2003] Mak, J. K. H., Choy, C. S. T., and Lun, D. P. K. (2003). Precise specification to compound patterns with ExLePUS. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 440–445. IEEE Computer Society.

[Mak et al., 2004] Mak, J. K. H., Choy, C. S. T., and Lun, D. P. K. (2004). Precise modeling of design patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering*, pages 252–261. IEEE Computer Society.

[Mapelsden et al., 2002] Mapelsden, D., Hosking, J., and Grundy, J. (2002). Design pattern modelling and instantiation using DPML. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, volume 21 of *ACM International Conference Proceeding Series*, pages 3–11, Sydney, Australia. Australian Computer Society, Inc.

[Maplesden et al., 2004] Maplesden, D., Hosking, J., and Grundy, J. (2004). MaramaDPTool. Available from: https://wiki.auckland.ac.nz/display/csidst/MaramaDPTool.

[Maplesden et al., 2007] Maplesden, D., Hosking, J., and Grundy, J. (2007). A visual language for design pattern modeling and instantiation. In *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA.

[Martin, 1998] Martin, R. C. (1998). Java and C++: a critical comparison. In *Java Gems: jewels from Java Report*, pages 51–68. Cambridge University Press.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

[Mens et al., 2002] Mens, T., Demeyer, S., and Janssens, D. (2002). Formalising behaviour preserving program transformations. *International Conference on Graph Transformation*, 2505:286–301.

[Milner, 1987] Milner, R. (1987). Is computing an experimental science? *Journal of Information Technology*, 2(2):58–66.

[Nicholson, 2006] Nicholson, J. (2006). *Verification of Java implementations against Class-Z specifications*. MSc, University of Essex, UK.

[Nicholson et al., 2007] Nicholson, J., Eden, A. H., and Gasparis, E. (2007). Verification of LePUS3/Class-Z specifications: Sample models and abstract semantics for Java 1.4. Technical Report CSM-471, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex.

[Nicholson et al., 2008] Nicholson, J., Gasparis, E., and Eden, A. H. (2008). The Two-Tier Programming Project. http://ttp.essex.ac.uk/ [Last accessed: 1 of apr 2010].

[Nicholson et al., 2010] Nicholson, J., Gasparis, E., and Eden, A. H. (2010). The Two-Tier Programming Toolkit, version 0.5.4. Available from: http://ttp.essex.ac.uk/.

[Nicholson et al., 2009] Nicholson, J., Gasparis, E., Eden, A. H., and Kazman, R. (2009). Automated verification of design patterns in LePUS3. In *Proceedings of the 1st NASA Formal Methods Symposium*, pages 76–85, Moffett Field, California, USA. NASA.

[Object Management Group, 2003] Object Management Group (2003). UML 2.0 specification: Infrastructure. Technical report, Object Management Group.

[Object Management Group, 2005] Object Management Group (2005). UML 2.0 specification: Superstructure. Technical report, Object Management Group.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM Special Interest Group on Software Engineering Notes*, 17(4):40–52.

[Pierce, 2002] Pierce, B. C. (2002). *Types and Programming Languages.* The MIT Press.

[Raje and Chinnasamy, 2001] Raje, R. R. and Chinnasamy, S. (2001). eLePUS – a language for specification of software design patterns. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 600–604, Las Vegas, Nevada, United States. ACM.

[Raje et al., 2007] Raje, R. R., Chinnasamy, S., Olson, A. M., and Hidgon, W. (2007). The applications and enhancement of LePUS for specifying design patterns. In *Design Patterns Formalization Techniques.* IGI Global, Hershey, USA.

[Raosoft Inc., 2004] Raosoft Inc. (2004). Sample size calculator. Available from: http://www.raosoft.com/samplesize.html.

[Sauvage, 2004] Sauvage, S. (2004). Agent oriented design patterns: A case study. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 3, pages 1496–1497, New York, NY, USA. IEEE Computer Society.

[Schmidt et al., 1996] Schmidt, D. C., Fayad, M., and Johnson, R. E. (1996). Software patterns. *Communications of the ACM*, 39(10):37–39.

[Seemann and von Gudenberg, 1998] Seemann, J. and von Gudenberg, J. W. (1998). Pattern-Based design recovery of Java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, Lake Buena Vista, Florida, United States. ACM.

[Shapiro, 2000] Shapiro, S. (2000). *Thinking about Mathematics: The Philosophy of Mathematics.* OUP Oxford.

[SOA Systems Inc., 2010] SOA Systems Inc. (2010). SOA patterns. http://www.soapatterns.org/ [Last accessed: 21 of apr 2010].

[Spivey, 1992] Spivey, J. M. (1992). *The Z Notation: a Reference Manual.* Prentice-Hall, second edition.

[Stelting and Maassen, 2002] Stelting, S. A. and Maassen, O. (2002). *Applied Java Patterns*. Prentice Hall.

[Sun Microsystems Inc., 2006] Sun Microsystems Inc. (2006). Java 6 SDK: standard edition documentation. Available from: http://java.sun.com/javase/6/docs/.

[Taibi, 2007a] Taibi, T. (2007a). *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA.

[Taibi, 2007b] Taibi, T. (2007b). An integrated approach to design patterns formalization. In *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA.

[Taivalsaari, 1996] Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479.

[Thomas, 2004] Thomas, D. (2004). MDA: revenge of the modelers or UML utopia? *IEEE Software*, 21(3):15–17.

[Turner, 2009] Turner, R. (2009). *Computable Models*. Springer.

[Turner, 2010] Turner, R. (2010). Logic and computation. Available from: http://cswww.essex.ac.uk/staff/turnr/Mypapers/TPLessex.pdf.

[Wing, 1990] Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer*, 23(9):8–23.

Index