Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 259 (2009) 193–205
www.elsevier.com/locate/entcs

# Static Class Elements for Object-Z

Thomas Ruhroth[1] Heike Wehrheim[2]

*Universität Paderborn*
*Institut für Informatik*
*33098 Paderborn, Germany*

**Abstract**

Static variables and methods are part of almost every modern object-oriented programming language. Static elements are for instance indispensable for certain kinds of *design patterns* applied during programming. Object-oriented *specification* formalisms on the other hand lack such concepts. This can prevent writing formal specifications close to the actual implementation, and can thus hamper a refinement-based stepwise development.
In this paper, we extend the state-based object-oriented specification language Object-Z with a concept for static class elements. We furthermore show how refinement can introduce static elements into a specification.

*Keywords:* Object-Z, Class, Static Variables

## 1 Introduction

In a model-based software design, the system to be built is first described by means of an abstract model. This abstract model is on the one hand the source of a first analysis with respect to given requirements, and on the other hand the starting point of a successive concretisation, developing more detailed models, closer to an actual implementation. In a *formal* software development, these steps from abstract to more concrete models have to be valid *refinements* [3,15,4], thus guaranteeing the concrete model to preserve behaviour of the abstract model.

In object-oriented (OO) programming languages, recurring solutions to frequent tasks have been identified by *patterns* [6]. Patterns are often employed in programs as they present a known, working way of coding specific programming tasks, and can thus enhance correctness and readability of programs. When following a stepwise, formal development of systems, a designer will thus often be faced with the task of writing specifications with object-oriented patterns, the more often, the closer he

[1] Email: thomas.ruhroth@uni-paderborn.de
[2] Email: wehrheim@uni-paderborn.de


doi:10.1016/j.entcs.2009.12.025

gets to the implementation level. This has stimulated work on the use of patterns in object-oriented formal methods [9,8,11,14,1]. However, one particular kind of patterns (creation patterns) cannot be directly modelled by standard object-oriented formalisms (e.g. Object-Z [11], VDM$^{++}$ [10], OhCircus [2]) as they lack the concept of *static* class elements. Static variables and methods in OO-programming languages are elements which – in contrast to *object* variables and methods – belong to the class itself and not to specific objects. Static variables of a class exist once for the class, and are not instantiated once per object. Originally coming from Smalltalk [7], today most modern OO-programming languages (e.g. Java, C$^{\#}$) include this concept. An enhancement of formal object-oriented languages with this concept would thus be beneficial, in particular, when the language is used for a stepwise design.

In this paper, we present such an enhancement of a formal specification language with static class elements. As language we use the object-oriented state-based method Object-Z [13,5], which already contains the main object-oriented features like object creation, inheritance and polymorphism. By making just a small extension of the language (and its semantics), we can incorporate static variables and methods into Object-Z. This extension essentially concerns the addition of information about the class an object belongs to: like `self` referring to the object identity, we now have a property `classname` referring to the class of an object. With this single new concept we can model static class elements in Object-Z specifications. As this always involves a number of additional class definitions (for instance, one class being the root of all objects, and another class being the root of all classes), we furthermore define a shorthand notation which allows to simply write static variables and methods within normal Object-Z classes. We then show that creation patterns (more specifically, the Singleton pattern) can be modelled with this extension of Object-Z.

Static variables and methods will most often be used during the final step of developing a model close to an object-oriented implementation. Thus, during the development we will have a transition from a specification without to one with static elements. This transition should of course be a valid refinement step. Hence, we furthermore study refinement in this new setting of Object-Z with static elements. As it turns out, refinement between classes with static elements can simply be seen as a *class refinement* [4], i.e. we can use standard class simulations to verify them. Unfortunately, this means that refinement in this setting is in general not *compositional*: a refinement on the class-side (the static elements) plus a refinement on the object-side (the "normal" Object-Z part) does not necessarily give us a refinement as a whole. However, we show that the introduction of a class-side constitutes a valid refinement (under some weak condition). In a stepwise design we can thus gradually introduce static elements into our specification.

The paper is structured as follows. In Section 2 we will give an example Object-Z specification which describes Object-Z's inability of (naturally) modelling creation

patterns, more specifically the Singleton pattern. We furthermore give a Java code fragment to show how the pattern can be modelled with static variables. The next section will then present our extension of Object-Z. Section 4 discusses the issue of refinement and compositionality, and the last section concludes.

## 2 The Log Example

We begin with explaining the problem in modelling creation patterns, and alongside introduce Object-Z. Our example is a log class, which can be used to store messages from a system like error or information messages. We model this class in Object-Z. Classes in Object-Z are written as schemas which themselves include other schemas, in particular those for describing the state, the initialization and the operations of the class. The class *Log* first of all declares its visibilities (operations *log*, *getMessage* and INIT are visible) and then starts with a state definition which includes a variable *logFile* holding a sequence of *Message*s. The INIT-schema ensures that the variable *logFile* is empty at the beginning. The class ends with two operations *log* to log a message and *getMessages* to obtain all messages collected so far. The operations comprise a declaration and a predicate part. In the declaration section, the input (marked with ?) and output (!) variables are declared. Furthermore the deltalist ($\Delta(\ldots)$) defines which state variables can be changed by the operation. The predicate section states enabling conditions for the operation and its effect. Here, primed variables refer to the after state. Operation *log* adds new messages to the *logFile* using the concatenation operator ($\frown$), *getMessage* simply returns all messages. In contrast to programming languages, operations in Object-Z are atomic, i.e. an operation specification can be seen as a constraint on the allowed state change and this change is taking place in one step [3].

$[Message]$

*Log*

$\upharpoonright (log, getMessage, \text{INIT})$

$logFile : \text{seq}\, Message$

INIT

$logFile = \langle\rangle$

*log*

$\Delta(logFile)$
$aMessage? : Message$

---

$logFile' = logFile \frown \langle aMessage? \rangle$

*getMessage*

$listOfMessages! : \text{seq}\, Message$

---

$listOfMessages! = logFile$

[3] In contrary to programming languages, we thus do not have to deal with issues concerning the ordering of initialisation of static variables.

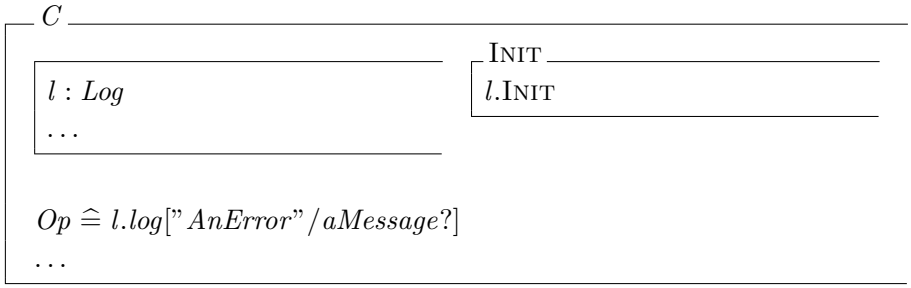



Fig. 1. Class $C$ using the *Log*

When using this class, we might encounter a problem concerning multiple instances. For example, consider a class $C$ (Figure 1)which has a variable of type *Log* and uses it to log events (in operation *Op*).
Each instance of this class $C$ holds its own variable *logFile*, therefore, if we have multiple instances, each *l.getMessage* returns a subset of all messages generated. Instead we would like the system to have just *one* instance of the *logFile* on which all instances of $C$ write (more precisely, we would like one instance of *Log* only). This particular issue of single object instantiation is a well-known problem in OO-programming languages and it can be solved with static variables. Static variables are created only once per class, not once per instance like normal object variables of classes. Static operations are similar, the operation is created once for a class and cannot use any internal information of an instance (unless it has a static variable with a reference to instances). In Java code static variables and operations are marked with the keyword *static*.

```
public class Log {

   private static Log instance = new Log();

   public static Log getInstance() {
       return instance;
   }
   private Log() {
        ...
   }
   public void log(String aMessage) {
        ...
   }
   ...
}
```

Fig. 2. A Java implementation of the Singleton pattern

To achieve the above desired unique instance of class *Log*, programming languages would use the *Singleton pattern* [6]. Figure 2 gives an example of a Java implementation of the Singleton pattern. The only instance of class `Log` is stored in the static class variable `instance`, which is declared `private`, so this class variable cannot be used from outside the class. If you want to get the single instance of `Log`, you have to use the method `getInstance`, which is declared `static` and `public`, so it can be used from every place in the code. The private constructor `Log()` ensures that no instance can be created outside the class, thus `instance` remains the one instance of class `Log`.

In object-oriented programming languages the concept of static variables and operations thus allows to formulate the Singleton pattern (and also other creation patterns). In Object-Z we would need to use global variables and/or functions to achieve the same effect. This would however break the object-oriented structuring: the unique *Log* instance would not be attached to the *Log* class anymore. This gets the more inappropriate the more such variables or methods we have. Moreover, as object-oriented programming languages either do not have global variables or consider their usage to be bad style, a specification close to the implementation should not have global variables.

## 3 Static elements in Object-Z

In the previous section, we stated the problem of multiple instances for a logging system. The solution in object-oriented programming languages is the use of the Singleton pattern, which uses static variables and operations. Also all other creation patterns require static elements and thus can neither be adequately modelled in Object-Z. We therefore need an extension of Object-Z. First, some terminology: we refer to the static variables and methods of a class as its *class-side* and to the ordinary object variables and methods as its *object-side*. Thus the purpose of the extension is to be able to define a class-side of classes (in Object-Z), and furthermore ensure, that this is indeed a class-specific side, i.e. the variables and methods in this part specifically belong to the class itself and not to their objects. In particular, class variables are not instantiated once per object.

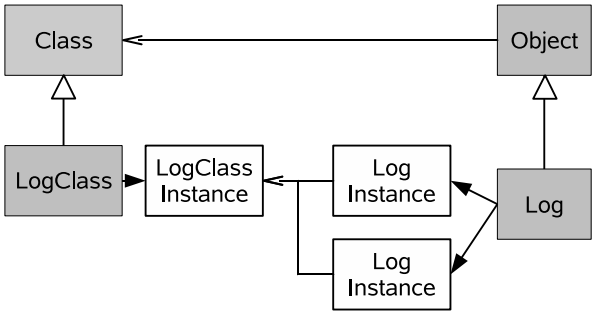



Fig. 3. Class framework (grey boxes) with some example instances (white boxes)

For the extension, we use a concept similar to the way static elements are defined in Smalltalk/80 [7], the first OO-language with static variables and methods. In Smalltalk everything is an object, even the class-side is an object. So we can define one root class *Class* for the class-side and use the instances of this class (or specializations thereof) as the class-sides of classes. We furthermore need to give every object a reference to its class-side, and most importantly, the references of *different* instances of a given Object-Z class have to refer to the *same* class-side instance. This uniqueness of the reference of course needs to be specified, and in order to avoid having to model this again and again for every new class, we introduce a specific root class called *Object* modelling this feature. All Object-Z classes which should have static elements need to inherit from *Object*. The grey boxes in Figure 3 show the log example and the suggested class structure. We have the class *Log* modeling our log system. The class *Object* is the superclass for all object-side classes (like the class *Log*). The class *Class* is the superclass for all class-side classes (like the *LogClass* we are soon going to develop). Every *Object* furthermore possesses a reference to its *Class*. Example instances of the classes are the white boxes, *LogInstance* are objects of the object-side of *Log* (any number allowed) and *LogClass* is the one instance of the class-side of *Log*.

This concept now needs to be incorporated into Object-Z. We do this in three steps:

(i) we first extend Object-Z in order to give each object the information to which class it belongs,

(ii) we second model the framework described above defining the root classes *Class*, *Object* and a function *classref*,

(iii) we model the class-side by subclasses of class *Class*, the object-side by subclasses of *Object*.

Only the first step is an extension of Object-Z, the rest can be done using existing Object-Z concepts. For the first step, we extend Object-Z with a construct named *classname*. First of all, *classname* is a new reserved word and we incorporate it into Object-Z expressions with the following new rule:

$$\texttt{Expression4} ::= \texttt{Expression4}.\mathit{classname}$$

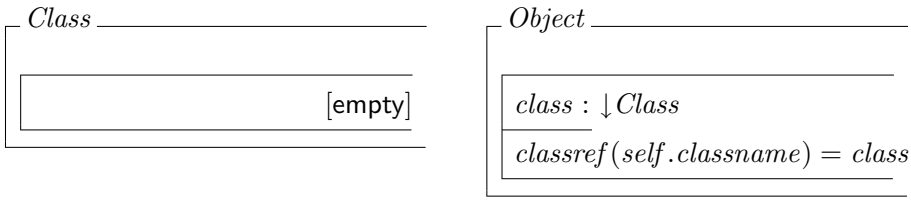
`Expression4` is part of the Object-Z grammar given in [13] and covers for instance variables names. Similarly to *self* which refers to the object itself, *classname* refers to the unique class-side of an object. An informal definition of *classname* (in the style of [13]) is:

$$a : \mathit{Classname}$$
$$a.\mathit{classname} = {}'\mathit{Classname}'$$

We use quotes here to distinguish the name of a class from the set of object identities

of a class. The variable *classname* will take values out of the universe of all possible names of classes *CNames*. A more formal definition could first define a new meta-function **name** (see [13] for other meta-functions for classes), and use this to derive the name of the class of an object. The above definition can also be extended to cover polymorphism and class union.

In the second step, we define the classes *Class* and *Object* from Figure 3 in Object-Z using the *classname* construct from the first step. The association between the class *Object* and the class *Class* is realized through a variable *class* in class *Object* (see below). The downarrow (↓) before the type identifier (*Class*) indicates that the variable can hold instances from the class *Class* or its subclasses. These two classes present the general framework for modelling static elements and should not be instantiated directly. Instead, the specifications will usually contain subclasses of *Class* and *Object*.




In the definition of the *class* variable we use a *global injective* function *classref* : $CNames \rightarrowtail \downarrow Class$. Note that this is the only part of our framework where we use a global definition. This function gives us the unique reference to the class-side instance. The concrete definition of the function must be given in the specification, i.e. every (object-side) class has to say what its class-side class is. This function *classref*, the *classname* construct and the constraint over the variable *class* are key to our framework: These constructs ensure that only one (and always the same) instance of the subclass of *Class* is referenced by all instances of a class. So this instance is the common class-side instance for all instances of the object-side.

This so far is the general framework. Now we can specify the logging class using the Singleton pattern in the third step. We first define the class-side *LogClass* and let it inherit from *Class*. This class defines the static variables and methods for the log class: analogous to the Java class given in section 2, a variable *instance* and a method *getInstance* is declared. The object-side $Log_2$ on the other hand is a subclass of *Object*. It contains the same variables and methods as before, and in addition, by imposing a new constraint on the inherited variable *class*, specifies its class-side to be *LogClass*. Note that unlike the Java program we cannot prohibit arbitrary instantiations of the $Log_2$ class (for one thing, because $Log_2$'s INIT needs to be visible to *LogClass*).

*LogClass*

$\upharpoonright (getInstance)$

*Class*

$instance : Log_2$

INIT

$instance.\text{INIT}$

*getInstance*

$inst! : Log_2$

$inst! = instance$

*Log*$_2$

$\upharpoonright (log, getMessage, \text{INIT})$

*Object*

...

...

$class \in LogClass$

...

...

With this we have finished the definition and usage of the framework. Now we can use the Singleton pattern. Consider again the class $C$ from the last section having a variable of type *Log*. This class is next changed: the initialization of this variable now proceeds via an operation of class *LogClass*. Class $C$ however does not need to know what the class-side of *Log*$_2$ is, this can be calculated by using function *classref*.

*C*$_2$

$l : Log_2$

...

INIT

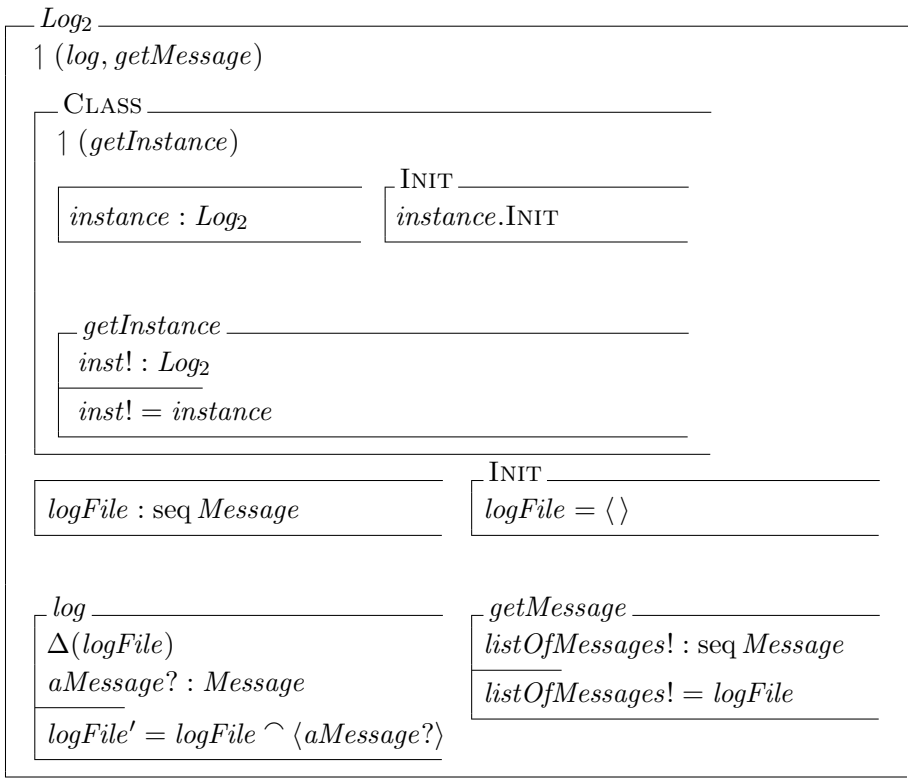
$classref('Log_2').getInstance[l/inst!]$

$Op \mathrel{\widehat=} l.log[''AnError''/aMessage?]$

...

The variable $l$ holds a reference to the one instance of *Log*$_2$. Moreover, it is guaranteed that in *all* instances of $C_2$, $l$ will have the same object identity.

Using the above modeling via classes gives us a good possibility to model class operations and class variables, but it requires a lot of writing. Thus we introduce a shortform for static elements by inserting a CLASS schema directly into an Object-Z class. The log class would then take the following form:




Since the class-side class is now written inside the object-side class, the constraint $class \in LogClass$ does not need to be given anymore. Using this shortform we can introduce another specification convention. All operations and variables of the class-side used by the object-side, which should not used by other classes, can be omitted in the visibility list. The transformation to the explicit class structure then adds them to the visibility list.

## 4 Refinement

We are ultimately interested in using the concept of static elements in a refinement-based design, where we start with an abstract model and gradually refine it into a specification close to an actual implementation. Being a programming concept, static elements will most likely appear in later stages when we introduce object-oriented programming structures into our design. Thus we have to see what refinement means in this setting at all, and furthermore whether the introduction of static elements is a valid refinement step.

Since our framework uses standard Object-Z concepts, we can also use the standard notion of refinement for Object-Z, and downward and upward simulations as proof technique. Our specifications will usually consist of several classes $A, A_1, \ldots, A_n$ , one of which can be considered as the *system* class (e.g. the class

$C_2$ in our example). We are in interested in the behaviour of this system class and the other classes are used for defining it (like a Java class with `main` method using other classes). Following [4] we write this as

$$(A \bullet A_1, \ldots, A_n)$$

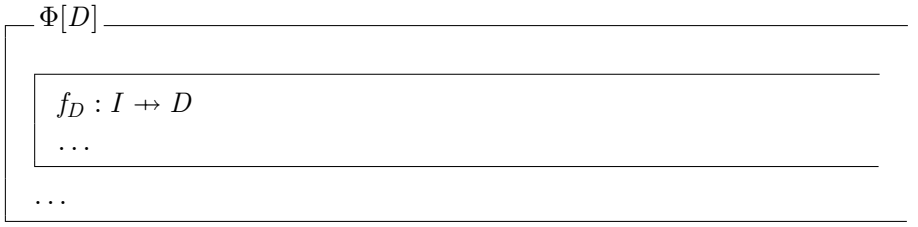
This means in particular that the class $A$ may have references to objects of classes $A_1, \ldots, A_n$, i.e. have variables of type $A_i$, $i = 1, \ldots n$. In our framework we in particular have the object-side containing a reference to the class-side via the variable *class*, and possibly also vice versa. The appropriate notion of refinement is thus *class refinement*, $\sqsubseteq$, proven using class simulations [4], which - like normal data refinement - can be split into upward and downward simulations. Here, we just consider downward simulations:

**Definition 4.1** Let $(A, A_1, \ldots, A_n)$ and $(C, C_1, \ldots, C_m)$ be two system definitions with Object-Z classes. Then $C$ is a *downward simulation* of $A$, $A \sqsubseteq_{ds} C$, if there is a retrieve relation $R$ on $A.\textsc{State} \wedge C.\textsc{State}$ such the following holds:

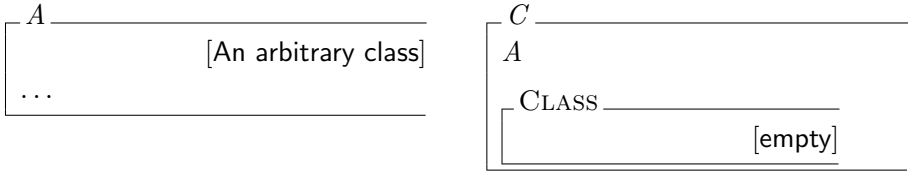
$$\forall\, C.\textsc{State} \bullet C.\textsc{Init} \Rightarrow (\exists\, A.\textsc{State} \bullet A.\textsc{Init} \wedge R)$$
$$\forall\, A.\textsc{State};\ C.\textsc{State} \bullet R \Longrightarrow (\text{pre}\, A.Op \Longleftrightarrow \text{pre}\, C.Op)$$
$$\forall\, A.\textsc{State};\ C.\textsc{State};\ C.\textsc{State}' \bullet R \wedge C.Op \Longrightarrow (\exists\, A.\textsc{State}' \bullet R' \wedge A.Op)$$

In general, class refinement is not compositional (although it can be made so [12]), and thus we cannot separately refine the class and the object-side and hope for a refinement of their composition: if $A \sqsubseteq C$ holds on the object-side and $AClass \sqsubseteq CClass$ on their corresponding class-sides, then we cannot in general deduce $(A \bullet AClass) \sqsubseteq (C \bullet CClass)$.

However, the introduction of static elements into our designs is a valid refinement step. When gradually refining a specification towards an implementation, we will get to the point where we like to introduce static elements for the first time. As we have seen in our previous example this might at least require a change of two parts of the specification: first, we extend some class $A$ by letting it inherit from *Object* (plus adding a constraint for *class*) and add a class-side, and second, we change the other classes using $A$. The first part will always be a refinement, as we will make precise in the following. Consider some arbitrary class $\Phi$ using a number of objects of some class $D$, indexed over some set $I$, plus possibly other variables:



Here, we function $f_D$ is used to represent arbitrary structures of $D$ objects, like sets, sequences or just single variables. This will be our system class. At the beginning, $\Phi$ uses objects of class $A$, i.e. the use $\Phi[A]$. Note that this in particular means that, when referring to $D$'s variables and operations within $\Phi$, these can only be variables and operations present in $A$. Now we extend $A$ with a class-side (first of all empty), thereby getting $C$.




This extension is a valid refinement:

**Theorem 4.2** *Let $\Phi[D]$ be the above given class, $A$ an arbitrary Object-Z class and $C$ a subclass of $A$ adding an empty* CLASS *schema to $A$. Then*

$$(\Phi[A] \bullet A) \sqsubseteq_{ds} (\Phi[C] \bullet C, CClass)$$

For a proof of this we have to give a retrieve relation relating $\Phi[A]$ to $\Phi[C]$. This can be done in two steps (following the way of showing refinements for promotions in Z and for references in Object-Z, see [4]). Essentially our result is a slight extension of Theorem 17.3.1 in [4]. We first construct a retrieve relation $R$ for proving $A \sqsubseteq (C \bullet CClass)$:

$$\begin{array}{|l}
\hline
R \\
A.\text{STATE} \\
C.\text{STATE} \\
\hline
\theta A.\text{STATE} = \theta C.\text{STATE} \lhd \{class\} \\
\hline
\end{array}$$

The bindings of variables in $A$ and $C$ coincide except for the additional variable *class* in $C$. Out of this representation relation we construct one for $\Phi$ itself:

$$\begin{array}{|l}
\hline
P_R \\
\Phi[A].\text{STATE} \\
\Phi[C].\text{STATE} \\
\hline
\operatorname{dom} f_A = \operatorname{dom} f_C \\
\forall\, i : \operatorname{dom} f_A \bullet \exists\, R \bullet \theta\ f_A(i) = \theta A.\text{STATE} \land \\
\qquad\qquad\qquad\qquad \theta\ f_C(i) = \theta C.\text{STATE} \\
\theta\ \Phi[A].\text{STATE} \lhd \{f_A\} = \theta\ \Phi[C].\text{STATE} \lhd \{f_C\} \\
\hline
\end{array}$$

Note that the term $\exists\, R$ here uses $R$ as a schema name. This states that $\Phi[A]$ and $\Phi[C]$ coincide except for the ranges of $f_A$ and $f_C$, where, however, $f_A(i)$ and $f_C(i)$

are related via $R$. The relation $P_R$ can be used to show a downward simulation between $(\Phi[A] \bullet A)$ and $(\Phi[C] \bullet C, CClass)$. Intuitively, this is true since $\Phi$ is not using any of the new functionality provided by $C$. The actual proof of the three conditions of downward simulation essentially relies on this argument. Moreover, we can also extend this result to non-empty class-sides, using exactly the same argument. Here, we however need one small sidecondition: since the class-side can now also have variables, we have to make sure that there are valid bindings for the class-side. In particular, there must be a binding in the class-side that satisfies the INIT predicate. Using the notation of above, but this time with an arbitrary class inside $C$, the condition is

$$\exists\, CClass.\text{STATE} \bullet CClass.\text{INIT}$$

If this condition holds true, we can use exactly the same representation relation as before to show a downward simulation between $\Phi[A]$ and $\Phi[C]$. Thus an introduction of static elements into some class without changing the other classes is a valid refinement step.

The last step towards a proper usage of these elements would consist of changing the classes holding references to classes with static elements (in our log example, $C$). Since this change essentially depends on the application itself, no general result can be established for it. Instead, this last step always needs an individual proof of refinement.

# 5 Conclusion

In this paper, we have proposed a framework for static elements in Object-Z. Except for a slight extension of Object-Z's expression language, the framework is completely embedded into Object-Z. It follows Smalltalk's principle of making everything an object. This extension with static elements allows for writing specifications which already possess the object-oriented structuring later employed in an implementation. We have furthermore shown that static elements can be introduced into a specification by refinement. Thus this new concept can be well applied in a stepwise design by refinement.

As future work we would like to evaluate the practical usefulness of this new concept, in particular with respect to the modelling of design patterns in Object-Z.

# References

[1] Sandrine Blazy, Frederic Gervais, and Regine Laleau. Reuse of Specification Patterns with the B Method. *Formal Specification and Development in Z and B*, 2651:40, 2003.

[2] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and System Modeling*, 4(3):277–296, 2005.

[3] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, 1998.

[4] J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Application*. Springer, 2001.

[5] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison Wesley, 1985.

[7] A. Goldberg. *Smalltalk 80*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1989.

[8] A. Hussey. Using Design Patterns to Derive PAC Architectures from Object-Z Specifications. In *TOOLS (32)*, pages 40–51. IEEE Computer Society, 1999.

[9] S.-K. Kim and D. A. Carrington. A rigorous foundation for pattern-based design models. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 242–261. Springer, 2005.

[10] Kevin Lano. *Formal Object-Oriented Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[11] T. McComb. Refactoring Object-Z Specifications. In M. Wermelinger and T. Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.

[12] T. McComb and G. Smith. Compositional Class Refinement in Object-Z. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2006.

[13] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[14] S. Stepney, F. Polack, and I. Toyn. A Z Patterns Catalogue I: Specification and refactorings, v0.1. Technical Report YCS-2003-349, University of York, 2003.

[15] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.