Design and Implementation Constructs for the Development of Flexible, Component-Oriented Software Architectures*

Michael Goedicke[+] Gustaf Neumann[*] Uwe Zdun[+]

[+] *Specification of Software Systems*
University of Essen, Germany
{goedicke|uzdun}@informatik.uni-essen.de

[*] *Department of Information Systems*
Vienna University of Economics, Austria
gustaf.neumann@wu-wien.ac.at

Abstract. Component-orientation is an emerging paradigm that promises components that are usable as prefabricated black-boxes. But components have the problem that they should be changeable and flexibly adaptable to a huge number of different application contexts and to changing requirements. We will argue, that sole parameterization – as the key variation technique of components – is not suitable to cope with all required change scenarios. A proper integration with multiple other paradigms, such as object-orientation, the usage of a scripting language as a flexible component glue, and the exploitation of high-level interception techniques can make components be easier (ex)-changeable and adaptable. These techniques can be applied without interfering with the component's internals.

1 Introduction

The task of a software engineering project is to map a model of the real world (existing or invented) onto a computational system. The complexity and diversity of concrete real world systems can be overwhelming. This is no complexity in the algorithmic sense, but an complexity of an overwhelming amount of details and of particularities in the universe of discourse. By developing a model we reduce this complexity by finding and extracting commonalities. The key instruments of modeling are abstraction and partitioning. Analyses of commonalities let us understand the common elements of a targeted system. The aim of any analysis of commonalities is to group related members of a family, regardless whether the members are components, objects, modules, functions, etc.

Orthogonal to the task of modeling the commonalities (where details are removed) is the task of engineering variability. It makes absolutely no sense to create abstractions to understand a family as a whole, if we do not introduce proper means for variation in the family members [3]. Finding commonalities in software eases

*Accepted for publication in: Proceedings of Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000), Erfurt, Germany, Oct 9-12, 2000.

understanding and reduces the need for changes, while finding proper variabilities enables us to use the software at all, because we have to re-adapt the found abstractions to the concreteness of the modeled real world situation. Commonality and variability are competing concerns and its hard to find a proper balance between them by approaches that (a) model the real world from the scratch and then (b) try to reuse the common aspects in such upfront design models. The forces in the steps (a) and (b) can normally not be well integrated. We rather propose in this work to model only the interfaces and keep them variable. Techniques for "programmable interfaces" let us flexibly glue the application parts together.

There are recurring ways for finding good abstractions and partitionings. "Good" means that they provide a tenable amount of commonalities to let us understand the problem and produce long-lasting software, but still enable us to easily introduce (expected and unexpected) changes. Such patterns of organizing abstraction around commonalities and variations are popularly called "paradigms". In software engineering a paradigm is a set of rules for abstraction, partitioning, and modeling of a system. E.g., the object-oriented paradigm structures the design/program around the data, but focuses on behavior [23]. It allows us to introduce variations in data structures/connections and algorithm details. Each paradigm has a key commonality and variation.

If we implement a system, we have to deal with a broad variety of paradigms. Coplien [3] discusses the need for multi-paradigms. In fact nearly any good real world software system is designed and implemented using multiple paradigms, simply because nearly no complex real situation exists, that can be described with one paradigm sufficiently. E.g., in nearly every large C++ program a mixture of object-oriented, procedural, template, and various outboard paradigms exists. Here, outboard paradigm [3] means a paradigm that is not supported by the programming language itself, but by a used technology, like the relational paradigm adopted from a relational database.

In the focus of this paper are language constructs and concepts for design and implementation that overcome current problems of the component- and object-oriented paradigms and their integration. Firstly, we will discuss these paradigms and their current integration problems. Afterwards we present some language concepts of the language XOTcl: Firstly we will discuss concepts which can be mapped manually to current mainstream languages, then we will present some interception techniques that are missing in current mainstream languages. Finally we will generalize our approach and compare to related work.

2 Combination of Component- and Object-Oriented Paradigm

2.1 Component-Oriented Paradigm

The very idea of component-based development is to increase productivity of building software systems, by assembling prefabricated, widely-used components. Components are self-contained, parameterizable building blocks with explicit interfaces. Component-based development aims at the replaceability of components and the transferability of components to a different context, thus enabling component reuse.

The idea of the component-oriented key abstraction is not new. E.g., in many large C systems self-contained components (or modules) that can be accessed via an explicit API can be found. Component-based development, as it is proposed today, mainly adds interface definition languages or other means to enforce that all component accesses conform to the component interface, platform and/or language independence, support for distribution, and accompanying services.

Current component approaches, such as component frameworks in scripting languages, like Tcl [19], or the component models of popular middleware approaches, such as CORBA, Java Beans, or DCOM, induce mainly a black-box component approach. Unfortunately often there are several factors for development organizations that drive them not to adopt the component-based approach for components developed by a third-party or even by a different in-house development team. Often used arguments are, that internals of a black-box component can not be changed, therefore, *reaction on business process changes* can become more difficult. Generally, *bugs in the component are harder to fix*, because it is hard to build a work-around for a bug, that is not located in your own code. The organization relies on the ability and will of the component developer to fix the bug. Moreover, using black-boxes often means that the *development team looses expertise on component's domain*.

These factors can be observed in many real world applications and they apply not only for black-boxes, but also (to a smaller degree) to components with available source code. The component abstraction seeks for building blocks that can be produced and maintained separately from the systems they are used in. Variations have to be treated separately and are mainly introduced by means of parameterization. The key problem of component-based software engineering is: On the one hand, components aim at extracting the commonalities to a level, where we can use them as prefabricated building blocks. On the other hand it is hard to maintain and to cope with changes in a piece of software without access to internals. Since parameters are the main variation technique of black-box components, the changeable parameters have to be foreseen by the component developer. But in reality often the requirement changes are not foreseeable at component development time.

2.2 Object-Orientation and Components

Object-orientation is a paradigm sharing properties with component-based approaches. Many component-based approaches are implemented using objects. Object-orientation arranges structures around the commonalities of data, but focuses on behavior. Traditional object-oriented programming and design maps entities of the modeled real world to a single programming language/design construct: the class. Object oriented design expresses computational artifacts through a mapping onto several classes and their relationships. In most object-oriented approaches, these relationships are association/aggregation (most often both are expressed through the same language construct), and inheritance or extension.

Object-orientation promises separation of the involved orthogonal concerns through encapsulation and class inheritance. The abstraction into general parts with inheritance or delegation in object-based system should help us to concentrate on common and special properties at different times. These abstractions also promise to gain modularity and to anticipate changes by designing general modules, that can be specialized in different ways and that support incremental extensions.

These promises were only partially achieved by traditional object-oriented approaches. Studies of the amount of reuse gained through object-orientation indicate that reuse is much smaller than expected in its early promises. E.g., Ousterhout [19] points out on basis of empirical evidence that the reuse of components – as they are used in scripting languages – is by far higher, than the reuse gained solely by object-orientation. We believe a main reason for this divergence is that large object-oriented frameworks tend to require an intimate knowledge of the framework's internals.

The component-oriented key abstraction of consequently exploiting parameterizable black-boxes is more suitable for reuse. But when taking a closer look at the success factors of scripting languages, one can observe that they combine the components with a highly-flexible glue language. Object-orientation especially helps us to understand structural complexity and to make it explicit by architectural means. These architectural means can be extremely valuable in complex design situation in the glue language (see [18]), because a relatively small glueing application can become very expressible and complex through the number of involved components. And object-oriented language constructs can also be valuable in the internal design of the self-contained components. Our approach relies on the two-level concept of scripting languages, like Tcl. We distinguish into reusable, self-contained components, mostly written in a system language, and a high-level, object-oriented scripting language that combines these components flexibly.

Complex design problems are a focus of object-oriented approaches, but a weak point of component combination with a scripting language, like Tcl. Object-oriented design patterns capture the practically successful solutions of the field of object-orientation. Our research on language support for design pattern, as for instance in [14], has shown that pattern variations cannot be described solely through parameterization. Reusable pattern implementation variants have to be fitted to the current context, especially when patterns are used in the hot spots [20] of software systems. These parts of the application – where an elegant and sufficient solution requires variabilities beyond pure parameterizations – are the parts that are hard to cover with the component-oriented abstraction, since its key variability is parameterization. The combination of the two paradigms with high-level design/implementation language functionalities lets the object-oriented constructs cover the weak points of the black-box component approach and vice versa.

3 Components and Component Configuration in XOTcl

Extended Object Tcl (XOTcl) [18] (pronounced *exotickle*) is an object-oriented extension of the language Tcl. Scripting languages gain flexibility through language support for dynamic extensibility, read/write introspection, and automatic type conversion. The inherent property of scripting languages such as Tcl is that they are designed as two-level languages, consisting of components written in efficient and statically typed languages, like C or C++, and of scripts for component glueing.

Our assumption is that just "glueing" is not enough. XOTcl enhances Tcl with language constructs giving architectural support, better implementation variants, and language support for design patterns, and explicit support for composition/decomposition. All object-oriented constructs are fully introspectable and all relationships are dynamically changeable. XOTcl offers a set of basic constructs, which are singular objects, classes, meta-classes, nested classes, and language support for dynamic aggregation structures. Furthermore, it offers two message interception techniques *per-object mixin* and *filter*, to support changes, adaptations, and decorations of message calls.

In XOTcl a component is seen as any assembly of several structures, like objects, classes, procedures, functions, etc., to a self-contained entity. Components are conveniently packed into packages that can be loaded dynamically. A component can also consist of a C or C++ extension of Tcl. Each component has to declare its name and optional version information with Tcl's `package provide` with the following syntax:

`package provide` *componentName ?version?*

The system automatically builds up a component database. With `package require` an XOTcl program can load a component dynamically with a name and optional version restrictions at arbitrary times. `package require` has nearly the same syntax:

`package require` *componentName ?version?*

Components expose an explicit interface that can be used by other programs without interfering with the components internals. But still we have to integrate the components with the application and make the component internals adaptable and dynamically fit-able to a changing application context.

3.1 Component Wrapping

Component wrappers can wrap black-box components written in various languages and structured with multiple paradigms. The component wrappers are object-oriented Wrapper Facades [21] that shield the components from direct access (see Figure 1). Note, that often a set of interacting component wrappers has to be used to wrap a complex component properly. Above the component wrapper layer a set of implementation objects define the hot spots of the design. All objects (including the component wrappers) can exploit the dynamic and introspective language functionalities of XOTcl. Since a black-box component is never accessed directly, but always with the indirection of the component wrapper, we gain a central place, that is a proxy or placeholder for a component. The component wrapper is a white-box for the development team of the application. Here, changes can be applied centrally and adaptations can be introduced without affecting the components' internals.

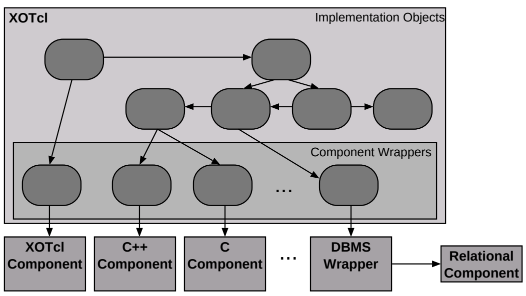

Figure 1. Integration of Components and Objects through Component Wrappers

Generally each component wrapper is implemented with an abstract interface and a concrete component wrapper implementation (see Figure 2). Clients use the components as Strategies [4] to make components easily exchangeable by providing

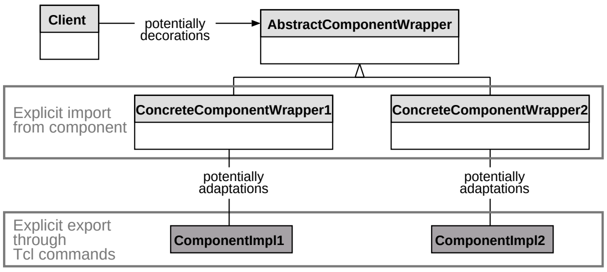

Figure 2. Class-Based Component Wrapper Interface

a new concrete component wrapper and by dynamically changing to a new Strategy. The concrete component wrappers forward the received messages as Wrapper Facades [21] to the components that implement the functionality. At the connection between client and component wrapper we can easily enhance the functionality of the component with Decorators [4]. At the connection between component wrapper and component we can use Adapters [4], e.g. to perform interface adaptations.

3.2 Export/Import Component Configuration

The implementation of the component's functionality (e.g. in C or C++) is integrated into XOTcl with Tcl commands (see Figure 3). A component explicitly defines its *export* by explicitly defining a set of Tcl commands (function names with argument lists). These commands can be mapped onto one (or more) wrapper objects, that configure the component usage and adapt the Tcl Commands to an object-oriented interface. The component wrapper explicitly declares which of the exported methods are the *import* of this component usage. This way the component's client defines the required interface that an implementing component has to conform to. The component implementation can be replaced by any other implementation that conforms to the required interface. Finally, the actual implementation objects, which are using the component, call the methods of the component wrapper.

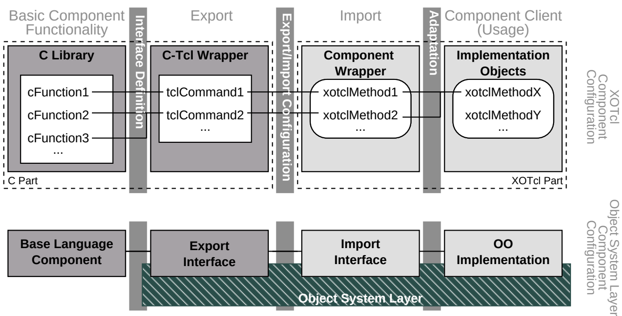

Figure 3. Three-Level Component Configuration with Explicit Export/Import

The implementation objects can be used to build an application or a new component. If a new component is built from existing components, it can export an interface through a component wrapper consisting of XOTcl methods. But, since any Tcl program can be embedded in a C program, a new component can also export C functions (which can be used by any C program).

The component concept relies at runtime on the concept of *component configuration* [5]. The first configuration step maps a C library component with an interface design into the scripting language. Then this functionality is imported and adapted by the component wrapper. Finally, the implementation objects use the adapted import in their application framework. Each configuration step allows us to actualize the configurations with different implementations that conform to the interfaces. The integration of C components is presented in the upper half of Figure 3.

The general technique of applying an *Object System Layer* to a base language and to implement the components with an object-oriented implementation is presented below. This technique is used in various languages and applications and is documented as the *Object System Layer* design pattern [12].

Component configuration – as used in this work – is the runtime technique of combining components. In XOTcl each component configuration can be changed dynamically at arbitrary times. The component import interfaces can be dynamically fitted to the new context. In order to keep track with this runtime flexibility an important functionality of the XOTcl language is introspection. It allows us to query the import interface for method names, argument list, and method implementations. The currently configured components can be queried to trace the components, their configuration, and the used interfaces at runtime. Runtime inspection tools can be written with a few lines of code.

4 Interception Techniques for Flexible Component Wrapping

The techniques discussed so far can (mainly) be implemented in any object-oriented design/programming language. The only difference of using XOTcl is that XOTcl language supports the discussed component concept already, i.e., it offers dynamic package loading mechanisms, language support for dynamic aggregation, dynamics and introspection in all language constructs, etc. In other languages we have to program the concept implementations by hand. But implementing flexible component wrappers solely with class constructs has several disadvantages:

- *Transparency:* The client should use the abstract interface without knowledge of concrete implementation details. The component wrapper should not appear to be scattered over several implementation objects.
- *Concerns that cross-cut the component wrapper hierarchy*: Often a complex class hierarchy is necessary to implement component wrappers sufficiently. E.g., most widget sets offer widgets as C or C++ components. In order to compose compound widgets out of simpler widgets, we may need object hierarchies as in the Composite pattern [4]. Concerns that are of a broad structural size and that cross-cut the hierarchy, such as painting of the whole compound widget, conventionally have to be programmed by hand.
- *Object-specific component wrapper extensions or adaptations*: Often adaptations have not to be performed for all objects of a certain component wrapper

type, but only for one object. We should be able to object-specifically enhance components, without sacrificing the transparency. The intrinsic component wrapper implementation and the implementation of extension/adaptation parts should remain decomposed.

- *Coupling of Component and Wrapper*: Component and component wrapper should appear as one runtime entity to clients, but they should be decomposed in the implementation.
- *Dynamics in Component Loading*: Components should be dynamically loadable, replaceable, and removable.
- *Runtime Traceability*: Components are loaded (possibly dynamically) into the system. To know which components are already loaded, the connections between wrapper and component should be traceable at runtime.

In this section we will briefly explain two interception techniques of XOTcl, that overcome these problems by flexible adaptation of the component wrapper calls to the concrete implementation. Both are transparent for the client. The per-object mixin implements concerns that are object-specific extensions, while the filter implements concerns that cross-cut class hierarchies. Filters and per-object mixins form runtime traceable entities with the intercepted objects at runtime, but are decomposed in the implementation.

4.1 Per-Object Mixins for Object-Specific Component Wrapper Extensions

A per-object mixin [13] is a language construct, that enhances a single object with a class that is object-specifically mixed into the precedence order of an object in front of the precedence order implied by the class hierarchy. Through object-specific, transparent, and dynamic interception of the messages that should reach the object, object-specific pattern variants [15] and object-specific roles [13] can be implemented conveniently. Per-object mixins allow us to handle orthogonal aspects not only through multiple inheritance, but since they are themselves classes and use class inheritance, they co-exist with the object's heritage order.

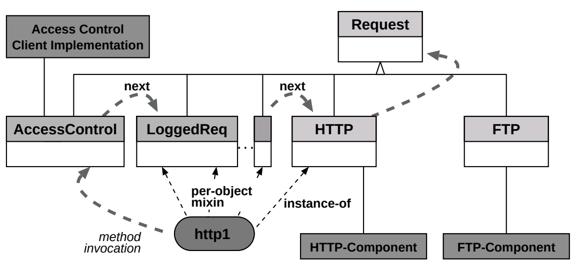

Figure 4. Request Logging/Access Control with Per-Object Mixins

In Figure 4 we can see an example of a per-object mixin. An abstract class `Request` has two subclasses, one handling `HTTP` requests and one for `FTP` requests. The class definitions may look like:

```
Class Request                            ;# Abstract class definition
Request abstract instproc open {}        ;# Abstract method
...
Class HTTP -superclass Request           ;# HTTP class definition
HTTP instproc open {} {                  ;# Method definition
  ...                                    ;# Method forwards to HTTP component
}
Class FTP -superclass Request            ;# FTP class definition
...
```

`HTTP` and `FTP` objects are Wrapper Facades [21] to components that implement the actual requests as black-boxes. Orthogonal to the tasks of a requests are the tasks of request logging, which can operate on both mentioned request types. In many cases only certain specified request objects should be logged, as in the example `http1`.

We do not want to interfere with the internals of the components that implement the requests in order to gain request logging. Therefore, a solution with single or multiple inheritance would not suffice, because it would either make all requests logged or create unnecessary intersection classes [13], like `LoggedHttpRequest` and `LoggedFtpRequest`. A solution with a reference from a logging object, as in the Decorator pattern [4], would require the client to maintain a reference to the perhaps volatile logging object and, therefore, it would be not transparent to the client. A solution with a reference to a logging object, as in the Strategy pattern [4], would not be transparent to the request object and unnecessarily interfere with the internals of the component wrapper. Both solutions suffer from the fact that – from the viewpoint of the client – one conceptual entity is split up into two runtime entities.

The solution with the per-object mixin, as in Figure 4, does not suffer from any of these problems. It attaches the role of being a logged request and an access control mechanism as a second orthogonal aspect object-specifically to the request object, either in Decorator or Strategy style (as required). The access control mechanism is actually performed in an imported component, while the rather simple task of logging is handled by the mixin class. The per-object mixin is transparent to client and request object. The logged request appears as one conceptual entity to the client. There is only one object `http1` that can be accessed and it always has the same intrinsic class `HTTP`. But still logging and request tasks are decomposed into different classes and can be dynamically connected/disconnected. Per-object mixins can be attached in chains and specialized through inheritance. The per-object mixin solution may look like:

```
HTTP http1                               ;# Instantiation of http1 object

Class LoggedReq                          ;# Logged request class definition
LoggedReq instproc open {} {
  # logging implementation
  next
}
...
Class AccessControl
...
http1 mixin {AccessControl LoggedReq}    ;# Mixin registration
```

`LoggedReq` and `AccessControl` are ordinary classes of XOTcl. As an example method, we define a method `open` for `LoggedReq` that logs all `open` calls and forwards the message afterwards with the `next` language primitive to the next mixin or the actual method implementation. We dynamically register the mixin classes for `http1`.

4.2 Filters for Cross-Cutting of Class Hierarchies

A second interception technique, called filter [14], is able to operate on a class hierarchy, instead of a single object (as the per-object mixins). A filter is a special instance method registered for a class C. Every time an object of class C or one of its sub-classes receives a message, the filter is invoked automatically. A prominent concern for usage of filters is to implement larger artifacts of the modeled world, like object-oriented design patterns, as instantiable entities of the programming language. In [14] we show how to express such concerns through a meta-class with a filter. The filter operates on all classes derived from the meta-class. Filters can be defined once and can then be registered dynamically. One can introspect the filters that are registered for a class. All filter invocations are transparent for clients.

As an example for filters, we present the implementation of a Composite pattern [4] variant in a reusable component. The context of the composite pattern is to build up an object tree structure and to derive a set of classes from an abstract component type. E.g., a composite widget component `Canvas` can aggregate other widgets. This way we can build up compound widgets. All widgets confirm to the same abstract widget component interface. A recurring problem of such structures is that leaf object, like button widgets, are not allowed to aggregate other objects. The solution to the problem in a purely class-based environment is, that only aggregating composite objects contain an aggregation relationship. Leaf objects, like buttons in the widget component example, have no children. An intrinsic property of composite hierarchies is that certain operations on the root component, as in the widget example a painting of the compound widget, have to be propagated through the object tree. Composite objects forward these operations to all their children, while leaf objects only execute the operation, but do not forward.

There are several problems that can be identified with the class-based implementation of the pattern. Concerns that cross-cut the composite hierarchy, like the forwarding of messages or the life-time responsibility of the whole for its parts (in the widget example: if a top-level widget is destroyed, all the constituent components have to be destroyed), are not expressed properly, since their semantics are not handled automatically. There is a certain implementation overhead, e.g. due to unnecessary forwarding of messages. The pattern as a conceptual entity of our design is neither traceable in the program code nor at runtime, but it is scattered across several implementation constructs. This variant of pattern implementation can hardly be reused, when the implementation language lacks proper (dynamic) means to fit a general variant implementation to a new context.

The implementation with a filter, a meta-class, and dynamic object aggregation, as in Figure 5, does not suffer from these problems. The filter is stored in a dynamically loadable component that contains a meta-class with a filter. The filter implements the reusable pattern implementation variant. Classes, like the widget component, that are superclasses of composite types are of the `Composite` meta-class type. Automatically they get the filter registered. The filter acts on the whole composite hierarchy and implements the concerns that cross-cut the hierarchy. Here, the forwarding of composite messages on a set of registered operations, like `paint`, is such a concern. The filter in the meta-class is transparent to clients. The compound widget appears as every other ordinary widget. The dynamic object aggregation language construct automatically handles the aggregation issues, like assurance of the tree structure and the life time responsibility of wholes for their parts.

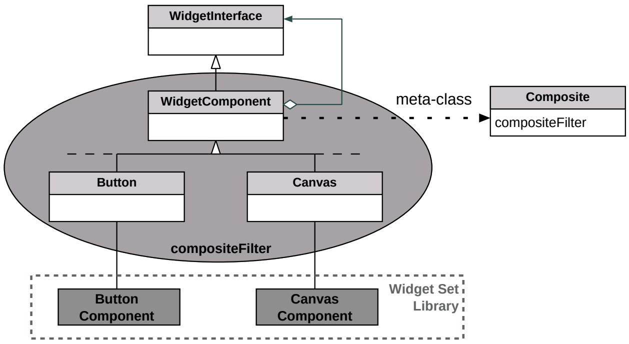

Figure 5. Composite Pattern with Filters/Dynamic Object Aggregations

The class definition of the `Composite` meta-class firstly has to define the meta-class. The composite filter is an ordinary instance method that determines the called method with an introspection option. It calls the message on all children with a loop and then on the current object. The component may look like:

```
package provide Composite 0.8
...
Class Composite -superclass Class          ;# Meta-class definition
Composite instproc compositeFilter args {  ;# Composite filter method
  ...
  set r [[self] info calledproc]           ;# Determine called operation
  foreach child [[self] info children] {   ;# Loop over all children objects
    eval $child $r $args                   ;# Forward message to children
  }
  return [next]                            ;# Call message on 'self'-object
}
```

We can load the Composite implementation from a component with `package require`. Afterwards we define the class hierarchy. `WidgetComponent` is defined by the meta-class `Composite` and automatically handles the forwarding of messages for all sub-classes transparently.

```
package require Composite
...
Composite WidgetComponent -superclass WidgetInterface
Class Button -superclass WidgetComponent
Class Canvas -superclass WidgetComponent
```

5 Components for Development of Flexible Software Architectures

The outcome of software development is a piece of software, a sustainable intellectual structure, which manifests the results of the design. Often it is handed over from the development team to the maintainers. The ability to modify or to understand the software especially for a person that was not involved in the development details, depends on the software architecture.

There are several properties that we expect from a "good" software architecture: It should be flexible, evolvable, understandable, predictable and maintainable. We expect the architectures to offer a significant amount of code reuse to speed up the development process and to achieve more reliable software systems. Certainly, the systems should be highly efficient. In summary we can make the following assessments on the architectural impact of the approach discussed in this paper:

- *Heterogeneous, Multi-Paradigm Black-Box Components*: Black-box components from various languages, like Tcl, XOTcl, C, or C++, are reused. The components are implemented with the most suitable paradigms. Unfortunately flexibility and evolvability of our software architectures suffer from our inability to change or react on problems of the components internals. In this paper we have discussed two approaches – scripting and high-level object-orientation – to overcome these problems.
- *Object-Oriented Scripting Language as a Component Glue*: Scripting languages combine components flexibly, by means of a highly flexible, introspective, and dynamic glueing language. But scripting languages are not very suitable to express the complexity of large application frameworks (see [18, 14]). Their original language design aims at smaller applications, partly because of runtime efficiency. However, time critical parts can always be put into components written in more efficient languages, like C. Still complex scripting applications, like several compound widgets in TK, were very slow in the early days of Tcl/TK. But nowadays CPU speed allows us to build very complex scripting applications without a reasonable speed penalty.

  In contrast, the combination with object-orientation gives us architectural support for composition/decomposition. The hot spots of the application, which are expectable changing parts, are kept in the high-level object-oriented language with its dynamic and introspective language means. Design experience of the object-oriented community with complexity of applications and with introducing flexibility in framework hot spots, helps us to make the component wrappers easily (ex-)changeable and evolvable.
- *Component Wrappers*: Object-oriented component wrappers integrate the components into the scripting language (if necessary). With the set of component wrappers a component's client explicitly defines its import from the component. In turn, the component explicitly defines its export through the Tcl wrapper. In a three-level process of configuration we actualize the interfaces with concrete implementations. These can be exchanged against other implementations transparently, what leverages evolvability and flexibility of our architectures. The implied indirection fosters understandability, since we can understand components and clients independently. The component wrappers are introspectable white-boxes to the application. Often changes in the component wrapper – without interference with the component's internals – are sufficient to cope with new requirements for a component.
- *Interception Techniques*: Object-oriented interception techniques, like filters and per-object mixins, enable us to deal with concerns that are hard to express with current object-oriented constructs. They are especially valuable on the component wrapper, since they allow us to transparently and dynamically introduce multiple views onto a component implementation (at runtime).

To back up the results in this work we have provided two case studies of systems build with the techniques described in this paper. In [16] we describe our web server implementation in XOTcl. We have compared efficiency with the pure C-based Apache web server. In the worst case our implementation was 25% slower than Apache, in some cases, our implementation was faster. In [17] we present a high-level framework for XML/RDF text parsing and interpretation. Again we have performed speed comparisons with implementations in Java and C. The Java based implementation was 2-4 times slower than our implementation in the scripting language (using off-the shelf C components), the pure C implementation was only 1.5-3.5 times faster than the scripting implementation.

6 Related Work

In [22] the (mainly black-box) component models of current standards, like CORBA, COM, or Java Beans are discussed. These approaches offer the benefits of black-box component reuse, but have problems, when the internals of a component have to be changed or adapted. Currently none offers an integrated concept for component reuse and adaptation, that solves all the problems raised in this paper, but all approaches have extension in this direction.

Java Beans implement limited dynamic loading and reflection abilities, but not all interesting data can be introspected (e.g., the important information on caller/callee of a call can not be retrieved automatically). Java Beans offer a distinct component model. But Java Beans offer only the Java Native Interface for integration of components written in other programming languages. Java does not support powerful language means for configuration/adaptation of components.

In [7] an interception system for COM objects that can intercept object instantiations and inter-object calls is presented. COM is a binary object model and the approach relies on direct manipulation of the function pointers that call the methods. In contrast to the approaches in this paper the approach is a very low-level approach and does not support suitable introspection mechanisms.

Orbix filters [8] (and similar techniques in other ORBs) implement limited interception abilities. They do only operate on distributed method calls and do not offer sophisticated introspection techniques. A more general form of such abstractions of the message passing mechanisms in distributed systems are composition filters [1]. Abstract communication types are used as first-class objects that represent abstractions over the interaction of objects. They encapsulate and enforce invariant behavior in object communications, can achieve the reduction of complexity of object interactions, and can achieve reuseability of object interaction forms.

The new CORBA 3.0 standard specifies a component model that is based in part on the Java EJB component concepts, but goes beyond that, by providing the component model to work with various languages. The new CORBA standard includes a scripting language specification, which is a framework to integrate scripting languages with a CORBA mapping. Interestingly, the goals for primary applications of the scripting language specification are the same issues, for which we have given architectural language support in this work, i.e., customizations of components, legacy wrapping, a service-based callback architecture, and flexible component glueing. It is likely that XOTcl and the techniques discussed in this work would ease it to reach these goals (though the specification is still in an early state).

A role, as in [11], is used to express the combination with extrinsic properties that can be dynamically taken/abandoned. The approach does not define a comprehensive read/write introspection mechanism. It does not provide an abstraction at a broader structural size, as the filter that applies a role on a whole class hierarchy.

Aspect-oriented programming [10] is a programming technique for decomposing concerns into aspects, that have to be coordinated with other concerns across component boundaries. Aspects *cross-cut* component boundaries, while components are characterized by being cleanly encapsulated. An "aspect weaver" (a kind of a compiler) weaves components and aspects together. The approach does only introduce limited (dynamic) changeability through re-weaving, but it can express concerns that cross-cut several components properly. In contrast to our approach, aspect-orientation is not inherently a black-box approach, but requires knowledge of the component's internals to build useful aspects.

Meta-object protocols, as in [9], divide a system into meta-level and base-level. Meta-level objects impose behavior over base-level objects. Generally meta-object protocols are low-level, but powerful; they can achieve reflection, dynamics, and transparency. Our approach provides meta-classes with similar abilities, but most often the interceptors are more powerful, easier composable, and provide more introspection facilities.

Bosch proposes in [2] a component adaption technique based on layers, which is similar to the presented interceptors: it is also transparent, composable and reusable, but it is not introspective, not dynamic and a pure black-box approach. Layers are given in delegating compiler objects, that are statically defined before compilation time. This approach can be hardly used for expressing runtime dynamics in component composition, since changes in layer definitions require recompilations.

Subject-oriented programming models [6] offer different views for a client onto a concern. Classes can be composed with composition rules. In contrast to composition with interceptors, it does not provide introspective and dynamical runtime composition, but only by a tool called subject compositor. Subject-orientation (and subsequent approaches) address composition of system parts with extensions and multiple views, thus it helps to overcome some of the problems of (descriptive) component composition. Central runtime problems of combining components, like adapting components without interference with component internals at different granularities, component introspection/runtime traceability, or dynamic component loading/unloading are nearly not addressed.

7 Conclusion

We have addressed the issue that software architectures should be build with reusable (off-the-shelf) black-box components, but should also exploit the characteristics of white-box object-oriented frameworks regarding evolvability and flexibility. We have presented a practical approach that integrates such black-boxes, written in several languages. An object-oriented scripting language serves as a component glue with explicit export/import interfaces and as a central place to introduce changes into the hot spots of the architecture. To overcome several problems of object-orientation, the language offers interception techniques, that are valuable for component composition and adaptation. All presented techniques can be used in various other languages through explicit programming by hand. Nevertheless, (runtime) language support for introspection, dynamics, component composition/decomposition, and interception techniques is useful, since it leads to shorter, more elegant, and less error-prone solutions. Since XOTcl is itself a C library it can be embedded in any C or C++ application as a distinct component glueing language.

XOTcl can be downloaded from `http://www.xotcl.org`.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. LCNS 791, Springer-Verlag, 1993.

[2] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 1999.

[3] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.

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

[5] M. Goedicke, J. Cramer, W. Fey, and M. Groše-Rhode. Towards a formally based component description language a foundation for reuse. *Structured Programming*, 12(2), 1991.

[6] W. Harrison and H. Ossher. Subject-oriented programming - a critique of pure objects. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages (OOPSLA)*, 1993.

[7] G. C. Hunt and M. L. Scott. Intercepting and insturmenting COM applications. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.

[8] IONA Technologies Ltd. The orbix architecture, August 1993.

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

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[11] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.

[12] M.Goedicke, G. Neumann, and U. Zdun. Object system layer. Accepted for publication in EuroPlop 2000, 2000.

[13] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.

[14] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.

[15] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proceedings of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.

[16] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. Accepted for publication in the World Wide Web Journal 3(1), 2000.

[17] G. Neumann and U. Zdun. Highly flexible design and implementation of an xml and rdf parser/interpreter. to appear, 2000.

[18] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

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

[20] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press Books. Addison-Wesley, 1995.

[21] D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.

[22] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.

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