Refactoring object-oriented specifications with inheritance-based polymorphism

Graeme Smith
School of Information Technology and Electrical Engineering
The University of Queensland, Australia
Email: smith@itee.uq.edu.au

Steffen Helke
Software Engineering Group
Berlin Institute of Technology, Germany
Email: helke@cs.tu-berlin.de

***Abstract*—Specification notations such as JML and Spec# which are embedded into program code provide a promising approach to formal object-oriented software development. If the program code is refactored, however, the specifications need also to be changed. This can be facilitated by specification refactoring rules which allows such changes to be made systematically along with the changes to the code.**

A set of minimal and complete set of refactoring rules have been devised for the Object-Z specification language. This paper reviews these rules as a basis for a similar approach for languages like JML and Spec#. Specifically, it modifies the rules for introducing and removing inheritance and polymorphism from specifications. While these concepts are orthogonal in Object-Z, they are closely intertwined in the other notations.

I. Introduction

A number of formal specification languages exist for specifying systems in an object-oriented style. These include high-level (programming-language independent) notations such as Object-Z [15] , VDM++ [9] and Alloy [8], as well as notations which are embedded in actual programs such as JML [3] and Spec# [1]. While well established notions of data refinement enable the incremental development of individual classes in such specifications, they do not support changes to the object-oriented structure. This has led to a number of approaches to refactoring formal specifications — in particular, to refactoring their structure [2], [6], [9], [10], [12]–[14].

McComb and Smith [14] present a set of structural refactoring rules for Object-Z specifications which are both *minimal* (in the sense there is no overlap in the type of changes the rules allow) and *complete* (in the sense that any reasonable design[1] can be derived from any specification).

The rules of McComb and Smith treat inheritance and polymorphism as orthogonal concepts. This is because not all inheritance hierarchies in Object-Z introduce polymorphism, and conversely polymorphism is not restricted to being a by-product of inheritance. In object-oriented programming languages such as Java and C#, however, inheritance and polymorphism are closely intertwined. When we introduce inheritance, we also introduce (potential) polymorphism. Hence, the relationship between the concepts needs to be catered for in the refactoring rules.

[1]By reasonable design, we mean any in which the outermost class defining the interface to the specified system does not allow public access to its state variables.

In this paper as a first step towards extending the results of McComb and Smith towards specification languages like JML and Spec#, we present an alternative set of refactoring rules for Object-Z. In Section II we present the rules for Object-Z developed by McComb and Smith, and explain why these are not appropriate for use with languages with inheritance-based polymorphism. We then present an alternative rule for introducing inheritance in Section III and illustrate how it can be used together with other refactoring rules to add (and remove) inheritance hierarchies within a specification. The approach is illustrated on a case study in Section IV before we conclude in Section V.

II. Object-Z rules

McComb and Smith [14] present three refactoring rules for Object-Z which deal with orthogonal concepts in the language. These rules allow a specification to be refactored by the introduction (and removal) of (i) generic parameters, (ii) inheritance and (iii) polymorphism. It is shown how, together with a rule for introducing (and removing) object instantiation, the refactoring rules are complete allowing a specification to be refactored to any reasonable design. The latter rule is defined by McComb [12] based on the "annealing" rule for VDM++ defined by Goldsack and Lano [9], [10].

The rules are each *semantics-preserving*, *i.e.*, when a rule is applied to a specification, it produces a semantically equivalent specification. Two Object-Z specifications are semantically equivalent when their *system* classes, *i.e.*, those defining the interface to the specification, have operations with identical names and behaviours. We assume that the state variables of such system classes are not publically accessible. The fact that the rules are semantics-preserving, allows them to be applied in a forward direction (to introduce structure) and a backward direction (to remove structure). This is central to the proof of completeness of the rules which relies on the ability to remove all structure from a specification (via backward application of the rules) before introducing the structure of the desired refactored design (see [14] for details).

The rules also have specific preconditions for their application. Hence, it is often necessary to use them in conjunction with data refinement of classes to reach a desired design. For example, the rule for introducing object instantiation splits a

A

S
T

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

$$\equiv$$

A

S
$component : B$

$OpA \mathrel{\widehat=} [\,\Delta(S) \mid P(S, S', T)\,]$
$OpB \mathrel{\widehat=} component.OpB$

B

T

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

Fig. 1. Annealing refactoring

(references to C)

C
$L == T$
$\vdots$

$$\equiv$$

(references to $C[T]$)

$C[X]$
$L == X$
$\vdots$

where X is fresh

Fig. 2. Introduce generic parameter refactoring

C
$\mathcal{F}$

$A_1 : \mathbb{P}\, C$
$\vdots$
$A_n : \mathbb{P}\, C$
$\langle A_1, \ldots, A_n\rangle$ partitions C

$$\equiv$$

A_1
$\mathcal{F}$

$\vdots$

A_n
$\mathcal{F}$

$C == A_1 \cup \ldots \cup A_n$

Fig. 3. Introduce polymorphism refactoring

class's state and operations into two classes — one holding a reference to an instance of the other as shown in Figure 1. A precondition of applying this rule is that every local operation in the class *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) — this determines in which class the operations appear after the rule is applied.

To achieve this restriction, any operation schema that changes variables in both S and T must be split 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 attained through refinement, by promoting logical operators to schema operators, and moving the schemas into new, separate operation definitions with fresh names. The new operations are then referenced from the original one. This very much resembles the *Extract Method* [4] refactoring step which splits a programming language procedure into two, where one contains a procedure call to the other.

For example, the operation X below refers to the post-state variables in both S' and T'.

$$X \mathrel{\widehat=} [\,\Delta(S, T) \mid P(S, S', T) \wedge Q(S, T, T')\,]$$

By promoting the logical conjunction to schema conjunction, it is equivalent to

$$X \mathrel{\widehat=} [\,\Delta(S) \mid P(S, S', T)\,] \wedge [\,\Delta(T) \mid Q(S, T, T')\,]$$

which, by introducing a fresh operation Y, is equivalent to

$$X \mathrel{\widehat=} [\,\Delta(S) \mid P(S, S', T)\,] \wedge Y$$
$$Y \mathrel{\widehat=} [\,\Delta(T) \mid Q(S, T, T')\,].$$

Such refinements are semantics-preserving. Application of this approach to a detailed case study can be found in McComb [13].

The rule for introducing generic parameters is illustrated in Figure 2. A class C has a locally defined type L which is defined to be the actual type T. When the rule is applied, the class C is replaced with a class $C[X]$, where the name X is fresh, and the local definition which previously defined L as T is changed to define L as X. All references to C in the specification are replaced with references to $C[T]$, including references for inheritance.

This refactoring rule only introduces one parameter, but repeated application can provide as many parameters as necessary.

The refactoring rule for introducing polymorphism is illustrated in Figure 3. The class C on the left-hand side has exactly $n+1$ means of referencing it: by C, or by n axiomatically defined aliases $A_1, \ldots, A_n$ which disjointly partition the references to objects of C.

The introduction of polymorphism is normally motivated by the identification of a class (C) that behaves in different ways depending upon the context in which it is used. The rule requires that the designer identify the contexts where alternate behaviours are expected, and divide the references between $A_1, \ldots, A_n$ accordingly.

Assuming this identification and partitioning of object references has occurred, the rule allows for the splitting of the behaviours into separate class definitions (A_1 to A_n on the right-hand side of Figure 3). To execute the refactoring transformation, all of the features of class C are copied verbatim to define the classes A_1 to A_n. The class C is removed from the specification, but C is globally defined to be the class union $A_1 \cup \ldots \cup A_n$ — thus providing for the polymorphism. The identical feature sets of the classes are represented with the symbol $\mathcal{F}$ in Figure 3.

The rule for introducing inheritance creates an inheritance relationship between any two classes in the specification, as long as the addition of the relationship does not result in a circular dependency. Figure 4 illustrates the application of the rule to two classes A and B with features $\mathcal{F}$ and $\mathcal{G}$ respectively.

The rule not only adds the inheritance relationship (indicated in Figure 4 by the inclusion of A in B) but also hides

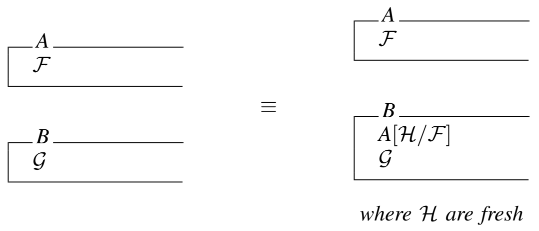


Fig. 4. Introduce inheritance refactoring

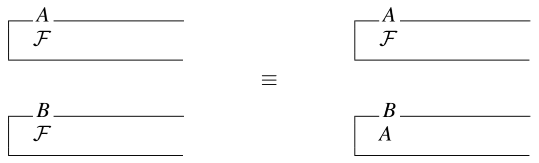


Fig. 5. Revised introduce inheritance refactoring

every feature of the superclass by assigning them a fresh name (the notation '$\mathcal{H}/\mathcal{F}$' indicates that all features $\mathcal{F}$ of the superclass A are hidden by assigning fresh names $\mathcal{H}$ for the features). The combination of inheritance and hiding makes the refactoring rule an equivalence transformation, so long as we assume we do not have inheritance-based polymorphism. To use the features inherited from the superclass, the designer must make refinements local to the subclass to reference the features in $\mathcal{H}$.

The rules for introducing object instantiation, generic parameters and polymorphism are readily adapted to specification notations such as JML and Spec#. In the latter case, Object-Z's notion of class union can be captured by the use of a Java or C# interface.

The same is not true, however, of the rule for introducing inheritance. This rule is inappropriate for specification notations supporting inheritance-based polymorphism, *i.e.*, where an object declared to be of class A may actually belong to class A or any subclass B of class A. In Figure 4, if $\mathcal{F}$ and $\mathcal{G}$ both contained a method named m, then given an instance a of class A a method call $a.m$ would result in the behaviour of A's m in the specification on the left-hand side of the rule. It would, however, possibly result in the potentially different behaviour of B's m in the specification on the right-hand side of the rule. Hence, the rule is not an equivalence transformation.

Furthermore, the *signature*, *i.e.*, the names of the (visible) features, of class B is not necessarily wider than that of class A. This can result in a specification that is not well-defined. For example, if $\mathcal{F}$ contained a method m that was not in $\mathcal{G}$ then if the specification on the left-hand side of the rule contained a method call $a.m$, where a was declared to be an instance of class A, then the specification on the right-hand side would not be well-defined since a's actual class might be B.

In the next section, we present an alternative refactoring rule for introducing inheritance which overcomes these problems. Additionally, we show how this new rule can be used to introduce and remove inheritance hierarchies with the aid of three additional simple refactoring rules which allows features to be added to a class, invariants to be moved between classes, and inherited features to replace identically defined features in a subclass.

III. Revising the inheritance rule

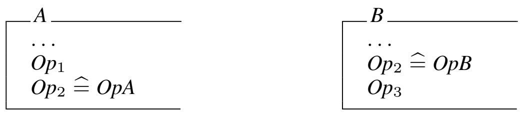
Specification notations such as JML and Spec# have inheritance-based polymorphism like the programming languages with which they are used. In this setting, a refactoring rule which introduces inheritance between two existing classes A and B is semantics-preserving precisely when the class which becomes the subclass, B say, is a subtype of of A in the sense that it has a wider signature and all its methods satisfy the *methods rule* of Liskov and Wing [11]. That is, each subclass method has a weaker precondition than its same-named counterpart in the superclass and, when the precondition of the superclass method holds, a stronger postcondition.

Checking whether the methods rule holds is, in general, non-trivial. An important property of the Object-Z refactoring rules is that they are largely syntactic and hence potentially automatable. In keeping with this, we define a less general rule for introducing inheritance, but one that is readily checked syntactically.

The rule, shown in Figure 5, allows an inheritance relationship to be added between classes A and B only when A and B have identical features (denoted by $\mathcal{F}$ in the figure). Unlike the existing rule for inheritance (Figure 4), the inherited features are not hidden in B.

The rule is obviously sound, *i.e.*, the left and right-hand sides of the rule are equivalent. However, the precondition is quite strong since A and B have to have identical features. Below we provide a general strategy for achieving this precondition. This strategy can be used to add inheritance between any two classes in a specification. Of course, this won't always lead to an improved design; as with any refactoring strategy, it is the specifier's responsibility to decide when to use it.

Let us assume we have two classes A and B whose features are unrelated except that they have one similarly named operation Op_2. This operation has definition Op_A in class A and Op_B in class B.


A
...
Op_1
$Op_2 \mathrel{\widehat=} OpA$
B
...
$Op_2 \mathrel{\widehat=} OpB$
Op_3

First we extend each class with the features from the other class apart from Op_2. Since these features will not be referenced in the specification, this results in an equivalence transformation. To allow this step in our framework, we introduce the following refactoring rule for adding features to a class.

Applying this rule to classes A and B results in the following.

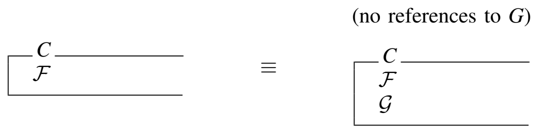


Fig. 6. Introduce features refactoring

$$\begin{array}{|l}\hline A \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} OpA \\ Op_3 \\ \hline\end{array} \qquad \begin{array}{|l}\hline B \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} OpB \\ Op_3 \\ \hline\end{array}$$

Next, again using the rule for introducing features, we extend each class with a visible constant t whose value is either t_A or t_B.

$$\begin{array}{|l}\hline A \\ \mid\ t : t_A \mid t_B \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} OpA \\ Op_3 \\ \hline\end{array} \qquad \begin{array}{|l}\hline B \\ \mid\ t : t_A \mid t_B \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} OpB \\ Op_3 \\ \hline\end{array}$$

Then we perform a refinement on each of the classes so that the value of t is t_A in class A and t_B in class B, and use the value of t to redefine Op_2 to be identical in each class. Since the constant t of A and B is not referenced anywhere in the specification at this point, the refinements are semantics-preserving.

$$\begin{array}{|l}\hline A \\ \mid\ t : t_A \mid t_B \\ \boxed{t = t_A} \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} [t = t_A] \wedge OpA \\ \qquad\quad [] \\ \qquad\quad [t = t_B] \wedge OpB \\ Op_3 \\ \hline\end{array} \qquad \begin{array}{|l}\hline B \\ \mid\ t : t_A \mid t_B \\ \boxed{t = t_B} \\ \ldots \\ Op_1 \\ Op_2 \mathrel{\widehat=} [t = t_A] \wedge OpA \\ \qquad\quad [] \\ \qquad\quad [t = t_B] \wedge OpB \\ Op_3 \\ \hline\end{array}$$

The resulting classes are now identical apart from the invariant on t. To remove this, we introduce a further refactoring rule. This rule allows us to introduce an invariant P to a class when all instances and subclasses of that class within the specification are already specified to satisfy P.

Applying this rule to both A and B in the backward direction, results in the classes being identical. To meet the condition on the left-hand side of the rule, we would need to, when applying the rule, add the appropriate constraint on t to any classes or operations which instantiated classes A and B. That is, where we had a declaration $a : A$ (meaning a is

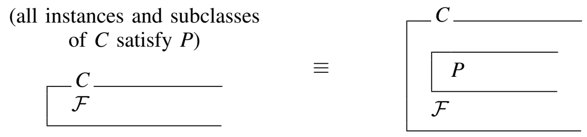


Fig. 7. Introduce invariant refactoring

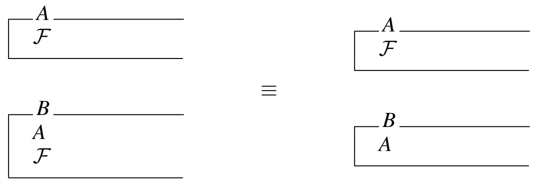


Fig. 8. Use inherited features refactoring

an object of class A) we would add the constraint $a.t = t_A$, where we had a declaration $b : B$ (meaning b is an object of class B) we would add the constraint $b.t = t_B$, and where we had a declaration $a : \downarrow A$ (meaning that a is an object whose class may be A or any subclass of A) then we would add the constraint $a.t = t_A$. The latter constraint is necessary since before applying the rule, B is not a subclass of A and hence $a : \downarrow A$ cannot behave as an object of class B. The constraint ensures this is the case after the rule is applied. Similarly, for any class which inherits A or B we would need to add the constraint $t = t_A$ or $t = t_B$ respectively.

The rule for introducing inheritance could then be applied to classes A and B. At this stage, we have an inheritance relation between two identical classes and invariants on all instances of those classes. We can then simplify the classes to reach our desired specification. This will be illustrated on a concrete case study in the next section. It is straightforward to generalise the strategy to cases where there is more than one operation, or other feature, with a common name in the two classes.

To remove an inheritance relationship between a class A and its subclass B, the strategy is similar (it is illustrated in Section IV). Classes A and B are transformed to be identical using the rules for introducing features and invariants described above. In this case, when the invariant from A is removed, an invariant $a.t = t_A$ is not added where there is a declaration of $a : \downarrow A$ in the specification. This is because objects of type $\downarrow A$ can behave according to the definitions in either class A or B before the rule is applied.

Once the classes are identical, the features of the subclass are removed using one further refactoring rule shown in Figure 8.

Application of this rule sets up the precondition for the backward application of the introduce inheritance rule (Figure 5) allowing the removal of the inheritance relation.

IV. Case study

In this section we illustrate the use of the refactoring rules of Section III on a simple case study: the well known game of Nim. In this game there are two players and a pile of sticks.

Each player in turn removes one, two, or three sticks from the pile. The player who removes the last stick loses.

We define two classes of players to represent different game strategies. The state of a player comprises a boolean variable *turn* indicating whether it is the player's turn. It can be switched using the operation *changeTurn*. This operation is identical for both classes. The operation *makeMove* captures the player's strategy. It has an input $pnum?$ denoting the number of sticks in the pile, and an output $num!$ denoting the number of sticks the player chooses to remove.

A *CleverPlayer* knows the strategy for winning the game. One stick left is clearly a losing situation. But 2, 3, or 4 sticks left are winning situations, since by making an appropriate choice, the player can leave the other player with the last stick. If there are 5 sticks left, it is a losing situation since no matter what the player does, the other player is left with a winning situation. Continuing in this manner, we see that when the number of sticks divided by 4 has a remainder of 1, we are in a losing situation. A *CleverPlayer* will endeavour to put, and keep, their opponent in a losing situation.

A *SimplePlayer* chooses the number of sticks to take on each turn completely at random, unless there are 4 or less sticks left when the player will take all the sticks but one. We define *SimplePlayer* (naively) as a subclass of *CleverPlayer* to reuse the declaration of *turn* and the *changeTurn* operation which are identical in both classes. The notation $[makeMove_0/makeMove]$ renames the inherited *makeMove* operation so that it can be overridden in class *SimpleMove*.

CleverPlayer

$$turn : \mathbb{B}$$

makeMove

$$pnum? : \mathbb{N}$$
$$num! : 1 \mathrel{..} 3$$

$$turn$$
$$pnum? \bmod 4 = 0 \Rightarrow num! = 3$$
$$pnum? \bmod 4 = 2 \Rightarrow num! = 1$$
$$pnum? \bmod 4 = 3 \Rightarrow num! = 2$$
$$pnum? \bmod 4 = 1 \Rightarrow num! = 1$$

changeTurn

$$\Delta(turn)$$

$$turn' = \neg\, turn$$

SimplePlayer

$$CleverPlayer[makeMove_0/makeMove]$$

makeMove

$$pnum? : \mathbb{N}$$
$$num! : 1 \mathrel{..} 3$$

$$turn$$
$$pnum? = 1 \Rightarrow num! = 1$$
$$(1 < pnum? \wedge pnum? \leq 4) \Rightarrow num! = pnum? - 1$$

The game is modelled in Object-Z by the class *Nim*. For brevity, we provide only the state declarations here. The state of the class consists of a variable *pile*, denoting the number of sticks in the pile, and two players: $player_1$ and $player_2$. $player_1$ is declared to be of type $\downarrow CleverPlayer$ which means that the player can be an object of the class *CleverPlayer* or alternatively an object of a direct or an indirect subclass of *CleverPlayer*. In contrast, $player_2$ is defined to be of class *CleverPlayer*. The invariant of the state schema ensures that the players do not have a turn simultaneously.

Nim

$$pile : \mathbb{N}$$
$$player_1 : \downarrow CleverPlayer$$
$$player_2 : CleverPlayer$$

$$player_1.turn \neq player_2.turn$$

...

A. Removing the inheritance relation

The inheritance relation between *CleverPlayer* and *SimplePlayer* is not a subtype relation. The postcondition of *makeMove* in *SimplePlayer* is weaker that in *CleverPlayer* violating Liskov and Wing's methods rule [11]. To remedy this undesirable situation, we remove the inheritance relation using the strategy of Section III.

Figure 9 shows the first two steps[2]. In step 1, using the refactoring rule for introducing features (Figure 6) we extend class *CleverPlayer* by adding a public constant t, which will be directly inherited by class *SimplePlayer*. Furthermore, we refine class *CleverPlayer* so that $t = t_{CP}$ allowing us to redefine its *makeMove* operation. Note that this step is semantics-preserving because t is not used in other parts of the specification. In step 2, we remove the invariant from *CleverPlayer* by applying the refactoring rule for introducing invariants (Figure 7) backwards. This introduces the state invariant $t = t_{CP}$ in class *SimplePlayer* and the state invariant

[2]In this and subsequent figures, ellipses (...) are used to hide parts of the classes that are unchanged.

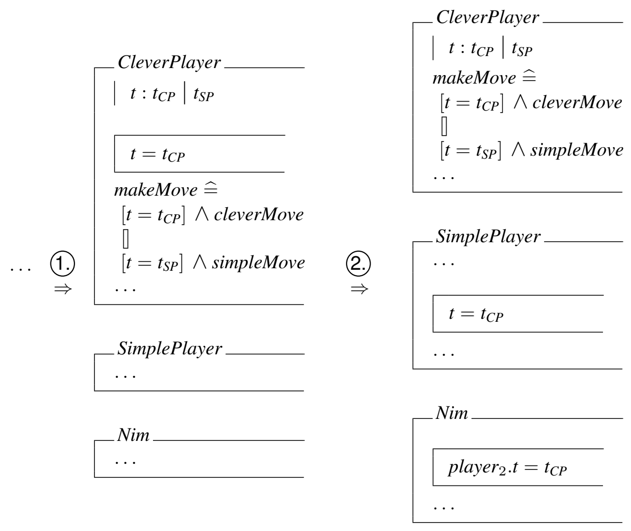


Fig. 9. Redefining *makeMove* in *CleverPlayer*

$player_2.t = t_{CP}$ in class *Nim* (but no invariant on $player_1$ which can be instantiated by an object of either class).

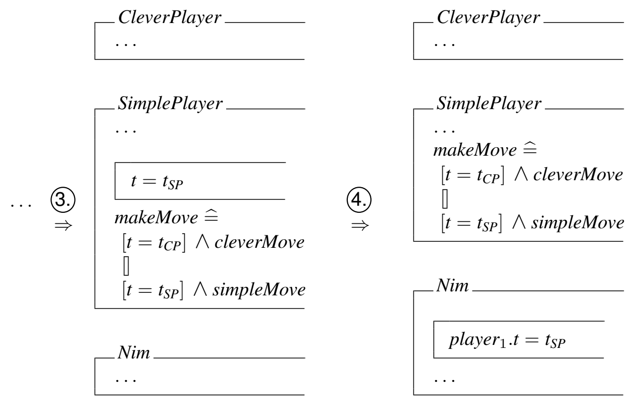


Fig. 10. Redefining *makeMove* in *SimplePlayer*

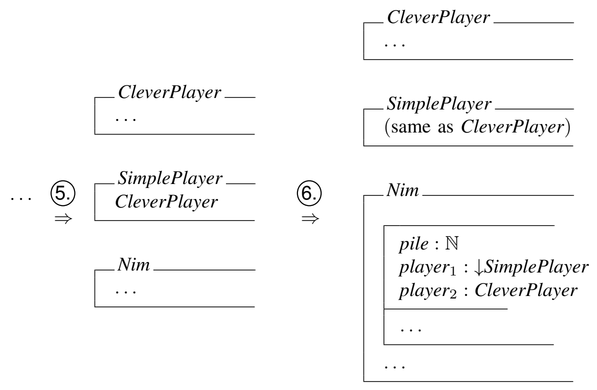
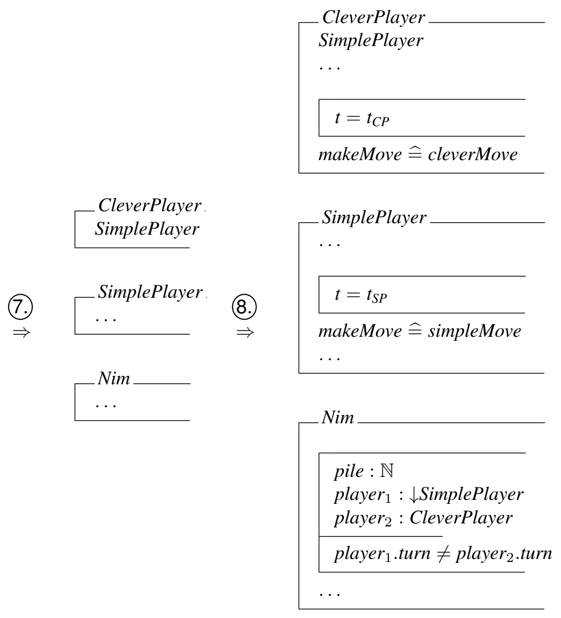


Fig. 11. Removing the inheritance relation

In step 3 (see Figure 10), we redefine the invariant in *SimplePlayer* to $t = t_{SP}$. This is semantics-preserving since there are no references to *SimplePlayer*'s t in the specification. This allows us to redefine *makeMove* to be identical to that in *CleverPlayer*. (We use *simpleMove* and *cleverMove* to denote the original *makeMove* operations of *SimplePlayer* and *CleverPlayer* respectively.)

In step 4, we remove the invariant from *SimplePlayer* by applying the refactoring rule for introducing invariants (Figure 7) backwards. This introduces the state invariant $player_1.t = t_{SP}$ in class *Nim*

In step 5 (see Figure 11), we use the rule for using inherited features (Figure 8) to remove all explicit features from *SimplePlayer*. We then remove the inheritance relation in Step 6. In this final step, since the two classes are now identical, we refine the declaration $player_1 : \downarrow CleverPlayer$ in class *Nim* to $player_1 : \downarrow SimplePlayer$. This is in preparation for introducing a new inheritance relation between the player classes (which we do next).



CleverPlayer
SimplePlayer
. . .
$t = t_{CP}$
$makeMove \mathrel{\widehat=} cleverMove$
CleverPlayer
SimplePlayer
7.
⇒
SimplePlayer
. . .
8.
⇒
SimplePlayer
. . .
$t = t_{SP}$
$makeMove \mathrel{\widehat=} simpleMove$
. . .
Nim
. . .
Nim
$pile : \mathbb{N}$
$player_1 : \downarrow SimplePlayer$
$player_2 : CleverPlayer$
$player_1.turn \neq player_2.turn$
. . .

Fig. 12. Establishing the new inheritance hierarchy

B. Adding a new inheritance relation

Since after step 6 the player classes of our specification are identical, we can apply the rule for introducing inheritance (Figure 5). This allows us to build a new inheritance hierarchy which is also a subtype hierarchy. Specifically, we make *CleverPlayer* a subclass of *SimplePlayer*. This is shown in step 7 of Figure 12.

Next, in Step 8, we apply the rule for using inherited features (Figure 8) to reintroduce *makeMove* into *CleverPlayer*, followed by the rule for introducing invariants (Figure 7) to both *SimplePlayer* and *CleverPlayer*. The latter removes the invariants from *Nim* and allows us to simplify the *makeMove* operation in both of the player classes.

In a final step, we remove the constants *t* and their invariants from both player classes, as they are no longer used. The resulting refactored specification is shown in Figure 13. The sequence of refactoring rules and semantics-preserving refinements to reach this point provide a strategy that could be more widely applied. An important area of future work is to formalise such strategies in order to perform more substantial changes to a design in one step. The application of such strategies, rather than the many steps required with individual refactoring rules, is essential to the practicality of the proposed approach.

V. Conclusions

The refactoring rules for Object-Z proposed by McComb and Smith [12], [14] treat inheritance and polymorphism as orthogonal concepts. Hence, they are not directly applicable to specification languages with inheritance-based polymorphism such as JML and Spec#. This paper proposes alternative rules for introducing and removing inheritance in Object-Z which overcomes this problem.

SimplePlayer

$turn : \mathbb{B}$

makeMove

$pnum? : \mathbb{N}$
$num! : 1 \mathrel{..} 3$

$turn$
$pnum? = 1 \Rightarrow num! = 1$
$(1 < pnum? \land pnum? \leq 4) \Rightarrow$
$\quad num! = pnum? - 1$

changeTurn

$\Delta(turn)$

$turn' = \neg\, turn$

CleverPlayer

SimplePlayer

makeMove

$pnum? : \mathbb{N}$
$num! : 1 \mathrel{..} 3$

$turn$
$pnum? \bmod 4 = 0 \Rightarrow num! = 3$
$pnum? \bmod 4 = 2 \Rightarrow num! = 1$
$pnum? \bmod 4 = 3 \Rightarrow num! = 2$
$pnum? \bmod 4 = 1 \Rightarrow num! = 1$

Nim

$pile : \mathbb{N}$
$player_1 : \downarrow SimplePlayer$
$player_2 : CleverPlayer$

$player_1.turn \neq player_2.turn$

. . .

Fig. 13. Final specification after refactoring

In related work, Goldstein, Feldmann and Tysberowicz [7] address transformations of object-oriented programs using formal contracts, but in a rather informal way. Similarly to our work, they describe a pragmatic procedure in which inheritance can be added between two classes. Freitas et al. [5] provide a more formal approach, but focus on refactoring rules to move invariants, attributes or redefined methods from a class to its superclass.

The next step for extending our approach to a set of JML or Spec# refactoring rules would be to adapt the Object-Z rules to the required syntax of the other notations. It is also necessary to formally prove that the rules are sound, and highly desirable to prove that they are complete. To enable these tasks we are working on a formal meta-model of object-oriented specifications in which we can express and reason about the rules. The meta-model will be general enough to map to concrete rules in any of the considered specification languages: Object-Z, JML and Spec#.

References

[1] M. Barnett, K. R. M. Leino and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, Springer LNCS, **3362**, 2004.

[2] P. Borba, A. Sampaio, A. Cavalcanti and M. Cornelio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52(1-3):53–100, 2004.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino and E. Proll. An Overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[4] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[5] G. F. Freitas, M. Cornelio, T. Massoni and R. Gheyi. Object-oriented Programming Laws for Annotated Java Programs. In *Tenth International Workshop on Rule-Based Programming (RULE)*, pages 65–76, Electronic Proceedings in Theoretical Computer Science (EPTCS), **21**, 2009.

[6] R. Gheyi and P. Borba. Refactoring Alloy specifications. *Electronic Notes in Theoretical Computer Science*, 95:227–243, 2004.

[7] M. Goldstein, A. Feldmann and S. Tyszberowicz. Refactoring with Contracts. In *Proceedings of the conference on AGILE*, IEEE Computer Society, pages 53–64, 2006.

[8] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

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

[10] K. Lano and S. Goldsack. Refinement of distributed object systems. *Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 99–114, Chapman and Hall, 1996.

[11] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 16(2):1811–1841, 1994.

[12] T. McComb. Refactoring Object-Z specifications. In *Fundamental Approaches to Software Engineering (FASE)*, Springer LNCS, pages 69–83, **2984**, 2004.

[13] T. McComb and G. Smith. Architectural design in Object-Z. In *Australian Software Engineering Conference (ASWEC)*, IEEE Computer Society Press, pages 77–86, 2004.

[14] T. McComb and G. Smith. A minimal set of refactoring rules for Object-Z. In *International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Springer LNCS, pages 170–184, **5051**, 2008.

[15] G. Smith. The Object-Z Specification Languague. Kluwer, 2000.