Enhancing Object-Based System Composition through 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

The management of complexity in large systems is traditionally focused on the modeling and management of classes and hierarchies of classes. In order to improve the compositional flexibility in large systems, this paper turns the focus on objects rather than classes. We will demonstrate that a more powerful object system can ease the development of large systems and can improve the degree of code reuse. The paper introduces a new object-level language construct, per-object mixins, for object-based system composition. It is implemented in the scripting language XOTCL*, which is an extension of MIT's* OTCL.

Per-object mixins extend the method chaining mechanism of OTCL *with the ability to mix classes into the precedence order of an arbitrary object. Per-object mixins can be used to implement state-specific behavior changes in a clean way. We present per-object mixins as a general approach to hide object specifics from client objects transparently.*

1. Introduction

Extended OTCL (XOTCL, pronounced *exotickle*) is an extension of OTCL [22] which is an object-oriented flavor of the scripting language TCL (Tool Command Language [16]). A central property of TCL is the use of strings as the only representation of data. For that reason TCL offers a dynamic type system with automatic conversion. TCL is extensible through components, which are application specific extensions typically written in C. All components integrated in TCL use a string interface for argument passing and therefore they automatically fit together.

The components can be reused in unpredicted situations without change. In [17] and [14] it is pointed out that the evolving *component frameworks* provide a high degree of code reuse, and offer easy usage and rapid application development. In the mentioned papers it is argued that in scripting languages application developers may concentrate primarily on the application task, rather than investing efforts in fitting components together. Therefore, in certain applications scripting languages are very productive and can lead to a high-quality development of software.

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, and method chaining. Instead of a protection mechanism OTCL provides rich read/write introspection facilities, which allows one to change all relationships dynamically (see [22] for details).

These OTCL properties provide a good basis for XOTCL (see Figure 1). The XOTCL extensions focus on mechanisms to manage the complexity that may occur in large object-oriented systems, especially when these systems have to be adapted for certain purposes. Such situations occur frequently in the context of component frameworks. 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.

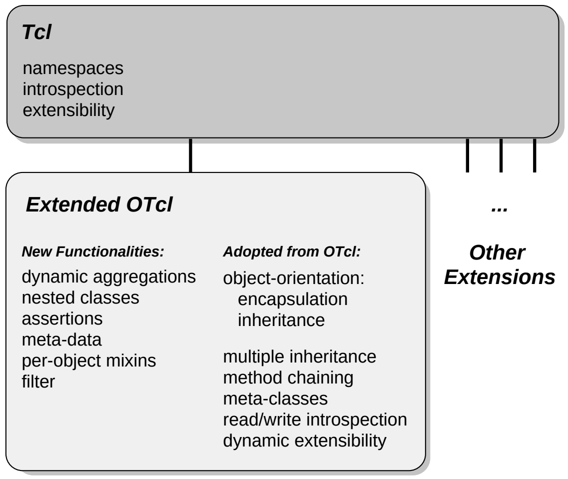

Figure 1. Language Extensions of XOTCL

These language functionalities aim at the management of complexity. An important tasks in this scope is the composition of objects (and their properties). This paper will motivate and explain the idea that (in appropriate applications) high-level language constructs working of the level of (single) objects may be used as a better means than class-level constructs for object composition. Afterwards this paper discusses the novel object-level language construct *per-object mixin* as an application of this idea. In another paper [15] we use per-object mixins to implement and language support object-level design patterns.

2. Object- and Class-Level Constructs

In this section we will give a brief introduction into the object and class concepts of XOTCL (refer to [13, 22] for more details), which are derived from OTCL. Afterwards we will introduce the distinction between object- and class-level approaches and present ideas how to improve composition abilities on the object-level.

2.1. The XOTCL Object and Class System

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. The root of the class hierarchy is the class `Object`, the most general class. A single object can be instantiated directly from this class.

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. Each class inherits behavior through single and 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 (such as `class`, `superclass`) are fully dynamic and can be changed at arbitrary times with immediate effect. Since classes are also objects, all functionality applicable for objects can be applied on the class-objects as well (including the per-object mixins presented in this paper).

2.2. Classes and Composability

Object-orientation organizes program structures around data, while the objects are characterized primarily by their behavior. Object-oriented programming style encourages the access of encapsulated data only through the methods of the object, since this allows data abstractions [21]. Central properties of object-orientation are inheritance and encapsulation of data and operations (conventionally applied on class-level). A weakness is the composition of objects. While the class-level constructs describe the properties and the behavior of their instances in detail, they suffer from powerful means to express how classes and objects are composed and how they are inter-related. One reason is that class structures are not fine-grained enough for several composability issues.

But inheritance and polymorphism still imply certain other problems. Hatton [6] points out that in traditional object-oriented approaches theses two concepts do not fit the human reasoning process very well because of the implied non-locality problems. Since inheritance breaks encapsulation up to a certain degree, the question whether inheritance is needed at all is raised [20]. Object-based languages [21] follow this approach. For example the language Self [19] uses prototypes to combine inheritance and instantiation in order to provide a simple and flexible alternative to inheritance. Lieberman [10] proposes delegation to replace traditional inheritance. His idea is that behavioral sharing between objects can be accomplished by forwarding of message. Weck and Szyperski [20] point out that in order to ensure contracts between objects, e.g. provided through assertions (as in Eiffel [12]), encapsulation is essential, but that the encapsulation is broken by inheritance. Moreover, assertions may be weakened by inheritance and polymorphism, since they have to cover all possible polymorphic structures/several different subtypes. This forces the programmer to formulate very general assertions, easily becoming too broad for the desired task[1].

Several authors propose ways how to enhance the expressive power of inheritance (e.g. [2, 18]). Concepts, like design patterns [5], rather rely on delegation/aggregation then on inheritance. The overall problem is, that composition on the object-level is not suited by class-level constructs sufficiently. We introduced new language features like the filter [14] and class nesting, that operate on whole class hierarchies. These features reduce the complexity in large systems radically, but they imply a certain coarseness, when they should work solely on single instances.

2.3. Enhancing Object Composability

The problems of object composability are discussed widely. In [11] so called 'inhibitors' for composability are examined. Currently there is a lack of a consistent terminology and insight in the relations between different composition techniques. Composition interfaces are neither flexible nor predictable. The composition semantics do not describe the intention of the composition clearly and hinder to manage change propagation.

Often a refinement of class-level constructs to the object-level seems a natural way to enhance object composability. In XOTCL such refinements do exist also in the basic functionalities. For example the dynamic class concept of XOTCL allows one to change the superclass relationship at arbitrary times. In XOTCL an object can dynamically switch its class by altering its `class` relationship. This "reclassing" of objects is an elegant solution for state changes, the implementation of a life-cycle, or of objects changing roles. The chaining of super-class relationship would not be fine-grained enough for such tasks, therefore dynamics on the object- and on the class-level are necessary.

As stated common and new functionalities in object-orientation work mainly on the class-level. The considerations presented above justify the idea that in many cases class-level constructs may be supplemented by a similar construct especially tailored to the object-level. Nevertheless, the enhancement of composability on class-level is also a valuable goal. Our approach relies on the idea that objects and classes must entail abilities of similar expression power. Our notion is that in many conventional object-oriented programming languages the object-level is not covered enough. On the other hand, the expressive power of the compositional programming abilities on class-level is too weak, when the structures are becoming very complex (see [14] for an approach to solve this problem).

Kiczales [7] points out a general underlying problem: the traditional view on abstraction is insufficient. The primary place for an abstraction boundary is between the aspects of a system particular to an implementation and the aspects common across all implementations. Often this useful idea goes together with the sense that the implementation is to be completely hidden behind *only* one interface from the client. Unfortunately, several clients need different knowledge about the implementation, e.g. for performance reasons. Kiczales proposes open implementations as a solution to this problem. The goal for the per-object mixins, presented in the next section, is to offer a more higher-level solution to this problem. We will see that they are able to easily adapt interfaces in an object-specific way, are transparent for clients, are dynamically attached/detached, have access to their object's internals through reflective techniques, *but* do not let the client break the abstraction boundary.

3. Per-Object Mixins

3.1. Method Chaining

A special feature of XOTCL derived from OTCL is the method chaining without explicit naming of the "mixin" method. It mixes the same-named (or "shadowed") superclass methods into the current method (modeled after CLOS [1]). A method can invoke explicitly the shadowed methods by the `next`-primitive, resulting in an unambiguous, linear next-path (see Figure 2). When no per-object mixins (or filters) are involved, the next-path is identical to precedence order of classes.

In the following example we define four classes and provide a constructor for each of them. The primitive `next` is used to call all constructors along the next-path. A call of `next` without arguments passes all arguments to the shadowed method:

```
Class Room
Class OvalRoom -superclass Room
Class Office
Class OvalOffice -superclass {OvalRoom Office}
Room instproc init args {
  [self] set roomNumber 0; next
}
OvalRoom instproc init args {
  [self] set diameter 0; next
```

[1] For these reasons we introduced assertions on the class- *and* object-level to XOTCL (not described in this paper – see [13]).

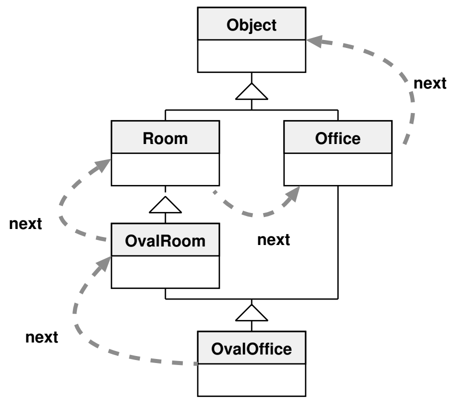

Figure 2. Class Hierarchy and Next-Path

```
}
Office instproc init args {
  [self] set deskType metal; next
}
OvalOffice instproc init args {
  [self] set ownerName -; next
}
```

When an object of class `OvalOffice` is created the four instance variables `ownerName`, `diameter`, `roomNumber`, and `deskType` are defined with default values. At runtime the definition of the instance variables occurs in the mentioned order. This ordering mechanism is used for all instprocs.

3.2. Usage of Per-Object Mixins

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 to register multiple mixins. The following defines the classes `A` and `Mix1` (with some methods) and registers `Mix1` on the instance `a` of class `A`.

```
Class A
A instproc proc1 {} {
  puts [self class]
  next
}
A instproc proc2 {} {
  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
```

The call "`a proc2`" results in the output "`::A`", since `proc2` is not a method of the per-object mixin. We extend the example by another per-object mixin and construct a chain of mixins (see Figure 3), e.g. :

```
Class Mix2
Mix2 instproc proc1 {} {
  next
  puts [self class]
}
a mixin {Mix1 Mix2}
```

Since the `puts` command in `proc1` is placed after the call to `next`, the output of this method is generated after the methods of class `A` are finished. The call "`a proc1`" produces the output:

```
::Mix1
::A
::Mix2
```

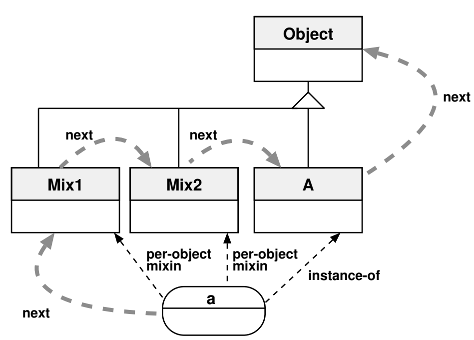

Figure 3. Next-Path with Per-Object Mixins

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.

3.3. Multiple Mixin Classes

Per-object mixins can cover a common problem which is not solvable elegantly using just class hierarchies and multiple inheritance: the problem of supplemental classes. *Supplemental classes* introduce additional orthogonal functionality into a class hierarchy of an application, which already performs a common or basic task. In order to introduce the additional functionality, multiple inheritance can be used. We will show that an approach based on per-object mixins solves this general problem more elegantly.

In order to introduce supplemental classes based on multiple inheritance, it is necessary to define a new class, like for example:

```
Class Basic+Add1 -superclass {Add1 Basic}
```

In languages like XOTCL the class hierarchy can be changed in a dynamical manner, every object of the class `Basic` may be changed to class `Basic+Add1` at arbitrary times, e.g.:

```
Basic basicObj
...
basicObj class Basic+Add1
```

Now we consider a situation with two supplemental classes. The following set of classes has to be defined to cover all possible combinations:

```
Class Basic
Class Add1
Class Add2
Class Basic+Add1 -superclass {Add1 Basic}
Class Basic+Add2 -superclass {Add2 Basic}
Class Basic+Add1+Add2 \
  -superclass {Add2 Add1 Basic}
```

In order to define supplemental classes based on multiple inheritance the number of helper-classes rises exponential. For n supplemental classes, $2^n - 1$ (or their permutations if order matters) artificially constructed helper-classes are needed to provide all combinations of additional mixin functionality. In general, when supplemental classes cause side effects, and they are added repetitiously, the number of constructible helper-classes is unlimited.

This demonstrates clearly that the sub-class mechanism based on multiple inheritance provides only a poor way to mix in orthogonal functionality. Therefore, we suggest per-object mixins to implement the supplemental classes, e.g. :

```
Class Basic
Basic instproc someProc {} {
  # do the basic computations
}
Class Add1
Add1 instproc someProc {} {
  # do the supplemental computations
  next
}
Basic bObject -mixin Add1
```

Below is a small application example to demonstrate how to use mixins for multiple supplemental classes:

```
Class Agent
Agent instproc move {place} {
  # do the movement
}
Class InteractiveAgent -superclass Agent

# Supplemental-Classes
Class MovementLog
MovementLog instproc move {place} {
  # movement logging
  next
}
Class MovementTest
MovementTest instproc move {place} {
  # movement testing
  next
}
```

An `Agent` class is defined, which allows agents to move around. Some of the agents may need logging of the movements, some need a testing of the movements, and some both (perhaps only for a while). These functionalities are achieved through the supplemental classes, which we will apply through per-object mixins.

We now create two interactive agents; one is logged and one is tested:

```
InteractiveAgent i1
InteractiveAgent i2
i1 mixin MovementLog
i2 mixin MovementTest
```

At arbitrary times the mixins can be changed dynamically. For example `i2`'s movements can also be logged:

```
i2 mixin {MovementTest MovementLog}
```

Figure 4 shows the situation of the object `i2` and its next-path.

Mixins are transparent for client objects. Object specific state changes can be modeled through the modification of the class relationship. When an object switches its class, this does not affect the supplemental classes registered as mixins of the object. But if the supplemental classes were accessed trough multiple inheritance, the re-classing of objects of the same basic type would have to cover all different derived types. In particular there would be one specialized class for each supplemental class combination (as

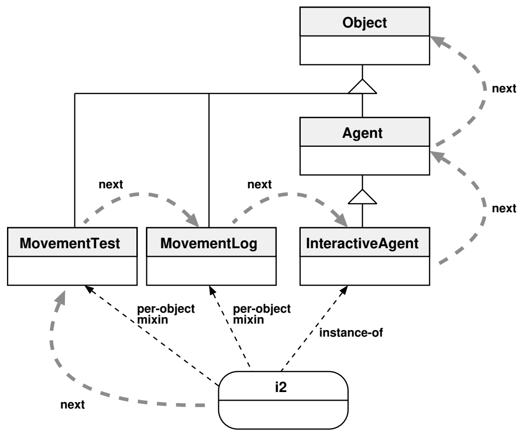

Figure 4. Next-Path for the Agent Example

shown above). In order to known which class of these is the correct one for a certain object, the object itself has to "remember" its classes' super-classes (and keep a history of super-classes, if they also may have changed).

Therefore, when using multiple inheritance together with dynamic class switching, the objects are forced to store local information of their classes. Such non-localities break encapsulation again. Therefore, dynamic object-oriented environments need a facility to attach classes to objects apart from the inheritance hierarchy. From that point of view per-object mixins are necessary to reach the primary object-oriented goal of encapsulation in dynamic environments.

3.4. Call Graph Dependencies for the Mixins

A general problem in the context of mixins are the calling dependencies: what kind of other methods can be called from a mixed-in method. In order to allow mixin methods of a per-object mixin class to access other (sister) instance methods of the same mixin class (this is necessary for more complex applications) the system has two alternatives:

A naive approach is, to dispatch the sister method directly. This approach has the consequence that the message forwarding mechanism of XOTcl is bypassed and same-named methods defined as object-specific `proc`'s become unaccessible. Another consequence is that mixins would not be applied on calls from other mixins. This strategy would be an unorthogonal exception to the language and was therefore discarded.

For these reasons we decided to implement a more general solution, which dispatches every invocation from the mixin as an regular invocation on the object. This approach has another advantage. Several application, of recursive nature, e.g. recursive computations or composite structures, like the composite or chain of responsibility pattern [5], induce the usage of a recursive approach. Through the dispatch along the object the mixin method gets the ability to map recursions directly. The mixin method can call itself recursively until the recursive task is ended. Afterwards it forwards the result using `next`. This avoids the need for delegation to other objects in order to use recursions.

3.5. Per-Object Mixin Inheritance

The usual way to specialize descriptive structures in object-oriented languages is inheritance. Since per-object mixins are themselves normal classes it is desirable that they can benefit from specialization through inheritance. But this already implies another reason for mixin inheritance: since per-object mixins are classes, instances can be derived directly. It is also desirable, that these instances behave similar to objects having the class as a per-object mixin.

Consider the following example. An application, like e.g. a web-browser, needs in several situation a general means to receive a data-stream incrementally and in some of these situations, it has to measure the time periods of receiving data. The later task could be modeled easily by a per-object mixin, the former would most likely be modeled by an instantiated class. But both classes need the same abstract interface and the initialization tasks – which are for both classes the same – need to be performed only once per data-stream.

A solution would be a general class `Sink`, which is subclassed to two classes. `MemorySink` stores/handles the data-stream, while `TimeSink` handles the time measurements (and inherits from `Time`):

```
Class Sink
Class MemorySink -superclass Sink
Class Time
Class TimeSink -superclass {Time Sink}
MemorySink s1 -mixin TimeSink
```

In the usual case `MemorySink` is instantiated and the instance gets the per-object mixin `TimeSink` to obtain a timed memory sink when needed. But both classes may be instantiated or used as a mixin.

A difficulty arises when `TimeSink` is used as a mixin class. Both classes (`MemorySink` and `TimeSink`) inherit from the `Sink` class, but with different precedence orders. It is necessary tp mark such common classes in the same hierarchy must be marked, in order to avoid double execution of the inherited methods (which can be a problem e.g. for a destructor). For that task we introduced the `mixinsinherit` option, which has the syntax:

ClassName `mixinsinherit` ?(0|1)?

If this option is turned off, mixin-classes will not inherit from this class. By default it is turned on (except for the general classes `Object` and `Class`. Without argument it returns the current state of the option on the class `ClassName`. Figure 5 shows the emerging situation for an instance of `MemorySink`, with a `TimeSink`-mixin and the `mixinsinherit` option turned off on `Sink`. The option has the effect, that the precedence order for `TimeSink`, which would normally be `Time` followed by `Sink` is not used, since mixins do not inherit from `Sink` or `Object`. Instead a dispatch along `MemorySink` is choosen. Without the option set on `Sink`, `Sink` would be searched prior to `MemorySink` (for some applications this might be the desired behavior).

```
Sink mixinsinherit 0
MemorySink sink1 -mixin TimeSink
```

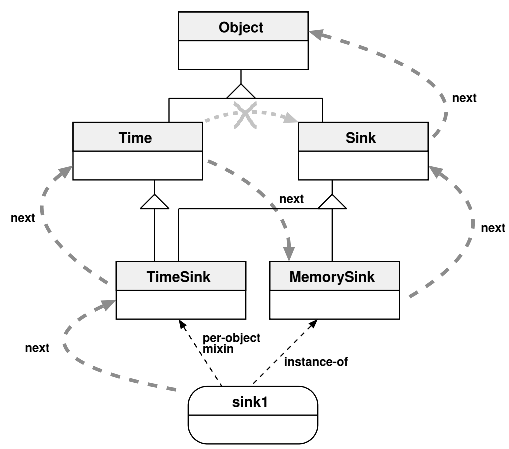

Figure 5. Mixin Inheritance Example

3.6. Hiding Object Specifics through Per-Object Mixins

As shown in the previous examples, per-object mixins act transparently for their client objects. In traditional object-oriented approaches two orthogonal tasks of one conceptual entity, like a computation and an (perhaps graphical) output, would be placed in two different objects. Such an usage of objects for the purpose of system modularization is one of the fundamental strengths of object-orientation. Meyer [12] calls this approach *object-based decomposition* and argues that many important aspects of software quality are achievable by decomposition, such as extendibility, reusability and compatibility.

One disadvantage of object-based decomposition is that it splits one conceptual entity (from the client object point of view) 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 self-problem identified by Lieberman [10]. The implementation of the supplier as several objects requires forwarding of messages, e.g. the computation object receives a request for a computation and forwards it to the output object. Once the message is forwarded, the reference to the receiving computation object is lost for the output object (and other objects of the conceptual entity). References to *self* refer to the delegated object, rather than to the original receiver.

Per-object mixins are able to decompose several tasks of one conceptual entity. They are themselves ordinary classes. Therefore, they can be used for object-based decomposition and are able to achieve its benefits. Since these classes are mixed into the current object's precedence order they do not suffer from the self-problem, because every self-reference refers to the same object. This object in conjunction with its mixins and classes forms a conceptual entity for the client.

Another way object-orientation offers to form such a conceptual entity is inheritance. We have already argued that class-level constructs are too general to solve every problem on the object-level. Furthermore, in the case of inheritance we have already shown in Section 3.3 a common example how the need for different supplemental classes added to suppliers lets the number of classes rise exponentially. Another important argument for the mixin approach is that mixins hide the data representations and algorithms they entail from the actual object. This leads to transparency of mixin actions. Many application require or benefit from transparency of certain object specific actions that do form a conceptual entity (from their point of view).

Now we present a general scheme how to apply the mixin approach sketched above in general. The following steps have to be done:

1. Find the main task of the application and define the (possibly abstract) application class.
2. Eventually specialize the application class.
3. Find all possible supplemental classes to the application class and implement them as classes. These are the mixin classes.
4. Add mixin methods to the mixin classes.
5. Register the supplemental classes as needed to the instances of the application classes as mixins.
6. Let clients use the application objects, without knowledge of the mixins.

4. An Application Case: Simple Persistence Store

In order to show how to apply the scheme on a real world example we present the base-line architecture of a small persistence store. Firstly we need a form of storage. The persistent store should be independent of the chosen storage form. So, we provide an abstract interface for storage access, specified using the `abstract` instance method:

```
Class Storage
Storage abstract instproc open  name
Storage abstract instproc store {key value}
Storage abstract instproc list  {}
Storage abstract instproc fetch key
Storage abstract instproc close {}
```

Afterwards, we derive several specialized storages from it, e.g. a GNU Dbm database access, a memory storage, etc.:

```
Class GdbmStorage -superclass Storage
Class MemStorage -superclass Storage
```

A persistence manager is responsible for handling several persistent objects and for opening/closing the storage access. It gets the storage as a mixin, in order to be able to change it dynamically at arbitrary times and to provide different storage forms to different persistence manager objects in the same system. Furthermore, the mixin is completely transparent solution for storage access.

```
Class PersistenceMgr -parameter {pName [self]}
PersistenceMgr instproc init {} {
  [self] mixin GdbmStorage
  next
  # open the storage access
  [self] open $[[self] set pName].db
}

PersistenceMgr instproc destroy args {
  # close the storage access
  [self] close
  next
}
```

Finally, we need a general class, for handling the persistence, and several sub-classes, for implementation of persistence strategies. Here, we present a lazy strategy only updating the storage at destruction time and an eager strategy for updating on every variable-writing. Since application objects should not be aware of the persistence, this a suitable task for per-object mixins. But since normal inheritance and mixin inheritance cooperate, as shown in Section 3.5, we can also derive direct instances from the `Persistent`-classes.

```
Class Persistent -parameters {persistenceMgr}

Persistent instproc persistent {list} {
  [self] instvar persistenceMgr
  foreach var $list {
    lappend [self]::__persistentVars $var
    $persistenceMgr fetch [self]::$var
  }
}
```

Above we see the most important tasks of the `Persistent`-class. It takes a parameter for associating the object with a persistent manager and allows the programmer to specify the persistent variables through the `persistent` instance method. Automatically, it re-fetches these variables at next instantiation time.

Now we derive an eager strategy that defines a variable trace for the persistence variables and a handle method `vtrace`. This method is registered through the TCL-command `trace variable` as a callback, which is invoked every time the specified variables are written.

```
Class PersistentEager -superclass Persistent
PersistentEager instproc vtrace {name sub op} {
  # store variable 'name' in storage
  # ...
}
PersistentEager instproc persistent {list} {
  next
  foreach v $list {
    trace variable [self]::$v w \
      [list [self] vtrace]
  }
}
```

Finally, a lazy persistency strategy is achieved by overloading the destructor of the persistent object:

```
Class PersistentLazy -superclass Persistent
PersistentLazy instproc storeall {} {
  # store all persistent variables in storage
  # ...
}
PersistentLazy instproc destroy args {
  [self] storeall
  next
}
```

Now an application class can chose the appropriate strategy, the default persistent variables and the persistence manager used for storage access in its constructor, e.g. :

```
Class AppClass
AppClass instproc init args {
  [self] set var1 1
  [self] set var2 2
  [self] mixin PersistentEager
  [self] persistenceMgr p
  [self] persistent {var1 var2}
  next
}
```

Still every instance can dynamically change the persistence, e.g. register more object-specific persistent variables, change the strategy, etc. The mixin solution acts transparent for client objects and is definable per-object. In comparison to a similar solution combining super-classes through method chaining, which is also transparent for the instance, the class-hierarchy of the application class has not to be affected and instances can easily be customized.

5. Related Work

We firstly will sketch related works regarding the two presented language constructs and afterwards we will describe related ideas to the notion of object-based system composition.

5.1. Related Work on Per-Object Mixins

We have already discussed the usage of mixins for method chaining in OTCL in Section 3.1. As seen afterwards the mechanism of per-object mixins functions the same using the `next` primitive and its next-path. The OTCL mixins are discussed more deeply in [22].

The mixins of OTCL provide an automatic method chaining without explicit naming of the mixin method. They are a very flexible programming mechanism and, combined with the unambiguous precedence OTCL offers, they avoid name clashes through (multiple) inheritance at all. Both the precedence order and the idea of mixins in OTCL are influenced by the lisp extension CLOS [1].

The filter approach [14] is the class-level construct which has led towards the idea of per-object mixins. It is defined as a registered instance method of a class or meta-class. It also uses the next-path for chaining and is also inherited. But it is applied on all calls of all instances of the registration class and all its sub-classes, instead of a limitation to specific calls of one instance, as per-object mixins are. Related work regarding filters is discussed in [14, 23]. A deeper comparison to the filter can be found in [15].

There are several extensions to the idea of mixins discussed. In Agora [18] mixins are treated as named attributes of classes. This has the consequence that a class can control how it is extended. Therefore it is possible to constrain class hierarchies and to make extensions specific to a class. As a consequence such kind of mixins may be nested.

Bracha and Cook [2] compare different inheritance mechanisms and propose mixins as a general inheritance construct. Inheritance is interpreted as mixin composition. In Jigsaw [3] Bracha and Lindstrom use mixins to unbundle the several roles of classes by providing a set of operators controlling effects like inheritance, name-resolution, modification, etc.

All mentioned approaches are working on the class-level and have more similarities to filters than to per-object mixins. But they imply the mixin characteristic that 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.

5.2. Related Work on Object-Based Composition

The idea of enhancing the role of objects is not new. The limitations of inheritance (and polymorphism) and how to apply it well are discussed by many authors from several points of view. We have summarized some of those ideas in Section 2 (in particular: [2, 6, 10, 18, 20]). A powerful object system allows the application programmer more flexibility and expression power. In OTCL [22] there are several powerful object-level constructs, like dynamics in the class relationship and per-object specialization. CLOS [1] enhances this flexibility to a higher level. It is one of the most flexible environments for object-oriented engineering. It is the basis for a meta-object-protocol [8], which provides many hooks to influence the behavior and semantics of objects. Our constructs filter and per-object mixin differ significantly, since they are more higher level constructs.

In [7] the discrepancy between the software engineering goal of abstractions hiding their implementation and the reality of the practice is examined. Resolving this discrepancy, e.g. through open architectures or reflective techniques, like introspection, provides a basis for the usability of higher level programming language constructs, like the presented per-object mixins. The general problem to address requirements of several clients, while staying focused enough for each specific client, may be solved by open implementations [9], which let clients access a module's implementation. We consider the per-object mixin as an elegant solution for this problem, since it allows per-client specialization of a module in a transparent way and accomplishes that different client may have different interfaces, without breaking the abstraction boundary between a supplying module and its clients.

All these ideas run into one general direction, which is identified in [4] as a "open-systems-trend", applying for operation systems, databases, communication systems and programming languages. Such approaches to open systems provide exchanges of objects in an open way. Type restrictions often become crucial to be enforced if the exchanging partners are distributed. Therefore, the intention to implement distributed open systems requires to delimit class-level constructs to the object-level and a more unique representation of objects. To reach that goal our solution propagates the string as only interface type, a powerful object system and special object-level constructs. This way it eases exchanges of objects. Type safety may be reintroduced by assertions which we also have developed as an object-level construct (resembling the class-level assertions in [12]) and implemented in XOTCL (see [13] for details).

6. Conclusion

This paper has described the general idea of enhancing object composability abilities of programming languages and presented many arguments for this idea. A distinction of two language levels, object- and class-level, was used to distinguish this approach from conventional approaches. We have pointed out that often it is possible to find resembling constructs on both levels, tailored for the necessities of their level. Furthermore, we have shown that these ideas go together with a powerful object- and class-system as a basis, e.g. offering dynamics and introspection.

Afterwards we have verified these ideas on the example of a new language constructs: per-object mixins. We have illustrated them with examples from the language XOTCL, which offers the mentioned desirable properties in its object- and class-system. Nevertheless, the general underlying constructs could be implemented in several other languages as well. Similarly the general notion of enhancing object composability is well suited for most class-based approaches.

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

References

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

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP'90*, volume 25 of *SIGPLAN Notices*, pages 303–311, October 1990.

[3] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. of IEEE International Conference on Computer Languages*, April 1992.

[4] S. Demeyer, P. Steyaert, and K. D. Hondt. Techniques for building open hypermedia systems. In *Proc. of ECHT'94 Workshop*, September 1994.

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

[6] L. Hatton. Does oo sync with how we think? *IEEE Software*, May/June 1998.

[7] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proc. of IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

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

[9] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proc. of ICSE'97*, Boston, May 1997.

[10] H. Lieberman. Using prototypical objects to Implement shared behavior in object oriented systems. In *Proc. of OOPSLA '86*, Portland, November 1986.

[11] C. Lucas and P. Steyaert. Research topics in composability. In *Proc. of the CIOO Workshop at ECOOP*, Linz, July 1996.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[13] G. Neumann and U. Zdun. XOTCL, an object-oriented scripting language. Submitted for publication, 1998.

[14] G. Neumann and 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.

[15] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. Submitted for publication, 1999.

[16] J. K. Ousterhout. TCL: An embeddable command language. In *Proc. of the 1990 Winter USENIX Conference*, January 1990.

[17] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.

[18] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in Agora. In *Proc. of ECOOP '93*, LNCS 707. Springer-Verlag, 1993.

[19] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proc. of OOPSLA '87*, Orlando, December 1987.

[20] W. Weck and C. Szyperski. Do we need inheritance? In *Proc. of the CIOO Workshop at ECOOP*, Linz, December 1996.

[21] P. Wegner. Learning the language. *Byte*, 14:245–253, March 1989.

[22] D. Wetherall and C. J. Lindblad. Extending TCL for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.

[23] 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.