Implementing Object-Specific Design Patterns Using Per-Object Mixins

Gustaf Neumann and Uwe Zdun
Information Systems and Software Techniques
University of Essen, Germany
{gustaf.neumann,uwe.zdun}@uni-essen.de

Abstract

Object-oriented software system composition is traditionally centered on class-based designs. In this paper we will take a look onto design issues from an object-level point of view and discuss the idea to build designs especially tailored for the object-level. Currently the object-oriented paradigm is still evolving. New ideas, like design patterns, enhance composability on the class-level. Based on the example of three design patterns from [11] (Decorator, Strategy, and Observer) we will show in this paper how to refine class-level designs at the object-level.

We believe that the underlying concepts of a programming paradigm and the capabilities of the programming language should be of comparable expressiveness. Regarding the class-level implementation of design patterns, we have introduced a language construct called filter, providing a powerful language support for class-level patterns. Similarly, in order to implement the object-level patterns presented in this paper, we use another language construct, tailored for the implementation on the object-level, called per-object mixins. This construct is implemented in the scripting language XOTcl, which is an extension of MIT's OTcl.

1 Introduction

Object-orientation is based on the principles of information hiding and abstraction through encapsulation and on specialization through inheritance. This approach has proven well in reducing complexity of software architectures, but unfortunately it still entails certain obstacles and limitations. Currently a central weakness is the composition of objects. Class-level language constructs are able to describe the properties and the behavior of their instances in detail, but they suffer from powerful means to express how classes and objects are composed and how they are inter-related.

The discrepancy between the aim of abstractions hiding their implementation and the reality, where client objects need access to module internals, is another challenging problem [14]. The necessity to let a module address the requirements of several clients, while staying focused enough for each specific client, entails the need for a new model of abstraction. Such a model are open implementations [15], for instances implemented with meta-object protocols [16].

Reflective techniques, like read/write introspection, in conjunction with dynamics of object and class-system, are a solution to allow modules to adapt their descriptive representation to client requirements. A general problem is, how to access these features properly. Approaches, like [16] or [22], use a distinction between a meta-level and a base-level, where the behavior of the base-level is controlled through meta-objects. This is a useful but low-level approach. The per-object mixins used in this paper provide a higher-level interface to let objects be adapted for client requirements and they are completely transparent for client objects.

Moreover, they avoid the distinction between meta-level and base-level. We consider the implied splitting of the tasks of one design entity into two (or more) objects, a base-level and a meta-level object, as the biggest disadvantage of meta-object protocols. We rather propose the usage of meta-classes (see [19]) or per-object mixins in order to be able to decompose base- and meta-part, while preserving the one design entity as one object of the implementation.

Another common problem to object-orientation is that numerous components, with several classes and all their relationships, have to work in concert. The evolving complexity of applications makes it difficult for software engineers to make the right design decisions. Therefore, it is important to make good designs accessible to other software engineers and this way reusable. The object-oriented design community proposes design patterns [11] as a solution to this problem.

Design patterns are a description of situations in which several classes cooperate on a certain task [26]. They typically can be found in the "hot spots" [25] of software architectures. Design patterns are collected in catalogs, like [11, 9]. Most efforts in the literature concentrate on cataloging of patterns. By using design patterns, they become a part of the programmer's paradigm [2].

Unfortunately, in order to reuse the design experience represented in the design pattern in an implementation, the programmer has to recode the pattern for every usage. Therefore, the design pattern is no entity of the programming language. In [3] more problems with design pattern implementation are investigated, i.e. :

- *Traceability:* The pattern is scattered over the objects and, therefore, hard to locate and to trace in an implementation [26]. We also see the problem of traceability of run-time structures, induced by the absence of introspection mechanisms in languages like C++.
- *Self-Problem:* The implementation of several patterns requires forwarding of messages, e.g. an object A receives a message and forwards it to an object B. Once the message is forwarded, references to *self* refer to the delegated object B, rather than to the original receiver A [17].
- *Reusability:* The implementation of the pattern must be recoded for every use.
- *Implementation Overhead:* The pattern implementation requires several methods with only trivial behavior, e.g. methods solely defined for message forwarding.

Some approaches, like [2, 3, 10, 12] provide a language support for design patterns to solve this problem. In [25] seven meta-patterns are identified that define most of the patterns of Gamma et.al. [11]. In [19] we have shown an approach, how to generally language support patterns based on these meta-patterns, using a new class-level language construct, called filter.

Filters are instance methods registered for a class C. Every time an object of class C receives a message, the *filter* is invoked automatically. When the filter is registered, all messages to objects of this class (and all its sub-classes) must pass the filter, before they reach their destination object. Therefore, the filter is a very powerful language construct. In combination with its rich introspection facilities and the dynamics of filter registration it is able to achieve a powerful language support for design patterns, but also has strong meta-programming abilities and can be used as a general tracing facility.

But this power comes with a certain coarseness, when applied to the object-level. Generally it is possible to specialize a filter enough to satisfy every client object's requirements. But we think the more intuitive way is, to provide a language construct, similar to the filter especially tailored for the object-level. For this task we will introduce a language construct called per-object mixin (investigated as a language construct in [20]).

Moreover, we will show that the general idea of refining class-level constructs to the object-level is not limited to programming language constructs. Large program structures, like class-level design pattern, can also be transformed into an object-specific pattern. In this paper we will show such a refinement on the example of the decorator, strategy and observer pattern [11]. We will use the new language construct per-object mixin to implement the object-specific patterns properly. Beforehand, we give a brief overview of the XOTCL language, in which we have implemented the per-object mixins.

2 Extended OTCL

Extended OTCL (XOTCL, pronounced *exotickle*) is an extension of OTCL [28] which is an object-oriented flavor of the scripting language TCL (Tool Command Language [23]). Generally, there is a fast and high quality development of software systems in scripting languages, like TCL. Since they offer a dynamic type system with automatic conversion, they become easily extensible through components (e.g. written in XOTCL, TCL, or C). All components use the same string interface for argument passing and therefore they automatically fit together. The components can be reused in unpredicted situations without change. In [24] and [19] it is pointed out that the evolving *component frameworks* provide a high degree of code reuse, and offer easy usage and rapid application development.

OTCL preserves and extends these important features of TCL. It offers object-orientation with encapsulation of data and operations, single and multiple inheritance, a three level class system based on meta-classes, method chaining and rich read/write introspection facilities, allowing the programmer to change all relationships dynamically (see [28] for details).

In XOTCL every object is associated with a class over the `class` relationship. Classes are ordered by the relationship `superclass` in a directed acyclic graph. Classes are a special objects with the purpose of managing other objects. "Managing" means that a class provides methods to create and destroy instances, and that it provides a repository of methods for its instances ("instprocs") to define their behavior. Furthermore, a classes can be combined through single or multiple inheritance. The instance methods common to all objects are defined in the root class `Object` (predefined or user defined). Since a class is a special (managing) kind of object it is managed itself by a special class called "meta-class" (which manages itself). One interesting aspect of meta-classes is that by providing a constructor, pre-configured classes can be derived. New user-defined meta-classes can be derived from the predefined meta-class `Class` in order to restrict or enhance the abilities of the classes that they manage. All inter-object and inter-class relationships are fully dynamic and can be changed at arbitrary times with immediate effect. Since classes are also objects, all methods applicable for objects can be applied on the class-objects as well.

The OTcl properties provide a good basis for XOTcl. The XOTcl extensions focus on mechanisms to manage the complexity that may occur in large object-oriented systems, especially when systems-parts have to be adapted for certain purposes. In particular we added the following support:

- *Dynamic Object Aggregations*, to provide dynamic aggregations through nested namespaces (objects).
- *Nested Classes*, to reduce the interference of independently developed program structures.
- *Assertions*, to reduce the interface and the reliability problems caused by dynamic typing and, therefore, to ease the combination of many components.
- *Meta-data*, to enhance self-documentation of objects and classes.
- *Per-object mixins*, as a means to improve flexibility of mixin methods by giving an object access to several different supplemental classes, which may be changed dynamically.
- *Filters* as a means of abstractions over method invocations to implement large program structures, like design patterns.

3 Per-Object Mixins

In this section we will give a brief introduction to the new language construct per-object mixin, discussed more deeply in [20]. The construct bases on the method chaining ability of OTcl, which mixes the same-named (or "shadowed") super-class methods into the current method (modeled after CLOS [5]), without explicit naming of the "mixin" method. A method can invoke the shadowed methods by the `next`-primitive, resulting in an unambiguous, linear next-path.

Per-object mixins are a novel approach of XOTcl to handle complex data-structures dynamically on a per-object basis. The term "mixin" is a short form for "mixin class".

> *A per-object mixin is a class which is mixed into the precedence order of an object in front of the precedence order implied by the class hierarchy.*

As a consequence, the per-object mixins extend the method chaining of a single object.

An arbitrary class can be registered as a per-object mixin for an object by the predefined `mixin` method. This method accepts a list of per-object mixins allowing the programmer to register multiple mixins. The following defines the classes `A` and `Mix1` (with a method) and registers `Mix1` on the instance `a` of class `A`.

```
Class A
A instproc proc1 {} {
  puts [self class]
  next
}
Class Mix1
Mix1 instproc proc1 {} {
  puts [self class]
  next
}
A a
a mixin Mix1
```

Since the per-object mixins extend the method chaining, they use the `next`-primitive to forward messages to shadowed methods. If a call on object `a` is invoked, like "`a proc1`", the per-object mixin is mixed into the precedence order of the object, immediately in front of the precedence order resulting from the class hierarchy. The resulting output of the example call is:

```
::Mix1
::A
```

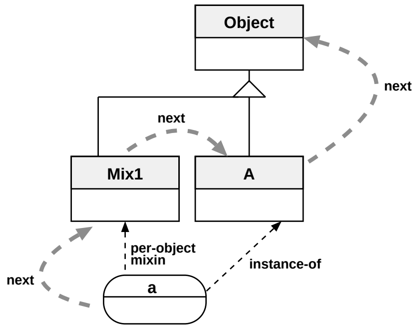

Figure 1: Next-Path with Per-Object Mixins

Mixins may be removed dynamically at arbitrary times by handing the `mixin` method an empty list. For introspection purpose XOTcl offers the `mixin` option of the `info` instance method. A command of the form

objName `info mixin` *?class?*

returns the list of all mixins of the object, when `class` is not specified. The command returns `1`, if `class` is a mixin of the object, or `0` otherwise.

The usual way to specialize descriptive structures in object-oriented languages is inheritance. Since per-object mixins are themselves normal classes they can benefit from specialization through inheritance. This is necessary, because, by being normal classes, instances can be derived directly from them. Without providing an inheritance ability the behavior of a class as a per-object mixin would differ from the behavior, when the class is instantiated. This would be an undesirable inconsistency to the language.

4 Per-Object Design Patterns

4.1 Implementation through Per-Object Mixins

The refinement idea of class-level constructs to the object-level is not only applicable to language constructs but also to certain class structures. E.g. certain design patterns are implementable on the object-level. For the implementation of design patterns on the class-level we propose the language support through filters as presented in [19].

A central property of per-object mixins is that they act transparently for their objects. Therefore, we consider them as a natural way for *object-based decomposition* [18]. An disadvantage of traditional object-based decomposition is that it splits one conceptual entity into multiple separated entities. Traditional object-oriented approaches offer no support to combine several objects to an entity, without loosing the decomposition. An important sub-problem in this context is the mentioned self-problem [17], since forwarding in a decomposed system entails the problem of loosing the self-reference.

Per-object mixins are able to decompose several tasks of one conceptual entity, without referencing to another object. Since design patterns often gain from decompositions, per-object mixins make these patterns conceptual entities. Therefore, we consider per-object mixins as a proper implementation of object-level design patterns.

4.2 The Decorator Design Pattern

Figure 2 shows the implementation of the decorator pattern from [11]. It attaches additional responsibility to an object. Another way to do so is using inheritance, but this is inflexible, because the additions would have to be statically attached. The decorator pattern solves this problem by defining an abstract component type and by letting decorators aggregate one such component. The emerging run-time object structure is a chain of decorators terminated by the concrete component, which is the object being decorated. The pattern is an alternative to sub-classing and, therefore, resembles the per-object mixin. For that reason, this pattern is very easily transferable to the object-level.

The implementation of Gamma et.al. [11] suffers from several problems, due to the used language C++. It

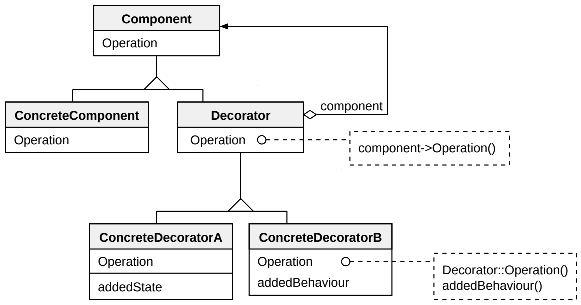

Figure 2: Decorator Design Pattern [11]

entails the self-problem, since the first decorator is the receiving object and this reference gets lost through message forwarding. The pattern is hard to trace in the program code and hard to introspect in the run-time structures. The abstract pattern semantics are mixed into application classes, therefore, the pattern is hard to reuse.

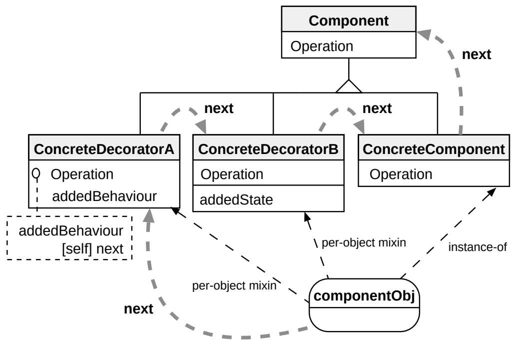

Figure 3: Per-Object Decorator Implemented with Per-Object Mixins

Figure 3 shows the per-object decorator implemented through mixins in general. The component object has several mixins defined automatically performing the added behavior. They are combined through `next`. Afterwards the operation is forwarded to the object's class and is resolved in original precedence order. All operation calls are performed on the same object, so this solution does not suffer from the self-problem. Since mixins are a language construct they are easy to reuse. Furthermore, they are introspectable, so the pattern is traceable in the run-time structures.

As an example we will implement an `Image` class which is decorated by a scrollbar and a menu. We create the three necessary classes. At first we build an abstract component type `Widget` for all three classes, using the `abstract` instance method. For the sake of simplicity we give the classes only one (unspecified) method `draw`:

```
Class Widget
Widget abstract instproc draw args
```

```
Class Image -superclass Widget
Image instproc draw args {
  # do the drawing of the image
}
Class Menu -superclass Widget
Menu instproc draw args {
  # attach menu to an image
  next
}
Class ScrollBar -superclass Widget
  ScrollBar instproc draw args {
  # attach scrollbar to an image
  next
}
```

In order to provide the main window of an image viewer with a scrollbar and a menu, it is only necessary to instantiate the `Image` and to specify the decorating classes `Menu` and `Scrollbar` as mixin classes of the object:

```
Image mainImage -mixin {Menu ScrollBar}
```

Out of its simplicity and shortness this solution reduces the complexity of the pattern radically. Moreover, it does not entail the stated problems like the self-problem, but all these benefits would also apply on decorators using filters. The main difference is, it is applied on the object-level. Only one image is decorated. Without maintenance of arbitrary structures, like decorator-lists, or other implementation overhead we can simply create other `Image`-instances, which are decorated in another way. E.g. a zoomed image that needs no menu is created by:

```
Image zoom -mixin ScrollBar
```

4.3 The Strategy Design Pattern

The strategy design pattern [11] encapsulates a set of algorithms in classes and lets clients use them through an abstract interface. This way the algorithms become dynamically exchangeable. Figure 4 shows the pattern implementation of Gamma et.al.

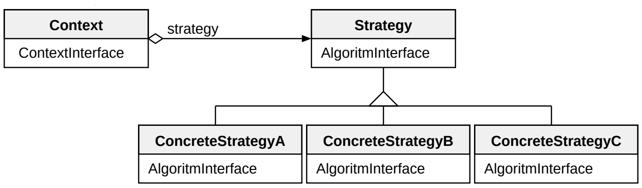

Figure 4: Strategy Design Pattern [11]

The implementation entails similar obstacles like the decorator pattern, which are described in [2]. I.e. it is hard to determine in the program code or at run-time if a class is instantiated as a strategy or as an application object. When the strategy object refers to self, it refers to itself instead of the receiving context object. The explicit forwarding of messages is an implementation overhead, where the software engineer has to explicitly distinguish between containing object and the strategy objects.

The object-specific implementation using mixins entails none of these problems. Since mixins are not instantiated and form a conceptual entity with the containing object, there is a clear distinction of strategy part and containing object part, which is introspectable at run-time. But still the self-problem does not occur. The forwarding is handled automatically – without naming of the mixin-method – by the `next`-primitive of the language. Figure 5 shows the evolving situation for an object `containingObj1` that is attached to one specific strategy. Note, that this solution is also applicable if the strategy depends on the output of the object, since the strategy computation may be put after the `next`-call.

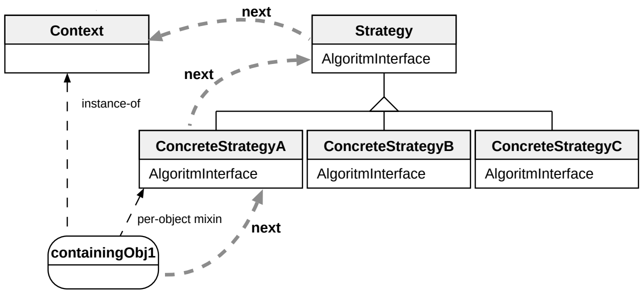

Figure 5: Per-Object Strategy Implemented with Per-Object Mixins

As an example we will implement a comparison strategy for strings. For instances, if a parser should parse a string into a node tree, it has to check which node type a string to be parsed belongs to. In a language offering class-objects, an abstract factory for node objects can question the node class-object, whether a string matches the type of the node or not, before creating a new instance.

Firstly, we create the abstract interface for nodes that just specifies an interface for parsing:

```
Class Node -parameter content
Node abstract instproc parse string
```

For this example we create two special node types, a description node, which holds a literal "Description" and a "or" node holding an "or"-expression.

```
Class DescriptionNode -superclass Node
DescriptionNode set content "Description"
DescriptionNode instproc parse string {
  # parse the description string
}

Class OrNode -superclass Node
OrNode set content "|"
OrNode instproc parse string {
 # parse the or expression string
}
```

On both classes the content of the node class is stored as a class variable. Here, we need two different comparison strategies: Literals, like "description", must exactly match their content, while expressions, like "or", must only contain their content. We implement these comparison strategies as classes:

```
Class ComparisonStrategy
ComparisonStrategy abstract instproc match string

Class Equals -superclass ComparisonStrategy
Equals instproc match string {
  [self] instvar content
  return [expr {$string == $content}]
}

Class Matches -superclass ComparisonStrategy
Matches instproc match string {
  [self] instvar content
  return [string match $string* $content]
}
```

If we now register the two comparison strategies for the corresponding node class-objects, like:

```
DescriptionNode mixin Equals
OrNode mixin Matches
```

a factory can query the class-object for a match and instantiate the proper node class in order to let it parse the given string, e.g. :

```
...
if {[OrNode match $string]} {
  OrNode orNodeObj -parse $string
}
...
```

4.4 The Observer Design Pattern

The observer pattern is a solution to the common problem that a set of depending objects ("observers") rely on the state of one or more observed objects ("subjects"). It fulfills the task of notifying all state changes. Figure 6 shows the observer design pattern as presented in [11].

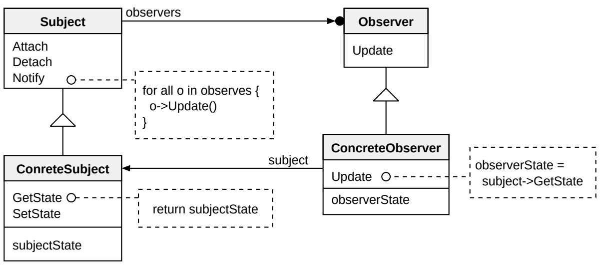

Figure 6: The Observer Pattern [11]

Bosch [2] identifies the problem that the traceability and resuseability of the pattern suffer from the fact that the methods `attach`, `detach` and `notify` do not build up a conceptual entity and that the calls of `notify` must be inserted at every point where a state change occurs. In [19] we present a solution for this problem which is using filters. In the case that only some observed methods of a certain set of subjects (and not all subjects of one type) should be observed, the object-specific solution presented in this section is more appropriate. When all the instances of a whole hierarchy (possibly with all

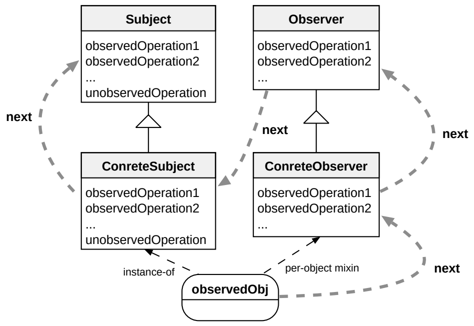

Figure 7: Per-Object Observer Implemented with Per-Object Mixins

their methods) are depending on the subjects, the filter solution should be used.

In Figure 7 a solution for an observer using per-object mixins is presented. The observers are mixins of the observed object and specify a set of observed operations. Additionally the subjects may contain unobserved operations.

As an example for this solution we present a network monitor which observes a set of connections and maintains several views on these (e.g. a diagram and a textual output). This example is strongly resembling the example in [19] and should underline the stated differences of the two language constructs filter and per-object mixins, which are examined more deeply in Section 5.1.

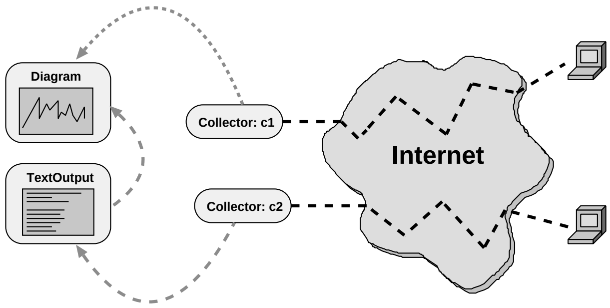

Figure 8: Observer Example

In the implementation the class `Pinger` encapsulates view and collector classes, the collectors are treated as subjects of the observer:

```
Class Pinger
Class Pinger::Collector
Pinger::Collector abstract instproc ping string
Class Pinger::NetCollector -superclass Collector
```

The operation `ping` is the network event, which must be handled by the collector. This is the operation to be observed. The `NetCollector` starts the observation of the network connection in its constructor `init`. The constructor is an unobserved operation:

```
Pinger::NetCollector instproc init args {
  set hostName 132.252.180.67
  set f [open "| /bin/ping $hostName" r]
  fconfigure $f -blocking false
  fileevent $f readable "[self] ping \[gets $f\]"
}
Pinger::NetCollector instproc ping {string} {
  # handle the network event
}
```

The two observers:

```
Class Pinger::Diagram
Class Pinger::TextOutput
```

must specify an observing `ping` method. The text output presentation may look like:

```
Pinger::TextOutput instproc ping {string} {
  puts "PINGER: [self] -- $string"
}
```

The diagram `ping` operation will most likely forward the message to a specified diagram object. For concrete applications the classes of the observers must be registered as mixins, e.g. like the situation in Figure 8, where `c1` has one diagram observer, while `c2` is observed by a textual output and a diagram:

```
Pinger::NetCollector c1 -mixin Pinger::Diagram
Pinger::NetCollector c2 \
  -mixin {Pinger::TextOutput Pinger::Diagram}
```

5 Related Work

We firstly will compare the per-object mixin approach to implement design patterns to our class-level approach "filter". Afterwards we will sketch related works from the literature regarding per-object mixins and finally regarding the idea of language support for design patterns.

5.1 Comparison of Filters and Per-Object Mixins

As shown in [19] filters are able to achieve a reusable language support for design patterns as programming language entities. Furthermore it enables running programs to trace (and manipulate) their structures. This power has sometimes the disadvantage of a certain coarseness, when filters should work on single objects.

Consider a situation where only a single method invoked on some objects should be observed. The filter would have to be defined on these object's classes. In order to fulfill it's observation task it would have to check explicitly on every call to every object of these classes whether the object is in the set of observed objects or not. This is an elaborate solution. Moreover, if the set of observed objects may change dynamically at least a list of them would have to be maintained. Perhaps there would have to be different filters for every object. In any case this would lead to an implementation overhead. All these problems would not occur when using per-object mixins.

Generally, a class-level construct aiming at reduction of complexity in large systems must be able to handle very broad structures, like entire class hierarchies. For some problems on the object-level a finer granularity of the language construct is more appropriate.

Nevertheless, both new language constructs, filter and per-object mixin contain several similarities. A filter which contains code for an explicit delimitation of it's actions to a certain method of a certain instance of one class, is relatively equivalent to a per-object mixin. On the other hand, the same applies for a per-object mixin which is registered on every instance of a class or hierarchy and contains methods for all instprocs of the hierarchy. The elaborateness of both directions emphasizes the sensibleness of the idea of a distinction of object- and class-level for language constructs aiming at complexity.

Both constructs use inheritance for specialization, classes can optionally limit their inheritance abilities, when applied as per-object mixin (see [20] for details). The differences, the two constructs entail, make them well suited for their language level. Per-object mixins are only applied on calls to one object's methods which are defined on the per-object mixin class. The filter handles all calls of all instances of the filtered class and it's sub-classes.

On the example of the observer pattern, investigated in this paper and in [19], it becomes obvious that the granularity of both language constructs makes them reasonable in usage, depending on the application's needs. These should be the basis for the decision which kind of design pattern, object- or class-level, is to be used. The decision for the appropriate language construct follows directly.

5.2 Related Work on Per-Object Mixins

Per-object mixins base on the method chaining mechanism of OTcl, discussed more deeply in [28]. The mechanism provides an automatic method chaining without explicit naming of the mixin method. It is a very flexible programming mechanism and, combined with the unambiguous precedence OTcl offers, they avoid name clashes through (multiple) inheritance at all. The idea of mixins in OTcl are influenced by the lisp extension CLOS [5].

There are several extensions to the idea of mixins discussed. In Agora [27] mixins are treated as named attributes of classes, in order to let the class control how it is extended. A central property of these mixins is that they may be nested. Bracha and Cook [6] analyze different inheritance mechanisms and propose mixins as a general inheritance construct. Inheritance is interpreted as mixin composition. In Jigsaw [7] mixins are used to unbundle the several roles of classes by providing a set of operators controlling effects like inheritance, name-resolution, modification, etc.

These approaches use class-level constructs, also resembling the filter approach. But as they use mixin classes the methods are only applied on certain messages (methods of the mixins), and not on all messages like in filters. This limits the expressiveness of mixin classes in comparison to filters. Since these mixins are applied only on classes their granularity is not fine enough for object-level applications. Nevertheless as a kind of "per-class mixins" they show the similarity between mixin and filter in general.

There are some other class-level concepts, with the ability to intercept and then change, redirect, or otherwise affect messages. The composition filter model [1] introduces the idea of a higher-level object interaction model through abstract communication types (ACTs). This idea was adopted by some approaches in the area of distributed computing, e.g. like Orbix filters [13].

5.3 Related Work on Design Pattern Implementation

Soukup [26] and Bosch [3] have identified problems in the implementation of popular design patterns [11]. Hedin [12] presents an approach to implement design patterns based on an attribute grammar in a special comment marking the pattern in the source code. This addresses the problem of traceability. The comments assign roles to the classes, which constrain them by rules.

The FLO-language [10] introduces a new component "connector" that is placed between interacting objects. The connectors are controlled through a set of interaction rules that are realized by operators. It is also centered on messages exchanges.

The LayOM-approach [2] is an approach for language support for design pattern, partially resembling the filter, since the approach is centered on message exchanges as well. It puts layers around the objects which handle the incoming messages. Every layer offers an interface for the programmer to determine the behavior of the layer through a set of operators which are (statically) given by the layer definition.

These approaches are class-level approaches and therefore suffer from the stated problems regarding the object-level composability. In [4, 3] a composition technique, called superimposition is proposed as a language support for implementation of frameworks. It composes the different behavioral roles a component has to play into one single entity. In conjunction with the layers this approach should also allow the software engineer to implemented a design pattern object-specific, in the sense that it is able to let supplier objects play different roles for different client requirements. Therefore, we think object-level design pattern should be easily implementable using this approach, but in comparison to per-object mixins this class-level approach seems to suffer from being static and offering no introspection.

6 Conclusion

In this paper we have argued for a stronger focus on composability issues regarding the object-level and explained several obstacles in object-oriented programming. We solved these by introducing flexible and fine-grained language constructs. In particular we presented the high-level programming language construct of per-object mixins, and showed its well-suitedness for object composition.

After this introduction we investigated in the application of the language constructs for design patterns, which are class-based. For three example patterns we presented object-level equivalents. We give application examples implemented via per-object mixins. This way, we have shown how to implement these new patterns in a way that they are object-specific, transparent for the client object and not suffering from the stated set of deficiencies occurring in traditional design pattern implementations.

For most class-level design patterns, e.g. those in catalogs like [11, 9], it should be possible and make sense to find an object-specific representation. The subset of patterns that rely on message exchanges, i.e. the patterns relying on the meta-patterns of Pree [25], should benefit from being implemented using per-object mixins, when the pattern is applied on the object-level.

XOTcl is available for evaluation from `http://nestroy.wi-inf.uni-essen.de/xotcl/`.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa: *Abstracting Object Interactions Using Composition Filters*, ECOOP '93, 1993.

[2] J. Bosch: *Design Patterns as Language Constructs*, Journal of Object Oriented Programming, also available as http://bilbo.ide.hk-r.se:8080/~bosch/, 1996.

[3] J. Bosch: *Design Patterns and Frameworks: On the Issue of Language Support*, LSDF'97, also available as http://bilbo.ide.hk-r.se:8080/~bosch/, 1997.

[4] J. Bosch: *Composition through Superimposition*, ECOOP '96 Workshop on Composability Issues in Object-Orientation (CIOO '96), 1996.

[5] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon: *Common Lisp Object System*. In: Common Lisp the Language, 2nd Edition, http://info.cs.pub.ro/onl/lisp/clm/node260.html, 1989.

[6] G. Bracha, W. Cook: *Mixin-Based Inheritance*, in: Proc. of OOPSLA/ECOOP '90, special issue of SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 303–311.

[7] G. Bracha, G. Lindstrom: *Modularity Meets Inheritance*, in: Proc. of IEEE International Conference on Computer Languages, April 1992.

[8] S. Demeyer, P. Steyaert, K. De Hondt: *Techniques For Building Open Hypermedia Systems*, ECHT'94 Workshop, Edinburgh, September1994.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture – A System of Patterns*, J. Wiley and Sons Ltd, 1996.

[10] S. Ducasse: *Message Passing Abstractions as Elementary Bricks for Design Pattern Implementations*, LSDF'97, also available as http://bilbo.ide.hk-r.se:8080/~bosch/lsdf/, 1997.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[12] G. Hedin: *Language Support for Design Patterns using Attribute Extension*, LSDF'97, also available as http://bilbo.ide.hk-r.se:8080/~bosch/lsdf/, 1997.

[13] IONA Technologies Ltd.: *The Orbix Architecture*, August 1993.

[14] G. Kiczales: *Towards a New Model of Abstraction in Software Engineering*, in: Proc. of IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.

[15] G. Kiczales, J. Lamping, C. Videira Lopes, C. Maeda, A. Mendhekar, G. Murphy: *Open Implementation Design Guidelines*, in: Proc. of ICSE'97, Boston, May 1997.

[16] G. Kiczales, J. des Rivieres, D.G. Bobrow: *The Art of the Metaobject Protocol*, MIT Press 1991.

[17] H. Lieberman: *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, in: Proc. of OOPSLA '86, Portland, November 1986.

[18] B. Meyer: *Object-Oriented Software Construction – Second Edition*, Prentice Hall, 1997.

[19] G. Neumann, U. Zdun: *Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages*, in: Proc. of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems, San Diego, May 1999.

[20] G. Neumann, U.Zdun: *Enhancing Object-Based System Composition through Per-Object Mixins*, submitted for publication, 1999.

[21] G. Neumann, U.Zdun: XOTCL*, an Object-Oriented Scripting Language*, submitted for publication, 1998.

[22] A. Oliva, L. E. Buzato: *Design and Implementation of Guarana*, in: Proc. of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems, San Diego, May 1999.

[23] J. K. Ousterhout: *Tcl: An embeddable Command Language*, in: Proc. of the 1990 Winter USENIX Conference, January 1990.

[24] J. K. Ousterhout: *Scripting: Higher Level Programming for the 21st Century*, in: IEEE Computer, Vol. 31, No. 3, March 1998.

[25] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[26] J. Soukup: *Implementing Patterns*, in: J.O. Coplien, D.C. Schmidt (Eds.), *Pattern Languages of Program Design*, Addison-Wesley 1995, pp 395-412, also available as http://www. codefarms.com/publications/papers/patterns.html, 1995.

[27] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen: *Nested Mixin-Methods in Agora*, in: Proc. of ECOOP '93, LNCS 707, Springer-Verlag, 1993.

[28] D. Wetherall, C.J. Lindblad: *Extending Tcl for Dynamic Object-Oriented Programming*, in: Proc. of the Tcl/Tk Workshop '95, Toronto, July 1995.

[29] U. Zdun: *Entwicklung und Implementierung von Ansätzen, wie Entwurfsmustern, Namensräumen und Zusicherungen, zur Entwicklung von komplexen Systemen in einer objektorientierten Skriptsprache*, Diplomarbeit (diploma thesis), Universität Gesamthochschule Essen, 1998.