Refactoring Object-Z Specifications

Tim McComb

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

Abstract. Object-Z offers an object-oriented means for structuring formal specifications. We investigate the application of refactoring rules to add and remove structure from such specifications to forge object-oriented designs. This allows us to tractably move from an abstract functional description of a system toward a lower-level design suitable for implementation on an object-oriented platform.

1 Introduction

Previous research added object orientation to specification languages to take advantage of the modularity and structure benefits that this approach offers. Whilst this has been achieved with formalisms such as Object-Z [22], VDM⁺⁺ [11], and Z⁺⁺ [11], which are built upon existing well-defined languages, there is not yet a practical process for turning these functional specifications into well-structured object-oriented software. This seems like an intuitively appropriate goal for such specification languages, as their semantics already closely model that of object-oriented programming languages.

There are two major aspects such a process must encompass. First, at a high level, a purely functional specification must be reorganised and refined to create a reasonable object-oriented design. Second, at a lower level, the object-oriented design must be further refined to object-oriented code.

We focus on the higher level objective: moving from an abstract functional specification in Object-Z to a specification of an object-oriented design. We achieve this by systematically modifying the specification to introduce design, taking advantage of rules that are similar to the *refactoring* [6,18] rules presented by Fowler [6]. Although we borrowed the term, our approach slightly differs from the normal interpretation of refactoring as the process of improving an existing design.

Fowler extensively catalogued object-oriented refactoring rules, but made no attempt to formalise the rules or the process of introducing them. Cornelio et al. [4], however, specified a set of Fowler's refactoring steps as rules in the ROOL programming language which were verified using a weakest precondition semantics.

Our idea differs from these approaches as we move from an abstract specification to an object-oriented design without the need to consider irrelevant detail at the programming-language level. Also, the ROOL language does not have a

reference-based model 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⁺⁺. In addition to this, our approach differs from others as it has the primary purpose of introducing a design into a specification rather than simply modifying an existing implementation.

In this paper we show that significant structural object-oriented design can be introduced at the specification level, in Object-Z, utilising two rules. One rule adds structure to the specification, whilst another rule removes structure. When combined with non-structural refactoring steps our approach a powerful method for the incremental introduction of specification-based structural design in Object-Z. Since the rules are behaviour-preserving, this process ensures that any software design produced will be correct with respect to the original specification.

2 Related Work

Refactoring is a relatively new idea as applied to object-oriented programming languages, but the systematic transformation of formal specifications to resemble programs, particularly in the non-object-oriented domain, is certainly not a new idea [13,17].

However, formal and incremental class reorganisation in object-oriented languages was discussed by Casais [3] and also by Mikhajlova and Sekerinski [15], who propose a class refinement semantics for object-oriented specification languages based upon data refinement. This involves the interpretation of subclasses as subtypes, this semantic definition going beyond the Object-Z interpretation of inheritance as strongly syntax-based [21]. This approach, however, does simplify refinement a great deal — as in VDM⁺⁺ and Z⁺⁺ [11] — but restricts the ease of expression in the languages because it inhibits reuse of syntactic definition.

Other investigations into modifying class structure in a formal context were made by Bonsangue et al. [1,2]. Here, object orientation was modelled by extending the action system formalism to support objects and classes. This formalism differs from Mikhajlova and Sekerinski's, even though it has the same interpretation of inheritance, as it considers the addition of methods when subclassing. However, it still considers inheritance as a form of subtyping.

There is significant work in this field in the VDM⁺⁺ language, where Lu [12] initially proposed that classes could be decomposed as a refinement step to introduce structure. Goldsack and Lano [7,10] 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, and a complete approach to specification-based object-oriented design was not developed.

3 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.

In Figure 1 we present a specification of the game written in Object-Z. The class *System* represents the two players in terms of their guesses (*guessesP1* and

Fig. 1. Initial monolithic Battleship specification.

guessesP2) and fleet positions (*fleetP1* and *fleetP2*). Each player provides the initial placement of their ships on the game board using the *InitFleetP1* and *InitFleetP2* operations. The *FireAtP1* and *FireAtP2* operations specify the behaviour of each player taking a turn. The delta-lists, i.e., lists of the form $\Delta(\dots)$, identify which state variables the operations may change; all other variables are unchanged.

Our specification is adequate to describe the core functionality of the game, but does not represent an appropriate object-oriented design. A good object-oriented design, and indeed a better structured specification, would identify players as separate objects in the system and encapsulate their functionality; thus reducing the clear redundancy. This paper describes a means for achieving this goal via the step-wise application of behaviour-preserving rules.

We shall introduce, in Section 4, the general rules for modifying the structure of an Object-Z specification. Using these rules, we illustrate their effectiveness in Sections 5 and 6, showing their application to reflect different design decisions whilst preserving the specified behaviour.

4 Structural Modification Rules

Our rules below describe syntactic transformations on Object-Z specifications. The rules rely on class simulation [5] to establish refinement relations between the two sides of the rules to ensure the preservation of behaviour. Refinement is interpreted with respect to a class's external interface, and we allow for the interface of a class to be widened in the refinement process.

We have proven both of the following rules correct by showing that the right-hand side is a refinement of the left-hand side using the definitional rules of Object-Z provided by Smith [22] together with Derrick and Boiten's class simulation laws [5]. Proof sketches are provided in Section 4.3.

For a complete approach, it is necessary to consider other rules that do not modify class structure. Such rules rename variables or operations in classes, simplify or reorganise operation expressions, or involve other equivalence transformations that are proven sound using the language definition. These rules are not elaborated upon for purposes of brevity, but are based upon those presented by Cornelio et al. [4] and Fowler [6].

In the following sections we present the Annealing rule for introducing class structure, and the Coalescence rule for eliminating class structure.

4.1 Annealing

The annealing rule allows for the decomposition of one class into two, effectively partitioning its data and functionality. It is similar in intention to Fowler's *Extract Class* refactoring [6,4], which performs the same function at the programming language level. The annealing rule is fundamental for introducing structure as part of the process of moving from specification towards design. In Figure 2, class *A* on the left-hand side is annealed to create the classes *A*₁ and *B* on the right-hand side.

Fig. 2. Annealing rule.

One class, called the *framing* class (represented in Figure 2 by A_1), holds a reference to an object of the *component* class (represented in Figure 2 by B). The interface of the original class (class A in Figure 2) is extended by the framing class, and it is the responsibility of the framing class to invoke operations upon, and manage the state of, the component class. Any references to the original class from the rest of the system are replaced by references to the new framing class.

The intention is to divide the state (S ; T) of the class A between two classes: one with state S and one with state T . S and T represent schemas that conjoin to form the state of class A . The predicates P , Q , and R have subscripts indicating the sets of variables that may appear free in them.

Ancestors of the class A are inherited by the newly formed classes A_1 and B . The actual set of classes inherited in each case corresponds with the partition (S or T) that the inherited classes contribute to. Any inherited classes that share a common ancestor must contribute to the same partition. This way, after the rule application, A_1 and B inherit only the classes that contribute to their respective states. Any class in the specification that was a descendant of the original class A must now inherit both A_1 and B .

A further precondition for performing the annealing rule is that every operation in the class to be partitioned *explicitly* changes, if any variables at all, either variables declared in S or variables declared in T , but not both (illustrated by the $\Delta(S)$ and $\Delta(T)$ notation). Operations that make no state changes may appear in either or both classes wherever referenced.

To achieve this precondition concerning operations, any operation schema that changes variables in both S and T must be split (using schema operators) such that the predicates that change variables in S are in a different operation from those that change a variable in T . This can be achieved in most specifications by promoting logical operators to schema operators, and moving schemas into new, separate operations and referencing the new operations from the original one. This is in the spirit of the *Extract Method* [6] refactoring step which splits a programming language procedure into two, where one contains a procedure call to the other.

For example, in Object-Z:

$$X \triangleq [\Delta(S, T) \mid P_{\{S, S', T\}} \wedge Q_{\{S, T, T'\}}]$$

is equivalent to, by promoting logical conjunction to schema conjunction:

$$X \triangleq [\Delta(S) \mid P_{\{S, S', T\}}] \wedge [\Delta(T) \mid Q_{\{S, T, T'\}}]$$

which, by introducing an operation reference, is equivalent to:

$$\begin{aligned} X &\triangleq [\Delta(S) \mid P_{\{S, S', T\}}] \wedge Y \\ Y &\triangleq [\Delta(T) \mid Q_{\{S, T, T'\}}] \end{aligned}$$

Operations that modify variables in S must remain in class A_1 , and operations that modify variables in T must be moved to the component class B . Operations that are moved into the component class must, in a similar way to the *Move Method* [6] refactoring step, be redefined in the framing class to dereference the component operation, e.g.

$$Y \triangleq \text{component}.Y$$

Operation references need to be contained only in the operations that change S (as above). This is so the references can be modified to dereference *component* upon application of the rule.

In the case where predicates cannot be readily split to cater for the segmentation of state, input and output variables may be added to the operation schemas to communicate variable values across the schema boundary using Object-Z's parallel ($\parallel$) operator. The parallel composition operator binds together output variables on one side (suffixed by $!$) with input variables on the other side (suffixed by $?$) where the variables have the same base name.

The default action of the rule is to make all variables that were accessible in the original class visible to both classes in the substituted system. In Figure 2, this is represented by the inclusion of the respective foreign schema in the states of A_1 and B underneath the delta (Δ) symbol. Any declaration that appears underneath the delta in the state of an Object-Z class may have its value changed by any operation. The state invariant, for example $\theta T = \theta \text{component}.T$ in class A_1 , ensures that the declared schema T is always synchronised with variable bindings of the coupled object's state *component.T*. This does create a high degree of coupling as references are required in both directions, however un-referenced variables may be removed after application of the rule eliminating unnecessary coupling between the two classes.

Fig. 3. Coalescence rule, assuming $A \sqsubseteq C$ and $B \sqsubseteq C$.

4.2 Coalescence

Coalescence (see Figure 3) is a rule that allows two classes A and B to be replaced with one class C , so long as A and B are both refined by C (denoted $A \sqsubseteq C$ and $B \sqsubseteq C$ respectively).

This approach allows the designer to extract common functionality from possibly independent parts of the system, perhaps using annealing, and then coalesce this functionality to form a common library or object-type. Because annealing is unidirectional and structure adding, this rule is important for structure reduction.

4.3 Proof Sketches of Rules

Annealing. To prove that the annealing rule is behaviour preserving (i.e., a refinement), we adopt Derrick and Boiten's [5] approach to class simulation.

For a class simulation to exist, that is, for the classes A_1 and B to together simulate the behaviour of class A , it is required that each operation in the system containing A_1 and 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 and 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 annealing rule Figure 2, the retrieve relation, which relates the states of A_1 (which indirectly includes B) with A , is defined as such:

$$R \triangleq [A.STATE; A_1.STATE \mid \theta A.S = \theta A_1.S \wedge \theta A.T = \theta A_1.T]$$

To establish a class simulation, the applicability and correctness of every operation in A_1 needs to be shown with relation to the corresponding operation in A . The class simulation laws provide the necessary proof obligations. We demonstrate the use of these proof obligations for OpA in the appendix.

The arguments for the applicability and correctness of OpB are similar, but for purposes of brevity are not presented.

Coalescence. The coalescence rule is proven by showing that C in Figure 3 is a refinement of both A and B , and is therefore substitutable for both. When the

rule is applied these refinements must be shown to hold, except for the trivial case where A is exactly the same as B . This trivial case always holds as a class is always a refinement of itself. Therefore, if two classes are identical, each is a refinement of the other and may thus be replaced by a single class which is identical to both.

5 Restructuring for Passive Players

In this section we systematically add a *Player* class to the monolithic *System* class in Figure 1 using the rules presented in Section 4.

ANNEALING

We first apply annealing steps to extract the functionality of the two players into two separate classes. This requires two applications of the rule, so the state variables of *System* must be partitioned into two groups, each group representing the data for a single player. These groups combined contain the entire state information for *System*; consisting of *guessesP1* with *fleetP1*, and *guessesP2* with *fleetP2*.

The initialisation condition is quite easily distributed between the two classes as it is a simple conjunction of predicates relating to independent parts of the state. The annealing rule requires that every operation in *System* that changes variables in both partitions be split, adding operation references where necessary. *InitFleetP1* and *InitFleetP2* require no change as this requirement already holds, but both *FireAt* operations need to be altered.

Progress can be made using logical operator promotions on *FireAtP1* (and similarly *FireAtP2*), but the conjunct in *FireAtP1* responsible for updating the *guessesP2* variable cannot be split by simply promoting conjunction since it also contains references to *fleetP1* and *fleetP1'*. The parallel composition operator ($\parallel$) in Object-Z can be used to achieve this end.

The schema containing the aforementioned conjunct should pertain exclusively to player two's partition, as this is what it changes. The required information contained in *fleetP1* and *fleetP1'* can be exported from the other schema that modifies player one's partition by introducing two new output variables *fleetP1!* and *fleetP1'!* to the schema and equating them to these variables. When input variables are added to the schema that references player two's variable set, (i.e., *fleetP1?* and *fleetP1'?*), the required variable values may be shared between the two schemas via a parallel composition operator. It is relevant to note that this process of introducing a parallel operator over schema conjunction is completely syntactic and automatable. The resultant operation is:

$$\begin{aligned}
\text{FireAtP1} \triangleq & \\
& \left[\Delta(\text{fleetP1}) \text{ fleetP1!}, \text{fleetP1'}! : \text{FLEETBOARD}; \text{guess?} : \text{COORDINATE} \right. \\
& \quad \left| \text{fleetP1'} = \text{fleetP1} \triangleleft \{\text{guess?}\} \wedge \right. \\
& \quad \quad \left. \text{fleetP1!} = \text{fleetP1} \wedge \text{fleetP1'}! = \text{fleetP1'} \right] \parallel \\
& \left[\Delta(\text{guessesP2}) \text{ fleetP1?}, \text{fleetP1'?} : \text{FLEETBOARD}; \text{guess?} : \text{COORDINATE} \right. \\
& \quad \left| \text{guess?} \notin \text{dom guessesP2} \wedge \right. \\
& \quad \quad \text{guessesP2'} = \text{guessesP2} \cup \{\text{guess?} \mapsto \\
& \quad \quad \quad \text{if fleetP1'?} \subset \text{fleetP1?} \text{ then} \\
& \quad \quad \quad \quad \text{if ran fleetP1'?} \subset \text{ran fleetP1?} \text{ then} \\
& \quad \quad \quad \quad \quad \text{if fleetP1'?} = \emptyset \text{ then} \\
& \quad \quad \quad \quad \quad \quad \text{fleet_destroyed else sink else hit else miss}\} \left. \right]
\end{aligned}$$

We then separate and name the anonymous schemas so that they form distinct operations as required by the annealing rule's precondition. The particulars of the names chosen are arbitrary at this stage, as long as they are meaningful. In the case above, for example, we redefine *FireAtP1* to be (*UpdateFleetP1* $\parallel$ *UpdateGuessesP2*) and define *UpdateFleetP1* and *UpdateGuessesP2* accordingly.

The annealing rule can then be applied to the *System* class, twice, producing a class for each player: *Player1* is created (refer to Figure 4) and then *Player2* (similar to *Player1* but not shown for brevity). The schemas that apply to each player are migrated from *System* into each respective new class by the rule, and each class is instantiated once using the variable names *player1* and *player2*. Dereferencing the *player1* and *player2* variables in *System* gives access to the migrated operations and variables. The state information and initialisation predicate for each player are also migrated, giving each class localised information pertaining to the player it represents.

COALESCENCE

The single *Player* class, shown in Figure 5, is formed by applying the coalescence rule to the *Player1* and *Player2* classes. The trivial application of the coalescence rule requires that the two classes be exactly the same, which in this case can be achieved through renaming attributes and operations (see Section 4) in each class to be those names shown in Figure 5. This circumvents the need to discharge proofs showing the *Player* class to be a refinement of both *Player1* and *Player2*.

This completes the process of creating a separate class to encapsulate the functionality of a player. However, the placement of game logic responsible for player interaction still resides in the *System* class. Section 6 illustrates the process of enacting an alternate design decision: moving this logic into the new *Player* class.

6 Restructuring for Active Players

At this stage of the design process, a fairly significant design decision has already been made. The *Player* class formed by annealing and coalescing from the *System* class creates *Player* objects that are *passive*. That is, the logic of the

Fig. 4. After the annealing step for *Player1*.

Fig. 5. After the coalescence step.

interactions between the players of the system is contained within the *System* class and not the *Player* classes themselves (the *System* acts as a mediator). An alternative design moves this logic into the *Player* class, such that each player instance holds a reference directly to the other player and the interaction functionality is implemented by bypassing the *System* class altogether.

Even this solution has two possible modes: each player could either have a *Fire* operation that launches a missile at the opponent, or a *Defend* operation that causes the opponent to fire a missile at the player object invoking the operation. All solutions are consistent with the original specification, but having active *Player* objects with a *Fire* operation is more intuitive, from a user-interface perspective, than the passive approach presented earlier or having a *Defend* operation.

To refactor *Fire* into the design, it is necessary that we go back to just after the two annealing steps applied in Section 5, and:

Step 1. Move the body of the *FireAtP1* operation into *Player2* and name it *Fire*. Retain the operation *FireAtP1* because it appears in the interface of *System*, but redefine it as *player2.Fire*. We repeat this process accordingly to move the *FireAtP2* operation into the *Player1* class.

For example, for class *Player1*:

$$Fire \hat{=} system.player2.UpdateFleet \parallel system.player1.UpdateGuesses$$

Step 2. We simplify the backward references to *system* by noting that the *Player1* and *Player2* classes are only ever instantiated once. Therefore *self* can replace *system.player1* in *Player1*, and *self* can replace *system.player2* in *Player2*.

Step 3. We introduce a local *opponent* variable to invariantly equal *system.player2* in *Player1*. Similarly for *Player2*, *opponent* is introduced to invariantly equal *system.player1*. This supplies a simple abbreviation, so showing the refinement is trivial.

Now, for both classes:

$$Fire \hat{=} opponent.UpdateFleet \parallel self.UpdateGuesses$$

With the invariants added:

$$\begin{aligned} \text{For class } Player1: & \text{ } opponent = system.player2 \\ \text{For class } Player2: & \text{ } opponent = system.player1 \end{aligned}$$

Step 4. We move these new invariant conditions into the *System* class for all instances of *Player1* and *Player2* (there is only one of each), and extend the interface of both classes *Player1* and *Player2* to include *opponent*.

Thus, the *System* class now contains the invariants:

$$\begin{aligned} player1.opponent &= player2 \\ player2.opponent &= player1 \end{aligned}$$

Step 5. In the same manner as described in Section 5, we apply the coalescence rule trivially by renaming operations and variables in the two player classes to make them correspond.

The changes to the specification from applying this process are illustrated in Figure 6. The resultant specification represents an improvement, in terms of design, over the original specification shown in Figure 1. This is evidenced by the reduction of redundancy by encapsulating the concept of a player inside a separate class. In addition, the design decision to move the interaction logic into the *Player* class is a step toward a distributed architecture, and also toward the provision of a separate user-interface for each *Player*.

Fig. 6. After the coalescence step, with active players.

7 Conclusions and Future Work

The presented transformation rules are intended to be general enough to allow for the introduction of a variety of different designs and architectures. It is recog-

nised, however, that adding a rule to repeat structure, rather than just divide or remove structure, could lend itself to many more architecture types (particularly in distributed domains). Other rules for modifying inheritance relationships between classes would be useful, particularly when targeting languages that do not support multiple inheritance (such as Java), and is left as an area for future work.

The provision of tool support for automating the refactoring process has been achieved in the past [16,19] and is recognised as important. Tool support for the rules presented above 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 [8] animator. The possibility exists for abstraction from the specification language itself to allow succinct formal specification and reasoning about these refactoring steps, in a similar vein to the work presented by Mens et al. [14] and Lämmel [20], and this is also left for future work.

The methodology is also intended to allow for an heuristic-based description of object-oriented design patterns as sequences of the refactoring steps. Some work has been performed toward this goal by introducing the Model-View-Controller architectural paradigm [9] for user interfaces into the above case study.

Acknowledgements

The author wishes to acknowledge Graeme Smith and Colin Fidge for their valuable input.

References

1. M. Bonsangue, J. Kok, and K. Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction (MPC'98)*, volume 1422 of *LNCS*, pages 68–95. Springer-Verlag, June 1998.
2. M. Bonsangue, J. Kok, and K. Sere. Developing object-based distributed systems. In *Proc. of the 3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 19–34. Kluwer, 1999.
3. E. Casais. An incremental class reorganization approach. In O. Madsen, editor, *Proc. ECOOP'92*, volume 615 of *LNCS*, pages 114–132. Springer-Verlag, June 1992.
4. M. Cornelio, A. Cavalcanti, and A. Sampaio. Refactoring by Transformation. In *Proc. of REFINE'2002*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2002.
5. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT Series. Springer, May 2001.

6. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
7. S. Goldsack and K. Lano. Annealing and data decomposition in VDM. *ACM Sigplan Notices*, 31(4):32–38, April 1996.
8. 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.
9. T. Hopkins and B. Horan. *Smalltalk: An Introduction to Application Development Using VisualWorks*. Prentice-Hall, 1995.
10. 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.
11. K. Lano. *Formal Object-oriented Development*. Springer Verlag, 1995.
12. J. Lu. Introducing data decomposition into VDM for tractable development of programs. *ACM SIGPLAN Notices*, 30(9), September 1995.
13. Z. Manna and R. Waldinger. Synthesis: Dreams $\Rightarrow$ programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, July 1979.
14. T. Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proc. Int'l Conf. Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.
15. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proc. of the Fourth International Formal Methods Europe Symposium (FME'97)*, volume 1313 of *LNCS*. Springer-Verlag, 1997.
16. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250. ACM Press, 1996.
17. C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 2nd edition, 1994.
18. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Computer Science Department, Urbana-Champaign, IL, USA, May 1992.
19. D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, 1997.
20. R. Lämmel. Towards generic refactoring. In *Third ACM SIGPLAN Workshop on Rule-Based Programming*. ACM Press, 2002.
21. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
22. G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.