Evolution of Object Systems
or
How to tackle the Slippage Problem in object systems

Kai-Uwe Mätzel, Walter Bischofberger

Ubilab, Union Bank of Switzerland
Bahnhofstr. 45, CH-8021 Zurich

e-mail: {Kai-Uwe.Maetzel, Walter.Bischofberger}@ubs.com

Object technology enables software developers to implement flexible, extensible, and adaptable systems. Important approaches to provide systems with these features are frameworks and design pattern-guided design. Nevertheless, object systems just like more traditional ones are hard to adapt to requirement changes, which are unpredictable at the system's development time. In this paper, we discuss the potential of object technology to build systems that are prepared for change. Evolution in general and undirected evolution of object systems in particular are investigated. We elaborate system properties which we consider crucial in consideration of adaptations to unpredictably changing requirements. We consolidate our experience in a collection of design and implementation ideas reaching from general to technically detailed. These ideas are helpful for designers and developers who intend to make systems more flexibly adaptable or to a certain degree self-adaptive to the requirement changes. [1]

1 Introduction

Software engineering is currently dominated by object technology. It is popular due to its promise to foster reuse and reduce the effort necessary for system evolution.

Object-oriented design usually leads to appropriate levels of abstraction. Abstraction makes the various layers independent from details of underlying layers. It is a generic concept that can be applied at application, infrastructure, and system level.

Inheritance, abstract coupling, and abstract or late creation are examples of object-oriented techniques that rely heavily on abstraction. Such techniques, together with widely available capabilities like dynamic loading or code shipping allow to build flexibly adjustable and dynamically extensible systems.

Are there limitations to the flexibility of object systems? Obviously, flexibility relies on appropriate abstractions. Therefore, in order to find these abstractions, a system has to be designed with flexibility explicitly in mind. Unfortunately, in consideration of evolution the required flexibility is only partially known. Only few changes and steps of evolution can be foreseen.

Obviously, we have to distinguish explicitly between predictable and non-predictable evolution. Predictable means that at least the direction of the resulting impact of future evolutionary steps is known. Therefore, we call this kind of evolution *directed*. Unpredictable evolution is called *undirected*.

[1] In *Proceedings of the Ubilab Conference '96, Zurich.* Universitätsverlag Konstanz, 1996.

Evolutionary steps, both directed and undirected, can have varying granularity. Usually, large evolutionary steps imply minor subsequent changes. For example, the substitution of an obsolete system component with a new one requires certain interface adaptations within the remaining system. Minor changes which just refine or slightly modify system components such as modifying a parameter list or adding new sub-classes cause a certain amount of slippage between the affected components.

Experience shows that if slippage-causing changes can be smoothly carried out, big steps become easier since they can be focused. Consequently, it is important to make systems more robust against the slippage-causing minor changes in order to prepare them for undirected evolution. According to this observation, we define the *Slippage Problem of Evolution* as the problem to cope with these slippage-causing minor changes and adaptations of undirected evolution.

The key to solve the Slippage Problem is a sophisticated system structure and flexible, adaptable coupling mechanisms between objects. Independent from the actual system architecture, a system needs a distinguished part which embodies the central design decisions of the system. This part is called the *design center*. In slippage-tolerant design the design center remains stable over time. This is accomplished by exclusively using highly flexible, dynamic mechanisms to couple the objects of the design center with outside ones. They are capable of absorbing most of the slippage-causing changes at runtime. Slippage-tolerant design raises the issues of runtime failure. Therefore, system designers always have to make a trade-off between flexibility and slippage tolerance on one hand and static checking on the other hand.

This paper discusses the potential of object technology to build systems that are prepared for change. We consolidate our experience in building complex distributed and non-distributed systems (Sniff [Bis92], Beyond-Sniff [Bis95], and GBO [Bis96a, Bis96b]) and integrate them with the work done by other researchers. We present a first result of our ongoing efforts to provide an overall conceptual framework consisting of design rules that guides developers in how to design and implement slippage-tolerant object systems. The recommendations reach from general design ideas to detailed technical constraints.

The next section discusses evolution compatible system design related to directed as well as undirected evolution. Section 3 presents the most critical aspects of object coupling. Section 4 list various concrete approaches and mechanisms that should help developers to solve the Slippage Problem in their own systems. Section 5 summarizes the content of the paper and Section 6 elaborates related topics.

2 Evolution Compatible System Design

In this section we dicuss approaches how systems should be designed to support evolution. More specifically, we show which properties an object system or a system in general must provide in order to be prepared for evolution.

2.1 Preparing a system for evolution

When is a system well prepared for evolution? We consider a system well prepared if it serves as an enactment of the following requirements:

(1) The system minimally depends upon its environment.
(2) The components of a system minimally depend upon each other.
(3) The necessary dependencies are represented exclusively by flexible mechanisms.
(4) The system consists of components which can dynamically be modified and extended.

Consequently, a system design should reduce inter- and intra-system coupling to a minimum and make the remaining coupling most flexible. The reduction of coupling has been a well studied issue for years, especially in relation with modular software design [Pom93] and research in module interconnection languages [Pri86].

In the following sections we discuss concrete approaches to a system design that meets these requirements concerning directed and undirected evolution.

2.2 Directed evolution

Directed evolution means that we can forecast which requirements might change and what the subject of change will be. Therefore, preparing a system for directed evolution requires appropriate abstractions which make the subject of change an explicit part of the design.

Object-orientation provides us with the opportunity of fine-adjustable levels of abstractions by means of sub-classing and abstract coupling. Abstract and partially abstract classes can be used to explicitly model the subjects of change. Changes can be performed by substituting objects of appropriate sub-classes for the original ones.

Design patterns as introduces by Gamma et al. [Gam95] embody substantial design experience in dealing with the explicit modeling of change. Each design pattern proposes a concrete solution for a specific class of design problems. The solution always considers at least one particular aspect to be a subject of change. This makes design patterns very valuable because the subject is not always obvious at the first glance.

Table 1 lists a few examples that show which design patterns could be used according to the particular subject of change.

Subject of Change	Used Design Patterns
System configuration	**Abstract Factory**
Implementations of particular functionality	combination of **Bridge** and **Abstract Factory**
Algorithms and abstract functionality	combination of **Strategy** and **Abstract Factory**, **Visitor**
Object state, extensible functionality	**Decorator**, **Extensible Object**

Table 1: Examples of design patterns for certain subjects of change

2.3 Undirected evolution

The problem of directed evolution can be tackled by explicitly modeling the subject of change. This cannot be done easily in the case of undirected evolution because the subjects of change are not explicitly known in advance. So, how can we prepare a system for things we do not know?

Since we cannot make any assumptions about future changes, we cannot work towards an appropriate problem-specific system structure. We propose to take a view of the system’s structure which allow us to better assess the potential of changes of the various system parts. We can then instrument these parts with generic evolution support mechanisms.

In order to find an appropriate system view, we consider a comparison between software architecture and architecture useful.

Steward Brand [Bra94] studied the evolution of buildings. He explains a building to be consisting of six shearing layers of change reaching from the geographical Site, over the Structure to the Stuff comprising all furniture. Each layer has its own pace of change: Site and Structure are almost static whereas Stuff is rapidly changing. Although these layers are closely related to each other, he requires a building to be built in such a way that the layers with high change rate can be changed or substituted without affecting the slower changing layers. Obviously, the resulting design provides the building with a certain slippage tolerance between the several layers.

We call a design that provides comparable mechanisms to Brand's shearing layers and flexible coupling between the layers *Design for Slippage*. Design for Slippage classifies changes by means of the layer where they occur and therefore makes them better predictable. The design decision manifested in the slowest layer are almost persistent and dominate the architecture. We call that slowest layer the *design center*.

Considering a system under the point of view of Design for Slippage there are two major issues:

- determine all system parts which belong to the design center
- determine the layer coupling mechanisms

2.3.1 Find a design center

If a software system is considered to have a design center, it can be successfully prepared for undirected evolution.

Apart from shearing layers, every software system consists of various hierarchical layers. At least, there are system, infrastructure, and application layers. Concerning this hierarchical layering, a design center is considered to be specific for a particular layer. Technically, the design center is a distinguished set of classes and object cooperation structures as well as the most important layer-specific infrastructure services.

Consider a desktop manager like the Macintosh Finder. It provides navigation and management support for hierarchical, recursive structures which are accessible from the workplace. Examples are files and folders. Assume the design of the desktop manager models this concept explicitly. The center of this design might be the desktop-item manager hierarchy together with its dynamic behavior. Furthermore, it comprises cooperation pattern between hierarchy and workplace manager as well as the workplace manager itself.

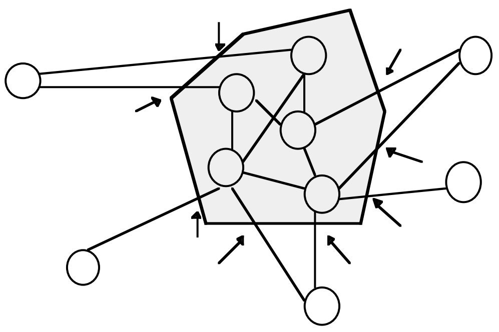

Figure 1: The edges of a design center

Having a design center, we can consider anything outside the center a potential subject of change. Generically ensuring the center's stability means to make no assumption about the nature of the outer parts. Therefore, the edges of the design center (Figure 1) are the place or layer where slippage occurs.

Finding a design center requires an explicit decision about what belongs to the design center and what does not. Unfortunately, there is only vague knowledge how to make these decisions. To provide the developers with guidelines and rules of thumb for the decisions is currently an area of active research. We just want to give two examples:

- The major software architectural styles[2] of a system should be defined by its design center. Therefore, everything considerably contributing to the architectural styles has to be part of the center.
- Mostly, the major abstractions, especially the interaction behaviors, are richer and more specific than a combination of single architectural styles. Therefore, additional to the part defining these styles, all elements embodying the specific features belong to the center.

2.3.2 Implement flexible object coupling

Slippage occurs at the edges of the design center. These edges are the coupling mechanisms between the design center and the outside environment. Object coupling is caused by depending on the interface of an object and the semantics of the related methods. Furthermore, coupling to the outside environment includes depending on particular system services, file formats, and so forth.

Consequently, a system has to be designed in such a way that the coupling mechanisms between inside and outside objects as well as their system environment are change-absorbing. This implies that the dependencies between the involved objects are reduced as much as possible. Depending on particular system services and properties can usually be reduced by introducing appropriate abstraction layers.

2.4 Tackling the Slippage Problem

Tackling the Slippage Problem means to support the finding of a design center and to provide appropriately flexible object coupling. The general idea is to turn undirected evolution into directed evolution on the meta-level by providing explicitly modeled, adaptable mechanisms for flexible object coupling. The determination of the design center guides where these mechanisms should be used. We focus on generic flexible object coupling because finding the design center heavily depends on the particular system.

Although there are various degrees of sensible flexibility, the process of achieving weaker coupling can be described as follows: The paradigm "Program to an interface" [Gam95] has to turn into "Program to a self-describing language-, system-, and compiler-neutral specification". Intrinsic to this paradigm is that coupling decision are made dynamic by shifting them to runtime. Changes such as interface modifications can so dynamically be neutralized by automatically using call adapters.

Assume building a component of a distributed system. The component requires services from other components of the system. Programming to an interface in this constellation

---

2 A software architectural style according to [Sha96b, Sha96a] describes design rules which identify the kinds of components and interaction behavior patterns including all interconnection constaints which may be used to build a system. The most popular ones are *Pipes and Filters* and *Client-Server*.

means that the client component is implemented by using the interface definitions of the service components. Programming to a specification means that the client component looks at the rest of the system as one whole service provider. It specifies the services it needs and does not care about which components actually provide which services and how requests are mapped onto the interface of service components. In the ideal case, service components and clients can be almost independently evolved without breaking the system.

Although the nature of object coupling is independent from any architecture, the design and implementation of flexible object coupling mechanisms are not. The architecture defines always additional requirements the coupling mechanisms have to meet. For example, the coupling of distributed objects has to be more flexible than between object in the same address space. Nevertheless, there are some generic mechanisms which can be implemented with any kind of language or infrastructure. They can be profitably used for the critical object couplings independently from the actual architecture.

3 The Aspects of Object Coupling

In this section, we present the major aspects of object coupling. They are relevant for the development of flexible coupling mechanisms as asked for in Section 2.4.

In order to cooperate with each other, objects have to communicate. Communication always requires an established connection between the objects and a common understanding of the exchanged data. We consider object communication, connection management, data representation, and dealing with dynamically changing objects most important.

3.1 Object communication

Objects communicate by means of messages. Communication can either be anonymous or directed. Using anonymous communication, the sender of a message knows neither who the receiver of the message is nor if a receiver exists at all. Using directed communication, the sender explicitly sends a message to a particular receiver.

The receipt of a message causes the call of a method of the object. Which method is eventually called depends on the actual dispatch strategy. Furthermore, the behavior of an object is determined by the fact whether sending messages is blocking or non-blocking and whether the object or its methods are reentrant or not.

We consider the following aspects of object communication relevant:

- identity of the communication partners (known or anonymous)
- message format
- dispatch strategy
- blocking or non-blocking sending
- reentrance of object and methods

3.2 Connection establishment

Although in the case of anonymous communication the details of connection establishment are hidden from the communicating parties, connection establishment takes place at least on the infrastructure level. For directed communication, connection establishment is of principal concern.

Establishing a connection comprises:

- the lookup process
- the bootstrapping process of objects or object clusters

The lookup process defines how an object gets in contact with other objects. There are object systems which do not explicitly support the concept of finding. It is hidden, for example by giving an object reference as a parameter to a method and making the referenced object known this way. Other systems make finding explicit for example by means of an object broker.

The bootstrapping policy describes how an object system can be brought into a “ready-to-use” state. This includes the initialization of an object or a cluster as well as its registration with the system.

3.3 Connection termination

The issue of connection termination mainly deals with the question how the closing of connections affects the involved objects. Objects can be permanent in the sense that they can even exist without any other object connected to it, or objects can be automatically deleted as soon as there is no longer any connection.

3.4 Data representation / data model negotiation

To exchange information, all involved objects have to agree on the format and the semantics of the exchanged information. Concerning evolution, the main issues are the flexibility and dynamic extensibility of the used representation as well as the support provided for explicit data model negotiation. Consider the case that two objects exchange structured information records. If the sequence or the number of the fields of the records is changed, this should not affect the two objects, as long as they are not directly interested in this change. Explicit data model negotiation allows to exchange necessary meta-information previous to the actual data exchange and thereby provides a high degree of flexibility. Meta-information for the previous example could contain the number and sequence of the record fields as well as the name and the type of each field. If a field has been renamed, the receiver can gather this information from the meta-information.

3.5 Modification of object structures and behavior

The above points are undoubtedly closely related to object coupling. This is not so obvious for mechanisms to manipulate the packaging, the structure, or the behavior of objects. Examples thereof are splitting a single object into multiple smaller ones, the extension of objects by new state variables or new methods, and the substitution of methods. Obviously, there must be an insurance that the changed and newly introduced methods participate in the dispatching strategy discussed in Section 3.1. Such manipulations can either address a single particular object, a group of objects, all instances or a certain class with all its sub-classes.

Object structure and behavior modification introduces a distinct kind of object coupling: (a) There is coupling by means of use- and aggregation-relationships. Methods are called and objects are accessed according to the cooperation patterns of the objects. (b) There is coupling by means of control relationships. Controllers manipulate the behavior or the structure of controlled objects. Obviously, this can modify the ordinary use- and aggregation-relationships.

Major aspects of object structure and behavior modification are capabilities to deal with:

- state-space modification
- function/code shipping [Fla96]
- dynamic object packaging

4 Designing and Implementing Slippage Support

The following sections of this chapter discuss some specific designs and implementations of flexible object coupling mechanisms. They are reaching from pure technical items such as self-describing method calls to items which support a particular design idea such as facades and roles. Each one of these mechanisms solves a particular set of the aspects of object coupling discussed in Section 3. In particular, we discuss:

- self-describing method calls
- anonymous communication
- trading
- dynamic facades
- roles
- dynamic objects
- extended semantic data representation mechanisms

This chapter mainly shows design experience which can be found implicitly in various existing systems. The given list of flexible object coupling mechanisms is not exhaustive.

4.1 Self-describing method calls

Method calls in object systems usually depend on a particular compiler, programming language or infrastructure. Furthermore, they are highly sensitive to even minimal signature changes such as renaming, adding, removing, or rearranging parameters. It is crucial for slippage-tolerant systems to have method calls which are independent from a particular system and resistant to these changes as much as possible. A very useful approach are *self-describing parameter lists*. Such a list consists of associations of formal parameter, actual parameter, and data type of the actual parameter. Therefore, self-describing implies dynamically type-checked, position- and length-tolerant parameter lists.

In a system that does not provide any or insufficient runtime type information, this has to be introduced by a semantic data representation mechanism. A semantic data representation mechanism (SDRM) [Mät96] defines a dynamically extensible set of scalar and composed data types together with an interface for creation, modification, testing, as well as internalization and externalization of instances of these types. SDRMs are language-independent and an excellent means of data integration.

This technique scales very well. Either an entire parameter list or parts of it can be self-describing. The degree of flexibility is finely adjustable: where considered necessary, checks are done statically, otherwise they are dynamic. Obviously, the adjustment is influenced by the application specific trade-offs between possible runtime failures and flexibility.

The strategy of method dispatching relies on the types of the actual parameters. Therefore, closely related to self-describing method calls are dispatch strategies specific to the used SDRMs. They have to consider the dynamic aspects of the parameters' representation.

Related to Section 3, self-describing method provide flexible solutions for the following aspects of object coupling:

- object communication
    - message format
    - dispatch strategy
- data representation

Examples of self-describing method calls

Flexible parameter list. Independence from the length of a parameter list can be accomplished by using run-arrays as parameters, sequence- and length-independence by using associative arrays or dictionaries. This is applicable in any object-oriented language.

Language feature. Programming languages usually provide some features for flexible parameter list. The quality and the power of these features vary greatly.

C++ allows length-independent parameter lists by using ellipsis (varargs). RTTI introduces type reflection and allows, combined with ellipsis, truly self-describing parameter lists based solely on C++ language features [Str94].

Dylan [App96] provides various language features for slippage-friendly method calls. First, Dylan provides complete runtime type information by representing classes as objects like Smalltalk or ET++ [Wei94]. Second, a method definition determines the tolerance with which the method is still callable. Dylan supports statically and latently typed parameters, position and key word parameters, and formal parameters that are bound to a list of actual parameters.

Existing SDRMs are either an integral part of a certain system or they represent a system themselves.

CORBA [OMG96] defines an integrated SDRM, represented by the IDL data type any. An IDL-any can contain any existing IDL type. Furthermore, the definition of CORBA's dynamic invocation interface (DII) is a good example for handling self-describing method calls. The parameter list of a service request, which eventually is mapped onto a particular method, is represented by a name-value-pair list (NVList). Each value is represented by an IDL-any. Externalization of IDL-anys is defined by the CORBA Externalization Service.

NEWI [Sim94] is an integration environment for cooperative business objects (CBOs). It has to cope with flexible data and control integration between distributed, heterogeneous components. It tackles the problems by introducing a SDRM called Semantic Data Streams (SDS).

In contrast to IDL-any and SDS, Anythings as well as the Any Framework [Mät96] are independent SDRMs. Using Anys, a self-describing parameter list can be modeled by taking one or multiple AnyFrames as parameters. An AnyFrame is basically a dictionary that is structured according to a defined schema. This allows very efficient conformance checks of the transferred actual parameter. Anys are implemented in various programming languages.

4.2 Anonymous communication

A step beyond flexible method invocation is to reduce the necessary amount of knowledge even more. Communication requires an established connection between at least two objects: the sender and the receiver. Communication can therefore be made more flexible by freeing the sender of a message from the knowledge of the receiver's identity or the number of receivers. For example, this is of very interest for change propagation in object systems or more generally, for event distribution. Anonymous communication can be achieved by using a *message bus*.

A message bus enacts a communication policy known as publish/subscribe communication. The sender object publishes a message. All interested subscribers receive this message. The sender object only has to know the message bus. How an object can subscribe for messages depends on the particular message bus. There are various flavors: publishing/subscribing using subjects/keywords, patterns of message structure, patterns of message content, and so forth[3].

Message bus communication requires just a few prerequisites: The message bus has a fairly simple view on objects: they can only publish messages and subscribe to messages. Objects as well as the message bus have to deal with messages: construction, parsing, deletion. All communication objects as well as the message bus have to agree on the message representation. SDRMs are an adequate means for representing messages.

Message busses are very flexible. The connection between objects can be transparently established and terminated without any additional knowledge except the message bus API. Subscribers or publishers can be substituted on the fly without affecting any other part of the system.

Message bus architectures have a few drawbacks: Depending on the actual message bus architecture, there is one single point of failure and a potential bottleneck. Existing communication patterns between objects are no longer explicitly visible in the system design. Causal relationships between various distributed events are hidden internally. For example, in order to get a request/reply behavior, the request publisher as well as the request subscriber(s) have to register themselves with the message bus prior to emitting the request.

Anonymous communication provides a slippage-tolerant solution for

- anonymous object communication
- connection establishment
- finding /lookup

Depending on the kind of used message bus, the following issues are addressed as well:

- bootstrapping
- connection termination

Examples

Anonymous communication is used in various contexts. Well known examples are change propagation in the sense as described by the Observer design pattern [Gam95] or implicit method invocation in object systems as described in [Rie96]. The Mediator design pattern [Gam95] resembles the message bus architecture.

Systems actually implementing a message bus, already have a long tradition. Linda [Mat88], a system originally developed at Yale University defines a so-called tuple space. Processes can publish tuples and subscribe to tuple patterns. Newer systems are for example the Rendezvous Software Bus [Oki93], ToolTalk [Sun93], HP SoftBench [HP92],

---

3 A message bus describes a particular broadcast universe, whereas all subscribers which are interested in the same set of messages form a multicast universe. Therefore, we consider in this paper group communication a special case of a message bus.

and OrbixTalk [Ion96]. The CORBA Event Channel Service makes anonymous publish/subscribe styled communication available for each CORBA application. Typical application areas of the above mention systems are tool integration (ToolTalk, HP Softbench) and the distribution of large amounts of information to a dynamically changing set of receivers such as trading floors (Rendezvous Software Bus).

4.3 Trading

For some applications anonymous communication is not applicable. For example, if an object requests various different but maybe related services and it wants to be sure that all services are provided by the same server(s), then the message bus is not an option.

In those applications a connection between objects should still be established at runtime. In contrast to anonymous communication, the objects are aware of the notion of an explicit connection. An object has to query for a service provider. If there is at least one object that can provide that service, the requester receives a reference to this object. This reference reveals only those properties of the object which are relevant to the requested service. Such properties can be a selection of the supported interfaces, the runtime behavior, and so forth.

Generally speaking, there are service providers and service requesters. The process of finding a service provider, e. g. the matching of requests with offers is called *trading* [ISO93a]. The instance that performs trading is called a *trader*. A trader can be considered to be placed between a broker, as implemented by CORBA’s ORB, and a message bus used for anonymous communication.

Trading is a highly complex process and in larger systems closely related to runtime management. The trader analyzes the service request and checks if there is an offer matching the request. This is the case, if the provider can talk the same protocol (independent from the actual type of the provider) as the request specifies. If there are no matching offers then the trader could check for offers that meet the request only partially. If possible, it constructs a “virtual” service provider based on the offers found. This allows, for example, to transparently install version-to-version converters. Trading can include the launching of appropriate service providers in the case that there are non at request time. Trading effectively implements “Program to a self-describing language-, system-, and compiler-neutral specification” as explained in Section 2.4.

Trading provides slippage-tolerance to the following aspects of object coupling.

- directed object communication
- connection establishment
- connection termination
- data model negotiation (partially)

Examples

Object trading can be used in small monolithic object-oriented systems, but of course is most useful in distributed object systems. The presented examples meet the requirements of trading only partially. Trading is still an area of active research.

Late creation as described in [Rie95] is a particular example of restricted trading combined with object creation. A client requests an object that is of a specific type. Furthermore, the object has to meet a certain requirement specification. Late creation defines the process of finding an appropriate sub-class and creating an object of that class. For example, an object request a Collection object which ensures the insertion sequence of the contained objects.

Depending on the particular selection process, the returned object can be a SequentialCollection, an OrderedCollection, or something else which meets the requirement.

Late creation is restricted trading in the sense that it does not support the construction of newly compound objects and the trading scope is limited to sub-classes of a particular class.

Open Distributed Processing - Reference Model. Trading is of particular interest in large distributed object systems. Therefore, the ODP-RM [ISO93a, ISO93b] defines a trader, including its interfaces and functionality as one of its major parts. Currently, there are no complete implementations available. CORBA is a particular, restricted instance of the ODP-RM. The trading functionality is partially built into the object request broker. The broker can find and launch servers which implement a requested interface. A trader definition within CORBA is currently being defined.

4.4 Dynamic facades

Indirection decouples objects. Objects do not have to know each other, they just have to know a deputy which either hides the coupling transparently or acts in behave. Gamma et al. [Gam95] identified various structural design patterns, whose major purpose is to decouple objects by introducing additional layers of indirection (see Section 2.2). These are especially Facades, Bridges, and Mediators.

Levels of indirection are such a fundamental design vehicle for flexible coupling that it must be easy and efficient to use. It should be possible to dynamically introduce levels of indirection as soon as a more flexible coupling is necessary. For instance, this could be the case in the context of runtime management. In this section and the following section we introduce the notions *dynamic facade*, *role object*, and *core object* in order to enable naming and referencing of particular indirection concepts in use. They are very powerful in combination with anonymous communication and trading.

A dynamic facade introduces an additional level of indirection and supports efficient object packaging. It generally resembles a facade introduced by [Gam95], only that it is more flexible. A facade according to [Gam95] is an object that dynamically encapsulates the objects of a whole subsystem as well as their relationships. The facade provides a proper interface for all or a particular subset of services provided by the objects of the subsystem. These services should only be accessed via the facade object. This makes it easier to evolve the subsystem without directly affecting its clients.

A dynamic facade can be dynamically created, modified, and deleted. It encapsulates a subsystem whose essential parts can be modified, exchanged, and restructured at runtime. Its interface provides methods to add objects to and remove them from the subsystem. Additionally, each object that is registered with the dynamic facade can promote operations to it. Promoted operations are part of the facades interface. Calling a promoted method initiates the forwarding of the call to the original object. Depending on the actual implementation, promotion can for example be executed by means of shipping code to the dynamic facade object (see Section 4.5), or the installation of callback functions. In contrast to ordinary facades, dynamic facades avoid the introduction of new facade classes in the case that the configurations of the subsystems change. Furthermore, they enable the encapsulation of previously unknown subsystem configurations.

A dynamic facade is interesting in consideration of the following aspects of object communication:

- dispatch strategy
- indirect connection establishment and termination
- object packaging

Examples

Beyond-Sniff [Bis95] distinguishes between two different kinds of dynamic facades: ServiceApplications and ProjectServiceTrees.

A ServiceApplication can contains any number of Services and provides them with a thread of control. A Service registers with the ServiceApplication under a certain subject. The ServiceApplication can then receive requests according to this subject and forwards the calls to the service. The Services are not only encapsulated by a ServiceApplication but they are completely folded into it. This shows that in large distributed systems the requirements of a dynamic facade can change: They do not just serve as proxies for their content. They provide the “computational container” from them.

A ProjectServiceTree is a dynamic facade for a changing set of services which are related to a certain project [Bis92, Bis95]. A clients always talks to ProjectServiceTree which dispatches and maps the calls on the proper services. For more information we refer to [Bis95].

4.5 Roles- and core objects

Objects usually have multiple clients. The particular context where an object is actually used can considerably differ from client to client. Nevertheless, an object has to provide a proper interface to meet all the requirements related to the different contexts. Therefore, the protocol of an object is a mixture of all the context specific protocols [Ree96].

Accordingly, objects can be designed by preferring object composition instead of inheritance: An object consists of a *core object* and various context specific protocol objects. A core object is designed to provide a number of specific services. It provides a concise and small interface for these services. The interface is not tailored according to the requirements of certain contexts of usage.

A protocol object is designed for one distinguished core object and one particular context. It provides the context specific interface of the core object’s services. We call a context-specific protocol of the core object its *role*. A role defines a client's view on the core object [Ree96]. From a technical point of view, role objects can be considered a special kind of facade which is specific according to the type of the client. Clients just see the protocol objects, the core object is not directly accessible.

Roles increase the reusability of core objects. Core objects are defined independently from role objects. New roles can be easily defined. Without specific technical support, we are faced with the following problems:

- The introduction of a new role probably extends the possible states of the object. State relevant instance variables are scattered over role and core objects. Specific properties such as persistency have to be implemented for the core as well as for all role objects which carry state information. Therefore, all clients of the same type have to share the same role object. The role object must exist as long as it is needed by a client.
- The distribution of state information raises future problems which can be described as a consistency update problem: A client for example invokes the SetName method of an

object, e. g. it invokes the SetName method of a role object which executes some proprietary code and finally forwards the call to the core object.

This shows that each role object can define certain proprietary functionality for the SetName method. Consequently, changing the name via a particular role nevertheless requires the appropriate execution of the SetName functionality of all possible roles.

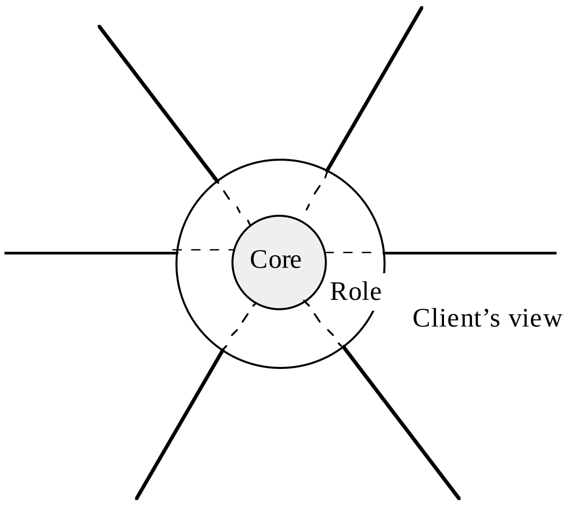

Figure 2: Core and role objects

In order to solve these problems we propose the explicit design and implementation of core and role objects according to the following requirements:

Core objects support state and behavior integration. A core object can be dynamically extended by new state information, e. g. arbitrary accessible information can be transparently attached to the core object. Furthermore, each method of the core object can be extended by new functionality. Extending can be accomplished by code shipping or callback functions.

State integration avoids the distribution of state information: all state information of an object is managed by its core object. Behavior integration provides a framework to achieve consistent updates.

Role objects are quite similar to dynamic facades. They are purely functional, light-weight objects. They just maintain a reference to their core object. The value of that reference can be either passed as a parameter at creation time or the role object itself looks it up as soon as the core object is needed. Role objects provide an interface in order to integrate their state information and their update functionality with their core object.

Creation and deletion of role objects should be extremely cheap. Due to state integration and cheap creation, there is no need to share role objects between clients. Core objects should be implemented as dynamic objects (see Section 4.6).

Role and Core objects are closely related to role-based modeling as described by Reenskaug [Ree96] as well as subject-oriented programming as introduced by Harrison and Ossher [Har93]. They primarily address the decoupling of clients and servers by means of

- indirect connection establishment & termination.

4.6 Dynamic objects

Dynamic facades and core objects have several things in common: Their behavior can be dynamically changed and the structure of core objects can be dynamically modified. We call objects with dynamic structure and behavior *dynamic objects*. How they are implemented highly depends on the used technology.

Examples for pragmatic approaches are the following: Using interpreted and dynamically compiled languages, dynamic behavior is for free. Using compiled languages, this can be accomplished by either exploiting dynamic loading of code or by building a hybrid system that incorporates both interpreted and compiled languages. An object can be dynamically changed by attaching code scripts to methods or by substituting built-in methods by code scripts. The latter raises the need to provide mechanisms to represent the object’s environment in the runtime environment of the interpreted code. Furthermore, the dispatch policy has to consider code scripts accordingly.

The dynamic structure is a good application area for SDRMs. For example, a class can define an instance variable which can store any kind of data that a particular SDRM can represent. Consequently, objects of this class can carry arbitrary information, which is selectively accessible due to the SDRM’s reflectivity. Therefore, these objects can be considered to have an extensible set of instance variables. Core objects (see Section 4.5) are such objects.

Dynamic objects face us with a similar kind of trade-off as self-describing method calls. The type of a dynamic object gives only insufficient information about the capabilities of a particular object. If the type defines a certain method then the dynamic object supports this method. The programmer has to ensure that a method modified or overwritten by a code script still embodies the same semantics. A type of a dynamic object just defines the minimal set of methods and instance variables an object of that type has to have. Since it can have more instance variables and support more methods, dynamic objects have to provide an interface to query information about their current structure and abilities.

Examples

The capabilities dynamic objects provide are widely known requirements. Gamma presents in [Mar97] the Extensible Objects pattern that proposes how to implement them.

Beyond-Sniff [Bis95] services are coarse-grained dynamic objects. A service contains an Any as an instance variable by which it can be structurally changed. Furthermore, each service supports so-called RequestHandlers. They can be added, changed, and removed. A RequestHandler contains a Python script and basically represents an implementation of a method of the service. It is called whenever the appropriate method is invoked.

Further approaches to dynamic objects are reflective object architectures and meta-object protocols (MOP) as implemented in CLOS [Kic91] or GBO [Bis96a, Bis96b]. Context relations as defined by Seiter et al. [Sei96] provide an interesting theoretical background for dynamic objects.

4.7 Extended semantic data representation mechanisms

Almost each one of the introduced mechanisms profits from SDRMs, although they are rather simple. Unfortunately, SDRMs are not well suited for the handling of large amounts of data which would make them even more useful.

Dealing with larger amounts of data requires some features from a SDRM, which it usually does not provide:

- scaleable structuring constructs, and
- declarative access to data items.

A SDRM providing these features is called Extended SDRM (ESDRM) [Mät96].

Examples

Anys (already discussed in Section 4.1) are an ESDRM. Beside a dynamically extensible set of data types and their externalization, Anys comprise the structuring concept “AnySoup”. An AnySoup can contain an arbitrary number of AnyFrames. An AnyFrame is basically a dictionary that is structured according to a defined schema. Schema definitions are unique within an AnySoup. Therefore, AnySoups serve as name spaces. Furthermore, each AnySoup comprises a query evaluator. A client can send OQL queries [Cat94] to an AnySoup and get all matching AnyFrames as results. A detailed discussion of Anys can be found in [Mät96].

5 Related Work

Evolution compatible system design has been an area of intensive research for a few years. We presented various examples in the Example sections of Section 4. As fare as we know, there are no systems which explicitly focus on a shearing layer architecture. In consideration of flexible object coupling we consider adaptive programming [Lie96], subject-oriented programming [Har93], and aspect-oriented programming [Kic96] very interesting. Furthermore, the ongoing efforts in the ODP trading community are an important source of influence. For detailed information about these topics we refer to the literature.

The introduced slippage mechanisms heavily depend on the underlying implementation technology. The most important aspect of an implementation technology are its available opportunities to influence the major mechanisms of object coupling. Dynamic, reflective programming languages and various object models such as CORBA provide various useful features to do so. These features are useful to implement the slippage mechanisms but they are not a solution by themselves.

In the area of object models we consider IBM’s System Object Model (SOM) [Lau94] and GBO’s System Object Architecture (SOA) especially interesting. These are models which provide various technical features to deal with evolution. SOM provides the following features, which make it a good choice for implementing slippage-friendly systems for non-distributed systems:

- Binary-to-Binary Compatibility avoids the need for recompiling clients of objects of changed classes as long as the change does not require modified program code. Changes belonging to this category are: methods can be added, methods which are not used by the client can be changed, the list of instance variables can be extended.
- Each class is represented by a class object. A class object can be used to query information about the represented class at runtime: protocol of the class, relationships to other classes, version number, and so forth. Furthermore class objects can be extended by dynamic methods, these are methods which are attached to a class at runtime. SOM supports three different styles of method resolution: offset, name-lookup, and dispatch function resolution. Class objects are directly involved in name-

lookup and dispatch function resolution which makes method dispatching system-specifically adaptable.

- Dynamic Link Libraries (DLL) are explicitly supported by SOM.
- Supports the Interface Definition Language (IDL).

GBO’s SOA tackles evolution by being reflective. As in SOM, each type is explicitly modeled by a type object. These type objects are an integrated part of the SOA runtime system, for example they decide about method dispatching, object migration strategies, and so forth. By means of the SOA transaction model, types, type interfaces, and type implementation objects can be smoothly evolved at system’s runtime. [Bis96a, Bis96b] explain SOA in detail.

6 Summary

The Slippage Problem is a major part of the system evolution problem. It can be tackled by providing mechanisms, which turn undirected evolution into directed evolution on the meta-level. An appropriate means is Design for Slippage. It generally defines a shearing layer architecture. Shearing implies that the various layers are flexibly coupled. Accordingly, we propose that a system should have a distinguished set of concepts and abstractions which forms its design center. The objects of the center have only a minimal amount of relationships to the outside. These relationships are represented by highly flexible coupling mechanisms. In order to support the further development of such mechanisms we presented the various aspects of object coupling which have to be addressed by those mechanisms.

Technical prerequisites for flexible coupling mechanisms are, for example, (extended) semantic data representation mechanisms and dynamic objects. The techniques are self-describing method calls, anonymous communication, connection establishment via traders, decoupling via dynamic facades, and the clear distinction between requirements caused by different contexts by means of role and core objects. These mechanisms rely on the notion of object-orientation and can be used in any kind of object system. However, the concrete examples of these mechanisms make clear that a specific implementation has to be individually tailored to the requirements of its particular object system.

The implementation heavily depends on the underlying technology. Although the requirements differ between various architectures, it is clear that the more flexibility the underlying implementation technology provides, the less effort has to be spent to implement the mentioned mechanisms. All these principles basically follow the same pattern: Object coupling previously established at compile- or link-time is shifted to runtime. Necessary compatibility can no longer be checked at compile-time but is now dynamically checked. Runtime system functionality like method dispatching can be explicitly changed. Therefore, the usage of these flexible coupling mechanisms is always a trade-off between the potential maximum of statically checked code and the necessary flexibility to make system slippage-tolerant and therefore, reduce maintenance efforts.

Acknowledgment

The authors especially thank André Weinand, who originally initiated this work unknowingly by giving a talk about pitfalls of large-scale object-oriented software development. We want to thank him and Dirk Riehle for their valuable contributions to this paper.

References

[App96] Apple Computer Inc.: Programming in Apple Dylan. Apple Developer Press, 1996

[Bis92] Bischofberger W: Sniff — A Pragmatic Approach to a C++ Programming Environment. In *USENIX 1992 C++ Conference Proceedings*, USENIX Association, 1992

[Bis95] Bischofberger W, Kofler T, Mätzel KU, Schäffer B: Computer Supported Cooperative Software Engineering with Beyond-Sniff.” *CSEE ’95*

[Bis96a] Bischofberger W, Guttman M, Riehle D, Stürmer C: *Global Business Objects System Object Architecture.* Union Bank of Switzerland, 1996

[Bis96b] Bischofberger W, Guttman M, Riehle D: Architecture Support for Global Business Objects: Requirements and Solutions. To appear in: *Proceedings of the Ubilab Conference ‘96*, Switzerland, 1996

[Bra94] Brand S: How buildings learn - What happens after they’re built. Viking, 1994

[Cat94] Cattell R (Ed.): The Object Database Standard: ODMG - 93. Morgan Kaufman Publishers, 1994

[Gam95] Gamma E, Helm R, Johnson R,Vlissides J: Design Patterns: Elements of Reusable Design. Addison-Wesley, 1995

[Fla96] Flanagan D: Java in a Nutshell. O’Reilly & Associates, Inc., 1996

[Har93] Harrison W, Ossher H: Subject-Oriented Programming (A Critique of Pure Objects). In OOPSLA’93 Proceedings, Washington, DC, 1993

[HP92] Hewlett Packard: SoftBench Encapsulator: Programmer’s Guide, 1992

[Ion96] IONA: OrbixTalk. White Paper, IONA Technologies Ltd., April 1996

[ISO93a] ISO/ITU: ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1

[ISO93b] ISO/ITU: ITU-TS SG7.Q16. ODP Trader

[Kic91] Kiczales G, des Rivières J, Bobrow D: The Art of the Metaobject Protocol. The MIT Press, 1991

[Kic96] Kiczales G, et al.: A Position Paper on Aspect-Oriented Programming. Working Paper, Internet Publication, 1996

[Lau94] Lau C: Object-Oriented Programming Using SOM and DSOM. Van Nostrand Reinhold, 1994

[Lie96] Lieberherr K: Adaptive Object-Oriented Software: The Demeter Method with Propagation Pattern. PWS Publishing Company, 1996

[Mar97] Martin R, Buschmann F, Riehle D (Ed.): Pattern Languages of Program Design 3. Addison-Wesley, 1997

[Mat88] Matsuoka S, Kawai S: Using Tuple Space Communication in Distributed Object-Oriented Languages. In *OOPLSA’88*, San Diego, CA, USA, September 1988

[Mät96] Mätzel KU, Bischofberger W: The Any Framework - A Pragmatic Approach to Flexibility. In *COOTS’96*, Toronto, Canada, June 1996

[Oki93] Oki B et. al.: The Information Bus - An Architecture For Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993

[OMG96] Object Management Group: *CORBA 2.0, Common Object Services and Common Facilities Specification.* 1996

[Pom93] Pomberger G, Blachek G: Software Engineering - Prototyping and Object-Oriented Software Development. Carl Hanser Verlag, 1993

[Pri86] Prieto-Diaz R, Neighbors J: Module Interconnection Languages. *Journal of Systems and Software,* Vol. 6, No. 4, 1986, pp. 307-334

[Ree96] Reenskaug T, Wold P, Lehne OA: Working with Objects. Manning, 1996.

[Rie95] Riehle D: How and Why to Encapsulate Class Trees. In *OOPSLA '95 Conference Proceedings.* Austin, TX, 1995

[Rie96] Riehle D: The Event Notification Pattern—Integrating Implicit Invocation with Object-Orientation. In *Theory and Practice of Object Systems*, Vol. 2, No. 1, 1996, To appear

[Sei96] Seiter L, Palsberg J, Lieberherr K: Evolution of Object Behavior using Context Relations. In *ACM SIGSOFT'96,* San Francisco, CA, October 1996

[Sha96a] Shaw M, Garlan D: Software Architecture - Perspectives on an Emerging Discipline. Prentice Hall, 1996

[Sha96b] Shaw M, Clements P: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Working Paper, Internet Publication, April 1996

[Sim94] Sims O: Business Objects - Delivering cooperative objects for client-server. McGraw-Hill Book Company, 1994

[Str94] Stroustrup B: The Design and Evolution of C++. Addison-Wesley, 1994

[Sun93] SunSoft: The ToolTalk Service An Inter-Operability Solution. SunSoft Press, 1993

[Wei94] Weinand A, Gamma E: ET++ - a Portable, Homogenous Class Library and Application Framework. In *Proceedings of the UBILAB Conference ´94,* Zurich, Switzerland, November 1994