Patterns to Guide Practical Refactoring: examples targetting promotion in Z

Susan Stepney, Fiona Polack, and Ian Toyn

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
{susan,fiona,ian}@cs.york.ac.uk

Abstract. Formal methods such as Z are generally criticised for their lack of practical applicability. As in other areas of software engineering, *patterns* help to construct, analyse and describe formal texts. Once a method has a catalogue of patterns, development can proceed by applying patterns, and by moving from one sort of pattern to another. This paper illustrates a developmental use of patterns. First, we describe the set of patterns that collectively represent the well-known Z structure, *promotion*. We then show how *refactoring* can be used to take an unstructured Z specification in to a promotion structure.

Keyword: Z, patterns, refactoring, development methods

1 Introduction

Pattern languages [Alexander *et al.* 1977] [Gamma *et al.* 1995] and refactoring [Fowler 1999] are programming techniques that can also be fruitfully applied to specification [Stepney *et al.* 2003a] [Stepney *et al.* 2002].

In this paper, we illustrate how patterns and refactoring can be applied, using the example of Z promotion as an *elaboration* of the Delta/Xi pattern [Stepney *et al.* 2003a]. We describe the simplest case of the promotion pattern in terms of a set of sub-patterns that can be used as steps in *generating* a promoted specification. We then provide various *elaboration* patterns, showing how the simple case can be adapted to different circumstances. Finally we use the promotion generative sub-patterns as a basis for *refactoring*, to take an unstructured Z specification into a promotion structure.

2 The Promotion pattern, and its generative sub-patterns

Promotion

Intent: Specify a global system in terms of multiple instances of a local state, and of operations that manipulate a local state.

Problem: A system that is essentially a hierarchy of components has operations and state at different levels in the hierarchy. This is difficult to specify directly, requiring Z structures such as μ and θ schema bindings.

Solution: Build a specification for local state and operations. Build the global state as a composite of local states. Define framing schemas to promote the local operations. Use schema calculus to construct appropriate global operations. (These steps are expanded further in the four sub-patterns below.)
Specimens: Unix File System [Morgan & Sufrin 1984] (first published example); [Woodcock & Davies 1996] (useful refinement laws); [Stepney & Cooper 2000] (an operating system for managing processes on a smart card).
□

We structure the promotion pattern into four sub-patterns, used to guide the generation of a promoted specification. For reasons of space, we merely summarise these sub-patterns here, giving the *intent* and the *solution* only. See [Barden *et al.* 1994, chapter 19] for a fuller description of the underlying structure. [Stepney *et al.* 2003b] gives an example of using these sub-patterns to generate a promoted specification, starting from a local state specification.

The sub-pattern illustrations are documented according to the presentation pattern comment the intent, and illustrated by diagramming the structure [Stepney *et al.* 2003a].

Promotion: local state and operations

Intent: describe the state, operations, initialisation and (precondition for) finalisation, for a single instance of the multi-instance system.
Solution:

$$\begin{aligned} &Local == [\ \text{state components} \mid \text{constraints}\ ] \\ &LocalOp == [\ \Delta Local;\ a? : IN;\ b! : OUT \mid \text{constraints}\ ] \\ &LocalInit == [\ Local\ ' \mid \text{constraints}\ ] \\ &LocalFinalise == [\ Local \mid \text{constraints}\ ] \end{aligned}$$

□

Promotion: global state

Intent: describe the global context of a multi-instance system.
Solution:

- the global state requires a set, $[ID]$ of identities
- gbl maps identities to instances of the local state

$$Global == [\ gbl : ID \pfun Local\ ]$$

□

Promotion: framing schemas

Intent: provide a context that describes how the global state is updated by the results of local operations.

Solution: the context, or "frame" is written as a "framing schema". This establishes the relationship of the state of a local instance to the global state, then shows how the after-state of an operation on that local instance is used to update the global state (the after-state of any suitable global operation) – no details of the global operation or the local operation are required; the frame merely deals with states established by operations defined elsewhere.

The following frame is for any operation that updates the state(s).

- $\Delta Global$, $\Delta Local$ introduce before and after global and local states
- $x?$ is the global identity of an (existing) local instance

$\Phi Update$

$$\Delta Global$$
$$\Delta Local$$
$$x? : ID$$

$$x? \in \operatorname{dom} gbl$$
$$\theta Local = gbl\ x?$$
$$gbl' = gbl \oplus \{x? \mapsto \theta Local\ '\}$$

- $x?$ is in the global set of local instances
- the before state of *Local* is the current local state associated with $x?$
- the global state is updated by overriding the *gbl* function with the maplet from $x?$ to the after-state of *Local*

The frame for introducing a new local instance to the global state is similar to an initialisation.

- declarations are as above, except that there is no *Local* before state

ΦNew

$$\Delta Global$$
$$Local\ '$$
$$x? : ID$$

$$x? \notin \operatorname{dom} gbl$$
$$gbl' = gbl \cup \{x? \mapsto \theta Local\ '\}$$

- $x?$ does not have a mapping in the global state (it is an unused identity)
- The *gbl* function is updated by set union, adding the mapping from $x?$ to the after-state of *Local*

The frame for an operation to remove a local instance from the global state, $\Phi Remove$, is similar, but removes the local instance from the global mapping. The frame is needed only where the removal of a local instance must be verified using operations at the local level.

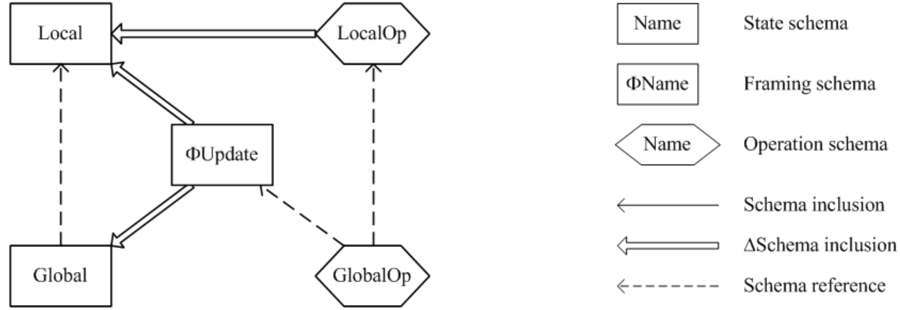
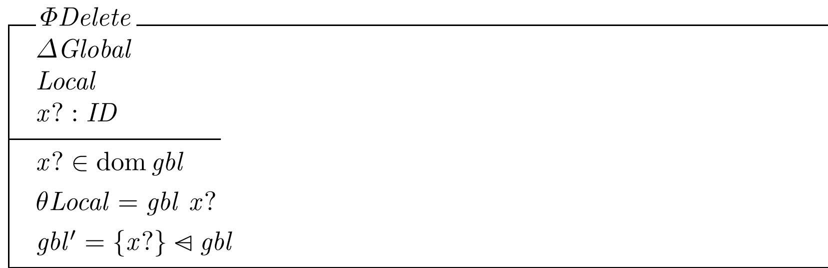


Fig. 1. Structure of the promotion pattern.

– declarations are as above, except that there is no *Local* after state



– $x?$ is in the global set of local instances
– the before-state of *Local* is the current local state associated with $x?$
– the global state is updated by removing the relevant maplet from *gbl*

□

Promotion: global operations

Intent: write a global operation in terms of the defined local operations.
Solution: Use schema calculus to extract the relevant local operation (or after-state of a local operation) and conjoin the appropriate framing schema.

$$GlobalOp == \exists\, \Delta Local \bullet \Phi Update \land LocalOp$$

$$GlobalNew == \exists\, Local\,' \bullet \Phi New \land LocalInit$$

$$GlobalRemove == \exists\, Local \bullet \Phi Remove \land LocalFinalise$$

Define an appropriate global initialisation:

$$GlobalInit == [\ Global\,' \mid \text{constraints}\ ]$$

Note that the type of *GlobalOp* is $[\ \Delta Global;\ x? : ID;\ a? : IN;\ b! : OUT\ ]$.

A diagrammatic representation of the promoted update operations is shown in figure 1. (For a fuller description of the diagrammatic notation, based on an extension of that in [d'Inverno & Luck 2001], see [Stepney *et al.* 2003a].)
□

3 Elaboration Patterns for Promotion

We now present various *elaborations* of the promotion pattern, for coping with cases that do not fit the simple pattern.

Promotion (elaboration): global constraints

Intent: add global state components to the collection of local instances
Solution:

```
┌─ GlobalWithState ─────────────────────────
│ Global
│ purely global state components
├───────────────
│ constraints on global state
│ constraints relating local states
└───────────────────────────────────────────
```

The addition of global state components may require addition of constraints (a) on the global state, and (b) relating the local instance states; the promoted local operations may affect the global state through such constraints. There may also be operations that act on the global state components alone.
Illustration: [Stepney *et al.* 2000] defines a collection of concrete (that is, refined from the abstract) electronic purses. The global state has additional mechanisms to control interoperability of the purses. The extended global state is:

- *conPurse* is the global state for the local purses, *ConPurse*, using identities from the set, *NAME*
- *ether* and *archive* are global-only state elements

```
┌─ ConWorld ────────────────────────────────
│ conPurse : NAME ⇸ ConPurse
│ ether : ℙ Message
│ archive : NAME ↔ Message
├───────────────
│ ...
│ dom archive ⊆ dom conPurse
└───────────────────────────────────────────
```

- the global state element, *archive*, is constrained to only known local purses

All the single purse operations are promoted to the global level. The inputs and outputs from the local operation, $m?$ and $m!$, are not simply mapped into global inputs and outputs; they are also linked to the *ether* (the message transport mechanism) by the framing schema, which extends the framing schemas pattern:

$$
\begin{array}{|l}
\hline
\textit{ΦUpdateCon} \\
\Delta ConWorld \\
\Delta ConPurse \\
m?, m! : MESSAGE \\
name? : NAME \\
\hline
name? \in \operatorname{dom} conPurse \\
m? \in ether \\
\theta ConPurse = conPurse\ name? \\
conPurse' = conPurse \oplus \{name? \mapsto \theta ConPurse\ '\} \\
ether' = ether \cup \{m!\} \\
archive' = archive \\
\hline
\end{array}
$$

In addition to the promoted operations, there are some global-only operations, for example to add messages in the *ether* to the *archive*
□

Promotion (elaboration): internal identifiers

Intent: use a native element of the local state as the identifier. (The global identifiers used in promotion are arbitrary. Where the local state has its own identity, this may be a suitable global identifier.)
Solutions:
There are two solutions, depending on whether the promotion uses only internal identity, or uses both internal and global identity redundantly.
Solution 1: Internal identity only
Adapt Promotion: local state and operations thus:
Include the identity as a native attribute of the local state

$$Local == [\ self : ID;\ \ldots\ ]$$

Include a constraint in local operations that the identity is not changed by the operation

$$LocalOp == [\ \Delta Local;\ \ldots \mid self' = self \land \ldots\ ]$$

Adapt Promotion: global state thus:
Make the global state the set of local instances, with different local states having different identities.

$$Global == [\ gbl : \mathbb{P}\, Local \mid \forall\, x, y : glb \mid x \neq y \bullet x.self \neq y.self\ ]$$

Global operations must include constraints to check the identity of local instances. For exapmle, the frame for the update operations modifies the predicates of the framing schemas pattern as follows.

$$
\begin{array}{|l}
\hline
\Phi Update \\
\Delta Global \\
\Delta Local \\
x? : ID \\
\hline
self = x? \\
\theta Local \in gbl \\
gbl' = (gbl \setminus \{\theta Local\}) \cup \{\theta Local\ '\} \\
\hline
\end{array}
$$

- the local state's *self* component matches the given identifier, $x?$
- the local instance occurs in the global state *gbl*
- the update replaces the local instance in the global set of instances with the result of the local operation

The frame for the operation to add a new piece of local state is as follows.

$$
\begin{array}{|l}
\hline
\Phi New \\
\Delta Global \\
Local\ ' \\
x? : ID \\
\hline
\forall x : gbl \bullet x.self \neq x? \\
self' = x? \\
gbl' = gbl \cup \{\theta Local\ '\} \\
\hline
\end{array}
$$

- no pre-existing local state has the given identifier, $x?$
- the new local state has identifier $x?$
- the update adds the new local state to the global set

Solution 2: External and internal identities
The local state and operation definitions are the same as for solution 1.
Adapt Promotion: global state thus:

Make the global function an injection, capturing the one-to-one correspondence between 'external' and 'internal' names, and ensure that, for any instance, external and internal identities are the same.

$$Global == [\ gbl : ID \rightarrowtail\!\!\!\!\mapsto Local \mid \forall x : \mathrm{dom}\, gbl \bullet (gbl\ x).self = x\ ]$$

Example: In the concrete specification of the electronic purse, the purse has a name; this is the name that the purse is known by at the external level. The local state is *ConPurse*:

$$ConPurse == [\ name : NAME;\ \ldots \mid \ldots\ ]$$

Adapt Promotion: global state thus:

- *ConWorld* uses the same type as the local identifier for the domain of the global *conPurse*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ConWorld \quad}} \\
conPurse : NAME \twoheadrightarrow\!\!\!\!\!\!+ \; ConPurse \\
\ldots \\
\hline
\forall n : \mathrm{dom}\, conPurse \bullet (conPurse\ n).name = n \\
\ldots
\end{array}
$$

– every purse's internal name must be the same as the name by which it is known in the global system.

The framing schemas pattern is applicable, as the additional constraint is carried forward in the $\Delta ConWorld$ inclusion.
Specimen: Hall's style [Stepney *et al.* 1992, chapter 3] combines both these variants. It has a set of local states, and it has a *derived* mapping from (external) identities to local states:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad HallStyle \quad}} \\
gbl : \mathbb{F}\, Local \\
idGbl : ID \nrightarrow Local \\
\hline
idGbl = \{\ l : gbl \bullet l.self \mapsto l\ \}
\end{array}
$$

□

Promotion (elaboration): combine promotions

Intent: specify a system that conforms to the local-global format for the promotion pattern, but has different kinds of local instance.
Solution:

– Each kind of local instance is associated with a separate component of the global state.

$$Global == [\ gbl1 : ID1 \nrightarrow Local1;\ \ldots;\ gbln : IDN \nrightarrow LocalN\ ]$$

– Constraints could be added as appropriate.

Example: [Stepney & Cooper 2000]'s specification of a Smart Card Operating System specifies three types of application instance that the operating system has to control: fixed ISO applications, user programmable applications, and (for modelling reasons) 'absent' applications. It is possible to promote these three types of local instances individually:

– *fixed* and *user* follow the global state pattern.
– *absent* has no local state other than an identifier, so the internal identifier pattern of promotion is applied.

$$
\begin{array}{|l}
\hline
\textit{CardGlobal} \\
\textit{fixed} : \textit{ID} \pinj \textit{Fixed} \\
\textit{user} : \textit{ID} \pinj \textit{Appl} \\
\textit{absent} : \mathbb{P}\, \textit{ID} \\
\hline
\mathrm{disjoint}\langle \mathrm{dom}\, \textit{fixed}, \mathrm{dom}\, \textit{user}, \textit{absent} \rangle \\
\hline
\end{array}
$$

- In this case, the global identifiers for each promotion are drawn from the same set, *ID*, and so are constrained to be disjoint.

Alternatively, combine the various local types with a free type disjoint union, and use this in the simple promotion pattern. (This results in much extracting of state from the free type machinery, however.)
□

Promotion (elaboration): multi-promotion

Intent: specify a system that comprises multiple instances, but has global operations that may affect more than one local instance.
Solutions: the local state and operations and global state patterns are applied. The framing schemas include multiple local instances. There are two possible patterns. The first applies when the global operations affect a fixed number of local instances. The second is more general, and applies when the number of local operations affected by the global operation is variable or unknown.
Solution 1: A fixed number of local instances, illustrated for two instances, but extensible to as many as are practical or desired.

Adapt Promotion: framing schemas thus:

- The (otherwise identical) local states are declared twice, distinguished by decorations. (Note that in $\Delta Local_1$, the decoration applies to the whole $\Delta Local$ phrase, and thus yields $Local_1$ and $Local'_1$.)

$$
\begin{array}{|l}
\hline
\Phi Update2 \\
\Delta Global \\
\Delta Local_1 \\
\Delta Local_2 \\
x?, y? : ID \\
\hline
\{x?, y?\} \subseteq \mathrm{dom}\, gbl \\
x? \neq y? \\
\theta Local_1 = gbl\ x? \\
\theta Local_2 = gbl\ y? \\
gbl' = gbl \oplus \{x? \mapsto \theta Local'_1, y? \mapsto \theta Local'_2\} \\
\hline
\end{array}
$$

- the provided identifiers are different instances from the set of identifiers used in the global system

– each local state instance is bound to one of the input identifiers
– the result of the global operation overrides the global state with the mappings from each identifier to the after-state of the local operation on its local state

Then the global operation becomes

$$GlobalOp == \exists \Delta Local_1;\ \Delta Local_2 \bullet \Phi Update2 \land ALocalOp_1 \land AnotherLocalOp_2$$

The type of $GlobalOp$ is $[\ \Delta Global;\ x? : ID;\ a?_1, a?_2 : IN;\ b!_1, b!_2 : OUT\ ]$. If the inputs or output need to be the same for both operations, renaming is required.

Illustration: This pattern is illustrated in the specification of the electronic purse transfer operations, section 4.5.

Solution 2: An arbitrary number of local instances

If there is an arbitrary number of local states all affected by the same operation, they can be bundled into a sequence ls, with a corresponding sequence of inputs, $xs?$, which identifies which local instances are being promoted. The framing schema becomes

$\Phi UpdateAll$

$\Delta Global$
$ls : \mathrm{seq}\,\Delta Local$
$xs? : \mathrm{iseq}\,ID$

$\#xs? = \#ls$
$\mathrm{ran}\,xs? \subseteq \mathrm{dom}\,gbl$
$\forall i : \mathrm{dom}\,ls \bullet \exists \Delta Local \mid \theta\Delta Local = ls\ i \bullet \theta Local = gbl(xs?\ i)$
$gbl' = gbl \oplus \{\ i : \mathrm{dom}\,ls;\ \Delta Local \mid \theta\Delta Local = ls\ i \bullet xs?\ i \mapsto \theta Local'\ \}$

The local operation is cast into a form for use with the framing schema:

$LocalOpS$

$ls : \mathrm{seq}\,\Delta Local$
$as? : \mathrm{seq}\,IN$
$bs! : \mathrm{seq}\,OUT$

$\#ls = \#as? = \#bs!$
$\forall i : \mathrm{dom}\,ls \bullet$
$\quad \exists \Delta Local;\ a? : IN, b! : OUT \mid$
$\quad\quad \theta\Delta Local = ls\ i \land a? = as?\ i \land b! = bs!\ i$
$\quad \bullet LocalOp$

Then the global operation becomes

$$GlobalOp == \exists ls : \mathrm{seq}\,\Delta Local \bullet \Phi UpdateAll \land LocalOpS$$

The type of $GlobalOp$ is $[\ \Delta Global;\ xs? : \mathrm{iseq}\, ID;\ as? : \mathrm{seq}\, IN;\ bs! : \mathrm{seq}\, OUT\ ]$.
Illustration: Consider a local state, $Counter$ that holds an incrementable natural number. An operation on the local state resets the number to zero. Following the local state and operations pattern, these are defined:

$$Counter == [\ c : \mathbb{N}\ ]$$
$$Reset == [\ \Delta Counter \mid c' = 0\ ]$$

A global system of counters can be defined using the global state pattern:

$$CounterSystem == [\ sys : NAME \rightarrowtail Counter\ ]$$

To simultaneously reset all the identified counters to zero (assuming a $\Phi UpdateAll$ defined following the pattern above, but using the names of this example):

$$\begin{array}{|l}
\hline
\textit{ResetS} \\
ls : \mathrm{seq}\, \Delta Counter \\
\hline
\forall i : \mathrm{dom}\, ls \bullet \exists \Delta Counter \mid \theta \Delta Counter = ls\ i \bullet Reset \\
\hline
\end{array}$$

$$ResetAll == \exists ls : \mathrm{seq}\, \Delta Counter \bullet \Phi UpdateAll \land ResetS$$

This is equivalent to

$$\begin{array}{|l}
\hline
\textit{ResetAll} \\
\Delta CounterSystem \\
xs? : \mathrm{iseq}\, NAME \\
\hline
\mathrm{ran}\, xs? \subseteq \mathrm{dom}\, sys \\
\{xs?\} \ntriangleleft sys' = \{xs?\} \ntriangleleft sys \\
\forall x : \mathrm{ran}\, xs? \bullet \\
\quad \exists \Delta Counter \bullet \\
\quad\quad \theta Counter = sys\ x \land Reset \land sys'\ x = \theta Counter\ ' \\
\hline
\end{array}$$

□

4 Using the promotion patterns

The promotion pattern and its sub-patterns can be used to generate a specification with a promotion structure. For example, [Stepney *et al.* 2003b] show how to use the patterns to generate a promoted bank account specification from a single account specification.

Here we focus on using the promotion patterns to generate a refactoring of an existing unstructured specification. We start with an initial specification

closely based on the 'Abstract World' specification from the electronic purse development [Stepney *et al.* 2000].

The structural promotion pattern is the target pattern for the refactoring, and the generative patterns guide the refactoring steps. Some of the refactor transformations do not seem obvious and might be missed without the generative patterns as a reference. Indeed, some of the transformations temporarily make the specification structure more baroque.

4.1 Starting point: the existing specification

- purses are identified by a name

$$[NAME]$$

The abstract world comprises two mappings

- *balance* maps purse names to their balances
- *lost* maps purse names to the total amount they have lost because of failed transfer operations
- modelling notes: We wish to distinguish empty purses from non-existent purses, so we use a partial function rather than a bag. We use a finite function to ensure that the total balance in the system is well-defined.

$$World == [\ balance, lost : NAME \mathrel{+\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow} \mathbb{N} \mid \operatorname{dom} balance = \operatorname{dom} lost\ ]$$

- the *balance* and *lost* domains are the same, and define the purse names known to the system

There is a standard precondition for operations involving transfer between two purses. This is specified separately and included in each operation to aid readability (following the name predicates pattern [Stepney *et al.* 2003b]).

- transfer operations always have two input purse names and an input representing the value to be transferred between the purses

```
┌─ TransferPre ─────────────────────────────
│ World
│ from?, to? : NAME
│ value? : ℕ
├──────────────
│ {from?, to?} ⊆ dom balance
│ to? ≠ from?
│ value? ≤ balance from?
└───────────────────────────────────────────
```

- the two identified purses are known and distinct (it does not make sense to transfer money from one purse to itself)

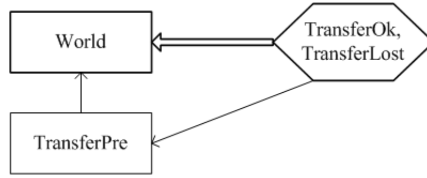


Fig. 2. Structure of the abstract world of electronic purses, before refactoring

– there is sufficient funds for the transfer

There are two transfer operations; one that succeeds, and one that fails. Both have the same declarations.

$$
\begin{array}{l}
\textit{TransferOkay} \\
\hline
\Delta \textit{World} \\
\textit{from}?, \textit{to}? : \textit{NAME} \\
\textit{value}? : \mathbb{N} \\
\hline
\textit{TransferPre} \\
\textit{balance}' = \textit{balance} \oplus \{\textit{from}? \mapsto \textit{balance}\ \textit{from}? - \textit{value}?, \\
\qquad\qquad\qquad\qquad \textit{to}? \mapsto \textit{balance}\ \textit{to}? + \textit{value}?\} \\
\textit{lost}' = \textit{lost}
\end{array}
$$

– a successful transfer decrements *value*? from the *from*? purse's balance, and adds it to the *to*? purse's balance, leaving all other balances unchanged
– success means the *lost* components are unchanged

$$
\begin{array}{l}
\textit{TransferLost} \\
\hline
\Delta \textit{World} \\
\textit{from}?, \textit{to}? : \textit{NAME} \\
\textit{value}? : \mathbb{N} \\
\hline
\textit{TransferPre} \\
\textit{balance}' = \textit{balance} \oplus \{\textit{from}? \mapsto \textit{balance}\ \textit{from}? - \textit{value}?\} \\
\textit{lost}' = \textit{lost} \oplus \{\textit{from}? \mapsto \textit{lost}\ \textit{from}? + \textit{value}?\}
\end{array}
$$

– a lost transfer decrements *value*? from the *from*? purse's balance, and loses it, modelled by adding it to the *from*? purse's *lost* component
– no other purse is affected

Figure 2 represents the structure of the specification before refactoring. It is a simple Delta/Xi pattern; there is also a schema included (as a precondition) in the operations.

The purse system is a Delta/Xi specification that matches the multi-promotion intent: it describes a system made up of instances of a local state and global operations based on operations on multiple (two) local states.

The refactoring steps are based on the four elements of the generative pattern for promotion. The first generative pattern, local state and operations, requires substantial refactoring over several steps. The second generative pattern, global state, emerges as a side effect.

4.2 Step 1: introduce local state

The first goal for refactoring is to define the local state. The system described above has a collection of purses, so the purse becomes the local instance.

The individual purses each have a *balance* and a *lost* component. The schema is formed by removing the identifying functions and predicates from the original state specification. The result matches the state of local state and operations:

$$Purse == [\ balance, lost : \mathbb{N}\ ]$$

Global elements of the original state are refactored to match global state:

$$[NAME]$$
$$World == [\ purse : NAME \ffun Purse\ ]$$

The preservation of meaning for *purse* can be established by a *schema expansion* refactoring [Stepney *et al.* 2002] to a single function from $NAME$ to the schema binding of *balance* and *lost*. This is isomorphic to the original state schema, *World*.

To complete these refactorings, and to preserve the full sense of the original specification, the original precondition and operation specifications have to be refactored to use the local and global states.

- The precondition schema declarations are not affected: purses are still identified by name, even though the name has been relocated to the global context.

TransferPre

$$World$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$

$$\{from?, to?\} \subseteq \operatorname{dom} purse$$
$$to? \neq from?$$
$$value? \leq (purse\ from?).balance$$

- Each reference to the *balance* function in the original precondition is replaced by a suitable reference to the *purse* function. Otherwise, the precondition predicates are unchanged.

The operations introduce global and local state schemas. To allow global operations to use the local state, explicit after-state purses are constructed.

– A successful transfer involves explicitly two purses. The refactoring is influenced by the form of the elaboration pattern, multi-promotion, in its version for a fixed number of local instances. The declarations are made accordingly. Making a schema binding: ZRM [Stepney *et al.* 2003b] is used to construct purse instances.

$$
\begin{array}{|l}
\hline
\textit{TransferOkay} \\
\Delta World \\
\Delta Purse_1 \\
\Delta Purse_2 \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
TransferPre \\
Purse'_1 = (\ \mu \Delta Purse \mid \theta Purse = purse\ from? \\
\qquad\qquad \wedge balance' = balance - value? \\
\qquad\qquad \wedge lost' = lost \\
\qquad \bullet Purse'\ ) \\
Purse'_2 = (\ \mu \Delta Purse \mid \theta Purse = purse\ to? \\
\qquad\qquad \wedge balance' = balance + value? \\
\qquad\qquad \wedge lost' = lost \\
\qquad \bullet Purse'\ ) \\
purse' = purse \oplus \{from? \mapsto \theta Purse'_1, to? \mapsto \theta Purse'_2\} \\
\hline
\end{array}
$$

– $Purse_1$ has $value?$ subtracted from its $balance$; no other elements is changed.
– $Purse_2$ has $value?$ added to its $balance$; no other elements is changed.
– The global world mappings to the instances of the two purses are overwritten with the after-states of each purse.

The lost case is refactored similarly.

– an unsuccessful transfer involves exactly one purse (we call this $Purse_1$ to use it with the general framing schema)

$$
\begin{array}{|l}
\hline
\textit{TransferLost} \\
\Delta World \\
\Delta Purse_1 \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
TransferPre \\
Purse'_1 = (\ \mu \Delta Purse \mid \theta Purse = purse\ from? \\
\qquad\qquad \wedge balance' = balance - value? \\
\qquad\qquad \wedge lost' = lost + value? \\
\qquad \bullet Purse'\ ) \\
purse' = purse \oplus \{from? \mapsto \theta Purse'_1\} \\
\hline
\end{array}
$$

– *Purse* has *value*? subtracted from its *balance*, and added to *lost*
– the global world mapping to the purse is overwritten with the purse's after-state

This intermediate refactoring preserves the meaning of the original specification, but is otherwise considerably more obscure than the original. Further refactoring is required[1].

4.3 Step 2: introduce local operations

Having extracted the local and global states, attention now turns to the extraction of local operations. These are specified to include the local part of the precondition schema.

– the declaration of *World* has been replaced by the local *Purse*
– The *from*? and *to*? purse declarations are not relevant at the local level.
– The other declaration is as for the original precondition schema.

TransferPre

$$
\begin{array}{|l}
Purse \\
value? : \mathbb{N} \\
\hline
value? \leq balance
\end{array}
$$

– The predicate is the remaining local one from the original precondition schema.

The local operations separately specify transfer from a purse, transfer to a purse and the lost transfer. The schemas are taken to be self-explanatory.

TransferFrom

$$
\begin{array}{|l}
\Delta Purse \\
value? : \mathbb{N} \\
\hline
TransferPre \\
balance' = balance - value? \\
lost' = lost
\end{array}
$$

[1] This first refactoring is, in fact, closer to the form of the actual Purse specification [Stepney *et al.* 2000]. It was written in this form because the multi-promotion pattern was unknown to the authors at the time. See [Stepney *et al.* 2003b] for diagrams of the full specification.

TransferTo

$$\Delta Purse$$
$$value? : \mathbb{N}$$

$$TransferPre$$
$$balance' = balance + value?$$
$$lost' = lost$$

TransferLost

$$\Delta Purse$$
$$value? : \mathbb{N}$$

$$TransferPre$$
$$balance' = balance - value?$$
$$lost' = lost + value?$$

4.4 Step 3: introduce framing schemas

The next generative pattern is framing schemas. Each kind of operation requires a frame. All the local operations are simple changes to the local state. However, at the global level, a successful transfer updates the state of two purses whilst the unsuccessful transfer updates only one purse.

The validity of the purses involved in a transfer must be checked at the global level, since the name identifiers have been assigned only to the global state. This check needs to appear as part of the frame(s).

$\Phi Transfer$

$$\Delta World$$
$$\Delta Purse_1$$
$$\Delta Purse_2$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$

$$\{from?, to?\} \subseteq \operatorname{dom} balance$$
$$from? \neq to?$$
$$\theta Purse_1 = purse\ from?$$
$$\theta Purse_2 = purse\ to?$$
$$purse' = purse \oplus \{from? \mapsto \theta Purse'_1, to? \mapsto \theta Purse'_2\}$$

– the identities of the two purses are checked against the known identities, and are distinct

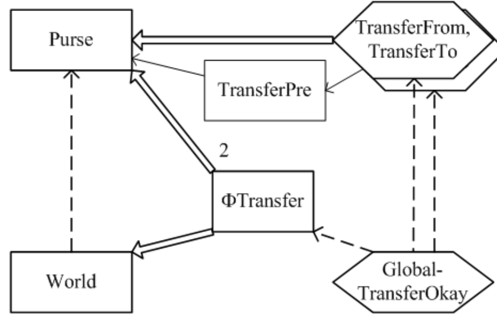


Fig. 3. Structure of the abstract world of electronic purses, after refactoring to a two-state promotion, for the *GlobalTransferOkay* operation

- the *purse* after-state is formed by overriding the mappings for the two instances with the results of the local operations

The single purse *TransferLost* could be provided with a framing schema by an application of the framing schemas pattern. However, it is equally valid, and requires fewer definitions, to think of the unsuccessful global operation as affecting two purses, one of which is unchanged.

4.5 Step 4: Define the global operations

The final step applies the global operations pattern.

$$\begin{aligned} GlobalTransferOkay == {} & \exists\, \Delta Purse\ _1;\ \Delta Purse\ _2 \bullet \\ & \Phi Transfer \wedge TransferFrom\ _1 \\ & \wedge\ TransferTo\ _2[value?\ _1/value?\ _2] \end{aligned}$$

$$\begin{aligned} GlobalTransferLost == {} & \exists\, \Delta Purse\ _1;\ \Delta Purse\ _2 \bullet \\ & \Phi Transfer \wedge TransferLost\ _1 \wedge \Xi Purse\ _2 \end{aligned}$$

4.6 Resulting specification, summary

Gathering together all the pieces produced above, the complete specification of the abstract purse world comprises (a) local specifications, (b) framing schema(s) and (c) global transfer operations that promote the local transfer elements. This could be demonstrated to be isomorphic to the original by a *schema expansion* refactoring [Stepney *et al.* 2002] of both versions, and simplifying. Alternatively, conjectures on the operations could be formulated and proved. Neither is demonstrated here.

The specification using promotion is slightly longer, but more readable. Writing further operations that change purses is straightforward: the local component operations are added, and the global operation pattern applied.

Figure 3 represents the structure of the specification after refactoring. This diagram looks more complicated than the pre-refactoring diagram (figure 2); this

demonstrates how much specification structure was implicit before (hidden in the state and operation schemas). More explicit structure makes a specification easier to comprehend.

5 Conclusion

The refactorings shown here take an unstructured specification into a specific pattern by a series of meaning preserving refactoring steps, guided by the target patterns and sub-patterns.

Refactoring can also be used to take a specification in one pattern in to a different pattern: Z patterns include a number of choice patterns that are equivalent (in some sense) formulations of the same concept; some are better for communication, and some are better for proof or refinement.

Meaning preservation is a crucial component in refactoring. Whilst it may be desirable to evolve the meaning of a specification by a series of small, controlled steps, this is not what is normally meant by refactoring. (In [Stepney *et al.* 2002] we refer to this as *benefactoring.*) In code refactoring, the meaning preservation is demonstrated if all the original functionality is preserved by the refactoring (same inputs lead to same outputs). In specification, meaning preservation may be defined in terms of isomorphic semantic models; it may be defined in terms of mutual refinement; it may be defined in terms of what (refactored) conjectures can be proved true (and, if relevant, proved false). France (private communication) uses a much looser definition of meaning preservation, relating to the intent of the specifiers, a preservation that could not be mechanically checked. The definition can be chosen to suit the application.

Some refactoring is clearly supportable by tools: some tools can already refactor, as part of a proof-assistant role. Thus, schemas can be expanded, preconditions calculated, and other equivalent structures substituted. These refactorings follow a strict mathematical meaning preservation. However, practical refactoring is more likely to need to preserve the properties of a specification, and to demonstrate preservation by reproving the conjectures. This occurred during the commercial specifications that are the motivation for this paper [Cooper & Stepney 2000], [Stepney & Cooper 2000].

More work on refactoring proofs is under way. Such refactorings, especially where guided by patterns, have the potential to guide proof, refinement, and other aspects of formal software engineering.

Acknowledgements

We thank the anonymous referee whose careful reading of an earlier version of this paper resulted in many improvements.

References

[Alexander *et al.* 1977]
Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid

Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[Barden *et al.* 1994]
Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. BCS Practitioner Series. Prentice Hall, 1994.

[Bowen *et al.* 2000]
Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors. *ZB2000: First International Conference of B and Z Users*, volume 1878 of *LNCS*. Springer, 2000.

[Cooper & Stepney 2000]
David Cooper and Susan Stepney. Segregation with communication. In [Bowen *et al.* 2000], pages 451–470.

[d'Inverno & Luck 2001]
Mark d'Inverno and Michael Luck. *Understanding Agent Systems*. Springer, 2001.

[Fowler 1999]
Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[Gamma *et al.* 1995]
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Morgan & Sufrin 1984]
Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Trans. Softw. Eng*, 10(2):128–142, 1984.

[Stepney & Cooper 2000]
Susan Stepney and David Cooper. Formal methods for industrial products. In [Bowen *et al.* 2000], pages 374–393.

[Stepney *et al.* 1992]
Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*. Springer, 1992.

[Stepney *et al.* 2000]
Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical Monograph PRG-126, Programming Research Group, Oxford University Computing Laboratory, 2000.

[Stepney *et al.* 2002]
Susan Stepney, Fiona Polack, and Ian Toyn. Refactoring in maintenance and development of Z specifications and proofs. In *REFINE 2002, Copenhagen*, volume 70(3) of *ENTCS*. Elsevier, 2002.

[Stepney *et al.* 2003a]
Susan Stepney, Fiona Polack, and Ian Toyn. An outline pattern language for Z. 2003. (these proceedings).

[Stepney *et al.* 2003b]
Susan Stepney, Fiona Polack, and Ian Toyn. A Z patterns catalogue I: specification and refactoring, v0.1. Technical Report YCS-2003-349, York, 2003.

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