Patterns and the B Method: Bridging Formal and Informal Development

Edward Chan(ekfchan@gmail.com)
& Brett Welch(brett.welch@gmail.com)
& Ken Robinson(k.robinson@unsw.edu.au)

UNSW–CSE–TR–0620
Technical Report December 2006

School of Computer Science & Engineering

THE UNIVERSITY OF
NEW SOUTH WALES

Abstract

In a world increasingly dependent on software controlled systems, the need for the verification of software safety and correctness has never been greater. Traditional software development methods leave much to be desired in this aspect, relying heavily on testing which can be costly and time inefficient. A more efficient and less error prone approach is to use *formal methods*, in particular the B method, to develop software.

This thesis explores concepts and methods to assist developers in using formal methods by borrowing concepts from the Object Oriented world of software development. Previous attempts at doing this have attempted to adapt the B method to the Object Oriented paradigm. This thesis presents an alternative approach that adapts concepts borrowed from the Object Oriented paradigm, to the B method. By concentrating on commonly occurring *patterns* in software development and drawing inspiration from the traditional Gang of Four design patterns, this thesis presents a series of patterns adapted to and specialised for the B method, demonstrating how the beginnings of complex and significant systems can be modelled in B.

Contents

List of Figures

Chapter 1

Introduction

Modern software engineering is largely an informal affair. Most projects follow a standard software development life-cycle, which take natural language requirements and follow through to specification and design, implementation and finally testing. System modelling such as the Unified Modelling Language (UML) are popularly used in the design phase to provide a visualisation of design. Most UML diagrams are verified by the human eye. The implementation phase carries through any errors from the design, and inevitably introduces new ones. Many of these errors then await the testing phase, which follows a testing plan formulated from the requirements. Such processes do incrementally assist in improving software quality, and many organisations further improve quality by adopting quality and process standards, and by using an iterative version of the development life-cycle.

While such informal processes are useful, they do not attack a fundamental problem in software engineering: the need for rigorous mathematical proof of software correctness with respect to the specification. Such a change would replace the tedious process of testing for errors and fixing them, incrementally approaching some level of usability. Instead we would know a program is correct; we just need to certify that it meets the user's needs and is a correct implementation of the requirements. Rigorous proof also allows software developers to guarantee that the program will not behave in potentially dangerous ways - this is impossible with the informal approaches mentioned earlier.

To date, software development using formal methods has mainly been confined to military use and a few mission critical applications. Examples of such include the driver-less METEOR subway system in Paris. Such confined usage may be due to '... practitioners, in their constant search for an edge in productivity, judge formal methods to be insufficiently beneficial to outweigh pragmatic problems' [Snook & Butler 2004]. Practitioners also resist formal methods because they are "hard"; formal methods use a mathematical paradigm to solve problems - far removed from the object-oriented paradigm, whose aim was to make abstracting problems easier. Finally, many practitioners seem to believe that formal methods are too abstract to solve large scale industrial problems.

1.1 Goals

Our goals in this thesis are to

1. Espouse concepts and methods to assist developers in using formal methods by borrowing concepts from the Object Oriented world of software development.

2. Utilise the above and build on the strengths of formal methods to examine how powerful, flexible and robust systems *can* be built with the B Method (B).

3. To begin the collation of a library of development *patterns* in B which others may build upon with use.

These goals will be further clarified in chapter 4.

1.2 Summary of research

This thesis presents a pattern based approach to assist the development of complex industrial systems using the formal method known as the B Method (B). By taking inspiration from object oriented Design Patterns, we examine classical design patterns from a formal perspective. We then explore the characteristics of these patterns in a formal context, and present some B specific patterns and how they can be classified. Finally, we present a series of case studies designed to show how these patterns can be applied in B to assist in developing systems.

1.3 Document overview

Chapter 2 provides some background on formal methods, the object oriented paradigm and design patterns.
Chapter 3 explores previous work done in the field of bridging formal and informal methods.
Chapter 4 further discusses the goals and approach of this thesis, and examines some classic OO design patterns and abstraction techniques and how they can be modelled with B.
Chapter 5 discusses an alternate pattern taxonomy better suited to the B method, and presents some B specific patterns.
Chapter 6 presents a series of case studies which use B patterns to solve commonly encountered problems.
Chapter 7 provides a final analysis of the work presented and potential future work.

Chapter 2

Background

2.1 Formal methods

2.1.1 What are formal methods

The term "formal methods" encompasses a set of development methodologies that are based around the application of mathematics to verify software systems. Thus all constructs are presented as *sets* and *predicates* in the program, with operations using mathematically defined *preconditions* and *post-conditions* to express the desired behaviour. An *invariant* is present to define what is always true and must never change - such as relationships between sets and safety conditions.

Examples of formal methods

The most prominent formal methods are *VDM* , *Z* and *B*. *Z* is the precursor to *B*, and is not truly a development method, rather a formal language to specify software. Jean-Raymond Abrial's B Method is a complete software development platform, allowing the developer to specify, refine, prove and implement their system. There are two flavours of B, *Event B* and *Classical B*. Both flavours are supported by tools to support development. *Classical B* is the platform used in this thesis.

2.1.2 The advantages of formal methods

Using formal methods has many advantages. These advantages include:

- The process of formalisation in itself is extremely useful. It forces the developer to deeply consider the problem at hand and how to solve it in a logically consistent manner.

- Verifiable behaviour. The developer is able to prove their work maintains the *invariant* and the *preconditions* of the operations. If the *invariant* and the *preconditions* are specified correctly in respect to the specification, this enables the developer to guarantee correct behaviour.
- Safety of behaviour. The developer is able to utilise the *Invariant* to specify dangerous states that should never be entered. Coupled with the above, the developer can prove his system will never enter a dangerous state.
- Fewer bugs. Due to the ability to prove specification and implementation the occurrence of bugs is significantly reduced, if not eliminated.

2.1.3 The disadvantages of formal methods

Using formal methods has some disadvantages. These disadvantages include:

- Steep learning curve. For those who haven't used a formal method extensively, learning to work some formal methods, such as B, is difficult. Working with B is a paradigm shift akin to moving from procedural languages to functional programming languages.
- Mathematics is hard. For most, mathematics is difficult. Considering problems through the lens of set theory and logic can be very difficult, which means specifying the right invariant and preconditions can be a challenge.
- Resources are scarce. Finding examples, tutorials, documents and other resources on formal methods is difficult. This makes learning formal methods like B very hard.

2.1.4 Development using B

Development within B is broken into three phases. These are:

- **Specification**. Specification involves modelling the desired system in an abstract manner, showing *what* the system should do, not *how*. It is one step more abstract than design. During specification, we are setting out to mathematically flesh out the requirements of the system.
- **Refinement**. Refinement involves mapping the specification to a more detailed model, one that starts to reveal the direction of the implementation. It is akin to the traditional design phase of a project. During refinement, we start to make some design decisions about the system we are building.
- **Implementation**. Implementation is a specialised case of refinement - it is legal to move directly from a specification to an implementation if your specification is "concrete" enough to implement directly. Implementation is the final step of development, from which we generate code to create the final product.

2.1.5 Usage of B and other formal methods

It is obvious that inevitably formal methods requires more thought and effort to create a worthwhile system than other popular methodologies. Thus it is necessary to point out that while formal methods are extremely beneficial to any development, formal methods only need to be used in cases where formality is great benefit or even a requirement. It is far more valuable to use B or other formal methods to create the nucleus or the core of the system and then wrap that with peripheral code to provide user interaction and less mission critical requirements. Most military systems, transport systems and medical systems would certainly benefit from a more formal treatment. When a system can potentially threaten life or create significant disturbance in case of failure, the rigour of formal methods is invaluable.

2.2 Object Oriented software development

Object Oriented - hereafter referred to as OO - software development is the current de-facto development method, due to its widespread usage and uptake. OO programming languages are abundant - Java, C++, C# and Python are popular examples of such. Almost all software practitioners are familiar with Object Oriented programming. By grouping behaviour and state into one construct called an object, we can more easily model the world around us. This development methodology is supported by a loose diagramming standard called the Unified Modelling Language, or UML. UML lets software designers express a high level class design by specifying inter-class relationships and interactions, as well as user-system interactions.

2.2.1 Advantages of OO development

Using OO development has some advantages. These advantages include:

- Ease of abstraction. Ours is a world filled with objects, and OO recognises this by allowing programmers to work the way the world works. Further to this, using UML to model a system can also be extremely useful to conceptualise a system.
- Fast development. Every OO programming language has features like inheritance and polymorphism, and a large library of functions the developer can call on to enable building useful systems faster.
- Widespread usage. The OO community is huge, with large amounts of documentation and support available no matter what language you are using.

2.2.2 Disadvantages of OO development

Using OO also has some disadvantages. These disadvantages include:

- Lack of rigour. UML diagrams are generally judged correct by the designer simply looking at it. Thus it is very easy for errors in the design to carry through to the developer, or even the end user.
- Significant reliance on testing. In general, the only way to verify an OO system works is through testing. It is near impossible to test every single case, and so inevitably bugs will slip through the gaps.

2.2.3 Design Patterns

Design patterns are structured descriptions of abstract solutions to commonly occurring problems. Most famous of these are the patterns described by the Gang of Four (GoF) in 1994, although their history began in 1987 with work by Cunningham and Beck. Patterns are not complete designs of a system, but rather describe a generic design that can be transformed into a specific solution given a specific problem. Patterns are useful for the following reasons:

- They are a vocabulary to communicate common design concepts. Many developers are familiar with some design patterns, and this enables a team to communicate a complex concept using a pattern's name.
- They are a form of knowledge management. Design patterns are a way to codify commonly used solutions in a highly generic fashion. As such, the concepts and design practices encoded in the patterns ultimately enables novices to become experts faster. In addition, repeated use ensures that the pattern can be improved with time, making the pattern user's design more robust.

Overall patterns are simply an attempt to collect knowledge and best practices surrounding software design and implementation. They are best employed as a tool to communicate and to educate others in software design.

Chapter 3

Previous work in bridging formal and informal methods

There have been a number of developments made with respect to incorporating informal and formal methods. These include object oriented versions of Z and VDM, such as Object-Z, Z++ and VDM++ where OO concepts such as inheritance, encapsulation and collections of objects are incorporated into the formal framework. However, the scope of this thesis will be narrowed to focus on developments that have attempted to enhance B using OO concepts.

3.1 Evaluating previous work

To evaluate previous work, the following high-level evaluation framework has been proposed.

- How much does this work or development enhance the ease-of-use of B? B is currently used by the majority of B practitioners with two publicly available toolkits, the B-Toolkit by B-Core of the UK and AtelierB, both toolkits and learning B itself requiring a high investment in training and education. Does the research make it easier to develop B-systems?

- Does the new development or work enable the development of more complex and mainstream applications using B? Currently, applications developed using B are restricted to the academic world and a few select mission critical systems.

- Does the development give this thesis any concepts or work to build upon?

This framework it is not meant to be used for quantitative analysis of the virtue of previous works. Rather it is a broad framework for qualitative analysis from which we can extract merits of each work that can be applied in this thesis.

3.2 UML-B and U2B

UML-B [Snook & Butler 2004] is the definition of a UML profile that takes a subset of full UML notations and a subset of the B Abstract Machine Language to use for system modelling. The purpose of this is strengthen UML so that is is 'precise and semantically well-defined' and can be converted using a tool called U2B to the equivalent B system which can be refined and implemented. On the UML side, class diagrams, state-chart diagrams and packages have been retained while formally, microB, the definition of the formal language, is based heavily on B. Butler and Snook have recognised the limitations of B with respect to object orientation, that is B does not support more than one machine calling another machines operations, it does not support operations calling other operations within the same machine among others. This precluded a direct mapping between classes and machines and instead, they mapped an entire class diagram to a single machine - the conceptual 'structure is provided by the UML rather than by B'. This is certainly a possible approach that this thesis can take.

An example of how classes map to a B machine using UML-B is shown in fig. 2.1:

All three classes have been modelled within the one machine using the MicroB language defined in the UML-B profile. Three different variables are used to represent the three different sets of instances for each class. Deferred sets are used to define the class types i.e. PHONE_SET maps to the PHONE class, CELL_SET to the CELL class. The variables are specified as an element of the power-set of these deferred sets to present the set of objects currently in existence. Snook & Butler have used functions from the set of objects to predefined sets such as NAT, BOOL and STRING to model the attributes of the class. Classes can have collections as attributes too because the objects may be mapped to sequences (arrays), and can refer to other classes by having a function map the instantiations of one class to another class e.g the PHONE variable may have a total function to the CELL variable.

These are just some of the examples of how Snook and Butler have incorporated object oriented concepts into B that are useful for this thesis. However, modelling multiple classes within a single machine is not without limitations. There can be no method calling between the classes because B does not allow operations within one machine to call other operations within the same machine. The initial solution for this was to 'cut and paste' operation bodies where a method call is being made - this is cumbersome and inelegant. Snook and Butler then proposed to use B 'DEFINITIONS' similar to macros to define operation bodies and include these definitions in the operations that required them which they have 'found to be very effective'

3.2.1 Evaluation

Snook and Butler have recognised that developers have trouble with abstracting to formal representations of a system. Indeed, the goal of UML-B is to address that. Industrial partners also gave input into the UML-B proposal with Praxis (UK) using an early definition of UML-B on a case-study system. UML-B also allows for graphical representations of formally developed software making it easier to abstract systems. It can be concluded that while UML-B is quite an interesting work in bridging formal and informal methods, it does not go far enough. In some ways UML-B has shoe-horned OO ideas into B, which results in B specifications that are hard to read and unwieldy. Furthermore, a novice to formal methods would still encounter much difficulty in using UML-B. Despite this, UML-B can still provides some of the concepts needed for mapping OO constructs to B and provides some valuable work for this thesis to be built on.

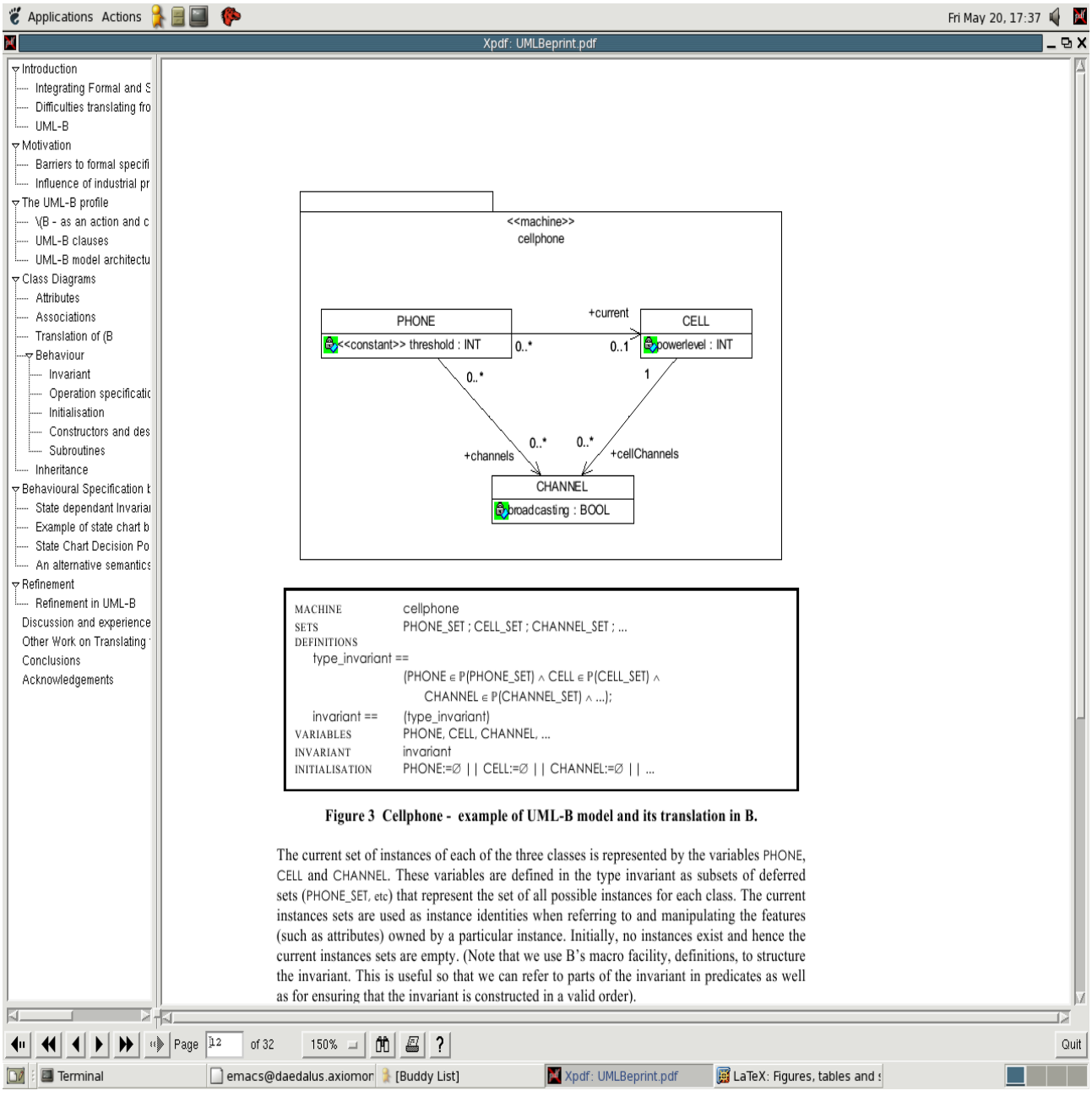


Figure 3.1: UML-B translation of class diagram to MicroB

3.3 The reuse of specification patterns within B

Blazy, Gervais and Laleau of the Institut d'Informatique d'Entreprise in France published a paper on using GoF Design Patterns with B, perhaps the most relevant previous work to be analysed. The have 'defined how to define specification patterns in B, how to reuse them directly in B ....how to reuse the proofs associated...' [Blazy, Gervais, Laleau]. Using the examples of the Composite design pattern and the Resource Allocation pattern, they present a process one can follow to specify systems with reusable design patterns using B.

Firstly, they specify a whole design pattern in one single machine. For example, when specifying the Composite Design Pattern, they specify sets to contain the instantiations of Composite and the Leaf much like the UML-B process. Referring to figure 2.2 the variables model the instances of the Component Class, the Leaf Class and the Composite Class. The invariant of the Composite machine specifies the relationships between each class and in the operations section, all the methods needed in each class to implement the composite pattern have been included.

MACHINE Composite_Pattern(COMPONENT)
VARIABLES Component, Composite, Leaf, Father
INVARIANT
Component $\subseteq$ COMPONENT $\wedge$
Composite $\subseteq$ Component $\wedge$
Leaf $\subseteq$ Component $\wedge$
Father $\in$ Component $\nrightarrow$ Composite $\wedge$
Leaf $\cup$ Composite = Component $\wedge$
Leaf $\cap$ Composite = $\emptyset$
...
OPERATIONS
children $\longleftarrow$ GetChild(father) = ...
cpt $\longleftarrow$ New_Composite(comp) = ...
Add_Composite(cpt,comp) = ...
leaf $\longleftarrow$ New_Leaf = ...
Add_Leaf(leaf) = ...
Remove_Composite(cpt) = ...
Remove_Leaf(leaf) =
pre
leaf $\in$ Leaf $\wedge$
leaf $\in$ dom(Father)
then
...
end
Operation(cpt) = ...

Figure 3.2: B-Model of the Composite Design Pattern

What's interesting is that this paper also presents a process for composing a system of two or more design patterns. They have specified three different approaches:

1. **Composition By Juxtaposition** - Two design pattern machines can be linked by specifying machine that uses the EXTENDS clause to extend both design pattern machines. In this case the new machine has access to all the operations of the extended machines and these operations form part of the new machines interface for other machines to access. No explicit link is expressed between the classes of one design pattern and the classes of another design pattern - they are considered to be disjoint.

2. **Composition By Inter-Pattern Links** - A new machine is specified to include the design pattern machines using the INCLUDES clause in B. This means that the machine has access to the variables, invariant and operations of the design patterns but the operations of the design patterns do not become part of the new machine. The new machine can specify mappings from classes in different design patterns by specifying this in its own new invariant.

3. **Composition By Unification** - A new machine is specified to include the design patterns. If the design

patterns contain the same classes - that is, the objects have the same type across different design pattern machines - then they can be unified by specifying the the sets must always be equal in the invariant of the including machine. There is a synchronisation requirement such when one operation changes the state of one design pattern machine, another operation must be called in parallel in the unified design pattern to ensure its state is the same.

3.3.1 Evaluation

The work presented promotes ease-of-use because it gives the developer freedom to breakdown the specification of a system using the usual GoF design patterns and then follow a process to manually translate that into a B system. No industry input was given on this project. The method espoused also promoted the creation of very large B machines since entire patterns were defined within one machine. While this would inevitably be 'difficult and hard to maintain', many useful concepts can be gained from this paper to incorporate design patterns into B, in particular the composition processes presented to build systems from design pattern components.

3.4 Industrial and privately funded projects

3.4.1 RODIN

The developer of the AtelierB toolkit, ClearSy System Engineering, recently (September 2004) announced the initiation of the RODIN project. RODIN stands for Rigorous Open Development Environment for Complex Systems. It is a project to create an open source formal methods toolkit/framework with 'generic mechanisms to support component reuse and composition' [RODIN Website].

RODIN is built for B and is backed by European industry partners. The open framework is to allow the plugging in of Model Checkers, UML-B, Code Generators while the IDE itself comprises of project management tools, specification tools using Event-B, proof obligation generators and a theorem prover. To date, the project is still very young and will not provide this thesis with much work to extend. However, it promises to deliver formal and informal methods integration to a new level because it will probably be the first integrated development environment to support both formal and informal methods side by side.

3.4.2 BOOSTER

BOOSTER stands for B Object Oriented Set Theory Entity Relationships. It is a semiformal class specification method which uses specifications to drive the base generator of the B toolkit. This results in complete, implemented systems. While it cannot be considered a one hundred percent formal process, the system is built on the precepts of formal methods. According to B-Core, the creator of BOOSTER and the developer of the B-Toolkit, pilot projects initiated at Oxford university have been promising.

Due to the nature of this project and the valuable IP behind it, we have been unable to gain more insight into BOOSTER. BOOSTER promises to be a significant step forward in software development, but we are unable to glean any gainful information to use in this thesis.

3.4.3 Siemens automatic generation of B-0 code

Siemens, who develop driver-less train systems using B, have developed a process to automatically generate an implementation from a B specification. Used privately within Siemens, we have learned that this method works by restricting the way the user specifies machines. This narrowing of focus enables B to be used more like a programming language, which enables Siemens to then auto-generate implementations for their specifications, which they can then prove. Once again, such work promises to be a significant step forward in the world of software development, but we are unable to draw any valuable lessons to build upon in this thesis.

3.5 Analysis

It can be seen that there have been several useful efforts to make B easier to use by approaching B from an Object Oriented perspective. However, there is certainly room for improvement:

- Object Orientation can be represented in B, but not in a complete manner - operations cannot be called machine to machine, and operations are also uncallable within the same machine. To address this, current methods need to use "workarounds" to achieve their goals. Implementation of inheritance and polymorphism is also a somewhat open question.
- Both methodologies presented advocate a single machine "monolithic" approach to developing systems. This is detrimental to maintainability, and readability, which is paramount to any specification. It can also cause trouble during implementation, as a "monolithic" implementation strategy is required - this means that the implementation cannot be carried out in steps, but must be completed all at once. Much flexibility in specification, refinement and implementation comes from composing a system out of multiple machines.
- Little work has been done in regards to implementation of object oriented machines and design patterns.

Chapter 4

Exploring formalisations of Object Oriented Design Patterns

The previous chapter outlined current state of the art and past attempts incorporating formal and informal methods. It has been shown that while useful, previous attempts are flawed in some aspect, or do not fully address the problem at hand.

4.1 Thesis goals

Elaborating on the goals expressed in the introduction, our goals for this thesis are to:

- Espouse concepts and methods to assist developers in using formal methods by borrowing concepts from the Object Oriented world of software development. By taking useful ideas such as encapsulation of behaviour and data (objects) and by utilising well known OO patterns, we want to show how rigorous systems can be built that are easier to conceptualise than pure mathematical models. In addition, most developers are familiar with OO concepts and many are familiar with design patterns. By referencing and building on this knowledge, we provide a point of reference for OO practitioners to enable a better understanding of B.
- Utilise the above and building on the strengths of formal methods to examine how powerful, flexible and robust systems *can* be built with B. The misconception that B cannot be used to develop "real" systems is incorrect. We plan to show how B can be used to build such "real" systems, by working with some OO concepts adapted to work in the B paradigm.
- To begin the collation of a library of development *patterns* in B which others may build upon with use.

Design patterns are useful to software professionals because they provide a robust, time-worn solution to common problems. Code examples given with most patterns show how and when one could use the pattern, and sometimes when not to use the pattern. Such a body of knowledge is invaluable, and our final goal is to begin the collation of such a library for B specific patterns.

The above goals are useful in describing our motivations, however would likely take a lifetime to achieve successfully. More specifically and in support of the above, this thesis aims to:

- Streamline development with B to enable complex, modern and mainstream systems to be built in a more rigorous manner by providing pattern based case studies of common patterns in system development.
- Provide insight into how OO concepts and patterns can be beneficially adapted to B.
- Develop understanding about the relationship between B and OO design patterns, and how patterns can manifest themselves in B.
- Enhance the understand-ability and usability of Formal methods, thereby encouraging the use of formal methods in the wider software development community.

4.2 Research approach

The approach of the work outlined in the previous chapter has been to force B into the shape of an OO paradigm. Our research has explored this path briefly and found it to be a challenging and yet somewhat fruitless path. As a result we changed our emphasis by taking some OO concepts and applying them into the B paradigm. This is a far more natural approach, and such a polar shift of emphasis yielded more satisfying results.
Originally this thesis was focussed on producing a specific process and tool for the reuse of OO design patterns in B. Before long we discovered this approach was too narrow, and might only work on a small set of patterns. While we still believe a tool could be created, we no longer focus on it. Instead we have taken a more general approach, examining common problems in software development. By taking inspiration from the traditional Gang of Four design patterns, we consider how these problems are solved in B and show how patterns manifest themselves within B. This approach can be broken down into the following steps.

1. First, we will consider the useful concept of an object and how it is modelled in B. Building on this, we will model some commonly used design patterns in a generalised way to gain an understanding of how common patterns are modelled in B.
2. We will then explore the different families of patterns as they are known to the OO world, and how these families might share similarities in the context of B. We will also explore a potential B specific family of patterns.

3. Finally we will provide some demonstration of how B can be used to model systems that solve commonly encountered problems using case studies. We will also demonstrate how more complex systems can be composed with a pattern based approach.

4.3 Modelling Classes and Objects in B

A process has been devised to model classes and objects in B. This is by no means a shift of B into the Object Oriented paradigm but rather a mechanism for the user to model systems in B using object abstractions of the physical world. These object machines are useful for encapsulating the data and behaviour for a single self-contained class and help to resolve the common difficulty that practitioners encounter when using mathematical constructs in B to model systems.

At the most primitive level this is carried out by treating a single B machine as a class. This B machine class would have a variable representing instantiated objects at runtime and a multitude of attribute variables to represent the mappings between the set of instantiated objects and their attributes. The machines operations are then used to model constructor and destructor methods, accessor and mutator methods as well as self-contained behavioural methods which are capable of modifying the objects state or returning the result of some processing.

Having a process to create object-like machines in B not only gives the user an object-centric abstraction by which they can model real-world objects within B but also provides a mechanism to model and formalise GoF design patterns in B where required. However, it must be noted that this process is not a *silver bullet* for either converting B into a pure object oriented language or directly porting GoF design patterns into B and the limitations of this approach will be discussed in the findings below.

4.3.1 Modelling a single class

Classes

Mapping a single class with attributes into a B machine is quite straightforward. A class type can be defined by using a deferred set and its runtime object instantiations by a variable which is specified in the machine invariant to be a subset of the deferred set. At runtime initialisation, the set of objects that this B Machine models is empty. Objects must be added to this set at runtime using a constructor-like operation to non-deterministically assign a new object to the instantiations set. The constructor will also map the newly constructed object to its initial attribute states if required. A destructor is specified by an operation which removes the object from the instantiations set modelling the deletion of an object from memory. The destructor operation must also remove

all attribute relationships from the attribute variables for the deleted object.

Attributes

To model the classes attributes, extra variables are added to the machine to map the object elements to their corresponding attribute states. At the specification stage of B, a number of primitive types are provided by the B-Toolkit through *SEEING* type machines provided in the B-Toolkits *SLIB* library. These include:

- *Int_TYPE* integers
- *Scalar_TYPE* scalars
- *String_TYPE* strings
- *Char_TYPE* characters
- *Bool_TYPE* booleans

The B-Toolkit also provides corresponding operation machines to change the values of the types specified above. To refer to a non-primitive type, then the user can define that type by using a deferred set placed in a user-defined context machine that is *SEEN* by the class machine being modelled.
To model a reference to a collection, an object can be specified to refer to an ordered sequence of attributes, or simply a set of attributes. There are also a number of different methods for specifying the relationships between the objects and their attributes. The most often used methods include: specifying a partial function from an object to its attribute which means that an attribute reference is optional, or a total function which states that it is mandatory for an object to be referring to an attribute state.

Methods

The class methods are modelled by Operations in the B-machine. Each operation is always given the object as the first argument so the machine knows which object to change the state for. The operation parameters are given in the subsequent arguments.
Please refer to the annotated generic B-Machine 'Class' which shows how the above-mentioned approach is applied to the class shown in the following figure.

B Representation of a Single Class

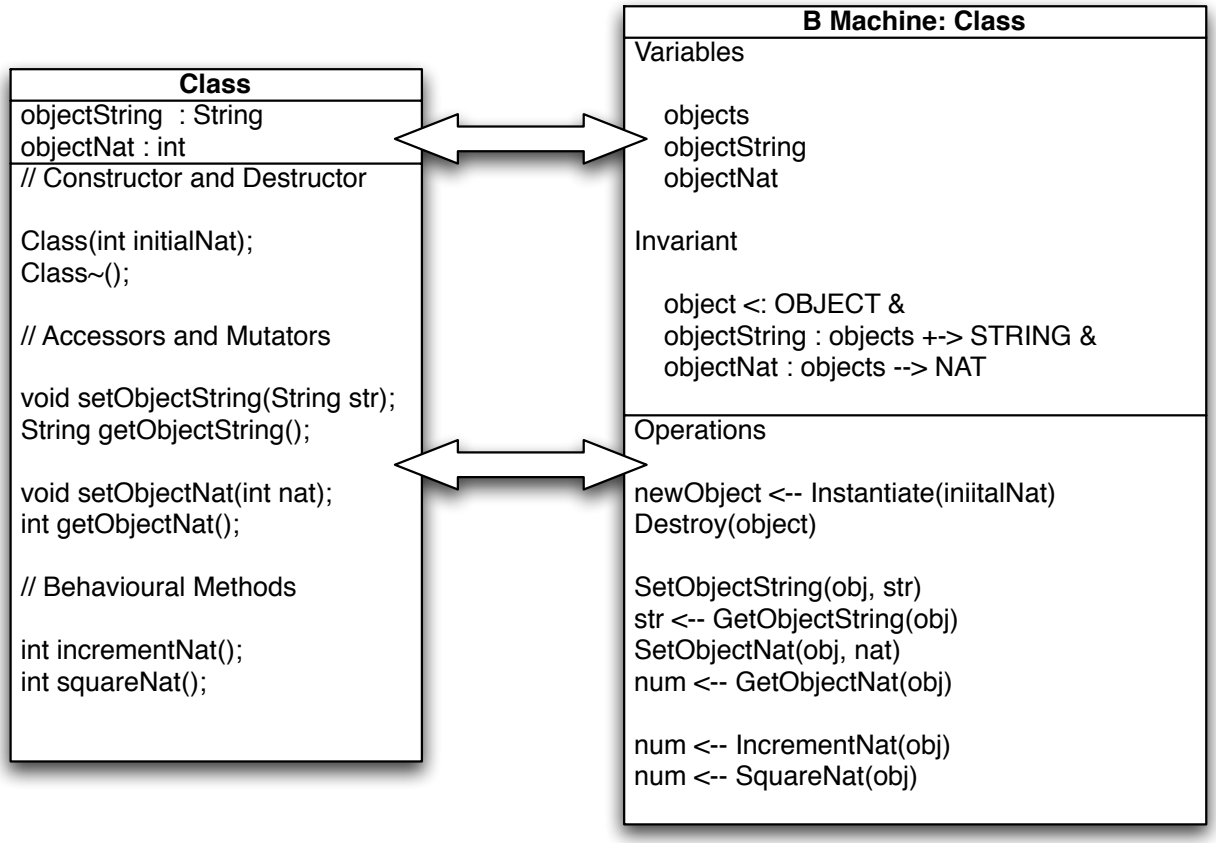


Figure 4.1: Modelling a class using a B machine

4.3.2 Modelling associations between classes using B-machine composition

Given a process to model a single class using a B machine, a methodology is required to model associations between classes. The similarities between object-oriented classes and B machine representations of classes in the above study are quite pronounced. However, where modelling of associations between machines is concerned, the methodology used here in B diverges from the Object Oriented paradigm. B-machines can be composed using several different types of machine composition:

Summary of B Machine Composition mechanisms

1. SEES - Machine M1 seeing M2 will have read-only access to M2s sets, constants and variables but cannot refer to these in its (M1) own invariant. Any number of machines can SEE M2 and 'Seeing' machines cannot invoke M2s operations unless those operations do not change the state of M2. SEES is useful for allowing all the machines within a system access to a *context machine* that contains all the deferred sets and consequently the user-defined non-primitive types.

2. USES - The uses relationship is a generalisation of sees relationship [Schneider,2001] A machine M1 that USES M2 as within the sees relationship also has read only access to M2s sets, constants and variables, the difference being that M1 can refer to M2s variables in in it's own invariant. B places a constraint on the USES relationship by requiring that if M1 uses M2, then a machine M3 must be created that INCLUDES both M1 and M2 for the purposes of discharging proof obligations since M2 has no control of M1 but a state change in M1 may violate the invariant of M2. For the purposes of developing a process to model class associations using B, the USES relationship will not be considered.

3. INCLUDES - If machine M1 'INCLUDES' machine M2, then M1 is able to change the state of M2 by calling the operations of M2. M1 also has access to the sets, constants and variables of M2 and can refer to these in its invariant. Includes relationships are useful because it allows one machine to call the operations of another machine to change it's state which fits neatly with object encapsulation in object oriented programming. However, the limitation is that a machine can only be included by at most one other machine. One machine can however include multiple machine. When M1 includes M2, M1 also has the ability to promote M2s operations to its own interface for a higher level machine to call.

4. EXTENDS - Extending is a special form of the includes relationship, in a case where M2 EXTENDS M1, then M2 has read access to the sets, constants and variables Machine M1 can EXTEND machine M2 meaning it can call all the operations of M2 as well as refer to the state of M2. EXTENDS and INCLUDES is that with EXTEND, M2's operations form part of the interface for M1 if M1 EXTENDS M2.

A process for modelling different types of relationships between classes

Given the above outline of B machine composition mechanisms, a process has been developed to model different types of relationships between classes. Only EXTENDS and INCLUDES composition mechanisms are used because these enable machines to refer to the state of another machine within their invariant and also allow the calling of operations in the included machine.

The most basic class association is a 1-to-1 unidirectional mapping between two classes A and B. This means A refers to a single instance of B in its attributes and uses this reference to call B's operations. This is modelled by having machine A and machine B representing class A and B respectively. Machine A includes Machine B so it will have access to Bs operations and can specify a relationship between the set of A instantiations and the set of B instantiations inside Machine A's invariant by using a function, injection or a bijection, each relationship with its own logical implications for the system that will be discussed in Chapter 6.

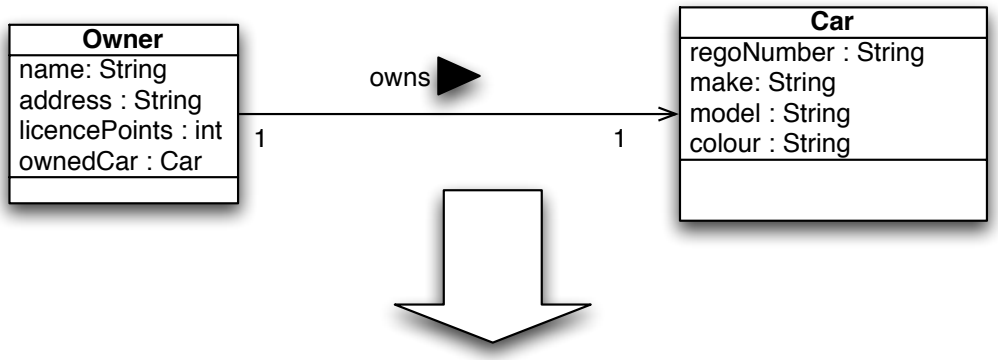
In the example shown in the following diagram, an Owner class and an Car class are represented by B machines *Owner* and *Car*. The owner class contains a reference to a car in its attributes and this is represented by *Owner* including *Car*. We use a total function to specify a relationship between the *owners* and the *cars* variable inside the invariant of the *Owner* machine. Thus we each owner instantiation has a relationship to a car instantiation and can also call operations inside the *Car* machine by providing the car instantiation as the first parameter.

To model bidirectional mapping between class A and class B, then a simple INCLUDES relationship between the machines modelling those classes is not enough as the variables of the including machine are hidden from the included machine. To overcome this problem a third machine, an 'association machine' is introduced which includes both the machines modelling class A and B. The association machine has access to the variables and operations of both and can have a bidirectional relationship specified between the two within its invariant.
An example is shown in the following diagram where there is a Campaign class and a CampaignStaff class being modelled in B. Here, not only is bidirectional mapping modelled but also 1-to-many relationships. The Campaign class has a reference to an array of CampaignStaff objects in its attribute while the CampaignStaff has a reference to a single Campaign object. To model this in B, the association machine specifies within its invariant a function from the campaigns variable to an element of the powerset of the campaign staff variable which represents a 1-to-many relationship. Conversely, a function is used to map campaignstaff to campaign as a 1-to-1 relationship in the opposite direction. Any operation that affects both campaign and campaignstaff variables must be specified in the association machine.

B Representation of Class Coupling

Unidirectional Association (1-to-1) - Car Owner Example using <<INCLUDES>>

Standard Object Oriented Representation



B Machine Composition Reprensentation using B Class Modelling

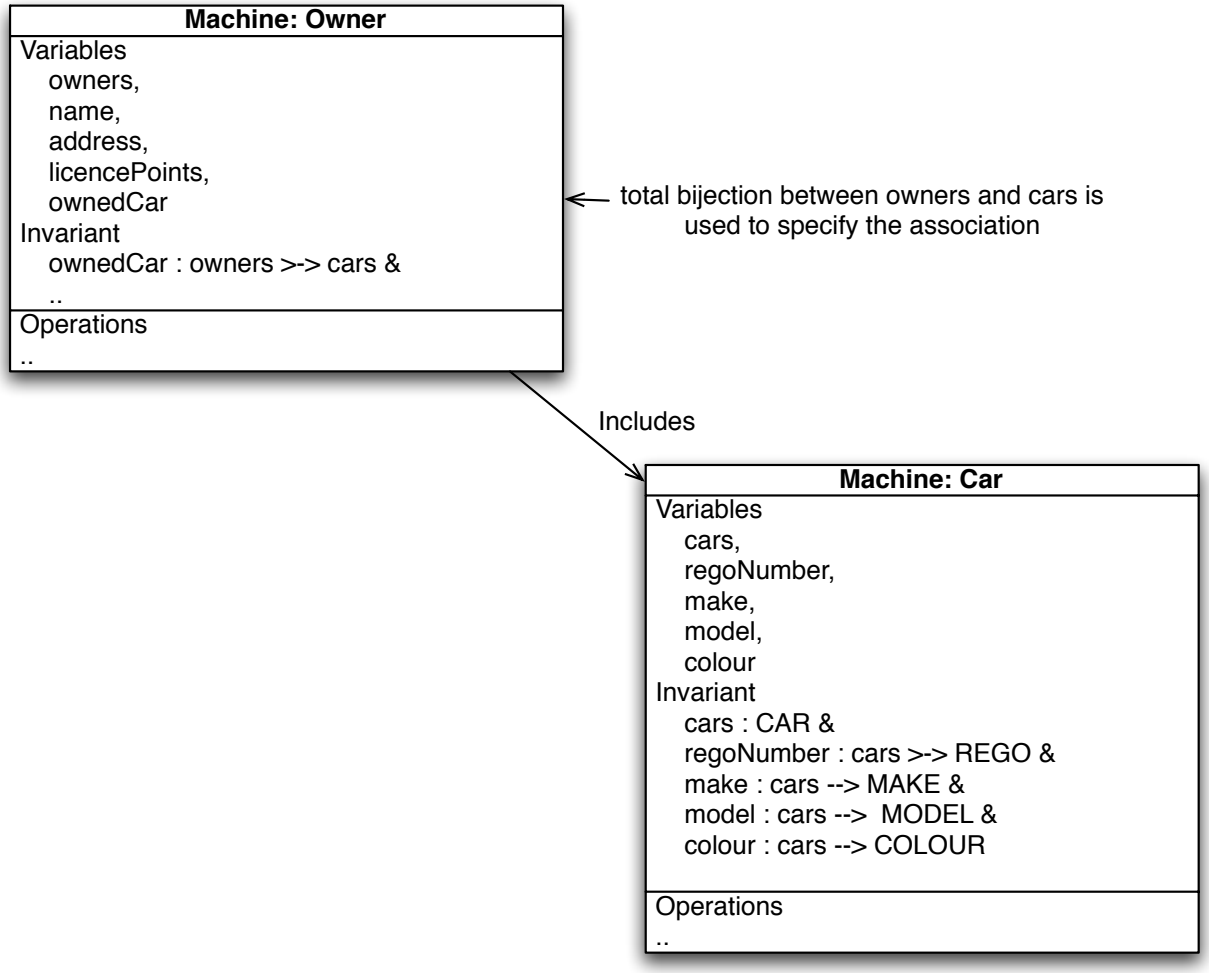


Figure 4.2: Modelling 1-to-1 unidirectional class relationships

Finally, the EXTENDS operation in B is a quasi-representation of subclassing or class specialisation in B. In the example in the following diagram, there is a Account class and a MortgageAccount class which is a subclass of the Account class so it will contain the operations and variables of the superclass. Using extends with an *Account* and a *MortgageAccount* machine, this is an equivalent representation in B with *MortgageAccount* having read-access to *Account*'s variables and also promoting the *Account* operations to its own interface.

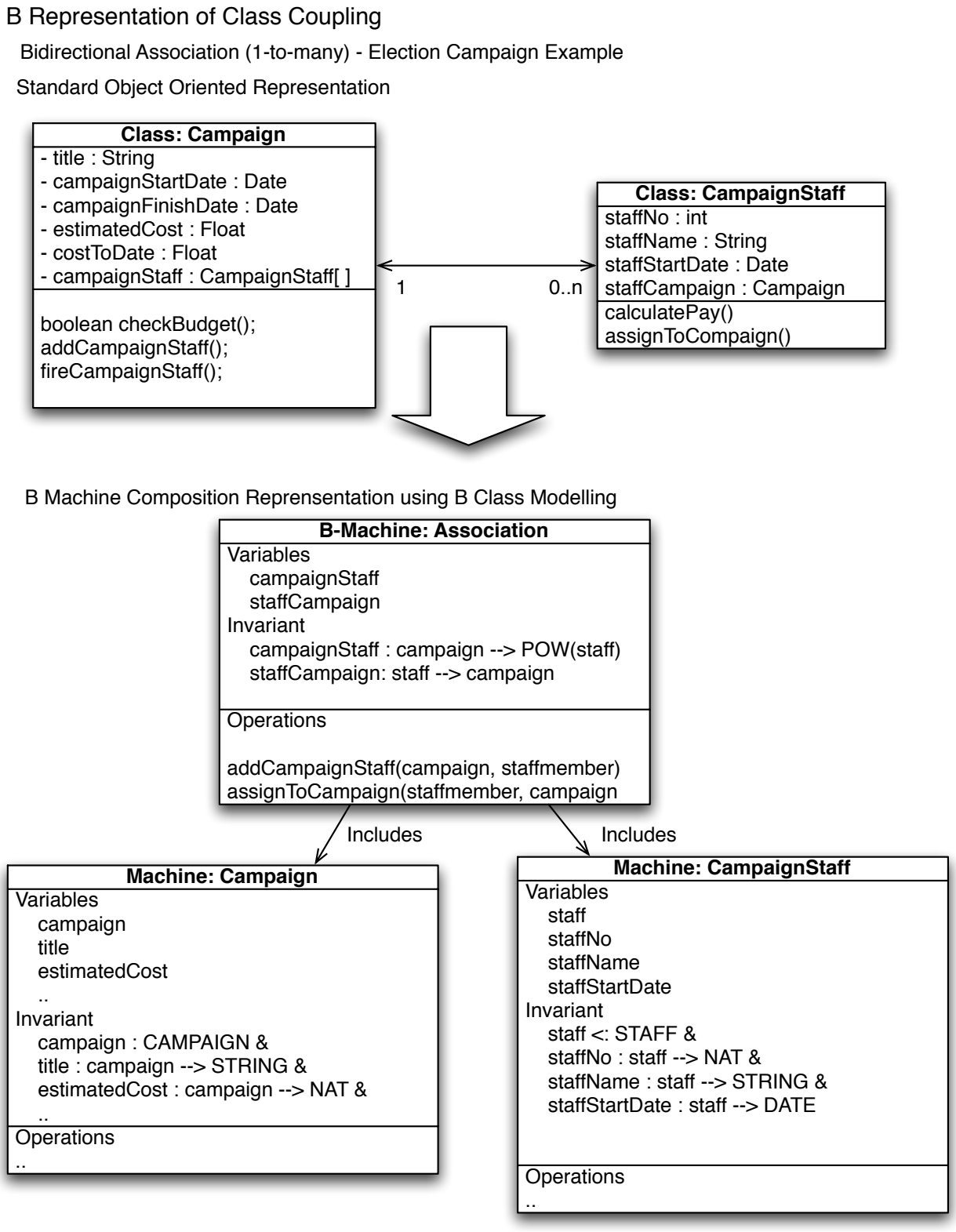


Figure 4.3: Modelling 1-to-m bidirectional class relationships

Limitations of this process

It must be noted that even though this process allows the modelling of OO systems in B, it does have some limitations. While EXTENDS provides a method to model subclassing, this is not totally equivalent as it is not a model of inheritance. Abstract classes and operations cannot be modelled in B and there is no ability to override the operations of extended machines. Dynamic typing is also absent from this model. These limitations prevent the modelling of GoF design patterns where there is a reliance on these object oriented features.
Also of note is that machines can not be included by multiple machines. Therefore situations where multiple classes refer to a single class within their attributes are also hard to model - however this has advantage of decreasing coupling between classes in a modelled object-oriented system.

4.3.3 Research Findings

A clear process for modelling certain types of object oriented classes in B has been developed. This itself can be considered a pattern to be part of a B developers toolkit as it allows for easier abstraction of problems into B. Given this process, a foundation has been laid for modelling some of the GoF patterns to bring their solutions across to B.

4.4 Examining individual patterns in a generic context

For each of the following design patterns, we have taken the functionality of that pattern and attempted to reproduce that functionality in B in a generic, context independent manner. Since they lack any problem specific context, the resulting B Machines could be used as a template to begin building a system, or as a reference point for how a certain effect or functionality can be modelled using B.

4.4.1 The Observer pattern

Observer pattern usage and goals

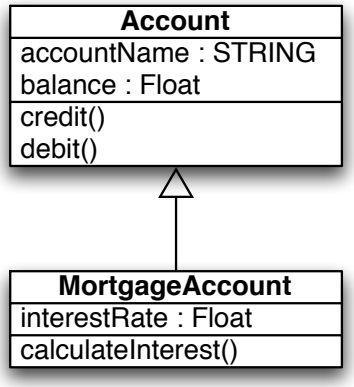
Also known as "publish-subscribe", the Observer pattern should be used when a one-to-many relationship between objects exists, such that a state change in one "observed" machine results in its "observers" being automatically updated. It aims to maintain loose coupling between the observers and the subject, such that each party needs to know little or nothing about the other, while allowing the state of the observer to depend on the subject's state.

Formalising the Observer Pattern - from OO to B

B Representation of Semi - Inheritance

Inheriting variables and functions from another class can be represented by using <<EXTENDS>>

Standard Object Oriented Representation



B Machine Composition Reprensentation using B Class Modelling

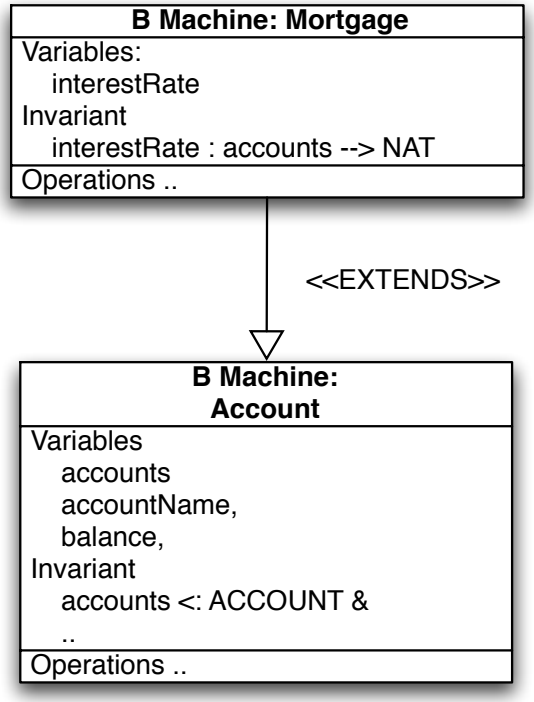


Figure 4.4: Modelling class specialisation in B

Formalising the Observer Pattern - B Specifications

Below we have included the B specifications of the machines that model the Observer pattern.

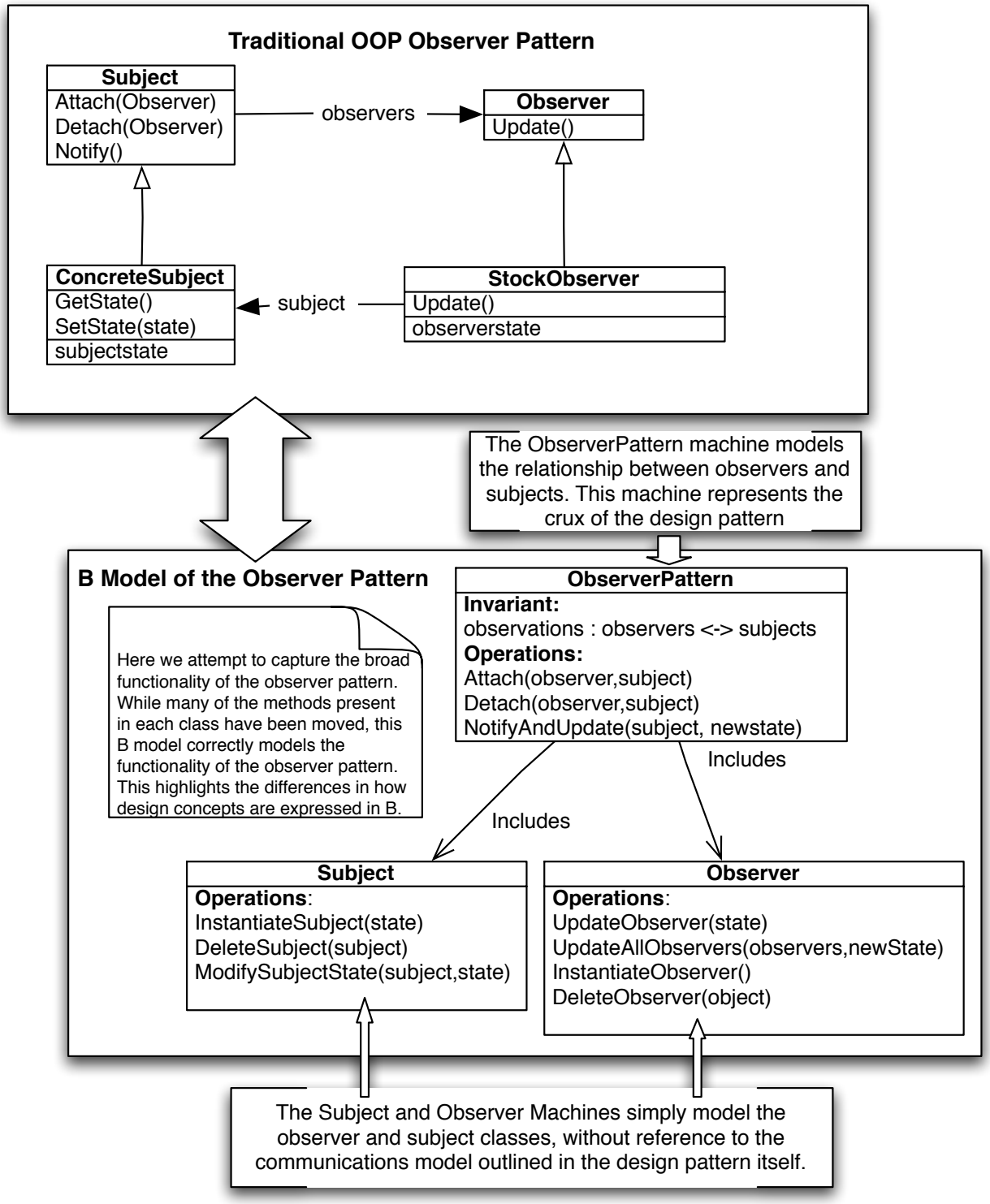


Figure 4.5: Graphical representation of how the Observer pattern is translated to B

MACHINE *Observer*

The Observer machine models the set of observer objects and their behaviour.

SEES *Observer_TYPE* , *ObjectState_TYPE*

VARIABLES

observers ,

observerstate

observers represents the set of observer objects that have been instantiated. *observerState* models that state of each observer.

INVARIANT

$observers \subseteq OBSERVER \wedge$

$observerstate \in observers \rightarrow STATE$

INITIALISATION

$observers := \{\}$

OPERATIONS

Update all observer's in the set *observerset*, setting their state to *newstate*

UpdateAllObservers (*observerset* , *newstate*) $\widehat{=}$

PRE

$observerset \subseteq observers \wedge$

$newstate \in STATE$

THEN

We construct a new function that maps every observer in the set to the new state

ANY *obsfn*

WHERE $obsfn \in observers \pfun STATE \land$

$\mathsf{dom}\ (\ obsfn\ ) = observerset \land$

$\mathsf{ran}\ (\ obsfn\ ) = \{\ newstate\ \} \land$

$\forall\ oo\ .\ (\ oo \in observerset \Rightarrow obsfn\ (\ oo\ ) = newstate\ )$

THEN

Finally we override the old observerstate function with our new function. The observers who do not care about this subject are not affected.

$observerstate := observerstate \rhd\!+ obsfn$

END

END ;

Update an observer's state

UpdateObserver (obs , $state$) $\widehat{=}$

PRE

$obs \in observers \land$

$state \in STATE$

THEN

$observerstate\ (\ obs\ ) := state$

END ;

Simple constructor returning a new observer object

$observer \longleftarrow$ **InstantiateObserver** $\widehat{=}$

BEGIN

ANY $newobserver$, $state$

WHERE $newobserver \in OBSERVER - observers \land state \in STATE$

THEN $observers := observers \cup \{\ newobserver\ \}\ \|$

$observer := newobserver\ \|$

$observerstate\ (\ observer\ ) := state$

END

END ;

Delete an Observer

DeleteObserver (*object*) $\widehat{=}$

PRE

$object \in observers$

THEN

$observers := observers - \{ object \}$

END

END

MACHINE *ObserverPattern*

ObserverPattern models the Observer Design Pattern using the Subject and Observer machines which model subject and observer classes. This machine is the linkage between the observer and the subject. Any methods in the Observer pattern that affect both the observer and the subject both go in this machine.

SEES

ObjectState_TYPE ,

Observer_TYPE ,

Subject_TYPE

INCLUDES

Subject ,

Observer

VARIABLES

observations

observations models the fact that one observer can observe multiple subjects and a subject can be observed by multiple observers. We also model the fact that the state of the *observers* relies on the state of the *subject*.

INVARIANT $observations \in observers \leftrightarrow subjects \wedge$

$\forall (obs , sub) . (obs \in observers \wedge sub \in subjects \wedge obs \mapsto sub \in observations$

$\Rightarrow$

$subjectstate (sub) = observerstate (obs))$

INITIALISATION $observations := \{\}$

OPERATIONS

Attach (*obs* , *subj*) $\widehat{=}$

PRE

$obs \in observers \wedge$

$subj \in subjects$

THEN

$observations\ [\ \{\ obs\ \}\ ] := observations\ [\ \{\ obs\ \}\ ] \cup \{\ subj\ \}\ \|$

$UpdateObserver\ (\ obs\ ,\ subjectstate\ (\ subj\ )\ )$

END ;

Detach (obs , $subj$) $\widehat{=}$

PRE

$obs \in \mathsf{dom}\ (\ observations\ ) \wedge$

$subj \in \mathsf{ran}\ (\ observations\ ) \wedge$

$obs \mapsto subj \in observations$

THEN

$observations := observations - \{\ obs \mapsto subj\ \}$

END ;

NotifyAndUpdate models both the observers *Notify* operation and the subject's *Update* operation. This is necessary due to the requirement in B specifications that sequential composition is not allowed the operation is only ever allowed to embody a single state change. So instead we have specified that the subject and its observers' state are completely synchronized, and that the notify and update happen in parallel.

NotifyAndUpdate ($subject$, $newstate$) $\widehat{=}$

PRE

$subject \in subjects \wedge$

$newstate \in STATE$

THEN

$ModifySubjectState\ (\ subject\ ,\ newstate\ )\ \|$

We take all the observers that are watching the given subject, and call the observer machine to update all of the observers

$UpdateAllObservers\ (\ observations^{-1}\ [\ \{\ subject\ \}\ ]\ ,\ newstate\ )$

END

END

Please note we have not included the subject machine as it does not add much to this discussion. The subject machine can be found in the Appendix.

Analysing the Observer Pattern

Modelling the Observer pattern presented some difficulty. This was due to the conceptual divide between OO and B, and the fact that B presents ways of modelling the intent of the observer pattern in multiple ways. The first major hurdle is presented by the update operation. In a B machine, each operation must model a state change. It cannot model sequential state changes, which is implied by the observer pattern - the state of the subject changes, and *then* the state of the observers change. This sequence cannot be modelled in a specification, so it was necessary to simultaneously update both.
An interesting side effect of this is that we now need to prove that the abstract "state" of the subject is always mirrored by the observer. This effect is not directly expressed in the intent of the observer pattern, but it is certainly implied. Technically speaking, the pattern only requires an observer to be notified of a change - whether a state change occurs or not is up to the developer. The B model, however, requires this state mirroring in order to explicitly model the update operation.

4.4.2 The Flyweight Pattern

Flyweight Pattern usage and goals

The Flyweight pattern uses object sharing to manage a high number of objects efficiently. Proper usage of the flyweight results in a lower memory footprint since objects that are the same are shared instead of duplicated.

Formalising the Flyweight Pattern - from OO to B

Formalising the Flyweight Pattern - B Specifications

Below we present three B machines that represent the flyweight pattern.

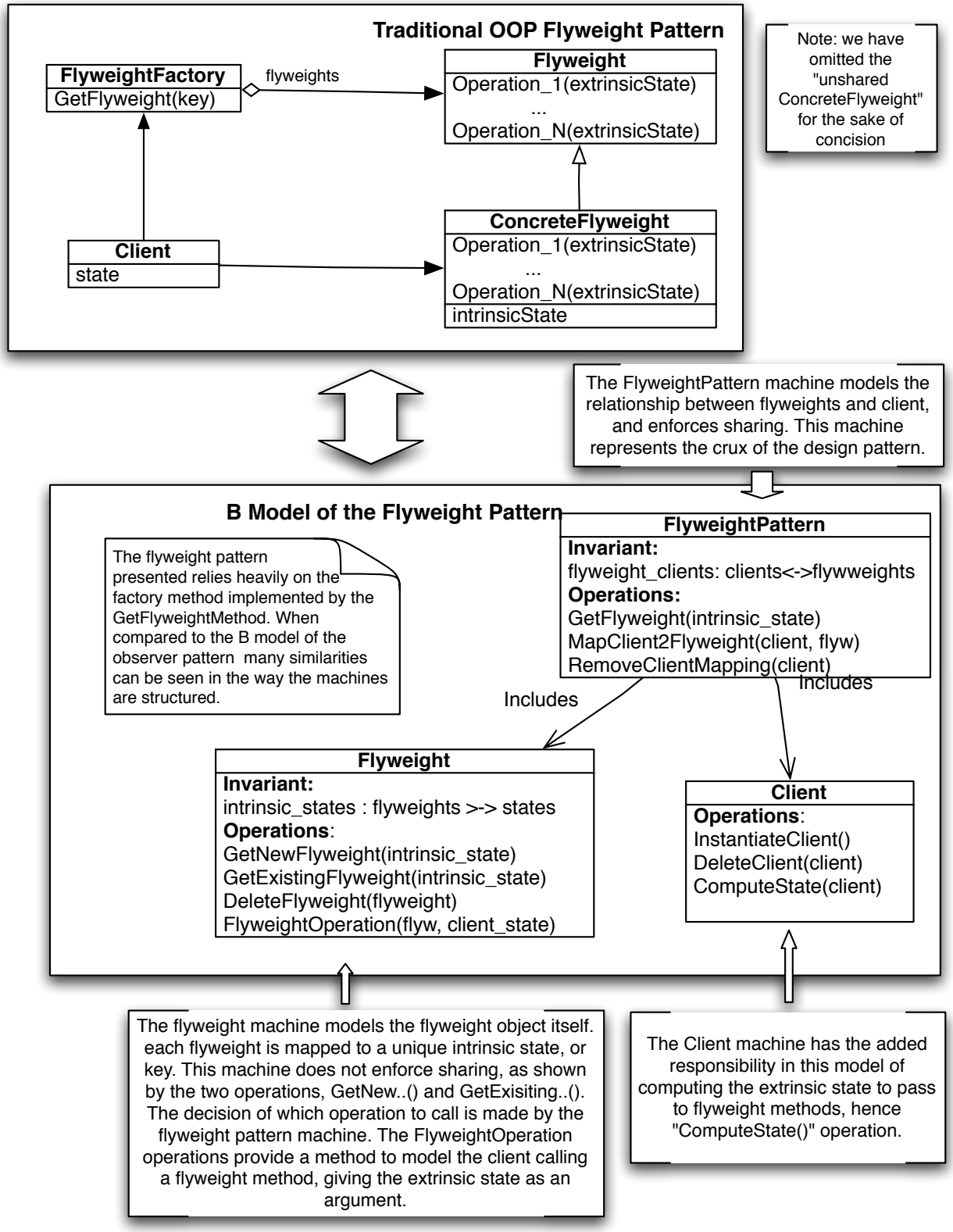


Figure 4.6: Graphical representation of how the Flyweight pattern is translated to B

The *Flyweight* machine models and manages the shared flyweight objects. Each Flyweight has its intrinsic state - a state that never changes. Flyweight operations usually provide a result by taking in an extrinsic state and applying it to their intrinsic state.

MACHINE *Flyweight*

SEES *ObjectState_TYPE* , *Flyweight_TYPE* , *Result_TYPE*

The flyweight object has an operation that takes an extrinsic state and it's own intrinsic state and does *something* with it. Because we don't know what that something is, we provide **eval**, a function that maps all external states to the intrinsic states that map to a resulting state.

CONSTANTS

eval

PROPERTIES

$eval \in STATE \rightarrow (\ STATE \rightarrow STATE \ )$

VARIABLES

flyweights ,

states ,

intrinsic_states

INVARIANT

$flyweights \subseteq FLYWEIGHT \ \wedge$

$states \subseteq STATE \ \wedge$

Flyweights are meant to be shared. As such, there should never be two flyweights with the same intrinsic state since the original flyweight with that intrinsic state should be reused. A Total Function is required since flyweights can't exist without an intrinsic state

$intrinsic\_states \in flyweights \rightarrowtail states$

INITIALISATION

$flyweights := \{\} \ \ \|$

$states := \{\} \ \ \|$

intrinsic_states := {}

OPERATIONS

To support sharing, two operations are provided, keyed on the *state* of the flyweight. If a flyweight exists with the given state, *GetExistingFlyweight* should be called. Otherwise a new Flyweight is constructed by *GetNewFlyweight*

flyw ⟵ **GetNewFlyweight** (*i_state*) ≙

PRE

i_state ∈ *STATE* ∧

i_state ∉ ran (*intrinsic_states*)

THEN

ANY *fly*

WHERE

fly ∈ *FLYWEIGHT* − *flyweights*

THEN

flyweights := *flyweights* ∪ { *fly* } ||

states := *states* ∪ { *i_state* } ||

intrinsic_states (*fly*) := *i_state* ||

flyw := *fly*

END

END ;

flyw ⟵ **GetExistingFlyweight** (*i_state*) ≙

PRE

i_state ∈ ran (*intrinsic_states*)

THEN

flyw := *intrinsic_states* $^{-1}$ (*i_state*)

END ;

DeleteFlyweight (*flyw*) ≙

PRE

flyw ∈ *FLYWEIGHT* ∧

flyw ∈ *flyweights*

THEN

flyweights := *flyweights* − { *flyw* } ||

intrinsic_states := { *flyw* } ⩤ *intrinsic_states*

END **;**

We apply the eval function to the client and the flyweight state get a result.

result ⟵ **FlyweightOperation** (*flyw* , *client_state*) ≙

PRE

flyw ∈ *FLYWEIGHT* ∧

client_state ∈ *STATE*

THEN

result := *eval* (*client_state*) (*intrinsic_states* (*flyw*))

END

END

The *Client* machine models the flyweight client the user of the flyweight objects. For the Client, the shared nature of flyweights is invisible.

MACHINE *Client*

SEES *ObjectState_TYPE* , *Client_TYPE*

VARIABLES *clients* ,

context_state

For the Flyweight pattern, clients are required to be able to generate or provide a context dependent state. This state is provided as part of the arguments to the flyweight itself, and becomes the extrinsic state. This is represented by the *context_state* variable

INVARIANT

$clients \subseteq CLIENT \wedge$

$context\_state \in clients \rightarrow STATE$

INITIALISATION

$clients := \{\} \parallel$

$context\_state := \{\}$

OPERATIONS

This operation allows the pattern machine to compute what state needs to be passed to the flyweight machine.

$state \longleftarrow$ **ComputeState** (*client*) $\widehat{=}$

PRE

$client \in CLIENT \wedge$

$client \in \mathsf{dom}$ (*context_state*)

THEN

$state := context\_state$ (*client*)

END ;

We create a new Client in the standard way, ensuring it's context_state is initialised also.

$cl \longleftarrow$ **InstantiateClient** $\widehat{=}$

PRE $clients \neq CLIENT$

THEN

ANY $client$, $state$

WHERE

$client \in CLIENT - clients \wedge$

$state \in STATE$

THEN

$clients := clients \cup \{ client \} \parallel$

$context\_state (client) := state \parallel$

$cl := client$

END

END ;

DeleteClient ($client$) $\widehat{=}$

PRE

$client \in CLIENT \wedge$

$client \in \mathsf{dom} (context\_state)$

THEN

$clients := clients - \{ client \} \parallel$

$context\_state := \{ client \} \lhd context\_state$

END

END

The *FlyweightPattern* machine maintains and enforces the sharing relationship between the Flyweights and their clients. This sharing relationship resulted in the creation of a factory method in the *GetFlyWeight* operation.

MACHINE *FlyweightPattern*

SEES

ObjectState_TYPE ,

Flyweight_TYPE ,

Result_TYPE ,

Client_TYPE

INCLUDES

Flyweight ,

Client

VARIABLES

flyweight_clients

INVARIANT

Every client that has a flyweight needs to reference it. a client can map to many flyweights, and flyweights are shared by many clients

$flyweight\_clients \in clients \leftrightarrow flyweights$

INITIALISATION

$flyweight\_clients := \{\}$

OPERATIONS

Flyweights are usually instantiated through a factory method to enforce sharing. We recreate flyweight sharing here using the same concept of a factory expressed and constrained mathematically here and in the Flyweight machine.

$flyw \longleftarrow$ **GetFlyweight** (i_state) $\widehat{=}$

PRE

$i\_state \in STATE$

THEN

SELECT

$i\_state \notin \mathsf{ran}\ (\ intrinsic\_states\ )$

THEN

$flyw \longleftarrow GetNewFlyweight\ (\ i\_state\ )$

WHEN

$i\_state \in \mathsf{ran}\ (\ intrinsic\_states\ )$

THEN

$flyw \longleftarrow GetExistingFlyweight\ (\ i\_state\ )$

END

END ;

After creating a flyweight, a client can be attached to the flyweight using *MapClient2Flyweight*

MapClient2Flyweight ($client$, $flyw$) $\widehat{=}$

PRE

$client \in clients \wedge$

$flyw \in flyweights$

THEN

$flyweight\_clients := flyweight\_clients \cup \{\ client \mapsto flyw\ \}$

END ;

RemoveClientMapping ($client$) $\widehat{=}$

PRE

$client \in CLIENT \wedge$

$client \in \mathsf{dom}\ (\ flyweight\_clients\ )$

THEN

$flyweight\_clients := \{\ client\ \} \lhd flyweight\_clients$

END

END

Analysing the Flyweight Pattern

The flyweight pattern presented an interesting challenge. To make flyweight work, we needed to support the sharing of objects. This meant that we needed an approach similar to the factory method - yet another design pattern. In fact, the flyweight pattern could be called a specialised formalisation of the factory method, since factory provides the most crucial mechanism - object sharing- of the entire pattern.

4.4.3 The Iterator Pattern

Iterator Pattern usage and goals

The Iterator pattern shows how to access the members of a collection of objects without exposing the representation of the collection. As such a tree collection could be accessed in the same manner as a stack, or a hashtable's key collection. Different iterators can provide different methods of traversing a collection.

Formalising the Iterator Pattern - from OO to B

Formalising the Iterator Pattern - B Specifications

Below are the salient B specifications for the Iterator pattern.

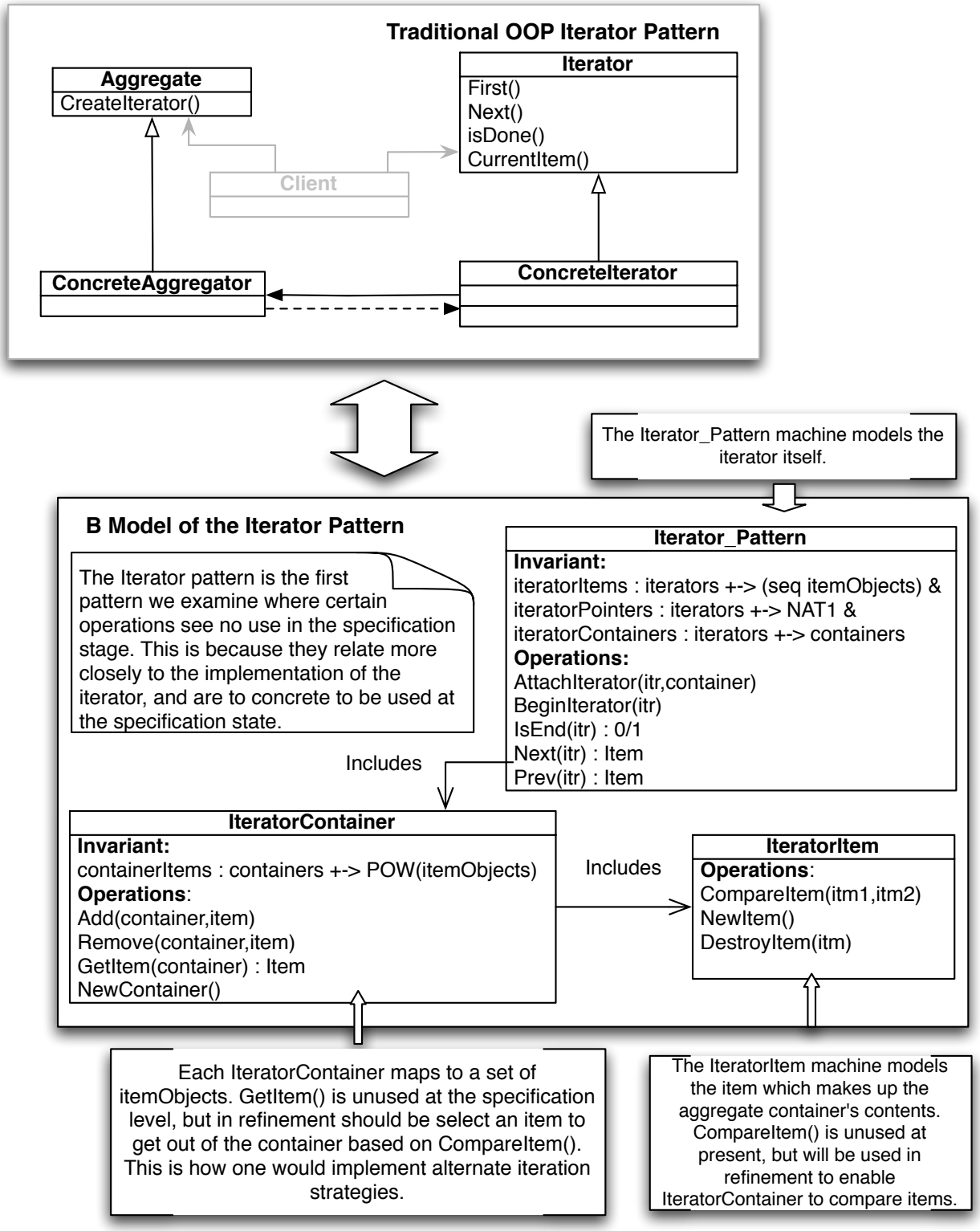


Figure 4.7: Graphical representation of how the Iterator pattern is translated to B

The *Iterator_Pattern* machine models the iterator behaviour. It manages the iteration strategy and the internal iterator pointer.

MACHINE *Iterator_Pattern* (*ATTRIBUTE* , *max_instance*)

CONSTRAINTS $max\_instance \in \mathbb{N}_1$

SEES *Class_CTX*

INCLUDES *IteratorContainer* (*ATTRIBUTE* , *max_instance*)

VARIABLES *iterators* , *iteratorItems* , *iteratorContainers* , *iteratorPointers*

iterators models the iterator objects. *iteratorItems* models an internal sequence of itemObjects, mapped to an iterator. This is necessary to impose order on an unordered set and hence consistently iterate over a set with next() and prev() operations. *iteratorPointers* maps each iterator to a number which represents the current position of the iterator itself. *iteratorContainers* maintains a reference to each iterator's attached container.

INVARIANT $iterators \subseteq OBJECT \wedge$

$iteratorItems \in iterators \nrightarrow \mathsf{seq}\ (\ itemObjects\ ) \wedge$

$iteratorPointers \in iterators \nrightarrow \mathbb{N}_1 \ \wedge$

$iteratorContainers \in iterators \nrightarrow containers$

INITIALISATION $iterators := \{\}$

$\|$ $iteratorItems := \{\}$

$\|$ $iteratorPointers := \{\}$

$\|$ $iteratorContainers := \{\}$

OPERATIONS

AttachIterator wraps the given iterator around the given container

AttachIterator (*itr* , *container*) $\widehat{=}$

PRE

$itr \in iterators \wedge$

$container \in containers$

THEN

iteratorContainers (*itr*) := *container* ||

iteratorItems (*itr*) := [ ]

END ;

BeginIterator simply sets the iterator's internal pointer to the first position. If the iterator has already iterated across some items, this allows us to iterate in the same order.

BeginIterator (*itr*) $\widehat{=}$

PRE

$itr \in iterators \wedge$

$itr \in \mathsf{dom}$ (*iteratorContainers*)

THEN

iteratorPointers (*itr*) := *1*

END ;

IsEnd must be called before calling *Next*. It tells us if the iterator has items or if it is finished

bb $\longleftarrow$ **IsEnd** (*itr*) $\widehat{=}$

PRE

$itr \in iterators$

THEN

If the iterator hasn't been initialised, then isEnd is true.

SELECT $itr \notin \mathsf{dom}$ (*iteratorContainers*) **THEN**

bb := *1*

If the iterator hasn't had BeginIterator called, then isEnd is true.

WHEN $itr \notin \mathsf{dom}$ (*iteratorPointers*) **THEN**

bb := *1*

If the iterator has begun, and the cardinality of our internal sequence of items is equal to the cardinality of the attached container, IsEnd is True.

WHEN size (*iteratorItems* (*itr*)) = card (*containerItems* (*iteratorContainers* (*itr*))) **THEN**

bb := *1*

ELSE

In all other cases, IsEnd is false.

bb := *0*

END

END ;

Next returns the next item in the iterator. If the iterator pointer is alreader pointing to the end of it's internal sequence in *iteratorItems*, we get a random item from the container that we have not visited yet and append it to the end of our sequence of visited items in *iteratorItems*.

itm ⟵ **Next** (*itr*) ≙

PRE

itr ∈ *iterators* ∧

itr ∈ dom (*iteratorItems*) ∧

itr ∈ dom (*iteratorContainers*) ∧

itr ∈ dom (*iteratorPointers*)

THEN

SELECT *iteratorPointers* (*itr*) < size (*iteratorItems* (*itr*))

THEN

itm := *iteratorItems* (*itr*) (*iteratorPointers* (*itr*) + *1*) ||

iteratorPointers (*itr*) := *iteratorPointers* (*itr*) + *1*

ELSE

ANY *obj*

WHERE

obj ∈ *itemObjects* ∧

obj ∈ *containerItems* (*iteratorContainers* (*itr*)) ∧

$obj \notin \mathsf{ran}\ (\ iteratorItems\ (\ itr\ )\ )$

THEN

$itm := obj \ \ \|$

$iteratorItems\ (\ itr\ ) := iteratorItems\ (\ itr\ ) \leftarrow obj \ \ \|$

$iteratorPointers\ (\ itr\ ) := iteratorPointers\ (\ itr\ ) + 1$

END

END

END ;

Previous returns the last viewed item. It decrements the *iteratorPointer*. *Previous* and *Next* can be called in any order and we can guarantee that the same order of items will be presented due to the use of our internal sequence of iterator items.

$itm \longleftarrow$ **Previous** $(\ itr\ ) \ \widehat{=}$

PRE

$itr \in iterators \wedge$

$itr \in \mathsf{dom}\ (\ iteratorContainers\ ) \wedge$

$itr \in \mathsf{dom}\ (\ iteratorPointers\ ) \wedge$

$iteratorPointers\ (\ itr\ ) > 1$

THEN

$itm := iteratorItems\ (\ itr\ )\ (\ iteratorPointers\ (\ itr\ ) - 1\ )\ \|$

$iteratorPointers\ (\ itr\ ) := iteratorPointers\ (\ itr\ ) - 1$

END ;

$itr \longleftarrow$ **NewIterator** $\widehat{=}$

BEGIN

ANY ii

WHERE

$ii \in OBJECT - iterators$

THEN

$itr := ii \ \ \|$

$iterators := iterators \cup \{\ ii\ \}$

END

```
    END

END
```

IteratorContainer Models the container that the iterator will iterate across. In this case, the container is simply an unordered Set of items, also known as a Bag of items.

MACHINE *IteratorContainer* (*ATTRIBUTE* , *max_size*)

max_size models the maximum size of the container.

CONSTRAINTS $max\_size \in \mathbb{N}_1$

SEES *Class_CTX*

INCLUDES *IteratorItem* (*ATTRIBUTE* , *max_size*)

VARIABLES *containerItems* , *containers*

containers model the container objects, while *containerItems* map each container to an unordered set of itemObjects. Note that in this container, it is impossible to have the exact same itemObject more than once in the container.

INVARIANT $containers \subseteq OBJECT \wedge$
$containerItems \in containers \nrightarrow \mathbb{P}$ (*itemObjects*)

INITIALISATION $containerItems := \{\} \parallel$
$containers := \{\}$

OPERATIONS

NewContainer instantiates a new Container of items

$container \longleftarrow$ **NewContainer** $\widehat{=}$
BEGIN
ANY tt
WHERE $tt \notin containers \wedge$
$tt \in OBJECT$
THEN
$container := tt \parallel$

containers := *containers* ∪ { *tt* }

END

END ;

Add adds an Item to a container.

Add (*container* , *item*) ≙

PRE

container ∈ *containers* ∧

item ∈ *itemObjects*

THEN

SELECT *container* ∈ dom (*containerItems*)

THEN

containerItems (*container*) := *containerItems* (*container*) ∪ { *item* }

ELSE

containerItems (*container*) := { *item* }

END

END ;

Remove removes an item from a container.

Remove (*container* , *item*) ≙

PRE

container ∈ *containers* ∧

item ∈ *itemObjects* ∧

container ∈ dom (*containerItems*) ∧

item ∈ *containerItems* (*container*)

THEN

containerItems (*container*) := *containerItems* (*container*) − { *item* }

END ;

GetItem randomly retrieves an item from a container. This is random since the container is a simple set, so no concept of order exists. The only get operation possible is a random get.

$itm \longleftarrow$ **GetItem** (*container*) $\widehat{=}$

PRE

$container \in containers \wedge$

$container \in \mathsf{dom}\ (\ containerItems\ ) \wedge$

$containerItems\ (\ container\ ) \neq \{\}$

THEN

ANY ii

WHERE $ii \in containerItems\ (\ container\ )$

THEN

$itm := ii$

END

END

END

This machine models a generic object contained in an Iterator container. When Iterator.Next() gets called, an itemObject is returned.

MACHINE *IteratorItem* (*ATTRIBUTE* , *max_instance*)

CONSTRAINTS $max\_instance \in \mathbb{N}_1$

SEES *Class_CTX*

VARIABLES *itemObjects* , *itemAttribute*

itemObjects models the pool of allocated items in the iterator collection. *itemAttribute* maps each object to a generic attribute, which will be used for comparison.

INVARIANT $itemObjects \subseteq OBJECT \land$
$itemAttribute \in itemObjects \nrightarrow ATTRIBUTE$

INITIALISATION *itemObjects* , *itemAttribute* := {} , {}

OPERATIONS

A Generic comparison operation. This would need to be modified to fit the problem at hand. The comparison could change depending on what sort of iteration you needed to perform or the attribute you need to compare.

$retVal \longleftarrow$ **CompareItem** (*itm1* , *itm2*) $\widehat{=}$

PRE

$itm1 \in itemObjects \land$

$itm2 \in itemObjects \land$

$itm1 \in \mathsf{dom}\ (\ itemAttribute\ ) \land$

$itm2 \in \mathsf{dom}\ (\ itemAttribute\ )$

THEN

SELECT $itemAttribute\ (\ itm1\ ) = itemAttribute\ (\ itm2\ )$

THEN

$retVal := 0$

WHEN $itemAttribute\ (\ itm1\ ) > itemAttribute\ (\ itm2\ )$

THEN

retVal := *1*

END

END ;

InstantiateItem constructs a new ytem to be put in an Iterator collection

newItem ⟵ **InstantiateItem** ≙

PRE

itemObjects ≠ *OBJECT*

THEN

ANY *itm*

WHERE

itm ∈ *OBJECT* ∧

itm ∉ *itemObjects*

THEN

itemObjects := *itemObjects* ∪ { *itm* } ||

newItem := *itm*

END

END ;

DestroyItem destroys an item

DestroyItem (*itm*) ≙

PRE

itm ∈ *itemObjects*

THEN

itemObjects := *itemObjects* − { *itm* } ||

itemAttribute := { *itm* } ⩤ *itemAttribute*

END ;

SetItemAttribute assigns the attribute of the given item to the state given.

SetItemAttribute (*itm* , *newAttribState*) $\widehat{=}$

PRE

$itm \in itemObjects \wedge$

$newAttribState \in ATTRIBUTE$

THEN

$itemAttribute\ (\ itm\ ) := newAttribState$

END ;

GetItemAttribute returns the current state of the item's attribute.

$attr \longleftarrow$ **GetItemAttribute** (*itm*) $\widehat{=}$

PRE

$itm \in itemObjects \wedge$

$itm \in \mathsf{dom}\ (\ itemAttribute\ )$

THEN

$attr := itemAttribute\ (\ itm\ )$

END

END

Analysing the Iterator Pattern

The Iterator pattern is a subtly powerful pattern. Capturing the power of Iterator required the specification of both an Item, a Container and an Iterator, all in a linear inclusion hierarchy. This is a departure from our previous patterns, which have all involved a *pattern machine* including its participants. In this development, the pattern machine itself is an object that uses the container.
The container provided is purely a set. It does not allow duplicate entries, and is unordered. That means that the Iterator is faced with two challenges:

1. How to iterate through a container with no inherent order.
2. How to maintain the path of iteration to consistently support Next() and Previous() operations.

To solve this required consideration on the nature of the Iterator. The Iteratorregardless of implementation or the container to iterate over - must impose its *own concept of order* on the collection. Two iterators can iterate differently across a container because their concept of *order* is different. Since a Set has no concept of order, we decided to use a random iteration strategy, pulling elements out of the set at random to place in a sequence of objects. This sequence solves the second problem, as it allows us to consistently present the same results as we traverse the set.

4.4.4 The Command Pattern

Command Pattern Usage and Goals

The Command pattern encapsulates a system "command" as a single object which can be passed through the system. This can assist in synchronising asynchronous requests through a request queue and allows undo and redo operations.

Formalising the Command Pattern - from OO to B

Formalising the Command Pattern - B Specifications

Below are the salient B specifications for the command pattern.

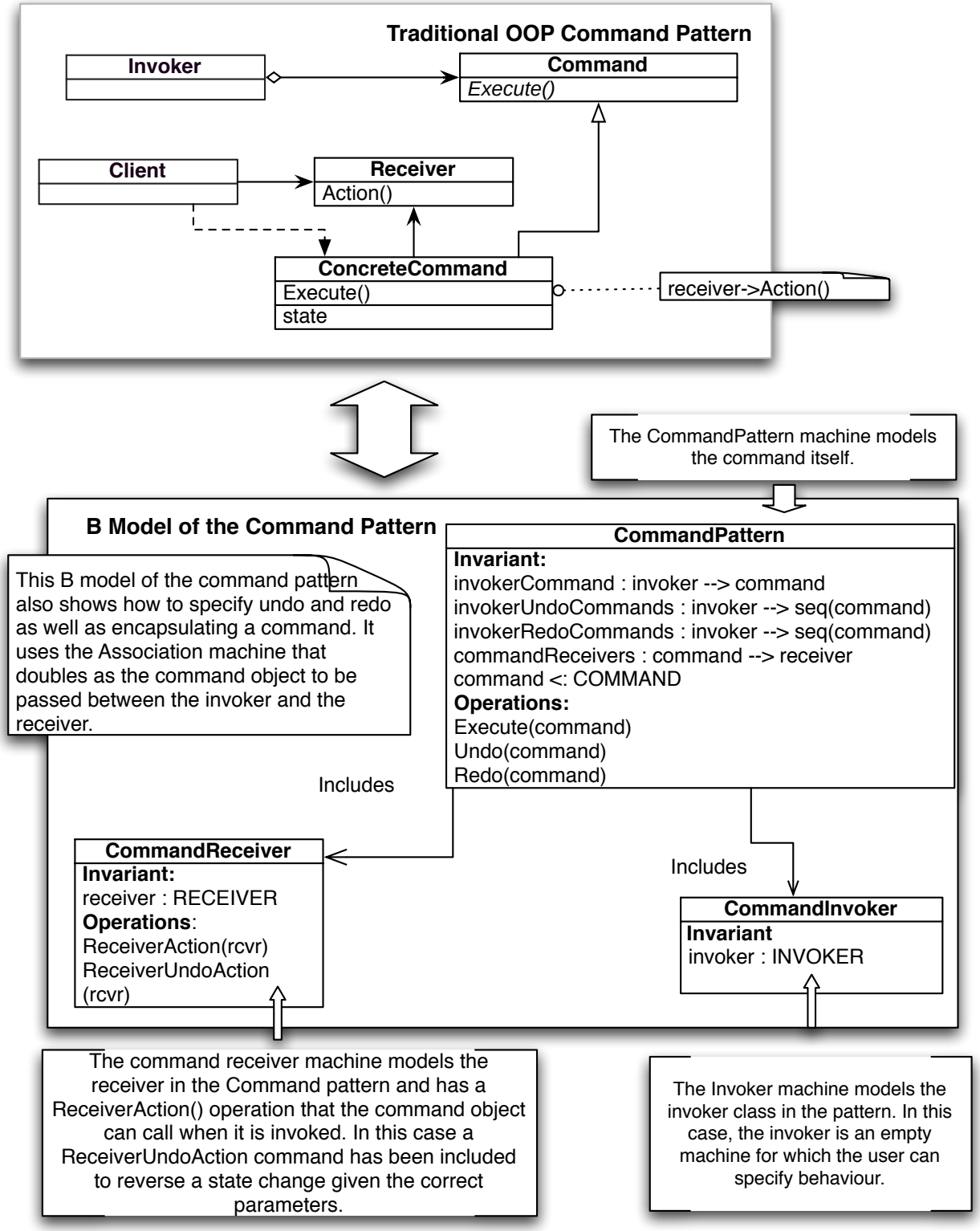


Figure 4.8: Graphical representation of how the Command pattern is translated to B

MACHINE *CommandReceiver* (*ATTRIBUTE* , *max_instance*)

The *CommandReceiver* machine models the receiver in the Command Design pattern. The interface includes a *ReceiverAction* operation which is called by the command object when it is invoked. There is also a *ReceiverUndoAction* operation that can be called which reverts the receiver object to a previous state

CONSTRAINTS $max\_instance \in \mathbb{N}_1$

SEES *Class_CTX*

In this command desig pattern template, variables are required to model the set of receiver objects and an arbitrary attribute. The user can change this machine so it carries out useful operations by remodelling the attributes.

VARIABLES *receiverObjects* , *receiverAttribute*

INVARIANT $receiverObjects \subseteq OBJECT \wedge$

$receiverAttribute \in receiverObjects \nrightarrow ATTRIBUTE$

INITIALISATION *receiverObjects* , *receiverAttribute* := {} , {}

OPERATIONS

ReceiverAction is a skip operation in this templatised example and is the operation called by the invoked command object in the *CommandPattern* machine that includes this machine

ReceiverAction (*rcvr*) $\widehat{=}$

PRE

$rcvr \in receiverObjects$

THEN

skip

END ;

UndoAction, given a parameter undo what was done by 'ReceiverAction' and return this receiver to its previous state. Because this machine holds no state, it is a skip operation that can be changed by the user.

ReceiverUndoAction (*rcvr*) $\widehat{=}$

PRE

$rcvr \in receiverObjects$

THEN

skip

END ;

Standard class operations: constructor, destructor, accessor and mutator are specified below using the standard process for modelling a class in B.

newReceiver $\longleftarrow$ **InstantiateReceiver** $\widehat{=}$

PRE

$receiverObjects \neq OBJECT$

THEN

ANY *rcvr*

WHERE

$rcvr \in OBJECT \wedge$

$rcvr \notin receiverObjects$

THEN

$receiverObjects := receiverObjects \cup \{ rcvr \} \parallel$

$newReceiver := rcvr$

END

END ;

DestroyReceiver (*rcvr*) $\widehat{=}$

PRE

$rcvr \in receiverObjects$

THEN

$receiverObjects := receiverObjects - \{ rcvr \} \parallel$

$receiverAttribute := \{ rcvr \} \lhd receiverAttribute$

END ;

SetReceiverAttribute (*rcvr* , *newAttribState*) $\widehat{=}$

PRE

$rcvr \in receiverObjects \wedge$

$newAttribState \in ATTRIBUTE$

THEN

$receiverAttribute\ (\ rcvr\ ) := newAttribState$

END ;

$attr \longleftarrow$ **GetReceiverAttribute** (*rcvr*) $\widehat{=}$

PRE

$rcvr \in receiverObjects \wedge$

$rcvr \in \mathsf{dom}\ (\ receiverAttribute\ )$

THEN

$attr := receiverAttribute\ (\ rcvr\ )$

END

END

MACHINE *CommandPattern* (*RCVRATTRIBUTE* , *IVRATTRIBUTE* , *maxInstance*)

Models the Command Design Pattern using generic Receiver and Invoker class machines. This machine is a model of the Command Class itself within the pattern as well as modelling the relationships between the Invoker and Receiver objects and their command objects.

CONSTRAINTS $maxInstance \in \mathbb{N}_1$

The system-wide context machine used in this development only contains the a generic 'Object' type so the types of the attributes for the *CommandReceiver* and *CommandInvoker* machines are passed to them via machine parameters. In an application of this command design pattern, a context machine would be used to define types.

SEES *Class_CTX*

Variables are required to map invokers to the:

- current command are going to invoke
- the stack of commands that have been executed (Undo stack)
- the stack of commands that have been undone and need to be re-executed (Redo stack)

A variable is also required to map command objects to their receiver so and also to hold the set of instantiated command objects in the system.

INCLUDES *CommandReceiver* (*RCVRATTRIBUTE* , *maxInstance*) , *CommandInvoker* (*IVRATTRIBUTE* , *maxIns*

VARIABLES

Invoker to Command associations

invokerCommand ,

invokerRedoCommands ,

invokerUndoCommands ,

Command to Receiver association

commandReceivers ,

Set of command instantiations

commandObjects

The invariant is used to specify the relationships between the class instantiations (invokers, receivers and commands) in this pattern. What is particular about this invariant is the use of B sequences to model stacks. Because command are placed in undo and redo stacks, the ordering of the collection that the invoker refers to is important which mandates the use of sequences.

INVARIANT $commandObjects \subseteq OBJECT \wedge$

$commandReceivers \in commandObjects \rightarrow receiverObjects \wedge$

$invokerCommand \in invokerObjects \nrightarrow commandObjects \wedge$

$invokerUndoCommands \in invokerObjects \nrightarrow \mathsf{seq}\ (\ commandObjects\ ) \wedge$

$invokerRedoCommands \in invokerObjects \nrightarrow \mathsf{seq}\ (\ commandObjects\ )$

INITIALISATION *invokerCommand* ,

invokerRedoCommands ,

invokerUndoCommands ,

commandReceivers ,

$commandObjects := \{\}\ ,\ \{\}\ ,\ \{\}\ ,\ \{\}\ ,\ \{\}$

OPERATIONS

Operations within this machine are divided into three segments.

- Invoker operations these always take an invoker object as the first argument and are part of the invoker interface.
- Command operations these take a command object as the first argument and are part of the command interface.

Invoker Operations

ExecuteCommand part of the invoker interface and specifies that the current command that the invoker is referencing should be executed on the receiver by calling *ReceiverAction* in the receiver machine. The command is then added to the undo stack for its invoker

ExecuteCommand (*ivkr*) $\widehat{=}$

PRE

$ivkr \in \mathsf{dom}$ (*invokerCommand*)

THEN

ReceiverAction (*commandReceivers* (*invokerCommand* (*ivkr*))) ||

Everytime we execute a comand we need to clear the Redo queue

invokerRedoCommands (*ivkr*) := [ ] ||

The command is added to the undo stack in the following B segment

SELECT

$ivkr \in \mathsf{dom}$ (*invokerUndoCommands*)

THEN

Pushing a comand onto a stack by prepending to the sequence

invokerUndoCommands (*ivkr*) := *invokerCommand* (*ivkr*) $\rightarrow$ *invokerUndoCommands* (*ivkr*)

ELSE

If a sequence for that invoker is non-existant, then specify the creationg of a new sequence

invokerUndoCommands (*ivkr*) := [*invokerCommand* (*ivkr*)]

END

END ;

Undo is a specification of how to undo an operation. A command must be popped from the undo stack and executed by the *ReceiverUndoAction* in the receiver machine. This command is then pushd onto the redo stack for the invoker that was given as the argument.

Undo (*ivkr*) $\widehat{=}$

PRE

$ivkr \in \mathsf{dom}$ (*invokerUndoCommands*) $\wedge$

size (*invokerUndoCommands* (*ivkr*)) > 0

THEN

Specifying non-deterministically how to pop the first command from the undo stack. This is done by specifying that the command required is the first command in the undo sequence

ANY *firstCmd*

WHERE *firstCmd* ∈ ran (*invokerUndoCommands* (*ivkr*)) ∧

firstCmd = *invokerUndoCommands* (*ivkr*) (*1*) ∧

firstCmd ∈ dom (*commandReceivers*)

THEN

After acquiringg the first command, then the sequence is re-assigned to its tail

invokerUndoCommands (*ivkr*) := tail (*invokerUndoCommands* (*ivkr*)) ||

Calling the correct receiver to carry out undo using the command object

ReceiverUndoAction (*commandReceivers* (*firstCmd*)) ||

Push undo action to the Redo LIFO

SELECT

ivkr ∈ dom (*invokerRedoCommands*)

THEN

invokerRedoCommands (*ivkr*) := *firstCmd* → *invokerRedoCommands* (*ivkr*)

ELSE

invokerRedoCommands (*ivkr*) := [*firstCmd*]

END

END

END ;

Redo is the reversal of the *Undo* operation and is specified exactly in exactly the same style but the command object is popped from the redo queue, processed using a receiver and then pushed onto the undo queue.

Redo (*ivkr*) ≙

PRE

ivkr ∈ dom (*invokerRedoCommands*) ∧

size (*invokerRedoCommands* (*ivkr*)) > *0*

THEN

ANY *firstCmd*

WHERE *firstCmd* ∈ ran (*invokerRedoCommands* (*ivkr*)) ∧

firstCmd = *invokerRedoCommands* (*ivkr*) (*1*) ∧

firstCmd ∈ dom (*commandReceivers*)

THEN

invokerRedoCommands (*ivkr*) := tail (*invokerRedoCommands* (*ivkr*)) ||

ReceiverAction (*commandReceivers* (*firstCmd*)) ||

SELECT

ivkr ∈ dom (*invokerUndoCommands*)

THEN

invokerUndoCommands (*ivkr*) := *firstCmd* → *invokerUndoCommands* (*ivkr*)

ELSE

invokerUndoCommands (*ivkr*) := [*firstCmd*]

END

END

END ;

Assigning a command object to an invoker object for execution.

AddCommand (*ivkr* , *cmd*) ≙

PRE

ivkr ∈ *invokerObjects* ∧

cmd ∈ *commandObjects*

THEN

invokerCommand (*ivkr*) := *cmd*

END ;

The Command Operations below take a command object as the first argument and are part of the comand interface

Constructor for a command object

newCommand ⟵ **InstantiateCommand** (*rcvr*) ≙

PRE

commandObjects ≠ *OBJECT* ∧

rcvr ∈ *receiverObjects*

THEN

ANY *cmd*

WHERE

cmd ∈ *OBJECT* ∧

cmd ∉ *commandObjects*

THEN

commandObjects := *commandObjects* ∪ { *cmd* } ||

commandReceivers (*cmd*) := *rcvr* ||

newCommand := *cmd*

END

END ;

SetCommandReceiver assigns a receiver to a command object so the command object will call the correct receivers methods when it is invoked

SetCommandReceiver (*cmd* , *rcvr*) ≙

PRE

cmd ∈ *commandObjects* ∧

rcvr ∈ *receiverObjects*

THEN

commandReceivers (*cmd*) := *rcvr*

END ;

rcvr ⟵ **GetCommandReceiver** (*cmd*) ≙

PRE

cmd ∈ *commandObjects* ∧

cmd ∈ dom (*commandReceivers*)

THEN

$rcvr := commandReceivers\ (\ cmd\ )$

END

END

Analysing the Command Pattern

Like the Iterator B model, the Command B model uses the association-machine pattern to model the relationship between the invokers and receivers. The association-machine itself is used to model the command objects so that that relationships can be specified between an invoker and its command, and command and the receiver it will call the action() methods. Invoker and Receiver specific behaviour can be specified by the user in the corresponding machines. What is central to the command pattern is modelled in the *CommandPattern* machine and these are the three operations *Execute*, *Undo* and *Redo* which are all given only the invoker as the argument. These operations will cause the current command referred to by the invoker to call the action the receiver it refers to allowing the encapsulation of method calls as is described in the GoF command design pattern.

4.5 Analysis of Findings

Our goal for this chapter has been to examine how an "Object" abstraction can be introduced and used in B. We examined how classes interact in OO can be modelled easily in B, and modelled four design patterns in B using this methodology. However, this process is far from bullet-proof. Not all patterns can or should be modelled using the "Object" abstraction, when B might have a more natural way of expression.

Furthermore, we have attempted to provide abstract specifications of these four design patterns, such that the specifications could conceivably be modified and used in a real, context-ful system. This is possible because the patterns presented all encapsulate some functionality that can be expressed. Many other patterns - the majority of patterns - cannot be modelled this abstractly and still be of use, because they rely too heavily on the problem context. Examples of such patterns include *Adapter* and *Mediator*.

Finally, we have found that divide between OO methodologies and B can cause issues when *specifying* OO design patterns. Some patterns do not make sense in a specification because they are far more relevant to an implementation strategy. Specification in B is not intended to decide an implementation route for the implementer, rather it should guide the implementer in their decisions so that the invariant and preconditions in the specification are held true by the implementation. Most design patterns almost do decide an implementation route for the implementer. In fact, it would be fair to say that some of the specifications listed above are not abstract enough for B specifications, and should be made more flexible. This is likely due to our attempts to *specify* a system *design*. This is somewhat akin to providing implementation details and some code in a UML class diagram. It is not exactly wrong, just not really the best way of doing things.

Chapter 5

Patterns in B and a B centric pattern taxonomy

Patterns exist in many forms, across development paradigms and even across disciplines. The concept of a design pattern was first recognised by the architect Christopher Alexander, who said

> "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice" *[A Pattern Language, Alexander et al, 1977]*

While he was discussing patterns in constructing cities and buildings, when we talk about patterns in computer science, the same could be said. Patterns exist in many different shapes and forms, and they can be categorised based on shared traits. In this chapter we examine how we can categorise the patterns that occur in the world of B, how that compares and contrasts with those that occur in the world of OO, as laid out in *Design Patterns, Elements of Reusable Object-Oriented Software, Gamma et al, 1995.*

5.1 The traditional taxonomy of Object Oriented Design Patterns

Design Patterns, Elements of Reusable Object-Oriented Software categorises patterns into three categories, based on the context and intent of the pattern. These categories are:

- **Creational** - These patterns all describe methods of creating objects for certain situations. Examples of creational patterns include the *Factory Method* and *Singleton* patterns.

- **Structural** - These patterns describe ways to structure your system to acheive a certain result. Examples include the *Adapter*, *Flyweight*, *Composite* and *Facade* patterns.

- **Behavioral** - These patterns describe methods to reproduce commonly required system behaviours. Examples include the *Observer*, *Strategy*,*Iterator* and *Command* patterns.

These are good categories for patterns that exist in the OO world. B, However, is an entirely different development paradigm. B's three phases of development outlined in chapter two do not map directly with the phases of an OO development, such that when working with B, some patterns sometimes shift in emphasis enough to be reclassified. For example the only difference between the *Flyweight* and *Observer* pattern in B is the functional intent and the sharing of Flyweight objects. In many ways, *Flyweight* can be considered a **Creational** pattern in B. Aside from these shifts in emphasis, many patterns can be achieved in entirely different ways that in the OO world. Some patterns are almost features of B. Overall, there are many compelling reasons to consider a new taxonomy of patterns specific to B to aid understanding how patterns manifest themselves in the world of B. These reasons include:

- **Higher levels of abstraction** B specifications are more abstract than a design. Thus some patterns cannot become apparent until later stages of refinement or even implementation. This unavoidably alters the perception of a pattern, its functional emphasis $ala flyweight$ and even its usage.

- **Redundant patterns** It can be said that some cases a pattern is in fact a sign of a dearth of sufficiently powerful programming abstractions. When working with B, some patterns are not needed as the intent behind the pattern is provided by inbuilt features.

Overall, this makes a case for us to consider how a B centric pattern taxonomy might appear, and how we could reclassify patterns in order to more naturally understand how they are modelled in B.

5.2 A B centric Pattern Taxonomy

Patterns in B appear in different shapes and at different stages in development. Some patterns are similar to those that appear in the OO world, others are strangers to the OO world, and yet more are foundational - they don't appear in the OO world as patterns because they are platform features, or they are irrelevant to the OO world. We propose the following categories best capture the different types of patterns in B.

- **Foundational Patterns** provide abstraction concepts. They are the foundation that other patterns build on. Examples are B specific patterns such as *Object* and *Interface* and will be discussed in detail in the following section.

- **Behavioural Patterns** are much the same as for OO design patterns. These patterns describe methods to reproduce commonly required system behaviours. Examples include the *Observer*, *Strategy*,*Iterator* and *Command* patterns.

- **Structural Patterns** are much the same as for OO design patterns. These patterns describe how to compose machines to achieve a certain outcome. Examples include *Bridge*, *Adapter* and *Facade*

- **Implementation Patterns** describe patterns that appear while undertaking the process of Implementation. They describe common ways to implement machines that are specified in a certain manner. The *Object Implementation Pattern* is an example to be discussed in the next section.

- **Invariant Patterns** describe patterns that are mainly specified within the Invariant of a machine. Thus they constrain the state of the machine and hence the behaviour it can support. These patterns are simply targeted usage of predicates to attain an outcome supported by certain operations Examples include the *Singleton* pattern.

The Creational patterns category has not been included, since such patterns are often Invariant patterns supported by operations to acheive a certain constraint on how elements of sets are distributed to the user. It must be mentioned that the above are no longer necessarily design patterns, since they can be seen and used in any phase of a B development. It is clear that more research needs to be done in this area to document B specific patterns. It would also be worth exploring how all of the GoF OO design patterns identified fit into these categories, but that is outside the scope of this thesis.

5.3 Patterns as platform features

Some mention before was made of patterns that are redundant or made significantly easier in B due to features of the platform. In this section we will cover some of those patterns and how they can be achieved in B.

5.3.1 The Singleton Pattern

We have classified the Singleton pattern as an Invariant pattern, because it's intent is achieved almost solely through an Invariant. The Invariant for the Singleton pattern would simply need to express the fact that the object's set must have a cardinality of 1. Once this is specified, operations surrounding that would need to check if the cardinality of the set is 0, and if not, return the only object in that set.

5.3.2 The Bridge Pattern

The Bridge pattern is largely dealt with as part of B. Once a machine is specified, it can be refined and implemented in as infinitely many ways as the developer desires. Thus the intent of Bridge, which is to allow abstract representations of a system to vary independently of the implementations of the system, is satisfied by the mechanisms B already has in place. However, due to the nature of the pattern, we have still classified it as a structural pattern since it relies on the structuring of developments to acheive its goal.

5.3.3 The Visitor Pattern

The Visitor pattern is about separating a data structure from the operations you want to perform on that data structure. This is achieved in the OO world by the use of a functor or function object passed in to an operation to 'visit' the data structure. When using B, this is unnecessary since any function can be passed in as an argument to the operation. Also, B allows lambda abstractions which can be used to aid the developing of Visitors for a data-structure. The Visitor pattern is a truly interesting example of a pattern which changes significantly in the B paradigm, and deserves further research that is outside the scope of this work.

5.3.4 The State Pattern

The State pattern allows an object to alter its behaviour when its internal state changes and is derived from a finite state machine model. B machines are particularly well suited to modelling FSMs because the invariant can be used to constrain the state of the machine. One possible method of doing this is to have a set of states that the machine can enter into and use the invariant to determine whether or not a machine should be in that state. The machines behaviour can then be altered by looking at which state the machine is in using SELECT..WHEN..THEN clauses that specify different behaviours within an operation dependent on the state. In fact, the B specification of the Interface pattern closely approximates the behaviour of the State pattern.

5.4 Examining B specific Patterns

While developing our case studies in the following chapter we began to discover several patterns that occur commonly when developing with B. We have briefly introduced them above when placing them in certain categories. Here we will define and discuss them in more detail.

5.4.1 Foundation Patterns

Modelling Classes in B : The Object Pattern

In chapter four we outlined how classes and objects can be modelled in B. After seeing this pattern re-occur in many of our developments did we come to realise that this was a pattern - one that was obviously not required in the OO world, but can be very useful when working with B.

Dynamic Typing, Polymorphism and Interfaces in B : The Interface Pattern

Many OO patterns require the concept of polymorphism, and still more refer to the concept of a shared interface. While developing the case study which involved usage of the Strategy pattern, it became apparent that we would need some concept of an Interface in B to support the pattern. Further work showed that an Interface can be specified in B in a generic fashion that can easily be reused.

Pattern Structure and Behaviour

In mathematical terms, an interface or a superclass can be represented by a set, in which all implementations resides. By building on the Object pattern, we begin by specifying an interface object. Each interface object maps to a *type*, and using this type we can ascertain object's *implementation identity*.

A corollary to this is that object creation must be managed through the Interface machine in order to correctly maintain the pool of interface objects, ensuring no object can be two objects at once. Furthermore operations are called on the Interface machine, not on the implementation machines. The operations are then *re-delegated* to the implementation machine based on the implementation type of the object. The abstract nature of the interface makes it difficult to express in a generic fashion. As such, we present a simple shape interface, with three implementers : Square, Rectangle and Triangle. The following figure shows an overview of the B machines and the development hierarchy.

A Shape Interface: Requirements

- We create a Shape Interface, representing a generic two dimensional geometric shape.
- The interface will have two methods which could be applied to any shape: Area and Perimeter.
- We will implement this interface for three shapes; Square, Rectangle and Triangle, demonstrating that calling the same Area() and Perimeter() operation for each shape can provide different results according to the particulars of each shape.

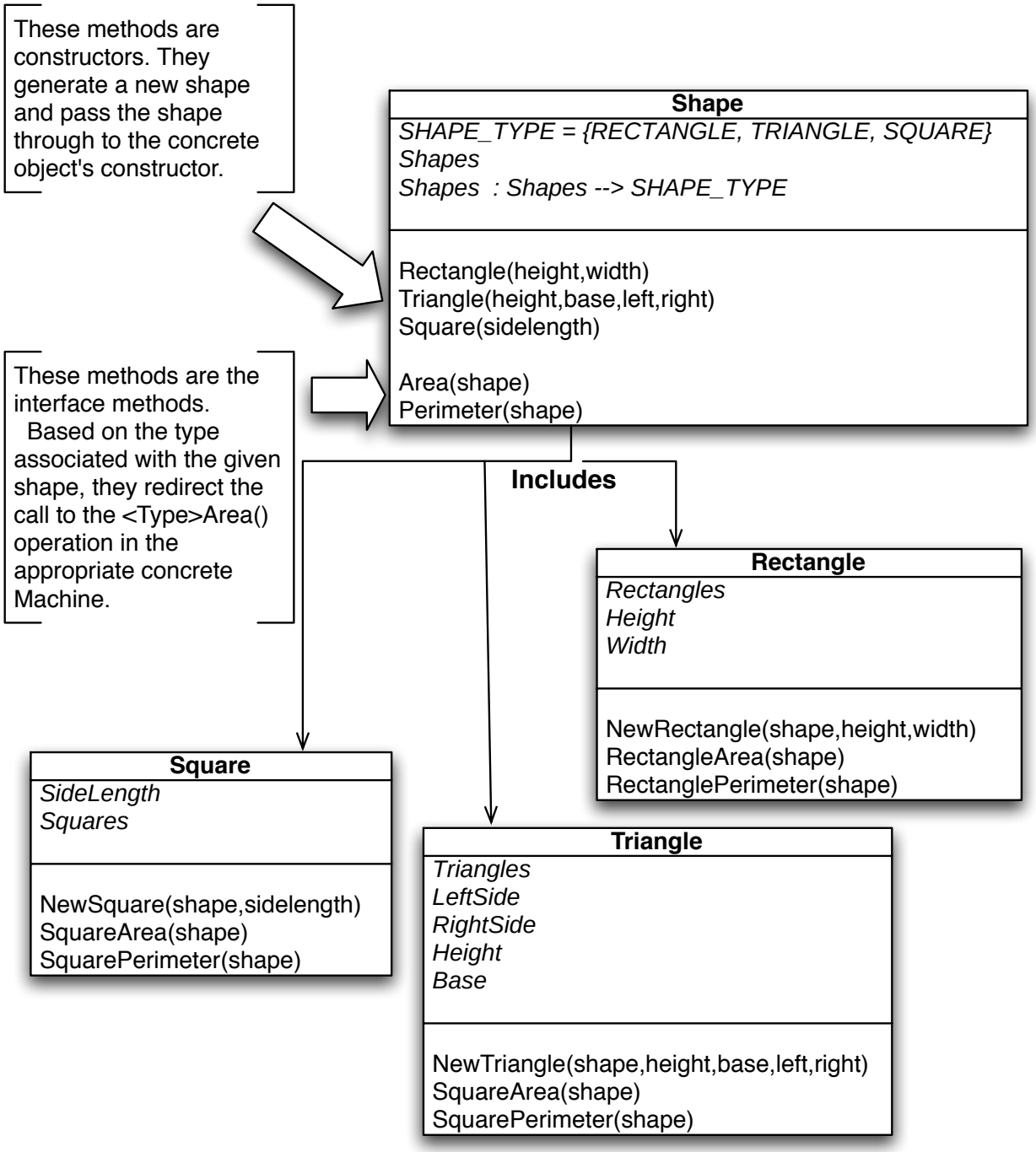


Figure 5.1: Graphical representation of the Interface pattern in B

A Shape Interface: B Specifications

The following B machines are the Interface machine and the Triangle and Rectangle implementations. The Square follows much along the lines of others and adds little to the discussion. However, the full development is listed in the appendix.

The Shape_Interface machine provides constructors for each type that implements the Shape interface, and the operations belonging to the Interface, namely Area and Perimeter.

MACHINE *Shape_Interface*

The Interface machine must include all of the machines that 'implement' the interface, in this case the Shapes Rectangle, Square and Triangle.

SEES *Shape_ctx*

INCLUDES *Shape_Rectangle* , *Shape_Square* , *Shape_Triangle*

Each shape needs to declare its shape type here, to allow us to distinguish what Shape a ShapeID corresponds to.

SETS $SHAPE\_TYPE = \{ SQUARE , RECTANGLE , TRIANGLE \}$

The variable Shapes is the set of all s belonging to instantiated shapes. The variable Shapes maps Shapes to their SHAPE_TYPE.

VARIABLES

Shapes ,

ShapeTypes

Our invariant shows that the set Shapes is in fact the union of all the Shapes: Rectangles, Squares and Triangles, which are managed in the shape's individual machine. We also assert that all the sets of individual shape identifiers are invariantly disjoint, i.e that their mutual intersection is constantly the empty set. This ensures that a Triangle cannot simultaneously be a Rectangle!

INVARIANT

$Shapes \subseteq SHAPE \land$

$ShapeTypes \in Shapes \rightarrow SHAPE\_TYPE \land$

$Shapes = Rectangles \cup Squares \cup Triangles \land$

$Rectangles \cap Squares \cap Triangles = \{\}$

INITIALISATION

$Shapes := \{\} \ \|$

$ShapeTypes := \{\}$

OPERATIONS

The Interface machine has two types of operations. The first type are constructors, which take in the shape's parameters, and allocate a unique ID to the shape. They then call upon the shape's own machine, passing through the unique ID and parameters to actually create the object. This allows us to maintain the SHAPE_TYPE of each shape in accordance with our invariant, while the individual characteristics of the shape are managed by the shape's own machine.

The second type are the operations of the interface itself. We pass in the shape's id that we want to operate on, and the operation delegates the call to a specific shape machine, based on the object's type. In this way we can simulate a B flavoured version of polymorphism a set of shapes can be treated as a set of shapes, yet they can provide different responses when queried.

$sid \longleftarrow$ **Rectangle** ($height$, $width$) $\widehat{=}$

PRE

$height \in \mathbb{N}_1 \ \wedge$

$width \in \mathbb{N}_1$

THEN

ANY rid

WHERE

$rid \in SHAPE - Shapes$

THEN

$sid := rid \ \|$

$Shapes := Shapes \cup \{ rid \} \ \|$

$ShapeTypes (rid) := RECTANGLE \ \|$

$NewRectangle (rid , height , width)$

END

END ;

$sid \longleftarrow$ **Square** ($side$) $\widehat{=}$

PRE

$side \in \mathbb{N}_1$

THEN

ANY rid

WHERE

$rid \in SHAPE - Shapes$

THEN

$sid := rid \ \|$

$Shapes := Shapes \cup \{ \ rid \ \} \ \|$

$ShapeTypes \ (\ rid \ ) := SQUARE \ \|$

$NewSquare \ (\ rid \ , \ side \ )$

END

END ;

$sid \longleftarrow$ **Triangle** ($left$, $right$, $base$, $height$) $\ \widehat{=}$

PRE

$left \in \mathbb{N}_1 \ \wedge$

$right \in \mathbb{N}_1 \ \wedge$

$base \in \mathbb{N}_1 \ \wedge$

$height \in \mathbb{N}_1$

THEN

ANY rid

WHERE

$rid \in SHAPE - Shapes$

THEN

$sid := rid \ \|$

$Shapes := Shapes \cup \{ \ rid \ \} \ \|$

$ShapeTypes \ (\ rid \ ) := TRIANGLE \ \|$

$NewTriangle \ (\ rid \ , \ left \ , \ right \ , \ base \ , \ height \ )$

END

END ;

These are the Interface's Operations, Area() and Perimeter(). Each one will query the shape to find the correct calculation of the particular shape's area.

ans ⟵ **Area** (*sid*) ≙

PRE

sid ∈ *Shapes*

THEN

SELECT *ShapeTypes* (*sid*) = *SQUARE* ∧ *sid* ∈ *Squares* **THEN**

ans ⟵ *SquareArea* (*sid*)

WHEN *ShapeTypes* (*sid*) = *RECTANGLE* ∧ *sid* ∈ *Rectangles* **THEN**

ans ⟵ *RectangleArea* (*sid*)

WHEN *ShapeTypes* (*sid*) = *TRIANGLE* ∧ *sid* ∈ *Triangles* **THEN**

ans ⟵ *TriangleArea* (*sid*)

END

END ;

ans ⟵ **Perimeter** (*sid*) ≙

PRE

sid ∈ *Shapes*

THEN

SELECT *ShapeTypes* (*sid*) = *SQUARE* ∧ *sid* ∈ *Squares* **THEN**

ans ⟵ *SquarePerimeter* (*sid*)

WHEN *ShapeTypes* (*sid*) = *RECTANGLE* ∧ *sid* ∈ *Rectangles* **THEN**

ans ⟵ *RectanglePerimeter* (*sid*)

WHEN *ShapeTypes* (*sid*) = *TRIANGLE* ∧ *sid* ∈ *Triangles* **THEN**

ans ⟵ *TrianglePerimeter* (*sid*)

END

END

END

The Shape_Rectangle machine provides the Shape Interface's operations for a rectangle. It tracks each Rectangle's height and width parameters and uses these to calculate it's area and perimeter.

MACHINE *Shape_Rectangle*

SEES *Shape_ctx*

VARIABLES

Rectangles ,

Height ,

Width

INVARIANT

$Rectangles \subseteq SHAPE \land$

$Height \in Rectangles \rightarrow \mathbb{N}_1 \land$

$Width \in Rectangles \rightarrow \mathbb{N}_1$

INITIALISATION

$Rectangles := \{\} \parallel$

$Height := \{\} \parallel$

$Width := \{\}$

OPERATIONS

The NewRectangle operation instantiates a new rectangle with the height and width provided. The new rectangle's ID is provided by the Shape_Interface machine to keep consistency with the other Shapes.

NewRectangle (*ids* , *ht* , *wd*) $\widehat{=}$

PRE $ids \notin Rectangles \land$

$ids \in SHAPE \land$

$ht \in \mathbb{N}_1 \land$

$wd \in \mathbb{N}_1$

THEN

$Rectangles := Rectangles \cup \{ ids \} \parallel$

$Height (ids) := ht \parallel$

Width (*ids*) := *wd*

END ;

RectangleArea() is called by the Shape_Interface machine's operation Area(). Using a rectangle's height and width attributes, it returns the area of the rectangle. RectanglePerimeter() operates in a similar manner.

ans ⟵ **RectangleArea** (*ids*) ≙

PRE *ids* ∈ *Rectangles*

THEN

ans := *Height* (*ids*) × *Width* (*ids*)

END ;

ans ⟵ **RectanglePerimeter** (*ids*) ≙

PRE *ids* ∈ *Rectangles*

THEN

ans := *Height* (*ids*) × *2* + *Width* (*ids*) × *2*

END

END

The Shape_Triangle machine provides the Shape Interface's operations for a triangle. It is built in the same fashion as the Rectangle, but providing for the characteristics of a triangle.

MACHINE *Shape_Triangle*

SEES *Shape_ctx*

VARIABLES

Triangles ,

SideLeft ,

SideRight ,

Base ,

Vertical

INVARIANT

$Triangles \subseteq SHAPE \ \wedge$

$SideLeft \in Triangles \rightarrow \mathbb{N}_1 \ \wedge$

$SideRight \in Triangles \rightarrow \mathbb{N}_1 \ \wedge$

$Base \in Triangles \rightarrow \mathbb{N}_1 \ \wedge$

$Vertical \in Triangles \rightarrow \mathbb{N}_1$

INITIALISATION

$Triangles := \{\} \ \|$

$SideLeft := \{\} \ \|$

$SideRight := \{\} \ \|$

$Base := \{\} \ \|$

$Vertical := \{\}$

OPERATIONS

NewTriangle (*ids* , *sideL* , *sideR* , *bs* , *ht*) $\widehat{=}$

PRE $ids \notin Triangles \ \wedge$

$ids \in SHAPE \ \wedge$

$sideL \in \mathbb{N}_1 \ \wedge$

$sideR \in \mathbb{N}_1 \ \wedge$

$bs \in \mathbb{N}_1 \ \wedge$

$ht \in \mathbb{N}_1$

THEN

$Triangles := Triangles \cup \{ \ ids \ \} \ \parallel$

$SideLeft \ (\ ids \ ) := sideL \ \parallel$

$SideRight \ (\ ids \ ) := sideR \ \parallel$

$Base \ (\ ids \ ) := bs \ \parallel$

$Vertical \ (\ ids \ ) := ht$

END ;

$ans \longleftarrow$ **TriangleArea** (ids) $\widehat{=}$

PRE $ids \in Triangles$

THEN

$ans := Base \ (\ ids \ ) \times Vertical \ (\ ids \ ) \ / \ 2$

END ;

$ans \longleftarrow$ **TrianglePerimeter** (ids) $\widehat{=}$

PRE $ids \in Triangles$

THEN

$ans := SideLeft \ (\ ids \ ) + SideRight \ (\ ids \ ) + Base \ (\ ids \ )$

END

END

Discussion

One difficulty presented by this pattern was that we needed a way to model calling a single method DoSomething, while having the implementation of that method changeable on the fly, to a faster DoSomething or a more memory efficient DoSomething. This resulted in the adopted approach - the "Interface" methods simply query the type of the object and redirect the Operation call to a concrete, implemented machine, and returns the answer. In this manner, the operations in the implementing machines do not need to name these methods in the same way. Instead the client calls the Interface machine, and the Interface decides how to handle it.

Another issue related to the instantiation of "objects". There is a need to manage all new objects centrally to ensure that an object is typed correctly and that no object can have two concrete types simultaneously, which is undesirable in our current situation. This means that the B machine that implements the interface cannot assign their own objects, but that this object must be handed down from a central location - the interface machine, and the concrete machine then uses this object to map its typespecific attributes.

Pattern consequences

The interface pattern yields some interesting consequences.

- A system API is significantly easier to develop when the Interface pattern is used. This is because the number of userfacing operations is reduced since each concrete type's operations are hidden behind the Interface machine. This helps reduce complexity for the API developer since protected operations for each type are unneeded.
- Creating a new type to implement an interface requires changes to the Interface machine, including "wiring in" the new type's information. However, the good news is that such a change is clearly a straight forward process that could be implemented in a tool to help in the exercise.
- Since we are using B, we now have the ability to make stronger logical assertions about the relationship between types that implement an interface.
- We are still left with the same problem that exists in the OO paradigm there is no guarantee that an operation fulfils its behavioural contract as intended by the creator of the interface.
- We are unable to enforce that an implementing machine implements all of the interface's operations; that these operation's preconditions are the same $in fact they cannot be the same$; that the operations have the same signature across each implemented type. The lattermost is largely unnecessary since the concrete type's operations remain unexposed.

- It is obvious that this pattern would benefit from a tool that can help a user enforce some rules across all machines that implement an interface.

- While named the Interface pattern, this pattern shares similarities with Object Oriented inheritance. This is most obvious in the relationship between concrete types and the abstract types, since the concrete type is a subset of the abstract type. Added methods could be promoted through the interface where required, yielding further specialisations of the object.

5.4.2 An Implementation Pattern

Outside the realm of OO, we believe that a discussion on B centric patterns would be incomplete without a pattern to provide users a template and a process to implement their B specifications, specifically implementation of B Class-specifications presented in the previous chapter. If users are able to repeat-ably use the B class specification process to derive B machines representing their system from an OO abstraction, then it follows that there should be a process for the user to follow for implementing that B specification of classes and class associations.

This section will present one such process for implementation by demonstrating how the *Class* specification machine in Chapter 4 is implemented. Because *Class* specifications are relatively concrete, our process enables the omission of the refinement process in B. If the specification were more abstract however, then the user may need to incrementally refine the specification before implementing it, a discussion of refinement remains outside the scope of this thesis as we are primarily concerned with developing patterns or repeatable processes for direct implementations of specifications.

Implementing an Object using the B: Object Implementation

Implementation machines of B specifications are fully encapsulated and are unable to access the state of any other implementations. They can only interact with other machines via the mechanism of importing specifications which provides the interfaces to their own implementations. This is because implementations must contain a set of Operation signatures that are congruent with the Operation signatures of the specification. Furthermore, implementations do not have any state of their own.

Bearing these constraints in mind, one might ask how we intend to implement the variables within a specification if an implementation does not have any state of its own. This is done through importing B SLIB machines that represent data structures, such machines model arrays, functions, sets and have their own corresponding implementation which is hidden from the user. These machines have their variables unified with the variables of the user specification in the invariant of the implementation to provide the facility to prove correctness. The implementations 'state' is then changed by calling the operations within those SLIB machines that are implementations of the variables. The following diagram shows how specifications can be made to map to their

B implementations.

The following is a fully annotated B implementation of the *Class* specification describing how to implement a B specification model of a OO Class.

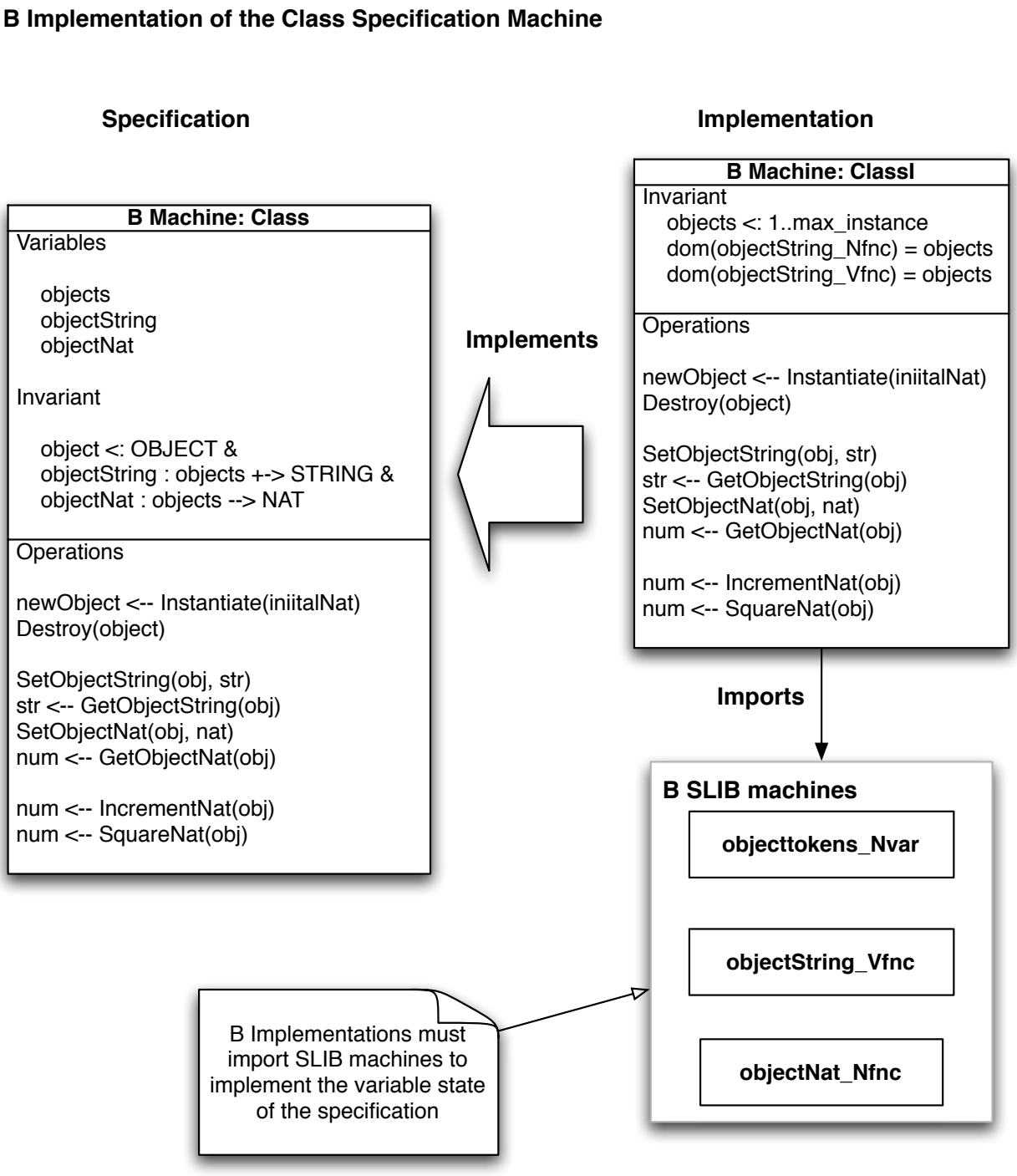


Figure 5.2: Implementing a B specification derived from a Class

IMPLEMENTATION *ClassI*

ClassI is an implementation of the *Class* specification machine presented in Chapter 4 demonstrating how to model classes in B. This generic implementation will show how to implement a specification that follows the B class model and forms the basis for a repeatable pattern/process to implement B specification models of Object Oriented classes.

Implementation is a special case of refinement where the implementing machine does not have any variables to store state. Instead it must import SLIB specification machines from the B library that are used to model data structures such as arrays, sets, sequences etc and uses those to store the state of the variables. The variables from the *specification* must be unified with the variables from the imported SLIB machines used to implement those variables to be able to prove that the implementation matches the specification.

REFINES clause states which machine we are implementing, in this case the *Class* machine

REFINES *Class*

The SEES composition mechanism is still available in B implementations, so we have access to certain data types, however, these seen machines also need to have a corresponding implementation. *objectString_str_ctx* is used in substitution of the *String_TYPE* machine in the specification.

SEES *Class_CTX* , *Bool_TYPE* , *objectString_str_ctx* , *Int_TYPE*

IMPORTS

objecttokens_Nvar is a enum variable machine that allows us to keep track of the value of the highest pointer that has been assigned to an object.

objecttokens_Nvar (*max_instance*) ,

freepointers keeps a track of all the unused pointers in the system. We do not need to track the assinged pointers in the system as that is up to the user

freepointers_set (*OBJECT* , *max_instance*) ,

objfree_Nvar is used to store the index of the first free placeholder in the array so we can recycle 'pointer addresses' when objects are destroyed

objfree_Nvar (*max_instance*) ,

objectNat_Nfnc is a function machine that maps object pointers to their objectNat attribute, Function machines used to map objects to attributes in the implementation process.

objectNat_Nfnc (*2147483646* , *max_instance*) ,

objectString_Vfnc is a function machine that maps object pointer to their objectString attribute. Vfnc machines allow pointer to map to a deferred type given as a parameter. In this case we have passed in the *STRING* deferred set from the String_TYPE machine.

objectString_Vfnc (*objectString_STROBJ* , *max_instance*)

The *PROPERTIES* clause in an implementation can be used to make deferred sets concrete. Below, the deferred set of *OBJECTS* is made to equal the set of integers rom 0 to max_instance provided as a parameter at the top. This is akin to *OBJECTS* being a set of 'pointers'.

PROPERTIES $OBJECT = 0 \,..\, max\_instance$

The invariant in the implementation is used for drawing a relationship between the imported implementation machines which hold the state of the implementation with the variables from the specification machine.

INVARIANT

$objects \subseteq 1 \,..\, max\_instance \wedge$

$objects \cup freepointers\_sset = 1 \,..\, objecttokens\_Nvar \wedge$

$objects \cap freepointers\_sset = \{\} \wedge$

Creating a relationship between the objectNat variable and the objectNat_Nfnc machine

dom (*objectNat_Nfnc*) = *objects* $\wedge$

Creating a relationship between the objectString variable and the object_Vfnc machine

dom (*objectString_Vfnc*) = *objects*

Operations in implementation must match the interface in the specification and implement the specification operations. As shown, all 'state' changes are made by calling the operations from the imported machine to change that machines state.

OPERATIONS

Constructor

newObject ⟵ **Instantiate** (*initInt*) ≙

VAR *bb* , *newfree* **IN**

Test to see if we have any free pointers

bb ⟵ *freepointers_EMP_SET* **;**

If there are no free pointers left then allocate more 'memory' and return the pointer to the new allocation

IF *bb* = *TRUE* **THEN**

objecttokens_INC_NVAR **;**

newfree ⟵ *objecttokens_VAL_NVAR*

ELSE

If there are free pointers, then return a random pointer

newfree ⟵ *freepointers_ANY_SET* **;**

freepointers_RMV_SET (*newfree*)

END **;**

objectNat_STO_NFNC (*newfree* , *initInt*) **;**

newObject := *newfree*

END **;**

Destructor

Destroy (*obj*) ≙

VAR *bb1* , *bb2* **IN**

bb1 ⟵ *freepointers_MBR_SET* (*obj*) **;**

bb2 ⟵ *objecttokens_GEQ_NVAR* (*obj*) **;**

IF *bb1* = *FALSE* ∧ *bb2* = *TRUE* **THEN**

freepointers_ENT_SET (*obj*)

END **;**

objectNat_RMV_NFNC (*obj*) **;**

objectString_RMV_FNC (*obj*)

END ;

Accessors and Mutators for both attributes

SetObjectNat (*obj* , *num*) $\widehat{=}$

VAR *bb* **IN**

bb $\longleftarrow$ *objectNat_DEF_NFNC* (*obj*) ;

IF *bb* = *TRUE* **THEN**

objectNat_STO_NFNC (*obj* , *num*)

END

END ;

num $\longleftarrow$ **GetObjectNat** (*obj*) $\widehat{=}$

VAR *bb* , *vv* **IN**

bb $\longleftarrow$ *objectNat_DEF_NFNC* (*obj*) ;

IF *bb* = *TRUE* **THEN**

vv $\longleftarrow$ *objectNat_VAL_NFNC* (*obj*) ;

num := *vv*

END

END ;

DEFINITIONS *max_instance* $\widehat{=}$ *2147483646*

IMPORTS is a composition mechanism that gives this implementation access to the imported machines operations. If the imported machine is implemented as well as is the case with the B SLIB machines then the implementation can have C-code generated for it. In this example, machines are imported to implement the set of instantiations as well as implementing the objectString attribute and the objectNat attribute.

END

Analysis of Implementation Pattern

A process has been devised to take B specifications of classes through to implementation. This provides the framework for which specifications of the GoF pattern templates presented in chapter 4 can be be implemented in B. The idea being that the user is presented with both an specification template of a pattern as well as the corresponding implementation template. Once they have modified the specification template to suit their requirements, then a modification of the implementation template can be made to produce implementations matching the specification which models the system. A case study of this implementation is presented in Chapter 6.

Chapter 6

Applying a pattern based approach to developing systems in B

Using the research findings from the previous two chapters, the following section aims to demonstrate how to apply the processes and patterns that have been developed to solving 'real-world' problems in B. This will be performed through a number of **casestudies** on simple systems that will have their requirements listed followed by a class diagram to present a design using a pattern that will satisfy those requirements or solve that particular problem. Following on from this, we will use the patterns and processes in chapters 4 and 5 to produce a B specification of the solution that would enable an implementation using B.

The case studies presented will include:

1. **Share Watcher** a system to model investors watching share prices to showcase the observer pattern.

2. **Calculator** a simple arithmetic calculator that supports unlimited undo and redo to showcase the command pattern. This case-study will also demonstrate how to use the object implementation pattern to implement the individual specification machines.

3. **Chess Game** models a game of chess between two AI players to showcase the strategy pattern in action.

4. **Spreadsheet Engine** a spreadsheet application back-end to demonstrate composition of different design patterns, in this case the observer and the command pattern.

It is hoped that these case-studies will provide the reader with insight into how B combined with the design pattern process makes it easier to solve significant problems in B whilst keeping formality as a goal.

6.1 Case study: A Share Price watching system

6.1.1 System requirements

The Share Price watching system must model a group of investors watching the prices of shares and automatically making decisions based on the price movements of those shares. Each investor watches only one share but one share can have any number of investors watching it.

The system must provide a facility to create new shares which stores a unique 3-letter share symbol for identification and its price in the form of an integer. The share class must also have a method for updating its price when the price on the share-market changes.

Investors must have a name, a unique identifier, a reference to the share they are watching and also a status which is either one of BUY, SELL or HOLD. An investor must be able to specify the minimum price for which they they want to sell a share at and a maximum price for which they want to buy a share at. If the share price is between these two thresholds, then the status for that investor is set to HOLD, if the share price is above the sell threshold then the investor should sell, and if the share price is below the buy threshold, the investor should buy.

The system does not model the process of trading shares.

6.1.2 Pattern usage: Observer

An observer pattern is used to solve this problem because when the share price is updated, a notification should be sent to all investors watching that share to indicate a change in price has occurred and the investors should update their statuses accordingly.

An object-oriented class diagram of this is shown in diagram 6.1 with two classes investor and share. Abstract observer and subject classes as specified in the GoF design pattern have been omitted for brevity. Each share references a collection of investors to send notifications but a investor only references one share to do its status update when it receives its notification.

To model the same system using B with the Observer pattern template, three machines will be required. The first machine *Investor* is a specification of the Investor class. This *Investor* machine is derived from the *Observer* machine presented previously. Its interface will allow for the construction and destruction of investor objects as well as:

- Changing the buy and sell price thresholds
- Getting the current status of the investor.
- Updating the status of the investor by giving the investor the new share price for the share they are observing.
- Setting the share that the investor is watching.

A *Share* machine is used to model the Share class. It's interface contains operations to construct and destroy share objects and set the price of a share.

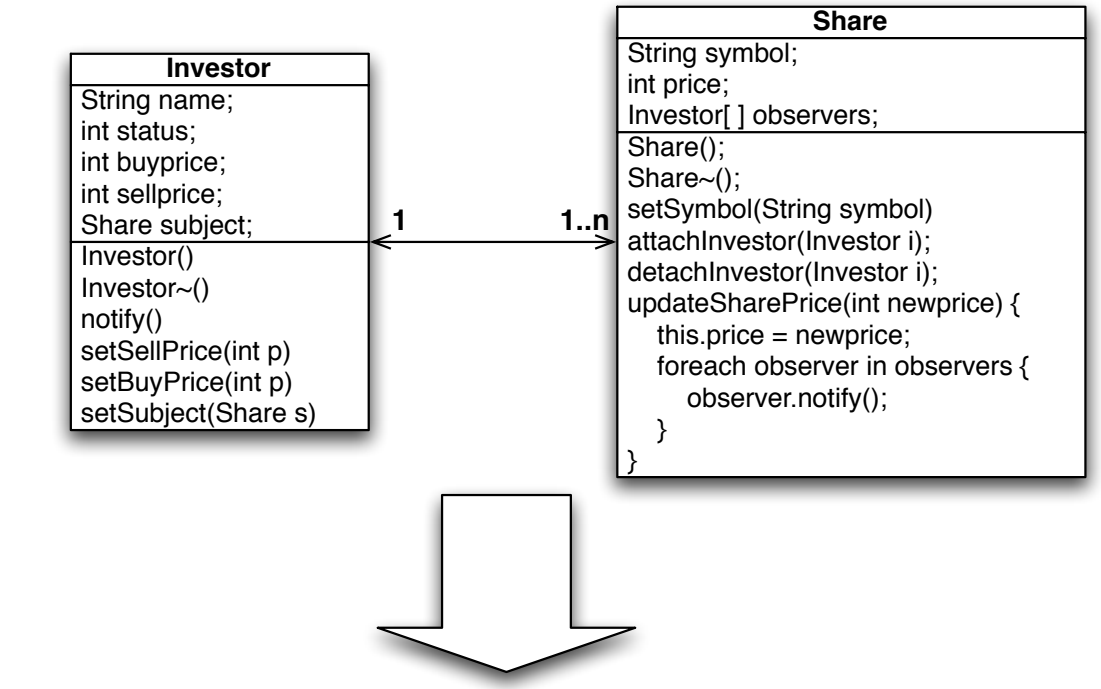
A *ShareWatcher* machine is derived from the *ObserverPattern* machine and specifies the relationship between the shares and investors in its invariant with each share referring to a set of investors for which it will send notifications to. The interface of the *ShareWatcher* machine includes operations to:

- Attach an investor to a share
- Detach an investor for a share
- Update a shares price and in the same operation, notify the shares as required.

6.1.3 Specification

A fully annotated B-specification of the ShareWatcher system is provided below. Only the *Investor* machine showing the multiple update function and the *ShareWatcher* machine (the ObserverPattern machine) are shown here. Please refer to the appendix for the specification of the *Share* machine which uses the B class modelling process to model the Share class.

Class Diagram of the Share Watcher system using the Observer Pattern



B machine structure for the Share Watcher system using the Observer Pattern

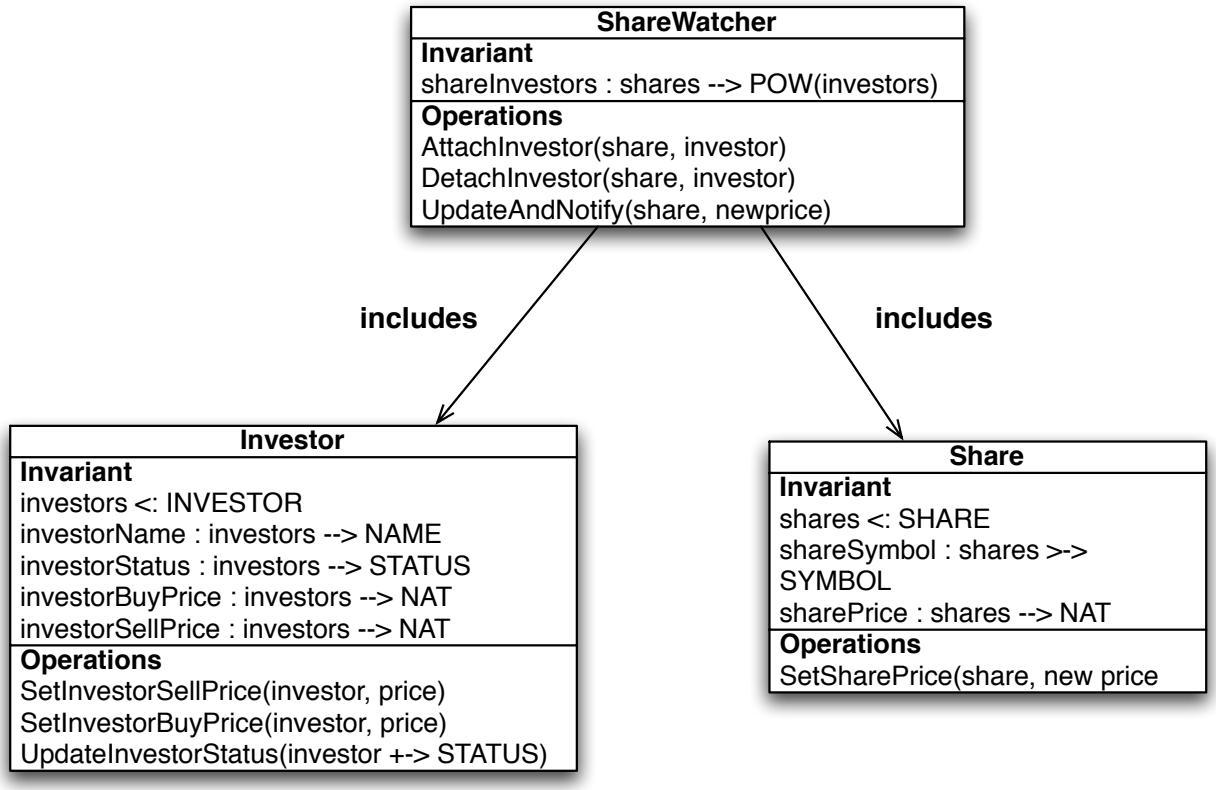


Figure 6.1: Comparing Class Diagram and B machine structure for Share Watcher System

MACHINE *Investor*

Investor machine is a representation of the investor class inside this Observer Pattern case study. The investor is the observer in the pattern and this machine models all the investor specific functions that are required.

SEES *ShareWatcher_CTX*

Each investor object will have attributes to store the following information

- *investorName* is a form of human readable identification for the investor object
- *investorStatus* is an indicator of whether the investor should be buying, selling or holding the share that they are watching.
- *investorShare* is a reference to a share so that the pattern machine including this participant will know which shares the investors needs to observe.
- *investorBuyPrice* is the an integer representation of the highest price that the investor is willing to accept for buying the share.
- *investorSellPrice* is the lowest price that the investor is willing to accept for selling that share

VARIABLES *investors* ,
investorName ,
investorStatus ,
investorShare ,
investorBuyPrice ,
investorSellPrice

The invariant in this *investor* machine simply draws relationships between each of the attribute variables and their types in accordance with the process for modelling a class in B. However what B allows that isn't present in OO is the specification of the extra invariants at the bottom which states that any investors buy price must be lower than their sell price for the share they are observing. This invariant must be enforced at all times throughout the machine or else proof obligations will be generated.

INVARIANT $investors \subseteq INVESTOR \wedge$

$investorName \in investors \rightarrow NAME \wedge$

$investorStatus \in investors \rightarrow STATUS \wedge$

$investorShare \in investors \rightarrow SHARE \wedge$

$investorBuyPrice \in investors \rightarrow \mathbb{N} \wedge$

$investorSellPrice \in investors \rightarrow \mathbb{N} \wedge$

$\forall\ ii\ .\ (\ ii \in investors \Rightarrow investorBuyPrice\ (\ ii\ ) \leq investorSellPrice\ (\ ii\ )\ )$

Because all the attribute variables use total functions to map to their values, an assertion that the domain of all these functions will be equal to the set of investors can be made

ASSERTIONS $\mathsf{dom}\ (\ investorName\ ) = investors \wedge$

$\mathsf{dom}\ (\ investorStatus\ ) = investors \wedge$

$\mathsf{dom}\ (\ investorShare\ ) = investors \wedge$

$\mathsf{dom}\ (\ investorBuyPrice\ ) = investors \wedge$

$\mathsf{dom}\ (\ investorSellPrice\ ) = investors$

INITIALISATION $investors\ ,$

$investorStatus\ ,$

$investorName\ ,$

$investorShare\ ,$

$investorBuyPrice\ ,$

$investorSellPrice := \{\}\ ,\ \{\}\ ,\ \{\}\ ,\ \{\}\ ,\ \{\}\ ,\ \{\}$

OPERATIONS

Investor object constructor and destructor methods

$investorID \longleftarrow$ **ConstructInvestor** $(\ name\ ,\ share\ ,\ shareprice\ ,\ buyprice\ ,\ sellprice\ )\ \ \widehat{=}$

PRE

$name \in NAME \wedge$

$investors \neq INVESTOR \wedge$

$share \in SHARE \wedge$

$shareprice \in \mathbb{N} \wedge$

$buyprice \in \mathbb{N} \wedge$

$sellprice \in \mathbb{N} \wedge$

$buyprice \leq sellprice$

THEN

ANY $newinvestor$

WHERE $newinvestor \in INVESTOR - investors$

THEN

$investorID := newinvestor \ \|$

$investors := investors \cup \{ newinvestor \} \ \|$

$investorName (newinvestor) := name \ \|$

$investorShare (newinvestor) := share \ \|$

$investorBuyPrice (newinvestor) := buyprice \ \|$

$investorSellPrice (newinvestor) := sellprice \ \|$

SELECT

$shareprice < buyprice$

THEN

$investorStatus (newinvestor) := BUY$

WHEN

$shareprice > sellprice$

THEN

$investorStatus (newinvestor) := SELL$

ELSE

$investorStatus (newinvestor) := HOLD$

END

END

END ;

DestroyInvestor ($investor$) $\widehat{=}$

PRE

$investor \in investors$

THEN

$investors := investors - \{ investor \} \ \|$

$investorName := \{ investor \} \lhd investorName \ \|$

$investorStatus := \{\ investor\ \} \lhd investorStatus \ \parallel$

$investorShare := \{\ investor\ \} \lhd investorShare \ \parallel$

$investorBuyPrice := \{\ investor\ \} \lhd investorBuyPrice \ \parallel$

$investorSellPrice := \{\ investor\ \} \lhd investorSellPrice$

END ;

UpdateInvestorsStatus is a function that takes in a set of 2-tuples (investors, status) that is required to model Observer pattern, Because all state changes to this machine must be carried out in parallel in the specification, a function that allows multiple investors to be updated at once is required.

UpdateInvestorsStatus ($isFunction$) $\widehat{=}$

PRE

$isFunction \in investors \pfun STATUS$

THEN

$investorStatus := investorStatus \lhd\!\!+ isFunction$

END ;

UpdateInvestorStatus is a function that calculates the new status of an investor based on the new shareprice of the share they are watching which is provided as a parameter. This will be a helper function in the pattern machine that includes this participant machine.

UpdateInvestorStatus ($investor$, $shareprice$) $\widehat{=}$

PRE

$investor \in investors \land$

$shareprice \in \mathbb{N}$

THEN

SELECT

$shareprice < investorBuyPrice\ (\ investor\ )$

THEN

$investorStatus\ (\ investor\ ) := BUY$

WHEN

$shareprice > investorSellPrice\ (\ investor\ )$

THEN

investorStatus (*investor*) := *SELL*

ELSE

investorStatus (*investor*) := *HOLD*

END ;

Standard object accessor and mutator methods

name ⟵ **GetInvestorName** (*investor*) ≙

PRE

investor ∈ dom (*investorName*)

THEN

name := *investorName* (*investor*)

END ;

SetInvestorName (*investor* , *name*) ≙

PRE

investor ∈ *investors* ∧

name ∈ *NAME*

THEN

investorName (*investor*) := *name*

END ;

status ⟵ **GetInvestorStatus** (*investor*) ≙

PRE

investor ∈ dom (*investorStatus*)

THEN

status := *investorStatus* (*investor*)

END ;

SetInvestorStatus (*investor* , *status*) ≙

PRE

investor ∈ *investors* ∧

$status \in STATUS$

THEN

$investorStatus\ (\ investor\ ) := status$

END ;

SetInvestorShare ($investor$, $share$, $shareprice$) $\widehat{=}$

PRE

$investor \in investors \wedge$

$share \in SHARE \wedge$

$shareprice \in \mathbb{N}$

THEN

$investorShare\ (\ investor\ ) := share\ \ \|$

SELECT

$shareprice < investorBuyPrice\ (\ investor\ )$

THEN

$investorStatus\ (\ investor\ ) := BUY$

WHEN

$shareprice > investorSellPrice\ (\ investor\ )$

THEN

$investorStatus\ (\ investor\ ) := SELL$

ELSE

$investorStatus\ (\ investor\ ) := HOLD$

END

END ;

$share \longleftarrow$ **GetInvestorShare** ($investor$) $\widehat{=}$

PRE

$investor \in investors$

THEN

$share := investorShare\ (\ investor\ )$

END ;

SetInvestorBuyPrice ($investor$, $buyprice$) $\widehat{=}$

PRE

$investor \in investors \wedge$

$buyprice \in \mathbb{N} \wedge$

$buyprice \leq investorSellPrice\ (\ investor\ )$

THEN

$investorBuyPrice\ (\ investor\ ) := buyprice$

END ;

$price \longleftarrow$ **GetInvestorBuyPrice** ($investor$) $\widehat{=}$

PRE

$investor \in investors$

THEN

$price := investorBuyPrice\ (\ investor\ )$

END ;

SetInvestorSellPrice ($investor$, $sellprice$) $\widehat{=}$

PRE

$investor \in investors \wedge$

$sellprice \in \mathbb{N} \wedge$

$sellprice \geq investorBuyPrice\ (\ investor\ )$

THEN

$investorSellPrice\ (\ investor\ ) := sellprice$

END ;

$price \longleftarrow$ **GetInvestorSellPrice** ($investor$) $\widehat{=}$

PRE

$investor \in investors$

THEN

$price := investorSellPrice\ (\ investor\ )$

END

END

MACHINE *ShareWatcher*

ShareWatcher is the 'pattern' machine within the Observer pattern case study. The machine itself does not model either the observer or the subject but rather is the mechanism by which the observer and subject classes can refer to each other. The *ShareWatcher* is used to model the intent of the Observer pattern which is to ensure that each share can be observed by

SEES *ShareWatcher_CTX*

ShareWatcher needs to include *Investor* and *Share* to have access to the operations of those machines to alter their state. *ShareWatcher* also needs to be access the variables of those machines and use it within the invariant to specify a one-to-many relationship.

INCLUDES

Share ,

Investor

Only one variable is required in *ShareWatcher* - we use *shareInvestors* to map each share to the set of investors that are viewing it.

VARIABLES

shareInvestors

To specify an optional one-to-many relationship between shares and investors, we use a partial function from shares to a powerset of investors with the further condition that investors cannot be observing multiple shares. This is specified by a predicate that states that all sets of investors watching each share must be disjoint.

INVARIANT

$shareInvestors \in shares \pfun \mathbb{P}\ (\ investors\ )$

OPERATIONS

AddInvestorToShare adds an investor to a shares' observing list so that when the share is updated, the investor is also updated as well.

AddInvestorToShare (*investor* , *share*) $\widehat{=}$

PRE

$investor \in investors \wedge$

$share \in shares \wedge$

$\forall\ ii\ .\ (\ ii \in \mathsf{ran}\ (\ shareInvestors\ )\ \Rightarrow\ investor \notin ii\ )$

THEN

IF $share \in \mathsf{dom}\ (\ shareInvestors\ )$

THEN

$shareInvestors\ (\ share\ ) := shareInvestors\ (\ share\ ) \cup \{\ investor\ \}$

ELSE

$shareInvestors\ (\ share\ ) := \{\ investor\ \}$

END $\|$

SetInvestorShare (*investor* , *share* , *sharePrice* (*share*))

END ;

RemoveInvestorFromShare allows a investor to stop receiving notifications and updating its status when the share it is observing is updated.

RemoveInvestorFromShare (*investor* , *share*) $\widehat{=}$

PRE

$investor \in INVESTOR \wedge$

$share \in \mathsf{dom}\ (\ shareInvestors\ ) \wedge$

$investor \in shareInvestors\ (\ share\ )$

THEN

Remove the investor from the set of investors that the specified share is watching

$shareInvestors\ (\ share\ ) := shareInvestors\ (\ share\ ) - \{\ investor\ \}$

END ;

UpdateAndNotify is an operation that allows a Shares price to be updated and then specifies that all interested Investors must have their status updated (the 'Notification') by using the *UpdateInvestorsStatus* function from the *Investors* machine.

UpdateAndNotify (*share* , *newprice*) $\widehat{=}$

PRE

$share \in shares \land$

$newprice \in \mathbb{N}$

THEN

SetSharePrice (*share* , *newprice*) $\parallel$

IF $share \in \mathsf{dom}\ (\ shareInvestors\ )$

THEN

Non-deterministically specify a set of (share, status) tuples to be used to overrie the current share status in the share machine using predicates

ANY *isfunction*

WHERE

Specifying that the function will contain a set of (share, status) tuples

$isfunction \in investors \pfun STATUS \land$

Specify that the only investors that need updating are the ones watching this share

$\mathsf{dom}\ (\ isfunction\ ) = shareInvestors\ (\ share\ ) \land$

Use universal quantification to specify that investors should be set to BUY status if the new price is below the buy threshold returned by *investorBuyPrice*

$\forall\ ii\ .\ (\ ii \in \mathsf{dom}\ (\ isfunction\ ) \land$

$newprice < investorBuyPrice\ (\ ii\ ) \Rightarrow isfunction\ (\ ii\ ) = BUY\ ) \land$

Specify that investors should be set to SELL status if the new price is above the sell threshold

$\forall\ kk\ .\ (\ kk \in \mathsf{dom}\ (\ isfunction\ ) \land$

$newprice > investorSellPrice\ (\ kk\ ) \Rightarrow isfunction\ (\ kk\ ) = SELL\ ) \land$

All investors should be set to HOLD status if the new price is between the sell and buy threshold

$\forall\ jj\ .\ (\ jj \in \mathsf{dom}\ (\ isfunction\ ) \land$

$newprice \leq investorSellPrice\ (\ jj\ ) \wedge$

$newprice \geq investorBuyPrice\ (\ jj\ ) \Rightarrow isfunction\ (\ jj\ ) = HOLD\ )$

THEN

$UpdateInvestorsStatus\ (\ isfunction\ )$

END

END

END

END

6.1.4 Case analysis

This case study has demonstrated how to apply the B Observer pattern to an interesting publishsubscribe style problem. Converting the object oriented design to a B model was a very straightforward process given the *ObserverPattern* template. Of special note is how we have also shown it is possible to specify the update of multiple investor objects in the absence of sequential composition. This was done through using a non-deterministic set of (investor, status) tuples to override the investortostatus mapping. The B specification is concerned most with the state of the investors before an update and then showing what the state should be after an update to the share price, not with how to implement this using a loop.

6.2 Case study: A simple calculator with undo/redo

6.2.1 System requirements

The calculator system models a set of users and their calculators which are able to remember their state in a client-server architecture. Users send requests to their calculator to make a calculation and this updates the state of the calculator as well as returns the result to the user. Because of the client-server architecture, requests for calculation need to be encapsulated as objects to be sent. This allows for queuing of requests as well as providing an undo/redo functionality.

The calculators are required to have unlimited redo and undo levels and basic arithmetic operations such as add, minus, divide and multiply. The system must allow for upgrading functionality for the calculator so a larger set of mathematical operations can be carried out.

6.2.2 Pattern usage: Command

To solve the problem of request encapsulation so that undo and redo can be supported as well as a 'client-server' architecture that doesn't couple the User class and the Calculator class, the command design pattern will be used. As well as the User class and Calculator class, this means a Command class is required to encapsulate the function call and its parameters.

The User class is the command invoker in this case and is used to model the the calculator users. It will have a minimal interface to set the name of the user. The calculator class will contain the logic to model an arithmetic calculator including a 'UndoOperation' which given an operator and an operand can reverse the state of the calculator so that previous calculator states are not required to be stored by the command objects in the undo stack.

The Command class models the encapsulation of method calls and stores the operation to be called as well as the parameters for those operations. It also has a reference to a receiver so it knows which receivers methods it must execute. To model the same system using B, the *CommandPattern* template with its three machines will be used.

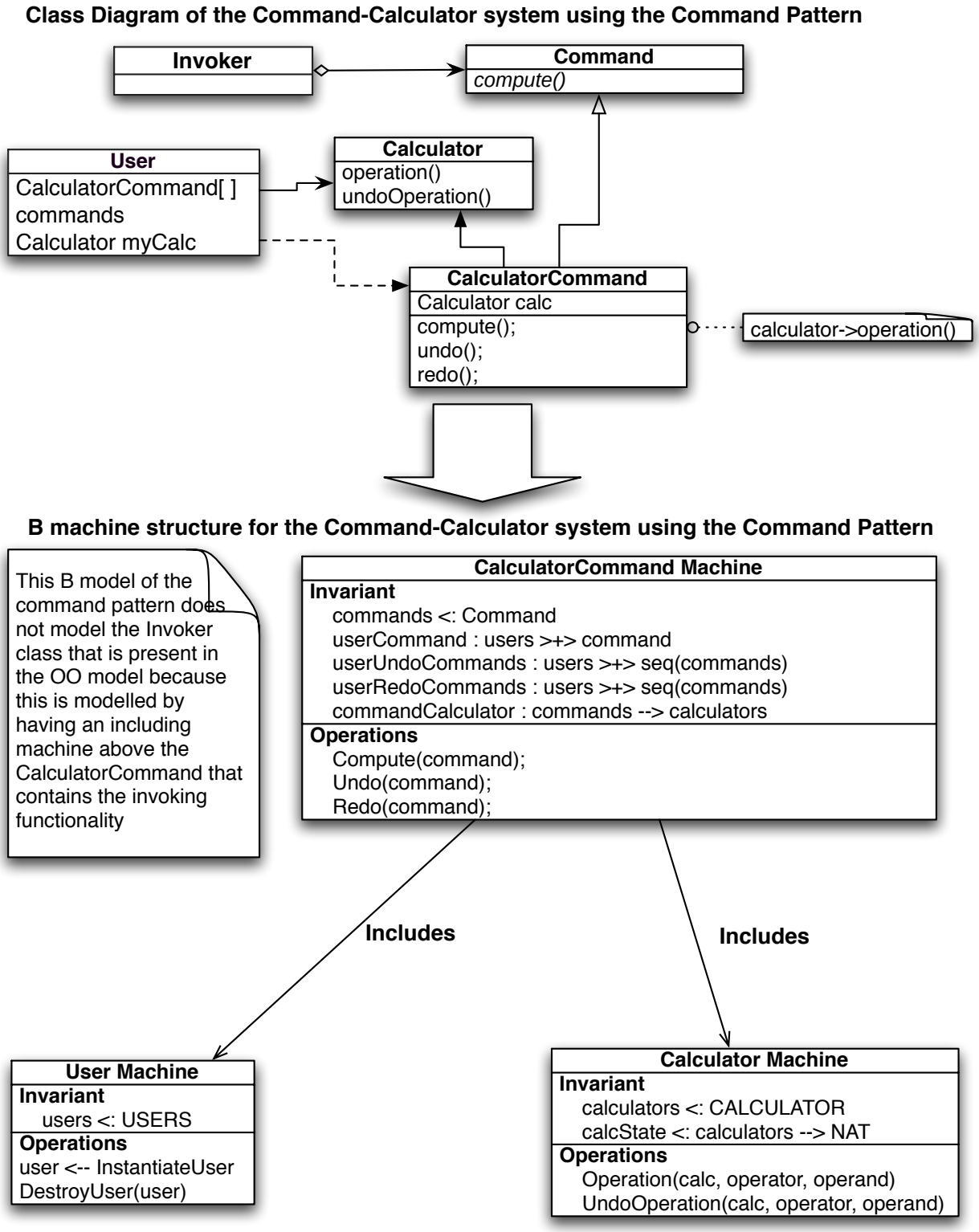


Figure 6.2: Comparing the Class Diagram and B machine structure for 'CommandCalculator'

6.2.3 Formal Specification

A fully annotated B-specification of the CalculatorCommand system is provided below. Only the *Calculator-Command* machine which is derived from the *CommandPattern* machine showing the compute, undo and redo operations and the *Calculator* machine derived from the *Receiver* are shown here. The *Calculator* machine contains the *Operation* and *UndoOperation* operations that are called by the command 'objects'. These operations are equivalent to the *ReceiverAction* and *ReceiverUndoAction* respectively. Please refer to the appendix for the specification of the *CalculatorUser* machine which uses the B class modelling process to model the User class.

MACHINE *Calculator* (*max_instance*)

The *Calculator* machine models a simple arithmetic calculator class that is capable of storing state. Only ADD, SUB, MUL, DIV operations are available, however the user is free to specify more operations for the calculator by appending to the *Operation* operation and also by specifying how to undo this operation by adding the inverse operation to *UndoOperation*.

Operations are carried out by giving a calculator object an operator and an operand for which it will update its own value to the result of the operation

The *max_instance* constraint allows the user to specify what the maximum number of calculator objects can be instantiated at any given time

CONSTRAINTS $max\_instance \in \mathbb{N}_1$

SEES *Class_CTX* , *Calculator_CTX*

Following the standard B representation of a class, there is a variable *calculatorObjects* that stores the set of instantiated calculator objects and a variable *calculatorValues* that is used to map the calculator objects to their current value

VARIABLES *calculatorObjects* , *calculatorValues*

INVARIANT $calculatorObjects \subseteq OBJECT \wedge$
$calculatorValues \in calculatorObjects \rightarrow \mathbb{N}$

INITIALISATION *calculatorObjects* , *calculatorValues* := {} , {}

OPERATIONS

The *Operation* takes in 3 parameters, the calculator object to be used, the operator in the form of a string and the operand. The pre-condition here states that if subtracting, then the operand cannot be greated than the calculator value to prevent it from going into negative numbers which are not available in B Specifications. A second pre-condition for this operation is that if the divide operation is being used, then the operand cannot be 0 to protect from divide-by-zero errors.

result $\longleftarrow$ **Operation** (*calculator* , *operator* , *operand*) $\widehat{=}$

PRE

$calculator \in calculatorObjects \wedge$

$operator \in OPERATOR \wedge$

$operand \in \mathbb{N} \wedge$

$(\ operator = SUB \Rightarrow operand \leq calculatorValues \ (\ calculator \ ) \ ) \wedge$

$(\ operator = DIV \Rightarrow operand \neq 0 \ )$

THEN

SELECT $operator = ADD$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) + operand \ \|$

$result := calculatorValues \ (\ calculator \ ) + operand$

WHEN $operator = SUB$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) - operand \ \|$

$result := calculatorValues \ (\ calculator \ ) - operand$

WHEN $operator = MUL$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) \times operand \ \|$

$result := calculatorValues \ (\ calculator \ ) \times operand$

WHEN $operator = DIV$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) \ / \ operand \ \|$

$result := calculatorValues \ (\ calculator \ ) \ / \ operand$

END

END ;

The *UndoOperation* allows the invoker to undo an operation by providing the parameters for the last operation to return to the calculator to its previous state. In some implementations of the GoF command design pattern, command objects must be capable of storing the state command receiver so that an undo operation involves setting the receivers state to that stored in the command. However, in this case-study of the UndoOperation, the command objects themselves do not store the state, only the operator and the operand so an *UndoOperation* is required to take these parameters and calculate the previous state

$result \longleftarrow$ **UndoOperation** $(\ calculator \ , \ operator \ , \ operand \ ) \ \widehat{=}$

PRE

$calculator \in calculatorObjects \wedge$

$operator \in OPERATOR \wedge$

$operand \in \mathbb{N} \wedge$

$(\ operator = ADD \Rightarrow operand \leq calculatorValues \ (\ calculator \ ) \ ) \wedge$

$(\ operator = MUL \Rightarrow operand \neq 0 \ )$

THEN

SELECT $operator = ADD$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) - operand \ \|$

$result := calculatorValues \ (\ calculator \ ) - operand$

WHEN $operator = SUB$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) + operand \ \|$

$result := calculatorValues \ (\ calculator \ ) + operand$

WHEN $operator = MUL$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) / operand \ \|$

$result := calculatorValues \ (\ calculator \ ) / operand$

WHEN $operator = DIV$ **THEN**

$calculatorValues \ (\ calculator \ ) := calculatorValues \ (\ calculator \ ) \times operand \ \|$

$result := calculatorValues \ (\ calculator \ ) \times operand$

END

END ;

Standard Class Operations

Calculator Object constructor

$newCalculator \longleftarrow$ **InstantiateCalculator** $\widehat{=}$

PRE

$calculatorObjects \neq OBJECT$

THEN

ANY $calc$

WHERE

$calc \in OBJECT \wedge$

$calc \notin calculatorObjects$

THEN

$calculatorObjects := calculatorObjects \cup \{ calc \} \parallel$

$calculatorValues (calc) := 0 \parallel$

$newCalculator := calc$

END

END ;

Calculator Object destructor - in B, to ensure the machine invariant is maintained, the reference to the calculators value must also be removed from *calculatorValues*.

DestroyCalculator ($calc$) $\widehat{=}$

PRE

$calc \in calculatorObjects$

THEN

$calculatorObjects := calculatorObjects - \{ calc \} \parallel$

$calculatorValues := \{ calc \} \lhd calculatorValues$

END ;

Standard set value operation to allow the user to change the value that the calculator is holding

SetCalculatorValue ($calc$, $newValue$) $\widehat{=}$

PRE

$calc \in calculatorObjects \wedge$

$newValue \in \mathbb{N}$

THEN

$calculatorValues (calc) := newValue$

END ;

Standard Get operation to allow the user to retrieve the value the calculator is holding

$val \longleftarrow$ **GetCalculatorValue** ($calc$) $\widehat{=}$

PRE

$calc \in calculatorObjects \wedge$

$calc \in \mathsf{dom}\ (\ calculatorValues\ )$

THEN

$val := calculatorValues\ (\ calc\ )$

END

END

MACHINE *CalculatorCommand* (*max_instance*)

The *CalculatorCommand* machine models the intent of the Command Pattern which is to be able to store commands in objects so that they can be undone and redone. The Command Objects used in the Command Design Pattern and also modelled here so they be passed between the invokers and receivers.

This machine follows the standard B representation of classes for modelling the Command Object as well as adding functionality that is specified in a more orthodox method with respect to the B-Toolkit

CONSTRAINTS $max\_instance \in \mathbb{N}_1$

SEES *Class_CTX* , *Calculator_CTX*

The *CalculatorUser* and *Calculator* machines are included so that full access to the CalculatorUser (Invoker) objects and the Calculator (Receiver) objects is given to this machine and so they can be referenced from this machine to allow for the passing of command objects between them.

INCLUDES *CalculatorUser* (*max_instance*) , *Calculator* (*max_instance*)

From the included machines, the following operations are promoted to this machines interface so that the user has access to them. These promoted operations do not affect of the state of the *CalculatorCommand* machine and therefore do not need to be wrapped.

Notice that actual arithmetic *Operation* and *UndoOperation* that are inside the *Calculator* machine do not get promoted as the user should not have direct access to them. Instead, the user must create a command object, store the parameters inside it and then invoke them to carry out an operation

PROMOTES *InstantiateCalcUser* ,

DestroyUser ,

InstantiateCalculator ,

DestroyCalculator ,

SetCalculatorValue ,

GetCalculatorValue

There are two sets of variables - one set of variables *userComand*, *userRedoCommands*, and *userUndoCommands* will be used to model the relationship between the invoker in this representation of the design pattern

to the command it needs to invoke, the queue of commands that can be undone and the queue of commands that can be redone.

Then there will be the modelling of the actual command class *commandObjects*. Each command will hold a calculator reference *commandCalculator*, an operator *commandOperators* and an operand *commandOperands*. The operator specifies the actual function to be carried out while the operand is the parameter to that function.

VARIABLES

Variables to map *CalculatorUser* objects to the current command they will be invoking and also to queues of undo commands and redo commands

$userCommand$,

$userRedoCommands$,

$userUndoCommands$,

Variables to model the command objects and their attributes

$commandObjects$,

$commandCalculators$,

$commandOperators$,

$commandOperands$

The invariant of this machine is used mainly to state the relationships between the variables. *userUndoCommands* maps each *calcUserObject* to its sequence of undoable commands. *userRedoCommands* maps each *calcUserObject* to its sequence of redoable commands. In the B-specification we can use ordered sequences to represent a stack.

INVARIANT $commandObjects \subseteq OBJECT \wedge$

$commandCalculators \in commandObjects \rightarrow calculatorObjects \wedge$

$userCommand \in calcUserObjects \nrightarrow commandObjects \wedge$

$userUndoCommands \in calcUserObjects \nrightarrow \mathsf{seq}\ (\ commandObjects\ ) \wedge$

$userRedoCommands \in calcUserObjects \nrightarrow \mathsf{seq}\ (\ commandObjects\ ) \wedge$

$commandOperators \in commandObjects \rightarrow OPERATOR \wedge$

$commandOperands \in commandObjects \rightarrow \mathbb{N} \wedge$

$\mathsf{dom}\ (\ userCommand\ ) = \mathsf{dom}\ (\ userUndoCommands\ ) \wedge$

$\mathsf{dom}\ (\ userCommand\ ) = \mathsf{dom}\ (\ userUndoCommands\ )$

INITIALISATION *userCommand* ,

userRedoCommands ,

userUndoCommands ,

commandCalculators ,

commandObjects ,

commandOperators ,

commandOperands := {} , {} , {} , {} , {} , {} , {}

OPERATIONS

The operations of the *CalculatorCommand* class are also split into two sections. There are operations that are for the *CalculatorUser* objects and also for the *CalculatorCommand* objects.

The reason for some of the *CalculatorUser* (invoker) operations being present in this machine and not in the *CalculatorUser* machine is because of the need to associate these with the command objects which are only visible inside this machine

CalculatorUser (Invoker) operations

Compute specifies that the user to invoke the operation that it's currently referencing.

val ⟵ **Compute** (*user*) ≙

PRE

user ∈ dom (*userCommand*)

THEN

Invoke the operation by calling *Operation* on the calculator that the command object is referencing

val ⟵ *Operation* (*commandCalculators* (*userCommand* (*user*)) ,
commandOperators (*userCommand* (*user*)) ,
commandOperands (*userCommand* (*user*))) ‖

After invoking the command, the redo command queue for this invoker needs to be cleared by setting it to the empty sequence

userRedoCommands (*user*) := [ ] ‖

The invoked command needs to be pushed onto the undo stack by prepending it onto the sequence

SELECT

$user \in \mathsf{dom}\ (\ userUndoCommands\ )$

THEN

$userUndoCommands\ (\ user\ ) := userCommand\ (\ user\ ) \rightarrow userUndoCommands\ (\ user\ )$

ELSE

$userUndoCommands\ (\ user\ ) := [\ userCommand\ (\ user\ )\ ]$

END

END ;

Undo is a specification of how to undo an operation. Within the context of this design pattern case study, undo simply states that a command object should be popped from the undo operations stack and invoked using the *UndoOperation* of the Calculator class to return the calculator object to its previous state.

The pre-condition states that the undo stack for the invoker cannot be empty for this operation to succeed

$val \longleftarrow$ **Undo** $(\ user\ ) \ \widehat{=}$

PRE

$user \in \mathsf{dom}\ (\ userUndoCommands\ ) \wedge$

$\mathsf{size}\ (\ userUndoCommands\ (\ user\ )\ ) > 0$

THEN

Use predicates to specifying the command to be the top of the undo stack and that the command actually references a calculator for which it will perform the undo

ANY *firstCmd*

WHERE $firstCmd \in \mathsf{ran}\ (\ userUndoCommands\ (\ user\ )\ ) \wedge$

$firstCmd = userUndoCommands\ (\ user\ )\ (\ 1\ ) \wedge$

$firstCmd \in \mathsf{dom}\ (\ commandCalculators\ )$

THEN

The undo 'stack' is specified to have its top element popped by assigning the undo command sequence to its tail

$userUndoCommands\ (\ user\ ) := \mathsf{tail}\ (\ userUndoCommands\ (\ user\ )\ ) \ \|$

The UndoOperation inside the calculator is invoked here by passing the parameters which are the attributes that the command object is referencing. 'firstCmd' is the actual command object that holds the command that needs to be undone.Using the relationship variables *commandCalculators*, *commandOperators* and *commandOperands* we are able to reference the calculator object, operator and operand for that command object.

val ⟵ *UndoOperation* (*commandCalculators* (*firstCmd*) ,
commandOperators (*firstCmd*) ,
commandOperands (*firstCmd*)) ‖

Push the undone command onto the redo stack for that invoker

SELECT
user ∈ dom (*userRedoCommands*)
THEN
userRedoCommands (*user*) := *firstCmd* → *userRedoCommands* (*user*)
ELSE
userRedoCommands (*user*) := [*firstCmd*]
END
END
END ;

The *Redo* operation is the exact reverse of the *Undo* operation that is specified above. It states that the command object should be popped from the redo operations stack, invoked and then pushed onto the undo operations stack.

val ⟵ **Redo** (*user*) ≙
PRE
user ∈ dom (*userRedoCommands*) ∧
size (*userRedoCommands* (*user*)) > *0*
THEN
ANY *firstCmd*
WHERE *firstCmd* ∈ ran (*userRedoCommands* (*user*)) ∧
firstCmd = *userRedoCommands* (*user*) (*1*) ∧

firstCmd ∈ dom (*commandCalculators*)

THEN

Redo stack has to have its first command popped

userRedoCommands (*user*) := tail (*userRedoCommands* (*user*)) ||

Command from top of redo stack has to be invoked

val ⟵ *Operation* (*commandCalculators* (*firstCmd*) ,

commandOperators (*firstCmd*) ,

commandOperands (*firstCmd*)) ||

The redo command object is pushed onto the undo stack for the invoker

SELECT

user ∈ dom (*userUndoCommands*)

THEN

userUndoCommands (*user*) := *firstCmd* → *userUndoCommands* (*user*)

ELSE

userUndoCommands (*user*) := [*firstCmd*]

END

END

END ;

Attach a command to a user (invoker), to carry out an operation

AddCommand (*user* , *cmd*) ≙

PRE

user ∈ *calcUserObjects* ∧

cmd ∈ *commandObjects*

THEN

userCommand (*user*) := *cmd*

END ;

Standard Command Class Operations

Constructor to create a command object

newCommand ⟵ **InstantiateCommand** (*calc* , *operator* , *operand*) ≙

PRE

commandObjects ≠ *OBJECT* ∧

calc ∈ *calculatorObjects* ∧

operator ∈ *OPERATOR* ∧

operand ∈ ℕ

THEN

ANY *cmd*

WHERE

cmd ∈ *OBJECT* ∧

cmd ∉ *commandObjects*

THEN

commandObjects := *commandObjects* ∪ { *cmd* } ||

commandCalculators (*cmd*) := *calc* ||

commandOperators (*cmd*) := *operator* ||

commandOperands (*cmd*) := *operand* ||

newCommand := *cmd*

END

END ;

Standard mutator operations for the command object to modify and retrieve its attributes.

SetCommandCalculator (*cmd* , *calc*) ≙

PRE

cmd ∈ *commandObjects* ∧

calc ∈ *calculatorObjects*

THEN

commandCalculators (*cmd*) := *calc*

END ;

$calc \longleftarrow$ **GetCommandCalculator** (cmd) $\widehat{=}$

PRE

$cmd \in commandObjects \wedge$

$cmd \in \mathsf{dom}$ ($commandCalculators$)

THEN

$calc := commandCalculators$ (cmd)

END ;

SetCommandOperator (cmd , $operator$) $\widehat{=}$

PRE

$cmd \in commandObjects \wedge$

$operator \in OPERATOR$

THEN

$commandOperators$ (cmd) $:= operator$

END ;

$operator \longleftarrow$ **GetCommandOperator** (cmd) $\widehat{=}$

PRE

$cmd \in commandObjects$

THEN

$operator := commandOperators$ (cmd)

END ;

SetCommandOperand (cmd , $operand$) $\widehat{=}$

PRE

$cmd \in commandObjects \wedge$

$operand \in \mathbb{N}$

THEN

$commandOperands$ (cmd) $:= operand$

END ;

$operand \longleftarrow$ **GetCommandOperand** (cmd) $\widehat{=}$

PRE

$cmd \in commandObjects$

THEN

operand := *commandOperands* (*cmd*)

END

END

6.2.4 A case-study on implementation

In this Command Pattern case study, as well as showing how to model the system using a B specification, the system was also carried through to the implementation stage. Each machine of the Command Calculator specification was implemented using the Object implementation pattern presented in chapter 5 and can be found in the appendix. Because B-implementations are quite complex only the *CalculatorCommandI* implementation will be discussed.

To implement the *CalculatorCommand* specification, are large number of SLIB machines for implementing the variables were required to be imported into *CalculatorCommandI*. The operations of these SLIB machines are then used to implement the functionality specified.

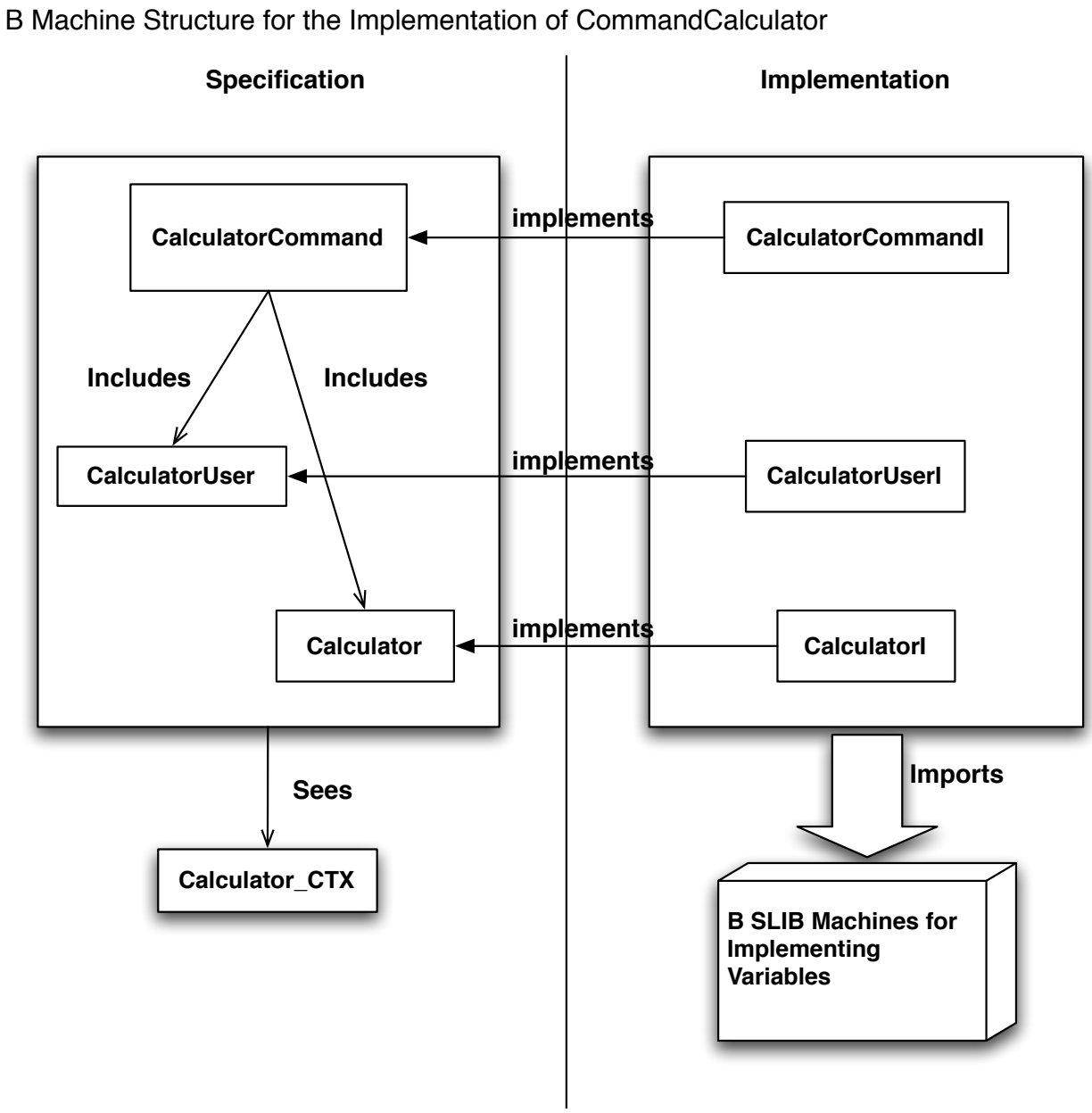


Figure 6.3: B machine structure for 'CommandCalculator' including implementation machines

IMPLEMENTATION *CalculatorCommandI*

CalculatorCommandI is a B implementation of the *CalculatorCommand* machine which contains the the main functionality needed to solve the problem of storing commands inside objects so that they can be stored to be undone and redone. The specifications concreteness means that a refinement was not needed to do an implementation.

REFINES *CalculatorCommand*

SEES *Class_CTX* ,

Calculator_CTX ,

Bool_TYPE ,

Scalar_TYPE ,

userRedoCommands_seq_ctx ,

userUndoCommands_seq_ctx

A large number of the B-Toolkits SLIB machines have been imported to implement the variables that are present in the specification. This is in addition to importing *Calculator* (command receiver) and the *CalculatorUser* machines (command invoker) so that their operations can be accessed or promoted to this implementations interface so that the interfaces of both *CalculatorCommandI* and *CalculatorCommand* match which is a requirement of B implementations.

IMPORTS *Calculator* (*max_instance*) ,

CalculatorUser (*max_instance*) ,

Below are the machines that have been imported from the SLIB to implement the variables that modelled the relationship between user (invoker) objects and their current command object as well as the undo and redo queues of command objects.

userCommand_Vfnc (*OBJECT* , *max_instance*) ,

userUndoCommands_seq_obj is a sequence machine that models a set of sequences and allows for the manipulation of those sequences via its interface

userRedoCommands_seq_obj (*OBJECT* , *max_instance* , *max_instance*) ,

userUndoCommands_seq_obj (*OBJECT* , *max_instance* , *max_instance*) ,

Because of the limitations of accessing the different sequences inside the sequence machine, a *userUndoSeqTokens_Nfnc* machine and its corresponding redo sequence tokens machine was needed to map the user objects to the token that allows access to the correct sequence in the sequence machine

userRedoSeqTokens_Nfnc (*MaxScalar* , *max_instance*) ,

userUndoSeqTokens_Nfnc (*MaxScalar* , *max_instance*) ,

The machines imported below are the ones that are used to implement the commandObjects.

- *commandCalculators_Vfnc* is a function machine that implements the *commandCalculators* variable and allows the each command object to map to its calculator.
- *commandOperators_Vfnc* implements the *commandOperators* variable.
- *commandOperands_Nfnc* implements the *commandOperands* variable.

commandObjects_Nvar (*max_instance*) ,

commandCalculators_Vfnc (*OBJECT* , *max_instance*) ,

commandOperators_Vfnc (*OPERATOR* , *max_instance*) ,

commandOperands_Nfnc (*MaxScalar* , *max_instance*) ,

freeCommandPointers_set (*OBJECT* , *max_instance*)

PROMOTES *InstantiateCalcUser* ,

DestroyUser ,

InstantiateCalculator ,

DestroyCalculator ,

SetCalculatorValue ,

GetCalculatorValue

To implement deferred sets, the deferred set *OBJECT* has been made to equal a set of integers from 1 to max_instance providing us with *max_instance* number of object pointers

PROPERTIES $OBJECT = 1 \,..\, max\_instance \wedge$

$userRedoCommands\_SEQOBJ = OBJECT \wedge$

$userUndoCommands\_SEQOBJ = OBJECT$

In the implementation invariant of the *CalcuatlorCommandI* machine, we are concerned mainly with unifying the variables from the imported SLIB machines that are used to implement the specification variables with the specification variables themselves this follows the standard Object implementation pattern and allows for the implementation to be proven correct against the specification if all proof obligations are discharged.

INVARIANT $commandObjects \subseteq 1 \,..\, max\_instance \wedge$

$commandObjects \cup freeCommandPointers\_sset = 1 \,..\, commandObjects\_Nvar \wedge$

$commandObjects \cap freeCommandPointers\_sset = \{\} \wedge$

$\mathsf{dom}\ (\ commandCalculators\_Vfnc\ ) = commandObjects \wedge$

$\mathsf{dom}\ (\ commandOperators\_Vfnc\ ) = commandObjects \wedge$

$\mathsf{dom}\ (\ commandOperands\_Nfnc\ ) = commandObjects \wedge$

$\mathsf{dom}\ (\ userCommand\_Vfnc\ ) = \mathsf{dom}\ (\ userCommand\ ) \wedge$

$userRedoCommands\_seqtok = \mathsf{ran}\ (\ userRedoSeqTokens\_Nfnc\ ) \wedge$

$userUndoCommands\_seqtok = \mathsf{ran}\ (\ userUndoSeqTokens\_Nfnc\ ) \wedge$

$\mathsf{dom}\ (\ userRedoCommands\ ) = \mathsf{dom}\ (\ userRedoSeqTokens\_Nfnc\ ) \wedge$

$\mathsf{dom}\ (\ userUndoCommands\ ) = \mathsf{dom}\ (\ userUndoSeqTokens\_Nfnc\ ) \wedge$

$\mathsf{dom}\ (\ userRedoCommands\ ) \subseteq \mathsf{dom}\ (\ userCommand\ ) \wedge$

$\mathsf{dom}\ (\ userUndoCommands\ ) \subseteq \mathsf{dom}\ (\ userCommand\ )$

The operations required in the CalculatorCommandI must match the interface of the CalculatorCommand specification. To correctly model the command pattern with undo and redo, the interface contains the *Compute*, *Undo* and *Redo* operations which are implementations of the specification.

OPERATIONS

For *Compute* to correctly implement the specification, then we needed to implement the function call to the calculator object as well as adding the object to the undo stack and clearing the redo stack

$val \longleftarrow$ **Compute** $(\ user\ ) \ \widehat{=}$

VAR $bb\ ,\ comm\ ,\ calc\ ,\ operator\ ,\ operand\ ,\ seqTok$ **IN**

$bb \longleftarrow userCommand\_DEF\_FNC\ (\ user\ )$ **;**

IF $bb = TRUE$ **THEN**

$comm \longleftarrow userCommand\_VAL\_FNC\ (\ user\ )$ **;**

$calc \longleftarrow commandCalculators\_VAL\_FNC\ (\ comm\ )$ **;**

$operator \longleftarrow commandOperators\_VAL\_FNC\ (\ comm\ )$ **;**

$operand \longleftarrow commandOperands\_VAL\_NFNC\ (\ comm\ )$ **;**

$val \longleftarrow Operation\ (\ calc\ ,\ operator\ ,\ operand\ )$ **;**

Clear the Redo queue for this user or add a new empty queue for this user

```
bb ⟵ userRedoSeqTokens_DEF_NFNC ( user ) ;
IF    bb = TRUE    THEN
   seqTok ⟵ userRedoSeqTokens_VAL_NFNC ( user ) ;
   userRedoCommands_CLR_SEQ_OBJ ( seqTok )
ELSE
```

Need to create a new Sequence and associate that sequence with user

```
   bb , seqTok ⟵ userRedoCommands_CRE_SEQ_OBJ ;
   IF    bb = TRUE    THEN
      userRedoSeqTokens_STO_NFNC ( user , seqTok )
   ELSE
      val := MaxScalar
   END
END    ;
```

Add command to the users undo queue, create an undo queue if there isn't one

```
bb ⟵ userUndoSeqTokens_DEF_NFNC ( user ) ;
IF    bb = TRUE    THEN
   seqTok ⟵ userUndoSeqTokens_VAL_NFNC ( user ) ;
   bb ⟵ userUndoCommands_PSH_SEQ_OBJ ( seqTok , comm )
ELSE
   bb , seqTok ⟵ userUndoCommands_CRE_SEQ_OBJ ;
   IF    bb = TRUE    THEN
      userUndoSeqTokens_STO_NFNC ( user , seqTok ) ;
      bb ⟵ userUndoCommands_PSH_SEQ_OBJ ( seqTok , comm )
   END
END
ELSE
   val := MaxScalar
END
END    ;
```

The Undo operation is very similar to the Compute operation except it pop the command to be executed from the Undo stack, undo the operation by calling *UndoOperation* from the *Calculator* machine, and then place it onto the Redo stack

val ⟵ **Undo** (*user*) ≙

VAR *bb1* , *bb2* , *comm* , *calc* , *operator* , *operand* , *seqTok* **IN**

bb1 ⟵ *userCommand_DEF_FNC* (*user*) ;

IF *bb1* = *TRUE* **THEN**

Get reference to correct Sequence Object

seqTok ⟵ *userUndoSeqTokens_VAL_NFNC* (*user*) ;

Pop the first command object from the queue

bb1 ⟵ *userUndoCommands_XST_SEQ_OBJ* (*seqTok*) ;

bb2 ⟵ *userUndoCommands_EMP_SEQ_OBJ* (*seqTok*) ;

IF *bb1* = *TRUE* ∧ *bb2* = *FALSE* **THEN**

comm ⟵ *userUndoCommands_VAL_SEQ_OBJ* (*seqTok* , *1*) ;

userUndoCommands_CUT_SEQ_OBJ (*seqTok* , *1*) ;

Extract operand, operator from command object

calc ⟵ *commandCalculators_VAL_FNC* (*comm*) ;

operator ⟵ *commandOperators_VAL_FNC* (*comm*) ;

operand ⟵ *commandOperands_VAL_NFNC* (*comm*) ;

Perform Undo on Calculator

val ⟵ *UndoOperation* (*calc* , *operator* , *operand*) ;

Now add Undone operation to Redo Queue

bb1 ⟵ *userRedoSeqTokens_DEF_NFNC* (*user*) ;

IF *bb1* = *TRUE* **THEN**

seqTok ⟵ *userRedoSeqTokens_VAL_NFNC* (*user*) ;

bb1 ⟵ *userRedoCommands_PSH_SEQ_OBJ* (*seqTok* , *comm*)

ELSE

bb1 , *seqTok* ⟵ *userRedoCommands_CRE_SEQ_OBJ* ;

```
            IF    bb1 = TRUE    THEN
               userRedoSeqTokens_STO_NFNC ( user , seqTok ) ;
               bb1 ⟵ userRedoCommands_PSH_SEQ_OBJ ( seqTok , comm )
            END
         END
      ELSE
         val := MaxScalar
      END
   ELSE
      val := MaxScalar
   END
END    ;
```

Redo is implemented as the reverse of Undo

```
val ⟵ Redo ( user )   ≙
   VAR    bb1 , bb2 , comm , calc , operator , operand , seqTok    IN
      bb1 ⟵ userCommand_DEF_FNC ( user ) ;
      IF    bb1 = TRUE    THEN
```

Get reference to correct Sequence Object

```
         seqTok ⟵ userRedoSeqTokens_VAL_NFNC ( user ) ;
```

Pop the first command object from the redo queue

```
         bb1 ⟵ userRedoCommands_XST_SEQ_OBJ ( seqTok ) ;
         bb2 ⟵ userRedoCommands_EMP_SEQ_OBJ ( seqTok ) ;

         IF    bb1 = TRUE ∧ bb2 = FALSE    THEN
            comm ⟵ userRedoCommands_VAL_SEQ_OBJ ( seqTok , 1 ) ;
            userRedoCommands_CUT_SEQ_OBJ ( seqTok , 1 ) ;
```

Extract operand, operator from command object

```
            calc ⟵ commandCalculators_VAL_FNC ( comm ) ;
```

```
        operator ⟵ commandOperators_VAL_FNC ( comm ) ;
        operand ⟵ commandOperands_VAL_NFNC ( comm ) ;
```

Perform operation on Calculator

```
        val ⟵ Operation ( calc , operator , operand ) ;
```

Now add Redone operation to Undo Queue

```
        bb1 ⟵ userUndoSeqTokens_DEF_NFNC ( user ) ;
        IF   bb1 = TRUE   THEN
          seqTok ⟵ userUndoSeqTokens_VAL_NFNC ( user ) ;
          bb1 ⟵ userUndoCommands_PSH_SEQ_OBJ ( seqTok , comm )
        ELSE
          bb1 , seqTok ⟵ userUndoCommands_CRE_SEQ_OBJ ;
          IF   bb1 = TRUE   THEN
            userUndoSeqTokens_STO_NFNC ( user , seqTok ) ;
            bb1 ⟵ userUndoCommands_PSH_SEQ_OBJ ( seqTok , comm )
          END
        END
      ELSE
        val := MaxScalar
      END
    ELSE
      val := MaxScalar
    END
  END   ;

AddCommand ( user , cmd )   ≙
  VAR   bb   IN
    bb ⟵ userCommand_DEF_FNC ( cmd ) ;
    IF   bb = TRUE   THEN
      userCommand_STO_FNC ( user , cmd )
    END
  END   ;
```

These are standard implementations of object constructor, accessor and mutator methods obtained by following the Object Implementation pattern presented in chapter 5.

```
newCommand ⟵ InstantiateCommand ( calc , operator , operand ) ≙
  VAR bb , free IN
    bb ⟵ freeCommandPointers_EMP_SET ;
    IF bb = TRUE THEN
      commandObjects_INC_NVAR ;
      free ⟵ commandObjects_VAL_NVAR
    ELSE
      free ⟵ freeCommandPointers_ANY_SET ;
      freeCommandPointers_RMV_SET ( free )
    END ;
    commandCalculators_STO_FNC ( free , calc ) ;
    commandOperators_STO_FNC ( free , operator ) ;
    commandOperands_STO_NFNC ( free , operand ) ;
    newCommand := free
  END ;

SetCommandCalculator ( cmd , calc ) ≙
  VAR bb IN
    bb ⟵ commandCalculators_DEF_FNC ( cmd ) ;
    IF bb = TRUE THEN
      commandCalculators_STO_FNC ( cmd , calc )
    END
  END ;

calc ⟵ GetCommandCalculator ( cmd ) ≙
  VAR bb IN
    bb ⟵ commandCalculators_DEF_FNC ( cmd ) ;
    IF bb = TRUE THEN
      calc ⟵ commandCalculators_VAL_FNC ( cmd )
    END
  END ;
```

```
  SetCommandOperator ( cmd , operator )  ≙
    VAR  bb  IN
      bb ⟵ commandOperators_DEF_FNC ( cmd ) ;
      IF  bb = TRUE  THEN
        commandOperators_STO_FNC ( cmd , operator )
      END
    END  ;
  operator ⟵ GetCommandOperator ( cmd )  ≙
    VAR  bb , vv  IN
      bb ⟵ commandOperators_DEF_FNC ( cmd ) ;
      IF  bb = TRUE  THEN
        vv ⟵ commandOperators_VAL_FNC ( cmd ) ;
        operator := vv
      END
    END  ;
  SetCommandOperand ( cmd , operand )  ≙
    VAR  bb  IN
      bb ⟵ commandOperands_DEF_NFNC ( cmd ) ;
      IF  bb = TRUE  THEN
        commandOperands_STO_NFNC ( cmd , operand )
      END
    END  ;
  operand ⟵ GetCommandOperand ( cmd )  ≙
    VAR  bb , vv  IN
      bb ⟵ commandOperands_DEF_NFNC ( cmd ) ;
      IF  bb = TRUE  THEN
        vv ⟵ commandOperands_VAL_NFNC ( cmd ) ;
        operand := vv
      END
    END
END
```

6.2.5 Case analysis

Like the Observer pattern case-study, the application of the B Command Pattern to modelling the user-calculator example was very straightforward. Of particular note are the *Operation* and *UndoOperation* operations in the *Calculator* showing how to add meaningful functionality to the calculator (receiver) that is executed by the *CalculatorCommand* machine. This development was very concrete and had the ability to be animated to show how specification matched the requirements. Using the *CalculatorCommand* machine to associate users to commands and commands to calculators as well as modelling the command objects themselves presented an interesting architecture.

However, when it came to implementing the system, although the system was implemented using the object implementation pattern, the complexity of the implementation machines themselves was quite pronounced because of the need to import so many SLIB machines to model the variables.

6.3 Case study: A Chess game

6.3.1 System requirements

The Chess game must model two virtual players playing a game of chess.

- Players should be able to make a move based on the chosen strategy, taking into consideration the state of the board.
- Strategies available to choose from should be an Aggressive strategy, a defensive strategy and a random strategy.
- Strategies should be able to be chosen at the beginning of the game.
- Pieces must be able to be set up in the correct initial position.
- Player can only make legal moves.

6.3.2 Pattern usage: Strategy

The Strategy pattern is particularly suited to this problem as it allows us to define a series of encapsulated, interchangeable chess strategies. Using the Strategy pattern, players can alter their strategy during the game without altering the game or board logic.

In the OO model of this pattern, we have the *ChessGame* class and the *ChessBoard* class, which together model the state of the game. We have a *ChessStrategy* abstract class, references the board and defines the interface

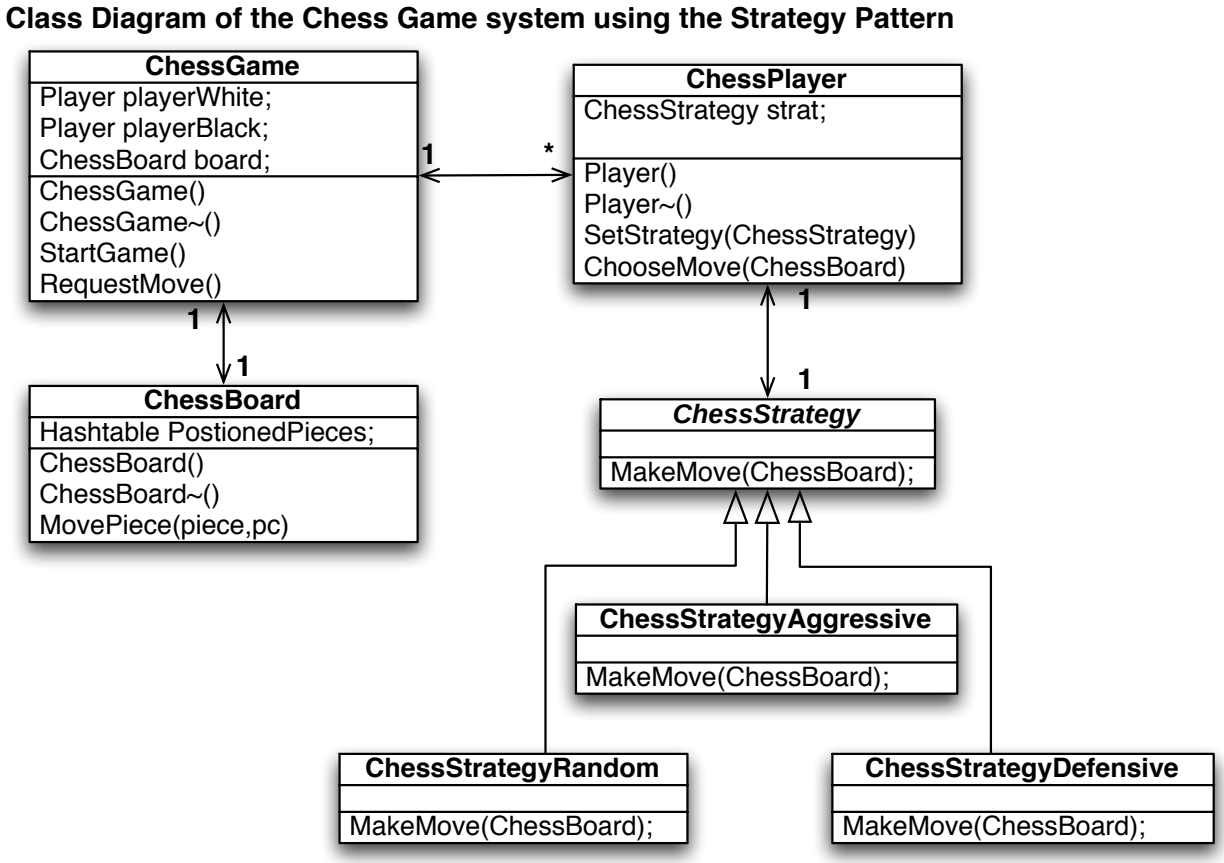


Figure 6.4: Class diagram for 'Chess Game'

of the strategy methods. Finally we have three strategy implementations, *RandomChess*, *AggressiveChess* and *DefensiveChess*.

The B model of this pattern naturally differs from the above, but is also somewhat simplified due to its role as specification, rather than a design. The *ChessGame* machine controls the state of the Chess game, while the *ChessBoard* machine maintains the board. The *ChessStrategies* machine is a stateless machine that encapsulates all of the chess strategies. However, since a B specification is not as detailed as a design, we do not need to express in detail exactly how each strategy's behaviour will be achieved. Instead, these decisions are modelled as constant functions in the *Chess_CTX* and *ChessStrategies_CTX* machines, to be implemented more fully in an implementation or refinement step.

6.3.3 Formal Specification

Included below is the full formal specification for the Chess development.

B Machine structure for the Chess Game system using the Strategy Pattern

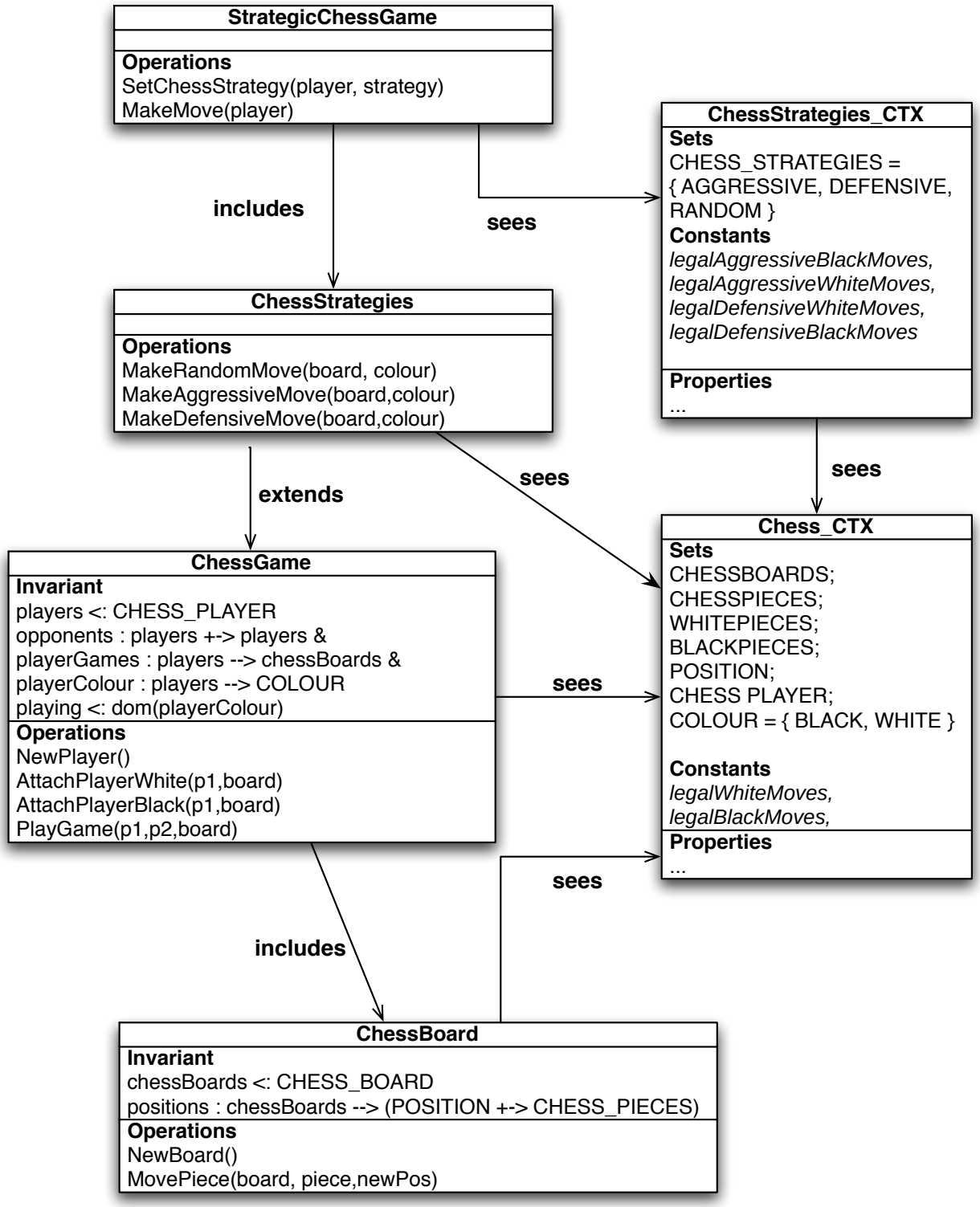


Figure 6.5: B Machine structure diagram for the 'Chess Game'

MACHINE *StrategicChessGame*

The *ChessGame* Models the players of the game and their relationship with the chess board. Players are associated with a strategy.

SEES

Chess_ctx ,

ChessStrategies_ctx

INCLUDES *ChessStrategies*

PROMOTES

NewPlayer ,

AttachPlayerWhite ,

AttachPlayerBlack

VARIABLES *strategies*

strategies tracks the player's chosen strategy, and playerGames models the relationship between players and the game they are currently engaged in.

INVARIANT

$strategies \in players \pfun CHESS\_STRATEGIES \land$

$playing \subseteq \mathsf{dom}(\ strategies\ ) \land$

$playing \subseteq \mathsf{dom}(\ strategies\ ) \land$

$playing \subseteq \mathsf{dom}(\ playerGames\ )$

INITIALISATION $strategies := \{\}$

OPERATIONS

SetChessStrategy simply sets a strategy to a given player, deciding how that player will play the game.

SetChessStrategy (*player* , *strat*) $\widehat{=}$

PRE $player \in players \wedge strat \in CHESS\_STRATEGIES$ **THEN**

$strategies\ (\ player\ ) := strat$

END **;**

MakeMove makes a move for the given player, based on the player's strategy.

MakeMove (*player*) $\widehat{=}$

PRE

$player \in \mathsf{dom}\ (\ playerGames\ ) \wedge$

$player \in playing$

THEN

SELECT $strategies\ (\ player\ ) = RANDOM$ **THEN**

$MakeRandomMove\ (\ playerGames\ (\ player\ )\ ,\ playerColour\ (\ player\ )\ )$

WHEN $strategies\ (\ player\ ) = AGGRESSIVE$ **THEN**

$MakeAggressiveMove\ (\ playerGames\ (\ player\ )\ ,\ playerColour\ (\ player\ )\ )$

WHEN $strategies\ (\ player\ ) = DEFENSIVE$ **THEN**

$MakeDefensiveMove\ (\ playerGames\ (\ player\ )\ ,\ playerColour\ (\ player\ )\ )$

END

END

END

MACHINE *ChessGame*

The *ChessGame* Models the players of the game and their relationship with the chess board.

SEES *Chess_ctx*

INCLUDES *ChessBoard*

PROMOTES *NewBoard* , *MovePiece*

VARIABLES *players* , *opponents* , *playerGames* , *playerColour*

opponents is a injective relationship mapping a player to their opponent.

playerColour maps each player to their colour, BLACK or WHITE.

INVARIANT

$players \subseteq CHESS\_PLAYER \land$

$opponents \in players \rightarrowtail players \land$

$playerGames \in players \nrightarrow chessBoards \land$

$playerColour \in players \nrightarrow COLOUR \land$

$playing \subseteq \mathsf{dom}\ (\ playerColour\ )$

INITIALISATION

$players\ ,\ opponents := \{\}\ ,\ \{\}\ \parallel$

$playerGames\ ,\ playerColour := \{\}\ ,\ \{\}$

OPERATIONS

$player \longleftarrow$ **NewPlayer** $\widehat{=}$

BEGIN

ANY pp **WHERE**

$pp \in CHESS\_PLAYER - players$ **THEN**

$player := pp\ \parallel$

$players := players \cup \{ pp \}$

END

END ;

AttachPlayerWhite attaches the first player to the white colour on a game

AttachPlayerWhite (*p1* , *board*) $\widehat{=}$

PRE

$p1 \in players \wedge$

$p1 \notin playing \wedge$

$board \in chessBoards$

THEN

$playerColour (p1) := WHITE \parallel$

$playerGames (p1) := board$

END ;

AttachPlayerBlack attaches the second player to the black colour on a game

AttachPlayerBlack (*p2* , *board*) $\widehat{=}$

PRE

$p2 \in players \wedge$

$p2 \notin playing \wedge$

$board \in chessBoards$

THEN

$playerColour (p2) := BLACK \parallel$

$playerGames (p2) := board$

END ;

PlayGame : The players are set as opponents and the game can begin

PlayGame ($p1$, $p2$) $\widehat{=}$

PRE

$p1 \in \mathsf{dom}\ (\ playerColour\ ) \wedge p2 \in \mathsf{dom}\ (\ playerColour\ ) \wedge$

$p1 \in \mathsf{dom}\ (\ playerGames\ ) \wedge p2 \in \mathsf{dom}\ (\ playerGames\ ) \wedge$

$p1 \notin playing \wedge p2 \notin playing \wedge$

$p1 \neq p2 \wedge$

$playerGames\ (\ p1\ ) = playerGames\ (\ p2\ )$

THEN

$opponents\ (\ p1\ ) := p2$

END

DEFINITIONS $playing \widehat{=} \mathsf{dom}\ (\ opponents\ ) \cup \mathsf{ran}\ (\ opponents\ )$

END

MACHINE *Chess_ctx*

The Basic Structure of the Game

SETS

$CHESS\_BOARDS$;

$CHESS\_PIECES$; $WHITE\_PIECES$; $BLACK\_PIECES$;

$POSITION$;

$CHESS\_PLAYER$;

$COLOUR = \{ BLACK , WHITE \}$

CONSTANTS

$legalMoves$, $legalBlackMoves$, $legalWhiteMoves$,

$changeBoard$

legalWhiteMoves and **legalBlackMoves** are constant functions which map a current board arrangement to a function which takes a *Chess_Piece* of a certain colour and returns its set of possible positions. These can be constant since the number of board arrangements and possible moves, while very large is still finite and could be instantiated.

PROPERTIES

$WHITE\_PIECES \cup BLACK\_PIECES = CHESS\_PIECES \wedge$

$WHITE\_PIECES \cap BLACK\_PIECES = \{\} \wedge$

$\mathsf{card} (WHITE\_PIECES) = \mathsf{card} (BLACK\_PIECES) \wedge$

$legalMoves \in BOARD \rightarrow MOVES (CHESS\_PIECES) \wedge$

$legalWhiteMoves \in BOARD \rightarrow MOVES (WHITE\_PIECES) \wedge$

$legalBlackMoves \in BOARD \rightarrow MOVES (BLACK\_PIECES) \wedge$

$legalMoves = legalWhiteMoves \cup legalBlackMoves \wedge$

$\forall bd . (bd \in BOARD \Rightarrow \mathsf{dom} (legalMoves (bd)) \subseteq \mathsf{ran} (bd)) \wedge$

$changeBoard \in BOARD \rightarrow (CHESS\_PIECES \nrightarrow (POSITION \nrightarrow BOARD)) \wedge$

$\forall (bd , pc) . (bd \in BOARD \wedge pc \in CHESS\_PIECES \wedge$

$pc \in \mathsf{dom}\ (\ legalMoves\ (\ bd\ )\ ) \Rightarrow$

$pc \in \mathsf{dom}\ (\ changeBoard\ (\ bd\ )\ )\ ) \wedge$

$\forall\ (\ bd\ ,\ pc\ ,\ pos\ )\ .\ (\ bd \in BOARD \wedge pc \in CHESS\_PIECES \wedge pos \in POSITION \wedge$

$pos \in legalMoves\ (\ bd\ )\ (\ pc\ ) \Rightarrow$

$pos \in \mathsf{dom}\ (\ changeBoard\ (\ bd\ )\ (\ pc\ )\ )\ )$

DEFINITIONS

$BOARD \ \widehat{=}\ POSITION \nrightarrow CHESS\_PIECES$ **;**

$MOVES\ (\ X\ ) \ \widehat{=}\ X \nrightarrow \mathbb{P}\ (\ POSITION\ )$

END

ChessStrategies models different strategies that the players might use. The machine itself is stateless, operating directly on the board machine.

MACHINE *ChessStrategies*

SEES

ChessStrategies_ctx ,

Chess_ctx

EXTENDS *ChessGame*

OPERATIONS

MakeRandomMove: Based on the players colour, we make a random but legal move.

MakeRandomMove (*board* , *colour*) $\widehat{=}$

PRE $board \in chessBoards \wedge$

$colour \in COLOUR$

THEN

If the piece's colour is black, we check for all legal moves for black, then

1. Take a piece that can legally moved,
2. Choose any legal new position for that piece.
3. Move the chosen piece to the chosen position.

SELECT $colour = BLACK$ **THEN**

ANY *piece* **WHERE**

$piece \in BLACK\_PIECES \wedge$

$piece \in \mathsf{dom}\ (\ legalBlackMoves\ (\ positions\ (\ board\ )\ )\ )$

THEN

ANY *pos* **WHERE**

$pos \in POSITION \wedge$

$pos \in legalBlackMoves\ (\ positions\ (\ board\ )\ )\ (\ piece\ )$

THEN $MovePiece\ (\ board\ ,\ piece\ ,\ pos\ )$

END

END

If the piece is white, we do the same as above but for white pieces.

WHEN $colour = WHITE$ **THEN**

ANY $piece$ **WHERE**

$piece \in WHITE\_PIECES \wedge$

$piece \in \mathsf{dom}\ (\ legalWhiteMoves\ (\ positions\ (\ board\ )\ )\ )$

THEN **ANY** pos **WHERE**

$pos \in POSITION \wedge$

$pos \in legalWhiteMoves\ (\ positions\ (\ board\ )\ )\ (\ piece\ )$

THEN $MovePiece\ (\ board\ ,\ piece\ ,\ pos\ )$

END

END

END

END ;

MakeAggressiveMove Based on the player's colour we try to make a move that results in taking a piece. The operation follows much the same logic as **MakeRandomMove**, but makes use of the *legalAggressiveMoves* constant functions in **Chess_CTX**.

MakeAggressiveMove ($board$, $colour$) $\widehat{=}$

PRE $board \in chessBoards \wedge$

$colour \in COLOUR$

THEN

SELECT $colour = BLACK$ **THEN**

ANY $piece$ **WHERE**

$piece \in \mathsf{dom}\ (\ legalAggressiveBlackMoves\ (\ positions\ (\ board\ )\ )\ )$

THEN **ANY** pos **WHERE**

```
            pos ∈ POSITION ∧
            pos ∈ legalAggressiveBlackMoves ( positions ( board ) ) ( piece )
        THEN    MovePiece ( board , piece , pos )
        END
      END
    WHEN    colour = WHITE    THEN
      ANY    piece    WHERE
        piece ∈ dom ( legalAggressiveWhiteMoves ( positions ( board ) ) )
      THEN    ANY    pos    WHERE
            pos ∈ POSITION ∧
            pos ∈ legalAggressiveWhiteMoves ( positions ( board ) ) ( piece )
        THEN    MovePiece ( board , piece , pos )
        END
      END
    END
  END    ;

MakeDefensiveMove ( board , colour )  ≙
  PRE    board ∈ chessBoards ∧ colour ∈ COLOUR
  THEN
    SELECT    colour = BLACK    THEN
      ANY    piece    WHERE
        piece ∈ BLACK_PIECES ∧
        piece ∈ dom ( legalDefensiveBlackMoves ( positions ( board ) ) )
      THEN    ANY    pos    WHERE
            pos ∈ POSITION ∧
            pos ∈ legalDefensiveBlackMoves ( positions ( board ) ) ( piece )
        THEN    MovePiece ( board , piece , pos )
        END
      END
    WHEN    colour = WHITE    THEN
      ANY    piece    WHERE
        piece ∈ dom ( legalDefensiveWhiteMoves ( positions ( board ) ) )
```

THEN **ANY** *pos* **WHERE**

pos ∈ *POSITION* ∧

pos ∈ *legalDefensiveWhiteMoves* (*positions* (*board*)) (*piece*)

THEN *MovePiece* (*board* , *piece* , *pos*)

END

END

END

END

ChessBoard models the chess board itself, with all its pieces mapped to positions. Its operations control the movements of pieces, instantiation and placement.

MACHINE *ChessBoard*

SEES *Chess_ctx*

VARIABLES

chessBoards , *positions*

chessBoards models all chess boards currently being played.

positions maps each chessBoard to its current state of play, mapping positions to pieces. *positions* is a total function because every board must have pieces on it. However, chess pieces can not occupy every position on a board, hence the second order part of the function is only partial.

INVARIANT

$chessBoards \subseteq CHESS\_BOARDS \wedge$

$positions \in chessBoards \rightarrow BOARD$

INITIALISATION *chessBoards* , *positions* := {} , {}

OPERATIONS

board $\longleftarrow$ **NewBoard** $\widehat{=}$

BEGIN

ANY *bb* , *startingPositions* **WHERE**

$bb \in CHESS\_BOARDS - chessBoards \wedge$

$startingPositions \in BOARD$ **THEN**

$board := bb \parallel$

$chessBoards := chessBoards \cup \{ bb \} \parallel$

$positions (bb) := startingPositions$

END

END ;

MovePiece moves a piece to a new location, and removes pieces captured by the move. This function must be protected by upper layer functions to ensure the move is legal. It will remove any piece currently occupying the new position specified.

MovePiece (*brd* , *pc* , *newPos*) $\widehat{=}$

PRE

$brd \in chessBoards \wedge$

$pc \in CHESS\_PIECES \wedge$

$pc \in \mathsf{ran}\ (\ positions\ (\ brd\ )\ ) \wedge$

$newPos \in POSITION \wedge$

$newPos \in legalMoves\ (\ positions\ (\ brd\ )\ )\ (\ pc\ ) \wedge$

$positions\ (\ brd\ ) \in \mathsf{dom}\ (\ changeBoard\ ) \wedge$

$pc \in \mathsf{dom}\ (\ changeBoard\ (\ positions\ (\ brd\ )\ )\ ) \wedge$

$newPos \in \mathsf{dom}\ (\ changeBoard\ (\ positions\ (\ brd\ )\ )\ (\ pc\ )\ )$

THEN

$positions\ (\ brd\ ) := changeBoard\ (\ positions\ (\ brd\ )\ )\ (\ pc\ )\ (\ newPos\ )$

END

END

6.3.4 Case analysis

Most interesting about this case is that it gave rise to the interface pattern in B. This because of how strategy works the client is ignorant of the implementation of the strategy, and the implementation can be freely changed without the client needing to change its behaviour. This led to thinking about how an interface should be represented in B. However, the interface pattern was not usable with the current machine structure, since the *ChessStrategies* machine must apply its move on the board. A machine in B can only be included *once*. Thus if each strategy was encapsulated in its own machine, then the ChessBoard could not be included by all of them.

Another solution utilising multiple patterns

The solution above could be improved upon with the following changes involving other design patterns.

- Use the Command pattern to encapsulate a "Move" command.
- Refactor the Strategy pattern to use an Interfacestyle structure, with each strategy encapsulated in a machine and the strategy interface including all of the concrete strategy machines.
- The Strategy Machines would construct a "Move" command and return it to the *ChessGame* machine which could then apply the move.

While further investigation is required, at first glance the proposed solution appears promising.

6.4 Case study: A Spreadsheet Engine

To demonstrate the composition of design patterns to build a complex system, a case-study on a spreadsheet engine specified in B will be presented.

6.4.1 Motivation and System Requirements

A spreadsheet engine is a good example of how B could be used to specify and implement the core of a system where correctness is important. Producing a spreadsheet engine in B would allow a user interface layer to be placed on top in a model-view-controller architecture where the lower levels have been formally verified. The requirement for this spreadsheet engine is that it provides an interface of operations for the interface layer to call. These interface need to allow the user to set the value of a cell whether this value is a formula, literal value or formula. The interface also needs to allow for the retrieval of data from the spreadsheet. Furthermore, we wish to specify having undo/redo as a requirement to be implemented in the spreadsheet engine.

To produce this system, we have decided not to proceed first with an object oriented design which is modelled using B but instead jump straight into using the B design patterns that have been developed to model the requirements. Another core requirement of a spreadsheet is that cells are automatically updated if the cells are referencing in their formula undergo a change. As stated above, undo/redo is also a requirement.

To model these two core requirements we have used a composition of the observer and the command design pattern to build the system. Cells need to be able to observe other cells as a consequence of having formulae entered so the observer pattern consist of a *Cell* machine as the participant and a *Formula* machine will be the pattern machine that includes the participant. Because cells are observing themselves, there is no need for a second participant in this Observer model. The *Formula* machine should also model the functionality required of formulae and provide an interface for the above layer to manipulate the cells and formulae.

With undo/redo functionality required, there will be an instance of the Command pattern that includes the *Formula* machine. This is called the *Spreadsheet* machine and provides the interface for which the user interface can make function calls. The *Spreadsheet* layer needs to be able to encapsulate the function calls to the formula machine so it will also include the *SpreadsheetCommand* machine that contains this functionality to complete the Command Pattern.

These two patterns are composed using Inter-pattern links [Sandrine, Blazy et al] as one of the Command Patterns participants, the *Formula* machine (Receiver) is also the Observer Pattern machine.

6.4.2 A Discussion of Pattern Composition

Going back to the research presented in Chapter 2, we have decided that the two most useful ways to compose two or more patterns is to use inter-pattern links or pattern juxtaposition. Our interpretation of the interpattern links method is the use of participant machines that also double as pattern machines within the system. Because B machines must be structured in a Tree form as opposed to OO classes which can be represented by a graph, having nodes (machines) that are both pattern and participants lends itself well. The other method canvassed was to use an association machine to include two pattern machines and juxtapose them together. The association machine can then just use the interface of both patterns and present this as a single interface. Using an association machine to include two pattern machines also leaves open the possibility of specifying links between the two pattern machines in the association machines invariant. All of these are valid methods to compose design patterns to build complete systems in B and the user must decide which method bests suits their requirements. In this spreadsheet case study, having interpattern links was decided to be the best method.

6.4.3 Formal Specification

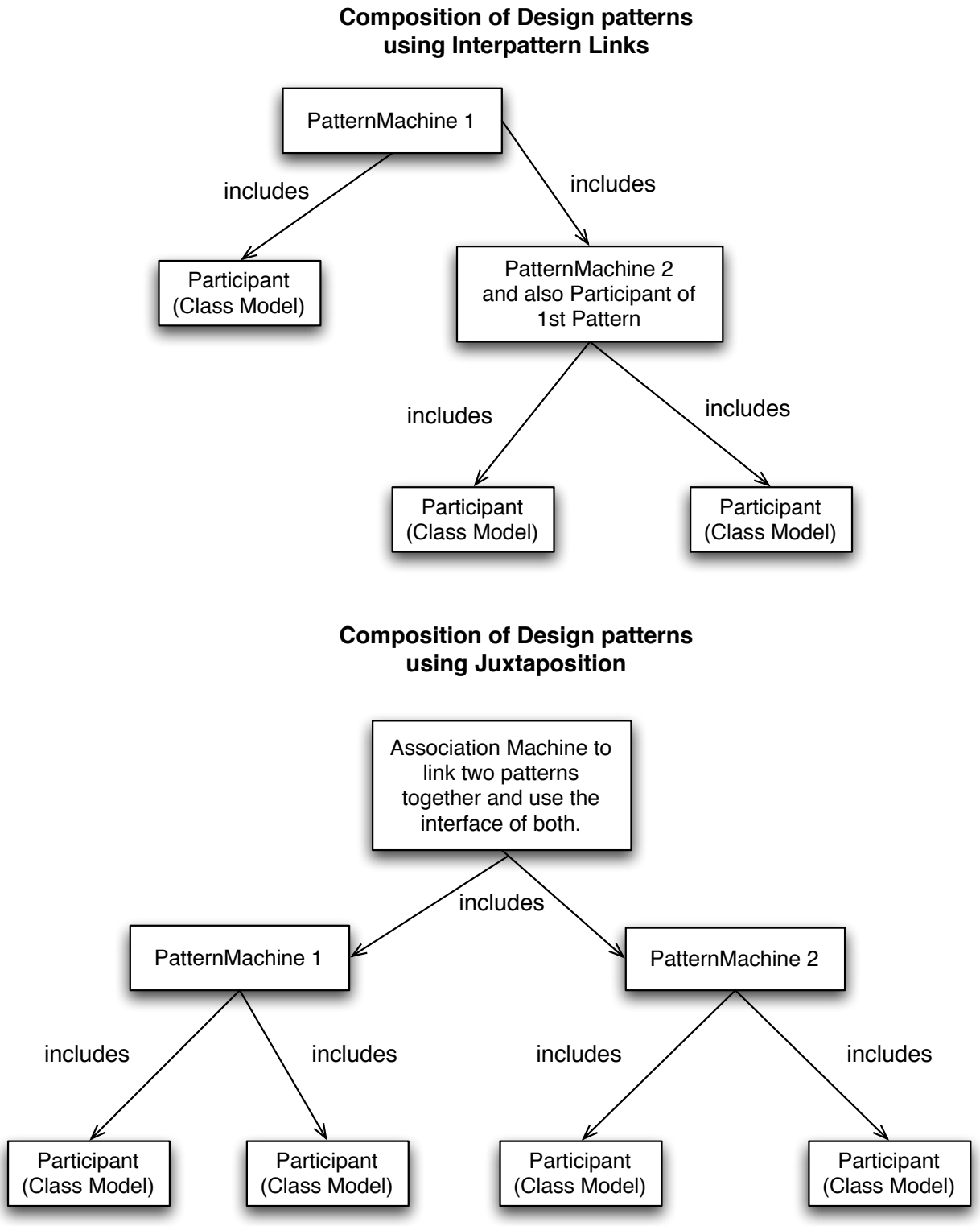


Figure 6.6: Different methods to compose patterns

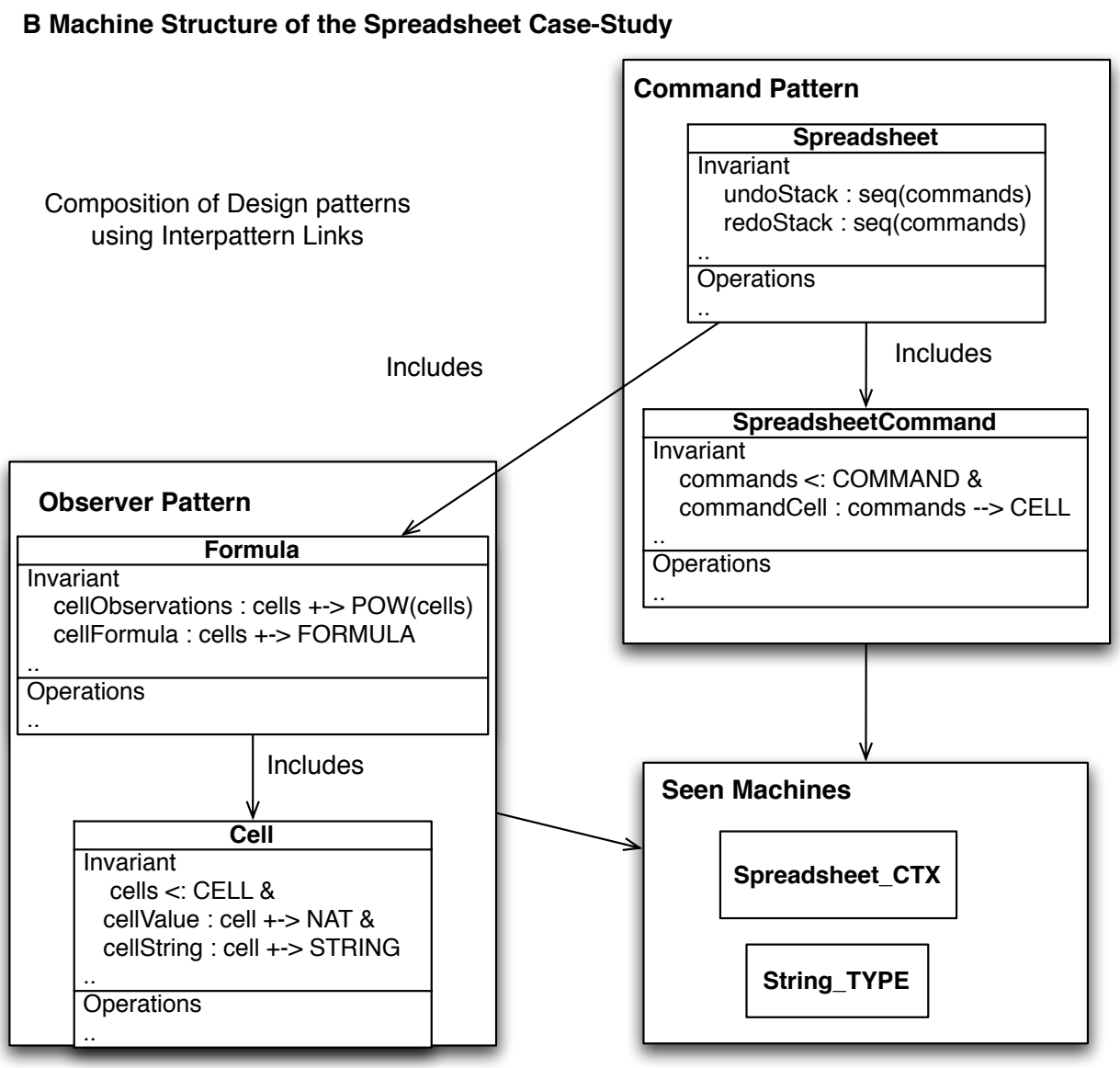


Figure 6.7: Overview of the Spreadsheet system structure

MACHINE *Spreadsheet*

The *Spreadsheet* machine represents the interface for the spreadsheet engine being specified. Ideally the fragile operations of this machine would be protected by a robust Spreadsheet_API machine but in the interests of brevity and for the purposes of this case study, this API machine has been omitted.

This machine will contain the functionality required for undoing and redoing operations and forms part of the observer pattern used in this development. Any of the spreadsheet commands that involve changing the state of the spreadsheet will need to be encapsulated and stored. The spreadsheet machine will not need to make references to the *Formula* machines invariant as it only needs to call operations in the *Formula* machine without needing to know its state.

This machine will need to include the *SpreadsheetCommand* machine as this provides the functionality for encapsulating commands

SEES *Spreadsheet_CTX* , *String_TYPE*

This machine itself models a singleton instantiation of the Formula class in the OO design. As such, we do not require a set of objects to model the different instances of the spreadsheet application as the machine itself is the object. Two attributes are required, these are for referencing the undo stack of operations (modelled by *undoStack*) and the redo stack, (modelled by *redoStack*).

INCLUDES *Formula* , *SpreadsheetCommand*

The *Spreadsheet_CTX* machine provides the systemwide context and is where all deferred sets that model the different class types within this development are stored

VARIABLES *undoStack* ,
redoStack

The invariant constrains the two stacks by specifiying that they are a sequences of spreadsheetCommands, of which *SpreadsheetCommand* models. To strengthen the invariant, a predicate to ensure that a single instance of a command cannot exist both in the redo queue and the undo queue are specified. A further predicate states that the intersection of the elements of both stacks will be equal to the set of instantiated commands.

INVARIANT $undoStack \in \mathsf{seq}\ (\ commands\ ) \wedge$

$redoStack \in \mathsf{seq}\ (\ commands\ ) \wedge$

$\mathsf{ran}\ (\ undoStack\ ) \cup \mathsf{ran}\ (\ redoStack\ ) = \{\} \wedge$

$\mathsf{ran}\ (\ undoStack\ ) \cup \mathsf{ran}\ (\ redoStack\ ) = commands$

At initialisation, both stacks are set to empty as no commands have been called yet

INITIALISATION $undoStack$,

$redoStack := [\ ]\ ,\ [\ ]$

The operations below form the interface for the spreadsheet. There are three operations for getting the values from within spreadsheet (required because of the three different types of data that a cell can hold), similarly there are three operations for setting the values inside the spreadsheet. These include being able to set a cell to hold a formula, a string or a literal. Finally there are undo and redo operations to reverse any changes the user has made.

OPERATIONS

Each of the set operations must call the corresponding operation within the cell machine and then encapsulate the call and store it inside the *SpreadsheetCommand* machine.

SpreadsheetSetCellFormula ($cell$, $formula$) $\widehat{=}$

PRE

$cell \in CELL \wedge$

$formula \in FORMULA \wedge$

$commands \neq COMMAND$

THEN

Store command in the undo stack

ANY cc

WHERE $cc \in COMMAND - commands$

THEN

CreateFormulaCommand (*cc* , *cell* , *formula*) ‖

undoStack := *cc* → *undoStack*

END ‖

Clear the redo stack

redoStack := [ ] ‖

Call the operation in the formula machine

FormulaSetCellFormula (*cell* , *formula*)

END ;

SpreadsheetSetCellValue (*cell* , *value*) ≙

PRE

cell ∈ *CELL* ∧

value ∈ ℕ ∧

commands ≠ *COMMAND*

THEN

Store command in the undo stack

ANY *cc*

WHERE *cc* ∈ *COMMAND* − *commands*

THEN

CreateLiteralCommand (*cc* , *cell* , *value*) ‖

undoStack := *cc* → *undoStack*

END ‖

Clear the redo stack

redoStack := [ ] ‖

Call the operation in the formula machine

FormulaSetCellLiteral (*cell* , *value*)

END ;

SpreadsheetSetCellString (*cell* , *string*) ≙

PRE

$cell \in CELL \wedge$

$string \in STRING \wedge$

$commands \neq COMMAND$

THEN

Store command in the undo stack

ANY cc

WHERE $cc \in COMMAND - commands$

THEN

$CreateStringCommand\ (\ cc\ ,\ cell\ ,\ string\ )\ \|$

$undoStack := cc \rightarrow undoStack$

END $\|$

Clear the redo stack

$redoStack := [\,]\ \|$

Call the operation in the formula machine

$FormulaSetCellString\ (\ cell\ ,\ string\ )$

END ;

SpreadsheetClearCell ($cell$) $\widehat{=}$

PRE

$cell \in CELL \wedge$

$commands \neq COMMAND$

THEN

Store command in the undo stack

ANY cc

WHERE $cc \in COMMAND - commands$

THEN

$CreateClearCommand\ (\ cc\ ,\ cell\ )\ \|$

$undoStack := cc \rightarrow undoStack$

END $\|$

Clear the redo stack

$redoStack := [\,] \;\|$

Call the operation in the formula machine

$FormulaClearCell\ (\ cell\ )$

END ;

The undo operation retrieves the first command from the undo stack and returns the spreadsheet to its previous state. It then stores this commmand on the redo stack.

SpreadsheetUndo $\widehat{=}$

PRE

$undoStack \neq [\,]$

THEN

ANY cmd

WHERE $cmd = undoStack\ (\ 1\ )$

THEN

Determine the type of command that was called and undo it by setting the value of the cell back to its previous value stored in the command object

SELECT $commandtype\ (\ cmd\ ) = SET\_VALUE$

THEN $FormulaSetCellLiteral\ (\ commandcell\ (\ cmd\ )\ ,\ commandliteral\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = SET\_STRING$

THEN $FormulaSetCellString\ (\ commandcell\ (\ cmd\ )\ ,\ commandstring\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = SET\_FORMULA$

THEN $FormulaSetCellFormula\ (\ commandcell\ (\ cmd\ )\ ,\ commandformula\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = CLEAR$

THEN $FormulaClearCell\ (\ commandcell\ (\ cmd\ )\ )$

END $\|$

Add this cmd to redoStack

$redoStack := cmd \rightarrow redoStack$

END ||

$undoStack := \mathsf{tail}\ (\ undoStack\ )$

END ;

The redo operation retrieves the first command from the redo stack and re-executes the command. It then stores this commmand on the undo stack.

SpreadsheetRedo $\widehat{=}$

PRE

$redoStack \neq [\,]$

THEN

ANY *cmd*

WHERE $cmd = redoStack\ (\ 1\ )$

THEN

Determine the type of command that was called and undo it by setting the value of the cell back to its previous value stored in the command object

SELECT $commandtype\ (\ cmd\ ) = SET\_VALUE$

THEN $FormulaSetCellLiteral\ (\ commandcell\ (\ cmd\ )\ ,\ commandliteral\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = SET\_STRING$

THEN $FormulaSetCellString\ (\ commandcell\ (\ cmd\ )\ ,\ commandstring\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = SET\_FORMULA$

THEN $FormulaSetCellFormula\ (\ commandcell\ (\ cmd\ )\ ,\ commandformula\ (\ cmd\ )\ )$

WHEN $commandtype\ (\ cmd\ ) = CLEAR$

THEN $FormulaClearCell\ (\ commandcell\ (\ cmd\ )\ )$

END ||

Add this cmd to undoStack

$undoStack := cmd \rightarrow undoStack$

END ||

$redoStack := \mathsf{tail}\ (\ redoStack\ )$

END ;

The operations that model the retrieval of string values do not affect the redo stack and undo stack so they are just return the state of the cell in the formula machine.

string ⟵ **SpreadsheetGetCellString** (*cell*) ≙

PRE

cell ∈ *CELL*

THEN

IF *cell* ∈ dom (*cellString*)

THEN

string ⟵ *GetCellString* (*cell*)

ELSE

string := [ ]

END

END ;

value ⟵ **SpreadsheetGetCellValue** (*cell*) ≙

PRE

cell ∈ *CELL*

THEN

IF *cell* ∈ dom (*cellValue*)

THEN

value ⟵ *GetCellValue* (*cell*)

ELSE

value := *0*

END

END ;

formula ⟵ **SpreadsheetGetCellFormula** (*cell*) ≙

PRE

cell ∈ dom (*cellFormula*)

THEN

formula := *cellFormula* (*cell*)

END

END

MACHINE *SpreadsheetCommand*

Them *SpreadsheetCommand* machine is responsible for encapsulating commands made in the spreadsheet system so that they can be undone and redone. This is based on the Command pattern and this machine models the Command class.

SEES *Spreadsheet_CTX* , *String_TYPE*

commands models the command objects that will exist in the system while the attributes variables *commandformula*, *comandliteral* and *comandstring* are used to store the parameters for those commands. *commandtype* stores the correct operation to call if the object is used in an undo or redo situation.

VARIABLES *commands* ,

commandtype ,

commandcell ,

commandformula ,

commandliteral ,

commandstring

The invariant maps the command objects to its attributes using the B-representation-of-class pattern

INVARIANT $commands \subseteq COMMAND \land$

$commandtype \in commands \rightarrow COMMAND\_TYPE \land$

$commandcell \in commands \rightarrow CELL \land$

$commandformula \in commands \nrightarrow FORMULA \land$

$commandliteral \in commands \nrightarrow \mathbb{N} \land$

$commandstring \in commands \nrightarrow STRING \land$

$\forall\ cc\ .\ (\ cc \in \mathsf{dom}\ (\ commandformula\ )\ \Rightarrow\ commandtype\ (\ cc\ ) = SET\_FORMULA\ ) \land$

$\forall\ cc\ .\ (\ cc \in \mathsf{dom}\ (\ commandliteral\ )\ \Rightarrow\ commandtype\ (\ cc\ ) = SET\_VALUE\ ) \land$

$\forall\ cc\ .\ (\ cc \in \mathsf{dom}\ (\ commandstring\ )\ \Rightarrow\ commandtype\ (\ cc\ ) = SET\_STRING\ ) \land$

$\forall\ cc\ .\ (\ cc \in \mathsf{dom}\ (\ commandcell\ )\ \Rightarrow\ commandtype\ (\ cc\ ) = CLEAR\ )$

INITIALISATION *commands* ,

commandtype ,

commandformula ,

commandliteral ,

commandstring := {} , {} , {} , {} , {}

The operations provided below form an interface for the *Spreadsheet* machine to store and retrieve command objects

OPERATIONS

CreateStringCommand (*command* , *cell* , *string*) $\widehat{=}$

PRE

$command \in COMMAND - commands \wedge$

$cell \in CELL \wedge$

$string \in STRING$

THEN

$commands := commands \cup \{ command \} \parallel$

$commandtype (command) := SET\_STRING \parallel$

$commandcell (command) := cell \parallel$

$commandstring (command) := string$

END ;

CreateFormulaCommand (*command* , *cell* , *formula*) $\widehat{=}$

PRE

$command \in COMMAND - commands \wedge$

$cell \in CELL \wedge$

$formula \in FORMULA$

THEN

$commands := commands \cup \{ command \} \parallel$

$commandtype (command) := SET\_STRING \parallel$

$commandcell (command) := cell \parallel$

$commandformula (command) := formula$

END ;

CreateLiteralCommand (*command* , *cell* , *literal*) $\widehat{=}$

PRE

$command \in COMMAND - commands \wedge$

$cell \in CELL \wedge$

$literal \in \mathbb{N}$

THEN

$commands := commands \cup \{ command \} \parallel$

$commandtype (command) := SET\_STRING \parallel$

$commandcell (command) := cell \parallel$

$commandliteral (command) := literal$

END ;

CreateClearCommand (*command* , *cell*) $\widehat{=}$

PRE

$command \in COMMAND - commands \wedge$

$cell \in CELL$

THEN

$commands := commands \cup \{ command \} \parallel$

$commandtype (command) := CLEAR \parallel$

$commandcell (command) := cell$

END ;

DeleteCommands (*coms*) $\widehat{=}$

PRE

$coms \subseteq commands$

THEN

$commands := commands - coms \parallel$

$commandtype := coms \lhd commandtype \parallel$

$commandcell := coms \lhd commandcell \parallel$

$commandliteral := coms \lhd commandliteral \parallel$

$commandformula := coms \lhd commandformula \parallel$

$commandstring := coms \lhd commandstring$

END

END

MACHINE *Formula*

The Formula machine is based on the GoF Observer pattern. To solve the problem of each cell notifying any cell which references the updated cell, an observer type relationship is required.

SEES *Spreadsheet_CTX* , *String_TYPE*

INCLUDES *Cell*

All operations which do not change the state of any machine can be promoted without writing a wrapper function

PROMOTES *GetCellValue* ,
GetCellString

VARIABLES *formulae* ,
cellFormula ,
cellObservations

INVARIANT

formulae is the set of formulae that are currently being used in the spreadsheet

$formulae \subseteq FORMULA \land$

cellFormula maps cells to formulae

$cellFormula \in cells \nrightarrow formulae \land$

cellObservations is based on the observer model - when a cell is updated it must notify the set of cells that it maps to

$cellObservations \in cells \nrightarrow \mathbb{P}\ (\ cells\ )$

INITIALISATION $formulae$, $cellFormula$, $cellObservations := \{\}$, $\{\}$, $\{\}$

OPERATIONS

After the formula has been entered into the system then it can be attached to a cell. It also needs to be evaluated to set the cells value. This is the purpose of the *SetCellFormula*

FormulaSetCellFormula (*cell* , *formula*) $\widehat{=}$

PRE

$cell \in cells \wedge$

$formula \in FORMULA$

THEN

cellFormula (*cell*) := *formula*

All cells affected by this formula should be updated by calling *SetMultipleCellValues* in the *Cell* machine and the *CellObservation* relationships between the cells should be updated here

END ;

In addition - we need to wrap the SetCellValue, SetCellString and the ClearCell operations because they are update operations which may require notifications to be sent to other cells that are referencing them

FormulaSetCellLiteral (*cell* , *value*) $\widehat{=}$

PRE

$cell \in cells \wedge$

$value \in \mathbb{N}$

THEN

SetCellValue (*cell* , *value*)

All cells affected by this update should be updated by calling *SetMultipleCellValues* in the *Cell* machine and the *CellObservation* relationships between the cells should be updated here with the correct function as a parameter

END ;

FormulaSetCellString (*cell* , *string*) $\widehat{=}$

PRE

$cell \in cells \wedge$

$string \in STRING$

THEN

SetCellString (*cell* , *string*)

All cells affected by this update should be updated by calling *SetMultipleCellValues* in the *Cell* machine with the correct function as a parameter

END ;

FormulaClearCell (*cell*) $\widehat{=}$

PRE

cell $\in$ *cells*

THEN

ClearCell (*cell*)

All cells affected by this formula should be updated by calling *SetMultipleCellValues* in the *Cell* machine with the correct function as a parameter

END

END

MACHINE *Cell*

This machine is a the representation of the cells in the spreadsheet at the most basic level. The cells here can only store values or store strings. Functionality that allows cells to reference other cells through formulas and automatically notify cells that reference themselves will be specified in a higher level machine that includes this one.

SEES *Spreadsheet_CTX* , *String_TYPE*

The variables will model a set of cell objects and their attributes which will include a reference to a string or an integer value.

VARIABLES *cells* ,

cellValue ,

cellString

The CELL deferred set is a representation of the entire spreadsheet while cells which is a subset represents only those cells that the user has populated with values or strings

INVARIANT $cells \subseteq CELL \land$

$cellValue \in cells \pfun \mathbb{N} \land$

$cellString \in cells \pfun STRING \land$

The invariant also specifies that any cell with an integer value cannot reference a string and vice versa.

$\mathsf{dom}(\ cellValue\ ) \cap \mathsf{dom}(\ cellString\ ) = \{\}$

INITIALISATION $cells := \{\} \ \parallel$

$cellValue := \{\} \ \parallel$

$cellString := \{\}$

OPERATIONS

The Set operations for the cell objects will diverge from the standard object model that has been presented throughout this thesis. Because CELL represents the entire spreadsheet, the set operations here will instantiate cells as needed if the user provides a cell reference that hasn't been used yet

SetCellValue (*cell* , *value*) $\widehat{=}$

PRE

$cell \in CELL \wedge$

$value \in \mathbb{N}$

THEN

IF $cell \notin cells$

THEN

$cells := cells \cup \{ cell \}$

END $\|$

$cellValue (cell) := value \ \|$

IF $cell \in \mathsf{dom} (cellString)$

THEN

$cellString := \{ cell \} \lhd cellString$

END

END ;

SetCellString (*cell* , *string*) $\widehat{=}$

PRE

$cell \in CELL \wedge$

$string \in STRING$

THEN

IF $cell \notin cells$

THEN

$cells := cells \cup \{ cell \}$

END $\|$

$cellString (cell) := string \ \|$

IF $cell \in \mathsf{dom} (cellValue)$

THEN

$cellValue := \{ cell \} \lhd cellValue$

END

END ;

Set Multiple Cell values allows a group of cells to have their values updated at once. This is in keeping with the observer pattern where the state of the observers is updated in parallel

SetMultipleCellValues (*cellvaluefunc*) $\widehat{=}$

PRE

$cellvaluefunc \in cells \pfun \mathbb{N}$

THEN

$cellValue := cellvaluefunc \lhd\!\!+ cellValue$

END ;

Get operations using the standard object model

$returnVal \longleftarrow$ **GetCellValue** (*cell*) $\widehat{=}$

PRE

$cell \in \mathsf{dom}$ (*cellValue*)

THEN

$returnVal := cellValue$ (*cell*)

END ;

$returnStr \longleftarrow$ **GetCellString** (*cell*) $\widehat{=}$

PRE

$cell \in \mathsf{dom}$ (*cellString*)

THEN

$returnStr := cellString$ (*cell*)

END ;

Empty cells are defined those that do not exist in the cells set but are part of the CELL set, to clear a cell, it is removed from the cells set and its references to either a cellValue or a cellString are removed by using domain subtraction

ClearCell (*cell*) $\widehat{=}$

PRE

$cell \in cells$

THEN

SELECT $cell \in \mathsf{dom}\ (\ cellString\ )$

THEN $cellString := \{\ cell\ \} \lhd cellString$

WHEN $cell \in \mathsf{dom}\ (\ cellValue\ )$

THEN $cellValue := \{\ cell\ \} \lhd cellValue$

END $\|$

$cells := cells - \{\ cell\ \}$

END

END

MACHINE *Spreadsheet_CTX* (*length* , *width*)

CONSTRAINTS $length \in \mathbb{N}_1 \ \wedge$

$width \in \mathbb{N}_1$

In the specification we do not wish to include implementation details for how to evaluate a spreadsheet formula. Instead, we will use a constant function *Evaluate* to that just maps a deferred set *FUNCTION* a collection of *cell*,*value* pairs to produce a result.

This will allow the calculation of the spreadsheet formulae result to be implemented or refined in new machines because at the specification stage we are only interested in specifying that cells with a formula should evaluate to a result and that if cells reference other cells and the referenced cells are updated, then the evaluated result of those formula cells may change

SETS *CELL* **;**

FUNCTION **;**

$COMMAND\_TYPE = \{ SET\_VALUE , SET\_STRING , SET\_FORMULA , CLEAR \}$ **;**

COMMAND

CONSTANTS *POSSIBLECELLVALUES* ,

FORMULA ,

EVALUATE

PROPERTIES $\mathsf{card}(CELL) = length \times width \ \wedge$

$POSSIBLECELLVALUES = CELL \times \mathbb{N} \ \wedge$

$FORMULA = FUNCTION \times \mathbb{P}(POSSIBLECELLVALUES) \ \wedge$

$EVALUATE \in FORMULA \rightarrow \mathbb{N}$

END

6.4.4 Case analysis

Unfortunately, due to time constraints, we were not able to present a fully specified formula machine. However, what this case study demonstrates is that design pattern subsystems specified in B can be composed together to produce a large system if the process is followed. It is also an example of what the architecture of a medium sized B system looks like when it is built using design pattern subsystems. What is interesting about this case study is that at no stage was OO abstraction employed before specifying in B. The requirements presented two common problems in requiring updates and undo/redo and design patterns were used directly to solve these problems.

Chapter 7

Conclusion

This thesis set out to explore how concepts and patterns from the OO paradigm can be beneficial in expediting development of complex systems in B. By taking a pattern based approach, we have examined classical design patterns in a formal context, and described how "Objects" can be used as a useful abstraction within B to solve problems. In the process of developing a series of pattern centric case studies, we have exposed several patterns that are specific to B developments, and proposed a B centric pattern taxonomy to assist in the understanding and conceptualisation of B patterns. Following this we have presented these case studies, which demonstrate how the patterns and concepts described in chapters 4 and 5 can be applied to specify and in some cases implement a solution. This culminated in the specification of a more complex system which demonstrates the composition of several B patterns, achieving a robust solution.

Central to this thesis has been the goal of beginning a reference library, based on generic patterns and case studies, which can then be re-used and adapted to different problems. By documenting and researching these patterns in B, we have equipped the novice B practitioner to attack more complex problems with more confidence in less time. By concentrating on only adapting those OO concepts that are useful within the B method, and not attempting to bend B into an OO shape, we have been able to build on the strengths of both B and OO, resulting in a more natural blend of the two methodologies. The majority of our case studies and patterns display both OO and B representations, which will aid an experienced OO practitioner better understand how the patterns they are familiar with can be modelled formally in B.

This thesis is only the beginnings of the work that needs to be undertaken in this area. Many of the patterns presented here have room for improvement, and still more patterns exist that have not been discussed, and should be documented and templated for others to reference. Further to this, it has become obvious that many patterns that are unique to B are yet to be documented. It would also be of great interest to further examine

patterns in the *refinement* and *implementation* phases of B, in particular how the patterns presented in this thesis can be driven toward implementation. Finally, in many cases we have noted that an automation tool to assist in building pattern based machines would be very useful - a tool to assist in building Interface pattern driven developments would be of great use, and we believe a tool to allow the extension of some behavioural and structural patterns would also aid productivity.

Chapter 8

Bibliography

1. Colin Snook, Michael Butler *UML-B: Formal Modelling and Design aided by UML* University of Southampton 2004 UK

2. Sandrine Blazy, Frederic Gervais, Regine Laleau *Reuse of Specification Patterns with the B-Method* Insititut d'Informatique d'Entreprise, Laboratoire CEDRIC, France

3. I. Johnson, C. Snook, A Edmunds, M Butler *Rigorous Development of Reusable, Domain-Specific Components, for complex applications* AT Engine Controls Ltd., Portsmouth, UK. University of Southampton, Southampton, UK

4. Jozef Hooman *Towards Formal Support for UML-based Devlopment of Embedded Systems* University of Nijmegen, 2002 Netherlands

5. Steve Schneider, *The B-Method: An Introduction* Palgrave 2001, New York, NY, USA

6. Christopher Alexander et al, *A Pattern Language*, Oxford University Press 1977, New York, NY USA

7. Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994, Indianapolis, IN USA

8. Simon Bennett et al, *Object-Oriented Systems Analysis and Design using UML, 2nd edition* McGraw-Hill 2002, Berkshire UK

9. John Vlissides *Pattern Hatching: Design Patterns Applied* Addison-Wesley ISBN 0-201-43293-5

10. Ken Robinson *Embedding Formal Development in Software Engineering*

11. UNSW B Resource Site http://www.cse.unsw.edu.au/b@unsw.

12. Wikipedia, *Design Patterns (computer science)* http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29

13. Data and Object Factory http://www.dofactory.com/

14. COMP2111. System Modelling and Development http://www.cse.unsw.edu.au/cs2111

15. COMP4001. Object Oriented Programming http://www.cse.unsw.edu.au/ cs4001

16. Peter Norvig, Design Patterns in dynamic languages http://www.norvig.com/design-patterns

17. John B. Wordsworth. Software Engineering with the B-Method. Addison-Wesley, 1996.