The Object-Z specification language

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University

July 21, 2022

Contents

List of Figures

1 Introduction

Object-Z is an extension to Z that can support object-oriented specifications. In this chapter we will introduce the main features of Object-Z through examples.

2 Defining a class

Object-Z uses the following (minimum) template to define a class:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ClassName}} \\
\hline
< \mathit{visibility\ list} > \\
< \mathit{parent\ class} > \\
< \mathit{state} > \\
< \mathit{initialization\ of\ state} > \\
< \mathit{list\ of\ operations} > \\
\hline
\end{array}
$$

The schemas included in the template can be divided into state (including state initialization) and operation schemas.

Additionally, a class can be parameterized.

3 Class Stack

Our first example is the definition of an unbounded stack, the complete definition of which is shown below. We describe the parts of the class in the subsequent paragraphs:

$Stack[T]$

$\upharpoonright (Push, Pop, Top)$

$$
\begin{array}{l}
elements : seq\ T \\
count : \mathbb{N} \\
\hline
count >= 0
\end{array}
$$

$INIT$

$$
\begin{array}{l}
elements = \langle\rangle \\
count = 0
\end{array}
$$

$Push$

$$
\begin{array}{l}
\Delta(elements, count) \\
el? : T \\
\hline
elements' = \langle el? \rangle ^\frown elements \\
count' = count + 1
\end{array}
$$

Pop

$$
\begin{array}{l}
\Delta(elements, count) \\
el! : T \\
\hline
count > 0 \\
el! = head(elements) \\
elements' = tail(elements) \\
count' = count - 1
\end{array}
$$

Top

$$
\begin{array}{l}
el! : T \\
\hline
count > 0 \\
el! = head(elements) \\
elements' = elements \\
count' = count
\end{array}
$$

Visibility list

Class features (attributes and operations) in Object-Z can be public or private. The visibility list indicates the public features of a class. We say that these features are *exported*. In the example, the visibility list

$$\upharpoonright (Push, Pop, Top)$$

indicates that operations *Push*, *Pop*, and *Top* are all public. The visibility list defines the interface of the class: The way in which instances of the class (objects) can interact with their environment. When all features are in a class' interface, then a visibility list is not required, i.e. in the absence of a visibility list, all features are public.

Parent class

In this example, the definition of class *Stack* does not inherit any other class definition.

State

We have chosen to define a parameterized class, as we want to be able to instantiate it by holding any type of elements in its collection. The state is defined by two variables. One variable holds a sequence of type T which will contain the elements of the collection:

$$elements : seq\ T$$

Another variable will hold the number of elements held in the collection:

$$count = \mathbb{N}$$

Initialization of state

The initial state schema refers to the state variables *elements* and *count* though these are never declared in this schema. The schema models conditions that hold initially:

Upon instantiation of the class, *elements* is initialized to the empty sequence:

$$elements = \langle\rangle$$

Additionally variable *count* is initialized to 0:

$$count = 0$$

The predicate of the state schema represents the class invariant. Thus, we need to specify that the value of variable *count* never becomes negative:

$$count >= 0$$

List of operations

The state schema is implicitly included in all operations, i.e. all operations can refer to *elements*, *elements′* *count* and *count′*. Each operation may include a list (called a Δ-list) which contains those attributes that can be modified by the operation.

For operation *Push* there is no precondition on the size of the stack as this is an unbounded collection. The postcondition states that upon successful termination of the operation, the collection should include the element as the head of the sequence and that the size of the collection must have been increased by one:

$$\begin{array}{l} elements' = \langle el? \rangle ^\frown elements \\ count' = count + 1 \end{array}$$

The statement $\langle el? \rangle ^\frown elements$ shows the concatenation of $\langle el? \rangle$ with *elements*. It is important to note that *el*? must be transformed into a sequence with $\langle el? \rangle$ as the concatenation operation requires that both its arguments must be sequences.

For operation *Pop*, the precondition states that the collection must be non-empty:

$$count > 0$$

The postcondition states that the element retrieved was the head of the sequence as well as the current collection contains the remaining elements while its size has been decreased by one:

$$\begin{array}{l} el! = head(elements) \\ elements' = tail(elements) \\ count' = count - 1 \end{array}$$

Operation *Top* is similar to *Pop* and they share the same precondition. The postcondition states that the operation has not changed the state of the collection (i.e. the operation returned the first element without removing it from the stack):

$$el! = head(elements)$$
$$elements' = elements$$
$$count' = count$$

3.1 Instantiating a class

We will define a class *IntStack* that will use the definition of generic *Stack*, instantiating it with integers:

IntStack

$items : Stack(\mathbb{N})$

$Push \mathrel{\widehat=} items.Push$
$Pop \mathrel{\widehat=} items.Pop$
$Top \mathrel{\widehat=} items.Top$

State

Class *IntStack* uses the generic *Stack* and instantiates it with integers:

$$items : Stack(\mathbb{N})$$

List of operations

Class *IntStack* uses the dot notation to invoke the operations *Push*, *Pop*, and *Top* from the class $Stack[T]$:

$$Push \mathrel{\widehat=} items.Push$$
$$Pop \mathrel{\widehat=} items.Pop$$
$$Top \mathrel{\widehat=} items.Top$$

3.2 Inheritance

A class may be specified as a specialization or extension of another class using inheritance. A class S can inherit another class P by including the name of the parent class after the visibility list in S.

The subclass inherits every feature (variables, constants, initial state schema and operations), except the visibility list. As a result, the subclass must define its own visibility list. This implies that a feature that is declared private in the parent class may now be declared as visible, and vice versa: A visible feature from the parent class can now be declared as private by not being included in the visibility list of the subclass. Furthermore, Object-Z supports multiple inheritance.

State and behavior

State variables in the parent class are merged with those of the subclass. The subclass may redefine a state variable, but only in a compatible way, expanding or restricting the type of a variable with the same name, for example restricting an integer ($\mathbb{Z}$) variable to one that can hold only positive integers ($\mathbb{N}_1$).

If an operation is redefined in the subclass, the declaration of an operation in the parent class is merged with that of the same operation in the subclass. An operation's predicate part is conjoined with that of the same operation in the subclass.

3.3 Inheritance for specialization: Class BoundedStack

In our example, assume now that we need to subclassify *Stack* to define a bounded stack *BoundedStack*. We will extend the state of the *Stack* class with a variable to hold the maximum allowable number of elements. The inherited state schema of *Stack* is extended with a predicate stating that the number of elements currently held in the stack cannot exceed the capacity of the stack. The state schema of the subclass which is explicitly specified is conjoined with that of the superclass. The class definition is shown below:

$$
\begin{array}{|l}
\hline
\textit{BoundedStack}[T] \\
\upharpoonright (\textit{Push}, \textit{Pop}, \textit{Top}) \\
\textit{Stack}[T] \\
\begin{array}{|l}
\hline
\textit{capacity} : \mathbb{N} \\
\hline
\textit{count} <= \textit{capacity} \\
\hline
\end{array} \\
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\
\hline
\textit{capacity} = 10 \\
\hline
\end{array} \\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Push}} \\
\hline
\hline
\textit{count} < \textit{capacity} \\
\hline
\end{array} \\
\hline
\end{array}
$$

We need to extend the class invariant, making sure that the value of variable *count* never exceeds the value of variable *capacity*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{state}} \\
\hline
\hline
\textit{count} <= \textit{capacity} \\
\hline
\end{array}
$$

Operation *Push* in *BoundedStack* would now enforce the following precondition:

$\textit{count} < \textit{capacity}$

4 Class Queue

Consider the definition of class *Queue*, the parts of which are discussed subsequently.

$$
\begin{array}{|l}
\hline
Queue[T] \\
\upharpoonright (Enqueue, Dequeue) \\
\\
\begin{array}{|l}
\hline
elements : seq\ T \\
count = \mathbb{N} \\
\hline
count >= 0 \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textsc{Init} \\
elements = \langle\rangle \\
count = 0 \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
Enqueue \\
\Delta(elements, count) \\
el? : T \\
\hline
elements' = elements \frown \langle el? \rangle \\
count' = count + 1 \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
Dequeue \\
\Delta(elements, count) \\
el! : T \\
\hline
count > 0 \\
el! = head(elements) \\
elements' = tail(elements) \\
count' = count - 1 \\
\hline
\end{array} \\
\hline
\end{array}
$$

As a queue is accessed on both ends, we need to decide which end of the sequence will be the front and which one will be the rear of the queue. We have chosen to treat the head of the sequence as the front of the queue (see Figure 1), thus we can define the *Dequeue* operation as

$$
\begin{array}{l}
el! = head(elements) \\
elements' = tail(elements)
\end{array}
$$

We treat the end of the sequence as the rear of the queue and we define the *Enqueue* operation as

$elements' = elements \frown \langle el? \rangle$

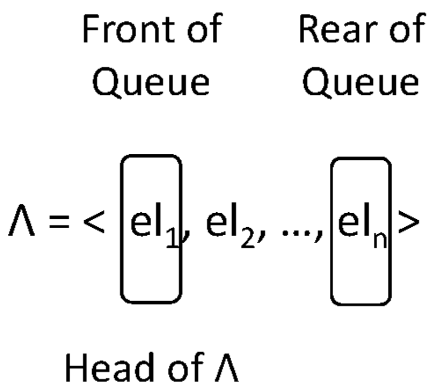

Figure 1: A list data structure that implements a Queue ADT.

4.1 Instantiating a class

We will define a class *IntQueue* that will use the definition of generic *Queue*, instantiating it with integers:

IntQueue

$items : Queue(\mathbb{N})$

$Enqueue \mathrel{\widehat{=}} items.Enqueue$
$Dequeue \mathrel{\widehat{=}} items.Dequeue$

4.2 Inheritance for specialization: Class BoundedQueue

Consider class *BoundedQueue* which is a specialized verion of *Queue*:

$$
\begin{array}{|l}
\hline
\textit{BoundedQueue}[T] \\
\upharpoonright (\textit{Enqueue}, \textit{Dequeue}) \\
\textit{Queue}[T] \\
\quad \begin{array}{|l}
\hline
\textit{capacity} : \mathbb{N} \\
\hline
\textit{count} <= \textit{capacity} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\text{INIT} \\
\textit{capacity} = 10 \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Enqueue} \\
\hline
\textit{count} < \textit{capacity} \\
\hline
\end{array} \\
\hline
\end{array}
$$

4.3 Inheritance for extension: Class ResettableQueue

Consider class *ResettableQueue* which is an extended version of *Queue* that introduces a new operation *Reset* that empties the collection and sets variable *count* to zero.

$$
\begin{array}{|l}
\hline
\textit{ResettableQueue}[T] \\
\upharpoonright (\textit{Enqueue}, \textit{Dequeue}, \textit{Reset}) \\
\textit{Queue}[T] \\
\quad \begin{array}{|l}
\hline
\textit{Reset} \\
\Delta(\textit{elements}, \textit{count}) \\
\hline
\textit{elements}' = \langle\rangle \\
\textit{count}' = 0 \\
\hline
\end{array} \\
\hline
\end{array}
$$

4.4 Multiple inheritance: Class ResettableBoundedQueue

With multiple inheritance (see Figure 2), the schemas of both parents are conjoined to form the schemas of the subclass.

$$
\begin{array}{|l}
\hline
\textit{ResettableBoundedQueue}[T] \\
\upharpoonright (\textit{Enqueue}, \textit{Dequeue}, \textit{Reset}) \\
\textit{BoundedQueue}[T] \\
\textit{ResettableQueue}[T] \\
\hline
\end{array}
$$

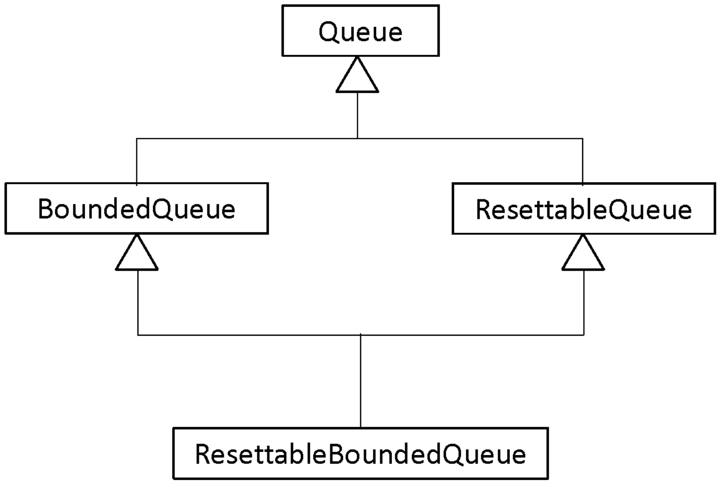

Figure 2: Multiple inheritance.

5 Handling errors and providing robust specifications

Similar to our approach in Z, in addition to the class operations where the precondition is expected to hold, we should always provide operations that handle precondition failures and specify a report to be produced. The exported interface should be an appropriate combination of the schemas as shown in the example that follows.

Let us define type *Report* to hold different kinds of messages as

$$Report ::= ok \mid error$$

Consider the following class, that exports interface Op_1:

Name

$\upharpoonright (Op_1)$

...

Op_1OK

...

...

Success

$result! : Report$

$result! = ok$

Error

$result! : Report$

...

$result! = error$

$$Op_1 \mathrel{\widehat=} (Op_1OK \land Success) \oplus Error$$

6 Additional examples

6.1 Employee management

In this example, we model a system that manages employees and their salaries. The class uses the basic type [*Employee*] , declared outside of the class, as a global type definition.

The visibility list indicates that this class exports the three operations *AddEmployee*, *DeleteEmployee*, and *ModifySalary*. The state of the class consists of a mapping from employees to their salaries, captured by the function *employeeSalary* (see Figure 3). Initially the system has no records.

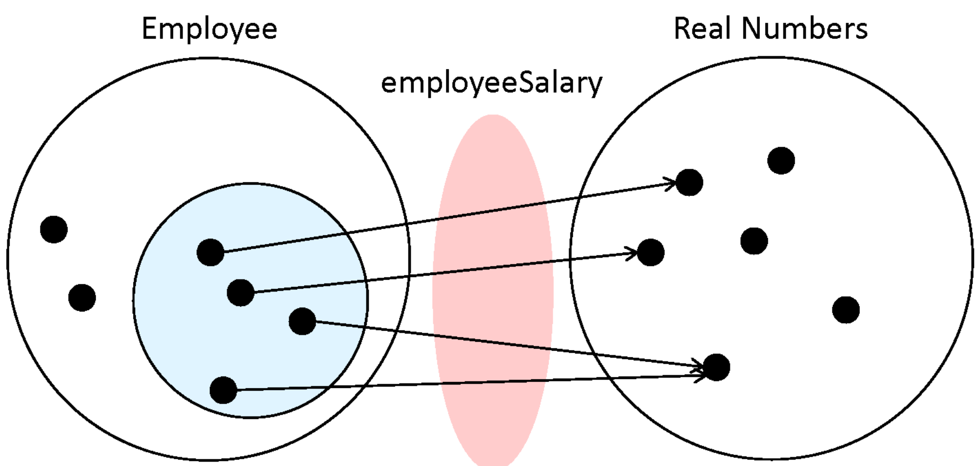

Figure 3: Function *employeeSalary*.

EmployeeSalaries

$\upharpoonright (AddEmployee, DeleteEmployee, ModifySalary)$

$employeeSalary : Employee \pfun \mathbb{R}$

$\forall\, d : \operatorname{dom}\ employeeSalary \bullet employeeSalary(d) > 0.0$

INIT

$employeeSalary = \varnothing$

AddEmployee

$\Delta(employeeSalary)$
$newEmployee? : Employee$
$salary? : \mathbb{R}$

$salary? > 0.0$
$newEmployee? \notin \operatorname{dom}\ employeeSalary$
$employeeSalary' = employeeSalary \cup \{newEmployee? \mapsto salary?\}$

DeleteEmployee

$\Delta(employeeSalary)$
$who? : Employee$

$who? \in \operatorname{dom}\ employeeSalary$
$employeeSalary' = \{who?\} \ndres employeeSalary$

ModifySalary

$\Delta(employeeSalary)$
$employee? : Employee$
$newSalary? : \mathbb{R}$

$newSalary? > 0.0$
$employee? \in \operatorname{dom}\ employeeSalary$
$employeeSalary' = employeeSalary \oplus \{employee? \mapsto newSalary?\}$

6.2 A credit card system

Consider class *CreditCard* that provides basic functionality for a credit card account. The class has attributes *number*, *balance*, and *limit*. The latter can be one of $1,000$, $5,000$, or $10,000$. The class has operations *Withdraw*, *Deposit*, and *GetAvailableFunds*. The state of the class should remain private, whereas its behavior should form its interface. We can provide a specification for class `CreditCard` as follows:

CreditCard

$$\upharpoonright (Withdraw, Deposit, GetAvailableFunds)$$

$$
\begin{array}{l}
number : \mathbb{N} \\
limit : \mathbb{R} \\
\hline
limit \in \{1000, 5000, 10000\}
\end{array}
$$

$$
\begin{array}{l}
balance : \mathbb{R} \\
\hline
balance + limit \geq 0
\end{array}
$$

INIT

$$balance = 0$$

Withdraw

$$
\begin{array}{l}
\Delta(balance) \\
amount? : \mathbb{R} \\
\hline
amount? > 0 \\
amount? \leq balance + limit \\
balance' = balance - amount?
\end{array}
$$

Deposit

$$
\begin{array}{l}
\Delta(balance) \\
amount? : \mathbb{R} \\
\hline
amount? > 0 \\
balance' = balance + amount?
\end{array}
$$

GetAvailableFunds

$$
\begin{array}{l}
amount! : \mathbb{R} \\
\hline
amount! = balance + limit
\end{array}
$$

Consider *CreditCard2*, which is a new version of *CreditCard* that extends the above specification by introducing attribute *withdrawals* that keeps track of the number of withdrawals. We can provide a specification for class `CreditCard2` as follows:

CreditCard2

$\upharpoonright (\mathit{Withdraw}, \mathit{Deposit}, \mathit{GetAvailableFunds})$
$\mathit{CreditCard}$

$\mathit{withdrawals} : \mathbb{N}$

INIT

$\mathit{withdrawals} = 0$

Withdraw

$\Delta(\mathit{withdrawals})$

$\mathit{withdrawals}' = \mathit{withdrawals} + 1$

Let us provide a specification for a class *CreditCompany* that maintains a collection of credit card accounts in an attribute *accounts*. A global property of this class is that all accounts have a unique number. Initially this collection must be empty. In its interface, the class provides operations *AddAccount* and *DeleteAccount* that respectively add or delete instances of *CreditCard* and *CreditCard2* from its collection.

CreditCompany

$\upharpoonright (AddAccount, DeleteAccount)$

$accounts : \mathbb{P}\ CreditCard$
$count : \mathbb{N}$

$\forall\, a_i, a_j : accounts \bullet a_i.number \neq a_j.number$
$count = \#accounts$

INIT

$accounts = \{\}$

AddAccount

$\Delta(accounts)$
$account? : CreditCard$

$account? \notin accounts$
$accounts' = accounts \cup \{account?\}$
$count' = count + 1$

DeleteAccount

$\Delta(accounts)$
$account? : CreditCard$

$account? \in accounts$
$accounts' = accounts \setminus \{account?\}$
$count' = count - 1$

7 Redefinition and removal of operations

We have discussed inheritance for specialization and extension through merging of specifications. What happens when we need to redefine a feature (attribute or operation) in a subclass, or even when we need to remove an operation from the interface of a subclass?

8 Cancellation and redefinition of features through renaming

Consider the definition of class $A[T]$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{A[T]} \\
\hline
\begin{array}{|l}
\hline
x : T \\
y : \mathbb{P}\,T \\
\hline
x \in y \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{Op} \\
\hline
\Delta(x) \\
x? : T \\
\hline
x? \in y \\
x' = x? \\
\hline
\end{array} \\
\hline
\end{array}
$$

Consider class $B[T]$ which inherits class $A[T]$:

$B[T]$

$\upharpoonright (Op)$
$A[y1/y, Op1/Op]$

$y : \text{bag } T$

$x \in y$

Op

$\Delta(x)$
$x? : T$

$x? \in y$
$x' = x?$

The visibility list of the inheriting class is totally independent of that of the inherited class. Hence, inherited features can be effectively canceled — that is, removed from the class interface and, through a combination of renaming and cancellation, redefined.

$A[T]$ is inherited with its variable y renamed to $y1$ and its operation Op renamed to $Op1$. These features are not included in the class's visibility list and hence effectively canceled.

Additionally, class $B[T]$ redefines variable y to be a bag (rather than a set) of elements of type T, and defines a new operation Op.

9 Explicit redefinition and removal of operations

In this section we discuss explicit redefinition and removal of operations (from Duke *et al.*, 1991).

9.1 Redefinition

When inheriting, an operation of the inherited class can be redefined or even removed completely. This is indicated by the keywords `redef` and `remove` respectively. Consider class $DoublePushStack[T]$:

$$
\begin{array}{|l}
\hline
\textit{DoublePushStack}[T] \\
\upharpoonright (Push, Pop, Top) \\
BoundedStack[T][\textbf{redef}\ Push] \\
\begin{array}{|l}
\hline
\textit{Push} \\
\Delta(elements, count) \\
item? : T \\
\hline
count < capacity - 1 \\
elements' = \langle item?, item? \rangle \frown elements \\
count' = count + 2 \\
\hline
\end{array} \\
\hline
\end{array}
$$

In this case, the operation $Push$ is not inherited from $BoundedStack[T]$ but is redefined in class $DoublePushStack[T]$, i.e. no merging with any inherited operation takes place. Class $DoublePushStack[T]$ behaves like $BoundedStack[T]$ but the effect of the $Push$ operation is to place an item twice at the top position of the collection.

9.2 Removal

In this case, operation Pop is removed completely:

$$
\begin{array}{|l}
\hline
\textit{OnlyPushStack}[T] \\
\upharpoonright (Push) \\
Stack[T][\textbf{remove}\ Pop] \\
\hline
\end{array}
$$

Class *OnlyPushStack*[*T*] behaves like a *Stack*, but with only a *Push* operation defined.

10 Bibliography

1. V. S. Alagar and K. Periyasamy, *Specification of Software Systems*, 2nd. ed., Springer, 2011.

2. R. Duke, P. King, G. Rose and G. Smith, *The Object-Z Specification Language*, Technical Report No. 91-1, University of Queensland, April 1991.

3. G. Smith, *The Object-Z Specification Language*, Springer, 2000.