Architectural Design in Object-Z

Tim McComb and Graeme Smith

School of Information Technology and Electrical Engineering,
The University of Queensland
{tjm, smith}@itee.uq.edu.au

Abstract

We present a process for introducing an object-oriented architecture into an abstract functional specification written in Object-Z. Since the design is derived from the specification, correctness concerns are addressed as part of the design process. We base our approach on refactoring rules that apply to class structure, and use the rules to implement design patterns. As a motivating example, we introduce a user-interface design that follows the Model-View-Controller paradigm into an existing specification.

1 Introduction

Specification languages like Object-Z [19], VDM^{++} [13], and Z^{++} [13] provide for structure and modularity in formal specifications by following the object-oriented programming paradigm. It is reasonable to expect that object-oriented designs could be forged from such specification languages using a sequence of step-wise structural transformations that are behaviour preserving. Such a design process has clear benefits: designs that are constructed are guaranteed to conform to the specification, and the design process is clearly documented as a sequence of tractable, repeatable steps.

Refactoring [6] is the term given to the process of remodelling object-oriented software to improve a design whilst preserving its behaviour. Rules for refactoring have been well documented [6] and formalised [2] at the programming language level, but we propose the use of more general rules, similar to refactoring rules, that apply at the specification level to *introduce* a design rather than improve an existing one.

The process of specification-based design, i.e. deriving an architecture from a specification, may be driven by the user or by the specification itself. Actually, these two interpretations are complementary. If user-driven, the developer chooses an appropriate design and works the specification into that design through a series of refinement steps. If specification-driven, the specification strongly influences – or even derives – the architecture [4]. Our approach supports the user-driven process, providing for the introduction of a wide variety of architectures for any particular specification. Although the specification may influence the choice of architecture, it need not.

We use a set of three fundamental refactoring rules for adding, copying and removing structure within a specification in the Object-Z language as a basis for more complicated rules. These rules specify refinements that apply at a structural level and conform to the Object-Z notion of class refinement, as specified by Derrick and Boiten [3], which allows for widening of the interface of a class. That is, the rules are either behaviour preserving, or provide for the introduction of additional behaviour whilst retaining compatibility with the original behaviour.

We describe the manifestation of some popular architectural design patterns, as coined by Gamma et al. [7], in terms of these rules. The use of design patterns helps guide the design process, and in this paper we implement the Model-View-Controller (MVC) [11] architectural paradigm for user-interfaces into a case study by applying the design patterns.

Following a discussion of related work in Section 2, we introduce the MVC paradigm in relation to other architectural design patterns in Section 3. We then look at a particular motivating example where we wish to derive a design that supports a user-interface using MVC in Section 4. We then describe, in Sections 5 and 6, the rules and process required for applying the design process to the case study.

2 Related Work

There is significant work into specification refactoring in the VDM^{++} language, where Lu [16] initially proposed that classes could be decomposed as a refinement step to introduce structure. Goldsack and Lano [8, 14] built upon this work to introduce and formalise *annealing* for decomposing classes in a VDM^{++} specification. The problem of invariant distribution in the decomposition process became the focus of their work, which asserted that invariants could potentially be used to guide the decomposition process [9]. However, a complete approach to object-oriented specification-based design was not developed.

Derrick and Boiten [3] investigated the implementation of the producer/consumer pattern using class refinement in Object-Z, depending upon low-level simulation proofs to demonstrate the specification changes were behaviour preserving. Our approach differs from Derrick and Boiten's as we define higher-level rules that do not necessarily carry this burden of proof.

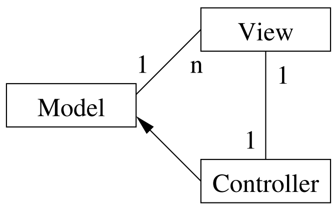
Fowler [6] extensively catalogued object-oriented refactoring rules, but made no attempt to formalise the rules or the process of introducing them. Cornelio et al. [2], however, specified a set of Fowler's refactoring steps as rules in the ROOL language which were verified using a weakest precondition semantics. Their work differs from ours as ROOL does not have a reference-based notion of object identity. Object references are necessary to implement certain design patterns and architectures and are an integral part of many widely-used object-oriented programming languages like Java and C^{++}. Also, our approach has the primary purpose of introducing a design into a specification rather than simply modifying an existing design or implementation, and applies at the specification level rather than the programming language level where Cornelio et al. concentrate their efforts.

Object-oriented design patterns were extensively catalogued, informally, in "Design Patterns: Elements of Reusable Object-Oriented Software" [7]. The MVC architectural paradigm, in particular, emerged earlier as part of the fundamental language design paradigms of SmallTalk [11], and has later evolved to become one of the most popular patterns, underlying Microsoft's document-view architecture in Visual C^{++} [12] and Sun's Java Swing toolkit [1]. Attempts have been made toward the formalisation of such patterns [15, 18], but it is recognised as a difficult task owing to the genericity the patterns require and the awkward mix of static architecture and dynamic behavior that the patterns describe. An approach by Eide et al. [5] attempts to circumvent this by separating static and dynamic structure of patterns allowing for a more precise description. We, however, avoid these problems as we only consider patterns that apply at a purely structural level and preserve behavioral aspects of the system.

3 Architectural Design Patterns

MVC is a too complicated an architecture to easily introduce into any specification as a single-step refactoring rule. Although this is theoretically possible, we desire rules that are general enough to be used to implement a variety of architectures, not just one, and we also want the rules to be as simple as possible. To counter the complexity we decompose MVC into two simpler design patterns. Each of these patterns will be explained briefly below, following an explanation of MVC.

Model-View-Controller The Model-View-Controller paradigm provides for a separation of concerns when attaching a user-interface to an object-oriented system. The abstract data model (the Model) is interfaced with the user through one or more View-Controller pairs. In any View-Controller pair, the View presents an interpretation of the data in the Model to the user and the Controller interacts with the user to change properties of the View and invoke changes upon the Model. The figure below shows how these classes are generally configured.



The relationship from the Model to the View is normally implemented as a simple dependency — the model notifying the view that a change has occurred. In this event, it is then the responsibility of the View to poll the Model to refresh the visual representation.

An MVC based design allows for the substitution of different user-interfaces without modifying the code contained in the Model. It also permits the creation of an arbitrary number of different views over the same data model.

Observer In general, the Observer pattern [7] describes a one-to-many relationship between a subject object and any number of dependent observer objects, such that when the observed subject changes state all the observer objects update themselves accordingly. The View(s) on the Model in the MVC paradigm follow precisely this pattern.

Proxy The Proxy [7] design pattern introduces a proxy class whose objects each maintain a reference to an object of another subject class. Objects of both the Proxy and its associated subject behave identically, as the proxy object simply forwards all operations and attribute references to the subject object at run-time. In MVC, a Controller can be viewed as a proxy to both the Model and the View.

4 Motivating Example: Battleship

The game of "Battleship" was chosen as a motivating example. Two players are required, and the game begins by presenting each player with a blank grid consisting of twelve rows and twelve columns. Each player then secretly places a fleet of five ships on the grid. Each ship is a different length ranging from occupying one cell to five cells, and may be placed either horizontally or vertically.

Players then alternate attempts to sink their opponent's ships by posing a query to their opponent as to whether a certain cell in the opponent's playing grid is occupied by a ship or not (this is analogous to firing a missile). Their opponent responds with either: *hit*, *miss*, *ship sunk*, or *fleet destroyed*. A ship is considered sunk when every cell it occupies in the grid has been guessed by the opposing player. A fleet is considered destroyed when all five ships are sunk and the game ends.

Figure 1 shows an Object-Z specification modelling the core functionality of the game. The class *System* contains a state that holds two references to instances of the *Player* class, $player1$ and $player2$, each of which represents a single player participating in the game. Each player has state information containing a set of *guesses*, a *fleet*, and a reference to the *opponent* player. The *System* class acts as a mediator, controlling the progress of the game. Initially, the operations $InitFleetP1$ and $InitFleetP2$ may be invoked to set up each player's confidential information regarding ship placement. $FireAtP1$ and $FireAtP2$ specify the behavior of each player taking a turn. The delta-lists, i.e., lists of the form $\Delta(\ldots)$, identify which state variables the operations may change; all other variables are unchanged. The parallel operator ($\|$) in the *Fire* operation acts as conjunction, but matches output variables (suffixed by !) on one side of the operator with input variables (suffixed by ?) on the other side, wherever those variables share a common basename. In the *Fire* operation, the value of $fleet!$ in $opponent.UpdateFleet$ is equated to the value of $fleet?$ in *UpdateGuesses*. Similarly, $fleet'!$ is equated with $fleet'?$.

Although the logic of the system is contained primarily in the interaction between the two *Player* instances, the interface to the game is still represented solely by the *System* class. This is the conventional way to structure Object-Z specifications. In Section 6, we introduce classes designed to provide direct interfaces to the *Player* objects. This is done in a step-wise fashion using refactoring rules which we describe in Section 5.

5 Refactoring Rules

Introducing an MVC-based user interface into the Battleship specification requires transformation rules that will manifest the design patterns discussed in Section 3. This

$$SHIP ::= destroyer \mid sub \mid cruiser \mid tanker \mid carrier$$
$$RESPONSE ::= hit \mid miss \mid sink \mid fleet\_destroyed$$
$$COORDINATE == 1 \mathrel{..} 12 \times 1 \mathrel{..} 12$$
$$FLEET == COORDINATE \mathrel{\twoheadrightarrow\!\!\!\!\!\!+} SHIP$$
$$GUESSES == COORDINATE \mathrel{\twoheadrightarrow\!\!\!\!\!\!+} RESPONSE$$

System

$\upharpoonright(\mathrm{INIT}, InitFleetP1, InitFleetP2, FireAtP1, FireAtP2)$

$player1 : Player$
$player2 : Player$
$player1.opponent = player2$
$player2.opponent = player1$

INIT
$player1.\mathrm{INIT}$
$player2.\mathrm{INIT}$

$InitFleetP1 \mathrel{\widehat{=}} player1.InitFleet$
$InitFleetP2 \mathrel{\widehat{=}} player2.InitFleet$
$FireAtP1 \mathrel{\widehat{=}} player2.Fire$
$FireAtP2 \mathrel{\widehat{=}} player1.Fire$

Player

$\upharpoonright(\mathrm{INIT}, InitFleet, Fire, opponent)$

$guesses : GUESSES$
$fleet : FLEET$
$opponent : Player$

INIT
$guesses = \varnothing$
$fleet = \varnothing$

InitFleet
$\Delta(fleet)$
$fleet? : FLEET$
$fleet' = fleet?$

UpdateFleet
$\Delta(fleet)$
$fleet!, fleet'! : FLEET$
$guess? : COORDINATE$
$fleet' = fleet \mathbin{\lhd\!\!\!-} \{guess?\}$
$fleet! = fleet$
$fleet'! = fleet'$

UpdateGuesses
$\Delta(guesses)$
$fleet?, fleet'? : FLEET$
$guess? : COORDINATE$
$guess? \notin \mathrm{dom}\, guesses$
$guesses' = guesses \cup \{guess? \mapsto$
if $fleet'? \subset fleet?$ then
if $\mathrm{ran}\, fleet'? \subset \mathrm{ran}\, fleet?$ then
if $fleet'? = \varnothing$ then
$fleet\_destroyed$
else $sink$
else hit
else $miss\}$

$Fire \mathrel{\widehat{=}} opponent.UpdateFleet \parallel UpdateGuesses$

Figure 1. Battleship Specification

section discusses rules for modifying specification structure to achieve this goal.

We require rules that are capable of implementing the Proxy and Observer patterns. A previous paper [17] introduced rules for adding structure to and removing structure from a specification. The two rules presented were: *annealing*, which adds structure (but no redundancy) to a specification by splitting a class into two classes, and *coalescence* which removes both structure and redundancy by merging classes that exhibit the same behaviour together. The annealing rule can be used to implement the Proxy pattern, and is explained in Section 5.1. However, neither of the rules can implement Observer as both manipulate structure but do not lend well to extending the system (by adding redundancy). Hence, in this paper, we introduce an additional rule, *reflection*, which adds structure and redundancy to a specification by copying a class state. It is described, including a proof of its correctness, in Section 5.2.

All of the rules have been proven to manifest refinements using the class simulation rules defined by Derrick and Boiten [3].

5.1 Annealing Rule

The annealing rule splits a class into two classes by partitioning the state and operations. The resulting structure consists of a *frame* class, containing one partition of the state information with related operations, together with a *component* class that contains the rest of the state and operations (see Figure 2). The interface that the framing class offers the rest of the system is compatible with the original class, and all references to the original class are replaced by references to the new framing class. The frame class holds a reference to the component class, such that the component class is effectively hidden from the rest of the system.

The backward reference, from the component class to the faming class, exists so that structures added to the component class at a later stage in the design process have access to the framing class if necessary. Also, it is made possible for operations in both classes to access variables that do not exist locally within a class through the use of the backward reference. In particular, state variables T are available (though not changeable) by operations in A_1 through the state invariant that matches the variable bindings θT of the local definition of T in A_1 with the bindings of $component.T$. Similarly, bindings of state variables S are available to class B via a dereference of the *frame* variable. In Figure 2, the notation $P_{\{x\}}$ signifies a predicate P that may reference a free variable x. Thus, $P_{\{S,S',T\}}$ may reference both S and T, but may only change S.

The precondition for applying the annealing rule to a class A with state S and T requires that all operations that affect T do not reference operations that affect S (the $\Delta(S)$

A

$\upharpoonright(\mathrm{INIT}, S, T, OpA, OpB)$

State: S; T

INIT: $R_{\{S,T\}}$

$OpA \mathrel{\widehat=} [\, \Delta(S) \mid P_{\{S,S',T\}} \,]$

$OpB \mathrel{\widehat=} [\, \Delta(T) \mid Q_{\{S,T,T'\}} \,]$

$$\sqsubseteq$$

A_1

$\upharpoonright(\mathrm{INIT}, S, T, OpA, OpB)$

State: S; $component : B$; Δ; T

$\theta T = \theta component.T$

$component.frame = self$

INIT: $R_{\{S,T\}}$; $component.\mathrm{INIT}$

$OpA \mathrel{\widehat=} [\, \Delta(S) \mid P_{\{S,S',T\}} \,]$

$OpB \mathrel{\widehat=} component.OpB$

B

$\upharpoonright(\mathrm{INIT}, frame, OpB)$

State: T; $frame : A_1$; Δ; S

$\theta S = \theta frame.S$

INIT: $R_{\{S,T\}}$

$OpB \mathrel{\widehat=} [\, \Delta(T) \mid Q_{\{S,T,T'\}} \,]$

Figure 2. Annealing Rule

notation indicates that an operation may change S). This is so that operations that affect T can be moved into the component class B without reliance upon the backward reference. Operations that are moved into the component class but required by the frame class A_1 (because they are exposed by the interface, or referenced by an operation of the frame class) are redefined in the frame class to reference the operation in the component class.

A special case of the annealing rule exists, where *every* variable and operation is moved into the component class. When this is the case, the requirement that references to the original class be replaced by references to the framing class is relaxed, and these references may be directed at either the framing class or the component class.

The specific details of the rule, and a proof of its correctness, are presented in [17] but, for brevity, are omitted from this presentation. There are conditions, however, that apply to class simulation refinements with regard to the context of the class being substituted. These conditions upon compositionality, provided by Derrick and Boiten [3], stipulate that:

- There must not be direct access from the context to state variables in the class to be refined.
- Schema composition ($\fatsemi$) must not be used within the context to constrain the class to be refined.

The first of these conditions is always met by the refactoring rules presented in this paper, as the rules do not alter the representation of the state (except to, perhaps, extend it). The second condition, however, must be checked whenever a rule is applied to ensure that the refinement is substitutable for the original class.

A

$\upharpoonright(\mathit{INIT}, S, \mathit{OpA})$

S

INIT: Q_S

$\mathit{OpA} \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,]$

$$\sqsubseteq$$

A_1

$\upharpoonright(\mathit{INIT}, S, \mathit{OpA})$

S
$\mathit{component} : B$
$\mathit{component.frame} = \mathit{self}$

INIT: Q_S
$\mathit{component.INIT}$

$\mathit{OpA} \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,] \land \mathit{Notify}$
$\mathit{Notify} \mathrel{\widehat{=}} \mathit{ExportState} \parallel \mathit{component.Update}$
$\mathit{ExportState} \mathrel{\widehat{=}} [\, S! : S \mid \theta S! = \theta S' \,]$

B

$\upharpoonright(\mathit{INIT}, \mathit{Update}, \mathit{frame})$

S
$\mathit{frame} : A_1$

INIT: Q_S

$\mathit{Update} \mathrel{\widehat{=}} \mathit{ImportState}$
$\mathit{ImportState} \mathrel{\widehat{=}}$
$[\, \Delta(S)\ S? : S \mid \theta S' = \theta S? \,]$

Figure 3. Reflection Rule

5.2 Reflection Rule

The annealing and coalescence rules are sufficient for us to introduce many different designs into a given specification, including the Proxy pattern described in Section 3, but they are not powerful enough to derive MVC as the Observer pattern cannot be implemented. The reason for this is that the two rules provide for structure modification by reorganising state information, operations, and classes, but offer no means for adding redundancy to a system or expanding a system at a structural level.

The reflection transformation (see Figure 3) adds this redundancy by introducing a component class whose objects contain a state that exactly corresponds to the state of objects belonging to the framing class. The communication and synchronisation of state data between objects is achieved by the invocation of a *Notify* operation which retrieves the changed state of the framing object and commits this state to the dependent object. Every operation in the framing object that is exposed by the interface and possibly modifies the state is conjoined with a *Notify* operation that performs this retrieve/communicate/commit step, which for example, if *component* is the reference to the dependent object, is:

$$\mathit{Notify} \mathrel{\widehat{=}} \mathit{ExportState} \parallel \mathit{component.Update}$$

This rule is particularly useful when class refinements (as defined by Derrick and Boiten [3]) are applied to the introduced dependent classes. This way, the superfluous state information and operations that are introduced can be refined to specialise the dependent objects. This process provides a means for introducing distributed architectures requiring communication and synchronisation of data between objects at runtime.

In Figure 3, the *frame* variable in the component class is not used by the component class, but it provides the ability for the component class to access the framing class if required by future refinements, which is often useful.

5.2.1 Proof Sketch

We first must form a single class $(A_1 \bullet B)$ representing the composition of classes A_1 and B on the right-hand side of the rule. To achieve this, we use the method proposed by Smith [19] and Derrick and Boiten [3]:

$$
\begin{array}{|l}
\hline
(A_1 \bullet B) \\
\upharpoonright(\mathit{INIT}, OpA) \\
\quad \begin{array}{|l}
\hline
S \\
component.S \\
component.frame : (A_1 \bullet B) \\
\hline
component.frame = self \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\mathit{INIT} \\
Q_S \\
component.Q_{\{component.S\}} \\
\hline
\end{array} \\
OpA \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,] \wedge Notify \\
Notify \mathrel{\widehat{=}} ExportState \parallel component.Update \\
ExportState \mathrel{\widehat{=}} [\, S! : S \mid \theta S! = \theta S' \,] \\
component.Update \mathrel{\widehat{=}} component.ImportState \\
component.ImportState \mathrel{\widehat{=}} \\
\qquad [\, \Delta(component.S)\ S? : component.S \mid \\
\qquad\ \ \theta component.S' = \theta S? \,] \\
\hline
\end{array}
$$

This can be simplified substantially by removing operations that are not exposed by the interface and substituting their definition in-line.

$$
\begin{array}{|l}
\hline
(A_1 \bullet B) \\
\upharpoonright(\mathit{INIT}, OpA) \\
\begin{array}{|l}
\hline
\ldots \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\mathit{INIT} \\
\ldots \\
\hline
\end{array} \\
OpA \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,] \wedge \\
\qquad ([\, S! : S \mid \theta S! = \theta S' \,] \parallel \\
\qquad\ [\, \Delta(component.S)\ S? : component.S \mid \\
\qquad\ \ \theta component.S' = \theta S? \,]) \\
\hline
\end{array}
$$

Using the definition of the parallel operator [19], and the fact that we know *component*.*S* is the same schema as S, and therefore represents the same type:

$$
\begin{array}{|l}
\hline
(A_1 \bullet B) \\
\ldots \\
OpA \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,] \wedge \\
\qquad [\, \Delta(component.S)\ z : S \\
\qquad\ \mid \theta z = \theta S' \wedge \theta component.S' = \theta z \,] \setminus (z) \\
\hline
\end{array}
$$

Above, z is a fresh variable introduced to denote the value communicated by the schemas composed with the parallel operator. It is hidden in the resulting operation.

We now apply the one-point rule to remove the z variable in *OpA*:

$$
\begin{array}{|l}
\hline
(A_1 \bullet B) \\
\ldots \\
OpA \mathrel{\widehat{=}} [\, \Delta(S) \mid P_{\{S,S'\}} \,] \wedge [\, \Delta(component.S) \\
\qquad\qquad\qquad\qquad\qquad \mid \theta S' = \theta component.S' \,] \\
\hline
\end{array}
$$

Schema conjunction in Object-Z can be rewritten as logical conjunction [19]:

$$
\begin{array}{|l}
\hline
(A_1 \bullet B) \\
\ldots \\
OpA \mathrel{\widehat{=}} [\, \Delta(S, component.S) \\
\qquad\quad \mid P_{\{S,S'\}} \wedge \theta S' = \theta component.S' \,] \\
\hline
\end{array}
$$

To prove that the reflection rule is behaviour preserving (i.e., a refinement), we adopt Derrick and Boiten's [3] approach to class simulation to establish that $(A_1 \bullet B)$ refines A.

For a class simulation to exist, that is, for the class $(A_1 \bullet B)$ simulate the behaviour of class A, it is required that each operation in $(A_1 \bullet B)$ is *applicable* whenever the corresponding operation from class A is applicable. Also, we must show that the application of any operation in $(A_1 \bullet B)$ is *correct* with respect to the corresponding operation in class A, where correctness is defined as adherence to a *retrieve* relation upon states.

For the reflection rule, the retrieve relation R, which relates the states of $(A_1 \bullet B)$ and A, is defined as such:

$$R \mathrel{\widehat{=}} [\, A;\ (A_1 \bullet B) \mid \theta A.S = \theta(A_1 \bullet B).S \,]$$

To establish a class simulation, the applicability and correctness of *OpA* in $(A_1 \bullet B)$ needs to be shown with respect to the corresponding operation in A. The class simulation laws provide the necessary proof obligations. For $(A_1 \bullet B).OpA$ applicability requires:

$$
\begin{array}{l}
\forall A.\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE} \bullet \\
\qquad R \Rightarrow (\operatorname{pre} A.OpA \Leftrightarrow \operatorname{pre}((A_1 \bullet B).OpA))
\end{array}
$$

And correctness requires:

$$
\begin{array}{l}
\forall A.\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE}' \bullet \\
\qquad R \wedge (A_1 \bullet B).OpA \Rightarrow (\exists A.\mathit{STATE}' \bullet R' \wedge A.OpA)
\end{array}
$$

When fully expanded to include state invariant information, $(A_1 \bullet B).OpA$ is:

$$
\begin{array}{|l}
\hline
(A_1 \bullet B).OpA \\
\Delta((A_1 \bullet B).S) \\
\hline
P_{\{(A_1 \bullet B).S, (A_1 \bullet B).S'\}} \\
\theta(A_1 \bullet B).S = \theta(A_1 \bullet B).component.S \\
\theta(A_1 \bullet B).S' = \theta(A_1 \bullet B).component'.S' \\
(A_1 \bullet B).component.frame = (A_1 \bullet B).self \\
(A_1 \bullet B).component'.frame' = (A_1 \bullet B).self' \\
\hline
\end{array}
$$

Using this operation definition, the applicability proof obligation for $(A_1 \bullet B).OpA$ expands to:

$$\begin{array}{l}
\forall A.\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE} \bullet \\
\quad \theta A.S = \theta(A_1 \bullet B).S \Rightarrow \\
\quad\ ((\exists A.\mathit{STATE}' \bullet P_{\{A.S,A.S'\}}) \Leftrightarrow \\
\quad\ \ (\exists (A_1 \bullet B).\mathit{STATE}' \bullet \\
\quad\quad P_{\{(A_1 \bullet B).S,(A_1 \bullet B).S'\}} \wedge \\
\quad\quad \theta(A_1 \bullet B).S = \theta(A_1 \bullet B).component.S \wedge \\
\quad\quad \theta(A_1 \bullet B).S' = \theta(A_1 \bullet B).component'.S' \wedge \\
\quad\quad (A_1 \bullet B).component.frame = (A_1 \bullet B).self \wedge \\
\quad\quad (A_1 \bullet B).component'.frame' = (A_1 \bullet B).self'))
\end{array}$$

Using the invariant of $A_1.\mathit{STATE}$ and $A_1.\mathit{STATE}'$:

$$\begin{array}{l}
\forall A.\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE} \bullet \\
\quad \theta A.S = \theta(A_1 \bullet B).S \Rightarrow \\
\quad\ ((\exists A.\mathit{STATE}' \bullet P_{\{A.S,A.S'\}}) \Leftrightarrow \\
\quad\ \ (\exists (A_1 \bullet B).\mathit{STATE}' \bullet P_{\{(A_1 \bullet B).S,(A_1 \bullet B).S'\}}))
\end{array}$$

Since we can assume the bindings between pre-state variables are the same, the two existentially quantified predicates are equivalent. This is reliant upon the assumption that, for a given pre-state, post-states either exist or do not exist for both A and $(A_1 \bullet B)$. Since invariants in $(A_1 \bullet B)$ do not add restrictions to the possible value of S', this is acceptable. As the consequent of the implication is true by assuming the antecedent, the statement is also true by inspection.

Correctness for $(A_1 \bullet B).OpA$ expands to:

$$\begin{array}{l}
\forall A.\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE};\ (A_1 \bullet B).\mathit{STATE}' \bullet \\
\quad \theta A.S = \theta(A_1 \bullet B).S \wedge \\
\quad P_{\{(A_1 \bullet B).S,(A_1 \bullet B).S'\}} \wedge \\
\quad \theta(A_1 \bullet B).S = \theta(A_1 \bullet B).component.S \wedge \\
\quad \theta(A_1 \bullet B).S' = \theta(A_1 \bullet B).component'.S' \wedge \\
\quad (A_1 \bullet B).component.frame = (A_1 \bullet B).self \wedge \\
\quad (A_1 \bullet B).component'.frame' = (A_1 \bullet B).self' \Rightarrow \\
\quad\quad (\exists A.\mathit{STATE}' \bullet \theta A.S' = \theta(A_1 \bullet B).S' \wedge \\
\quad\quad\quad\quad P_{\{A.S,A.S'\}})
\end{array}$$

Consider the consequent of the above implication, where we choose an $A.\mathit{STATE}'$ such that $\theta A.S' = \theta(A_1 \bullet B).S'$. $P_{\{A.S,A.S'\}}$ is shown by noting that $P_{\{(A \bullet B).S,(A \bullet B).S'\}}$ is an hypothesis and noting $\theta A.S = \theta(A_1 \bullet B).S$ is also an hypothesis. This confirms that an $A.\mathit{STATE}'$ exists rendering the entire predicate true.

6 Evolution of Case Study

In this section we describe the evolution of the motivating example presented in Section 4. We utilise the reflection rule to implement the Observer pattern, which allows us to introduce Views to the Battleship system. Since each player should have an independent user-interface (the game requires that players cannot see each other's game state), we apply the reflection rule to the *Player* class to manifest a *View* upon each instance of *Player*. Figure 4 shows the resultant changes to the *Player* class to become $Player_1$, together with the newly introduced *View* class.

The proof of the reflection rule in Section 5.2.1 shows that any application of it will result in a refinement, and, as a context for the refinement, the *System* class meets the requirements set out by Derrick and Boiten (refer to Section 5.1) with respect to the absence of forward composition ($\fatsemi$). Thus, the classes $Player_1$ and *View* as a pair are substitutable for *Player* without the need to discharge further proof obligations.

The two operations in *Player* that alter the state and are exposed by the interface, *InitFleet* and *Fire*, are conjoined with the *Notify* operation that updates the associated *View*.

The *View* class can be extended to do more than reflect the immediate state of a *Player* object, and Figure 5 shows a possible refinement of the *View* class that specialises it. Here, in the class $View_1$, the history of board locations that a player has guessed is maintained in an injective sequence named *guessHistory*. We extend the *Update* operation to alter the *guessHistory* variable via *UpdateHistory* every time the player changes state. If the set of *guesses* remains unchanged after the state change notification, then *guessHistory* also remains unchanged. Otherwise, the most recent board location guessed is added to the end of the sequence of guesses in *guessHistory*. We have also added an operation *ClearHistory* that will clear the sequence of locations in the history buffer to start afresh. It is evident by inspection that $View_1$ is a refinement of the *View* class, as the operations added affect only the *guessHistory* state variable, which is unrelated to the existing set of state variables. Also, the operations don't constrain the occurrence of existing operations, as the precondition of *UpdateHistory* is *true*.

In Figure 6, we show the result of implementing the Proxy pattern into the case study to yield a *Controller* class by using the annealing rule. In this case, by applying the annealing rule to the $View_1$ class, we modified the $View_1$ class to become $View_2$, and forged the new *Controller* class. Like the use of the reflection rule above, adherence to the annealing rule allows us to avoid proof obligations regarding the suitability of $View_2$ and *Controller* as a refinement of $View_1$.

The *Controller* acts as a proxy to $View_2$, and effectively to $Player_1$, since the $View_2$ class contains a backward reference. This illustrates the importance of backward references in some circumstances for tracing back references to core objects in the system. Without the backward reference in this instance, objects of the $Player_1$ class associated with the $View_2$ would not be accessible by the *Controller*.

We give the *Controller* class access to everything that a player would be interested in interacting with. This includes the operations *InitFleet* and *Fire* in $Player_1$ (the *player* reference is accessed via *view*), and the operation *ClearHistory* which applies to the $View_2$.

Player$_1$

$\upharpoonright(\text{INIT}, system, InitFleet, Fire, opponent)$

$\ldots$
$view : View$

$\ldots$

$$InitFleet \mathrel{\widehat=} [\,\Delta(fleet)\; fleet? : FLEET \mid fleet' = fleet?\,] \wedge Notify$$

$$Fire \mathrel{\widehat=} (opponent.UpdateFleet \parallel UpdateGuesses) \wedge Notify$$

$$Notify \mathrel{\widehat=} ExportState \parallel view.Update$$

ExportState

$guesses! : GUESSES$
$fleet! : FLEET$

$guesses! = guesses'$
$fleet! = fleet'$

View

$\upharpoonright(Update, player)$

$guesses : GUESSES$
$fleet : FLEET$
$opponent : Player_1$
$player : Player_1$

INIT

$guesses = \varnothing$
$fleet = \varnothing$

$$Update \mathrel{\widehat=} ImportState$$

ImportState

$\Delta(guesses, fleet)$
$guesses? : GUESSES$
$fleet? : FLEET$

$guesses' = guesses?$
$fleet' = fleet?$

Figure 4. View class

View$_1$

$\upharpoonright(Update, player, ClearHistory)$

$\ldots$
$guessHistory : \operatorname{iseq} COORDINATE$

INIT

$\ldots$
$guessHistory = \varnothing$

$$Update \mathrel{\widehat=} ImportState \wedge UpdateHistory$$

ImportState

$\ldots$

UpdateHistory

$\Delta(guessHistory)$

$$guesses' \supset guesses \Rightarrow (\exists x : COORDINATE \bullet x \in \operatorname{dom} guesses' \setminus \operatorname{dom} guesses \wedge guessHistory' = guessHistory \frown \langle x \rangle)$$
$$guesses' \not\supset guesses \Rightarrow guessHistory' = guessHistory$$

ClearHistory

$\Delta(guessHistory)$

$guessHistory' = \varnothing$

Figure 5. View$_1$ class: one possible variation of the View class

System
$\ldots$

Player$_1$
$\ldots$

View$_2$
$\ldots$

$\ldots$
$controller : Controller$

$\ldots$
$controller.view = self$

$\ldots$

Controller
$\upharpoonright(InitFleet, Fire, view)$

$view : View_2$
$player : Player_1$

$player = view.player$

$InitFleet \mathrel{\widehat=} player.InitFleet$
$Fire \mathrel{\widehat=} player.Fire$
$ClearHistory \mathrel{\widehat=} view.ClearHistory$

Figure 6. Complete system

An interesting effect of introducing a user-interface like this to a specification is that the user interaction is no longer intended to be through the *System* class – although this is still possible – but via the *View*$_2$ and *Controller* classes. This diverges from the canonical method for structuring specifications in Object-Z, as information regarding the specifics of design are normally disregarded.

7 Conclusions and Future Work

In this paper we have presented a process for introducing an object-oriented architecture into an abstract functional specification written in Object-Z. This enabled us to implement, in a step-wise fashion, a user interface that followed the Model-View-Controller paradigm into a case study. The rules presented provide a means for changing the structure of a specification whilst ensuring that the resultant specification is a refinement of the original.

Tool support for applying the rules is possible, and has been prototyped by formalising the rules in the *Z* specification language as operations upon a state containing the Object-Z specification (a meta-model). The rules are written as schemas that are executable by a specification animator, so the user may interact with the animation software to automate the refactoring process. The evolution of the case study presented in this paper has been successfully automated using this technique, taking advantage of the Possum animator [10]. The use of an animator readily allows for the searching of possible state spaces (designs), which is very helpful for identifying candidate classes for application of the rules.

Whilst we only discuss architectural design in this paper, there are many design patterns that pertain to dynamic behaviour. As an area of future work, we intend to investigate a mapping of specification patterns to behavioural design patterns, and establish a supporting methodology for the Object-Z language.

References

[1] Sun Java technology, `http://java.sun.com/`, 2003.
[2] M. Cornelio, A. Cavalcanti, and A. Sampaio. Refactoring by Transformation. In *Proc. of REFINE'2002*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2002.
[3] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT Series. Springer, May 2001.
[4] R. G. Dromey. From requirements to design: Formalizing the key steps. In *Intl. Conference on Software Engineering and Formal Methods (SEFM '03)*, pages 2–11. IEEE Computer Society Press, 2003.
[5] E. Eide, A. Reid, J. Regehr, and J. Lepreau. Static and dynamic structure in design patterns. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 208–218, Orlando, FL, May 2002. ACM Press.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] S. Goldsack and K. Lano. Annealing and data decomposition in VDM. *ACM Sigplan Notices*, 31(4):32–38, Apr. 1996.

[9] S. Goldsack and K. Lano. Invariants as design templates in object-based systems. In *VDM in Practice*, pages 13–27, Sept. 1999.

[10] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the SUM specification language. In W. Wong and K. Leung, editors, *Asia Pacific Software Engineering Conference (APSEC 97)*, pages 42–51. IEEE Computer Society, 1997.

[11] T. Hopkins and B. Horan. *Smalltalk: An Introduction to Application Development Using VisualWorks*. Prentice-Hall, 1995.

[12] D. Kruglinski, S. Wingo, and G. Shephard. *Programming Microsoft Visual C++*. Microsoft Press, 5 edition, 1998.

[13] K. Lano. *Formal Object-oriented Development*. Springer Verlag, 1995.

[14] K. Lano and S. Goldsack. Refinement of distributed object systems. In *Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems*. Chapman and Hall, 1996.

[15] K. Lano and N. Malik. Mapping procedural patterns to object-oriented design patterns. *Automated Software Engineering: An International Journal*, 6(3):265–289, July 1999.

[16] J. Lu. Introducing data decomposition into VDM for tractable development of programs. *ACM SIGPLAN Notices*, 30(9), Sept. 1995.

[17] T. McComb. Refactoring Object-Z specifications. In *FASE '04: Fundamental Approaches to Software Engineering*. In press, Springer-Verlag, 2004.

[18] T. Mikkonen. Formalizing design patterns. In *Proc. of the 1998 International Conference on Software Engineering*, pages 115–124. IEEE Comput. Soc, 1998.

[19] G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.