Object System Layer

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

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

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

Often a project is faced with non-object-oriented languages or with object systems that are not powerful enough for the project's purposes. But nevertheless we want to apply advanced object-oriented techniques in these languages. Therefore, build or use an object system as a language extension in the target language, and then implement the design on top of this OBJECT SYSTEM LAYER.

Application Example: Document Archive System

A document archive system allows users to archive a large number of documents on optical storage devices and retrieve them through several search criteria that are stored in a database. The system was originally designed and implemented in C on a Unix platform supporting only one (Oracle) database management system. It was ported to several Unix variants and finally to Windows NT. Later, support for the Informix database management system was added. The system used optical storage devices with proprietary interfaces. On Unix and Windows the archive server and retrieval server were written in C, while clients on Windows were developed in C++, and the original Unix clients were written in C.

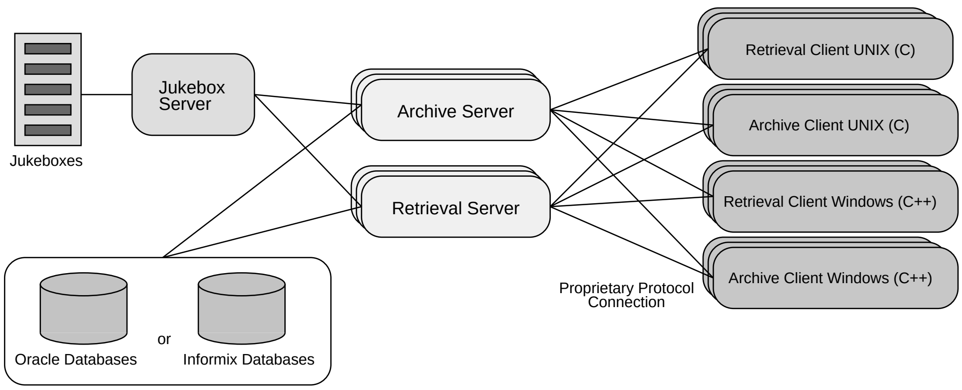

Figure 1. Document Archive System – Overview

During maintenance and evolution of the software system, the company faced several problems:

- It was hard to change the mechanisms with which clients and servers communicated, because they used a proprietary protocol based on sockets.
- It was hard to maintain the database code, because the system used two DBMS that were accessed by different protocols and that had different SQL dialects.
- It was hard to support the system on several platforms and with several versions of the programming language.
- It was hard for the company to configure the system for the customer requirements, since the document archive system has to be tightly integrated with the customer's IS infrastructure. No two installations of the document management system are exactly the same. The adaption of the system was programmed by hand during deployment using low-level C APIs.
- It was hard to replace cache management strategies for archival/retrieval, because they were not encapsulated in distinct components with well-defined interfaces.
- It was hard to integrate new storage technologies, because the system directly accesses the proprietary low-level interfaces to interact with the storage devices.

During a reengineering project, the forces in this software system led to adoption of an OBJECT SYSTEM LAYER, because the firstly envisioned solution of migrating the whole code base to C++ or Java would have required considerable costs in (useless) legacy migration, an additional concept for stepwise migration of the legacy parts, and a considerable redesign effort. The existing software is functioning and efficient. As far as possible existing parts should be reused in the reengineered solution.

Foreseeable future changes, like replacement of the existing proprietary communication infrastructure with a middleware, like CORBA, introduce other OBJECT SYSTEM LAYERS. An overall object-based structuring of the system would ease combination with such object systems. The system should benefit from the efficiency of system programming languages [25], like C or C++, but it should also be flexibly adaptable to new requirements with as little effort as possible. Document archiving is a very important part of the information system of several customers. The importance makes several customers suspicious about "new" and unproven technologies. Therefore, the new technologies should be wrapped in the appearance of the well-known and reliable technology.

Context

Object-orientation helps in the design and implementation of complex software systems. Often object-oriented approaches can be used throughout the design, but (parts of) the implementation have to be done in a language that does not support object-orientation, like C or Cobol. Or the implementation has to be done in an object system that does not support "advanced" object-oriented techniques. For instance C++ or Object Cobol do not natively support reflection or interception techniques.

Problem

Suppose you want to design with object-oriented techniques and use the benefits of advanced object-oriented language constructs, but you are faced with target programming languages that are non-object-oriented, or with legacy systems that cannot quickly be rewritten, or with target object systems that are not powerful enough or not properly integrated with other used object systems (e.g. when COM or CORBA is used). But the target language is chosen for important technical or social reasons, such as integrating with legacy software, reusing knowledge of existing workers, and customer demands, so it cannot be changed. One solution is to translate the design into a non-object-oriented design, and then to implement that design. If this mapping is manual, then it will be error-prone and will have to be constantly redone as the requirements change.

Forces

- *Legacy Integration*: Legacy applications are often written in procedural languages, like C or Cobol. A complete migration of a system to an object-oriented language often makes the integration with the existing legacy components difficult and forces a re-implementation of several well functioning parts. The costs of such an evolution (that are very hard to estimate) are often too high to consider the complete migration to object technology at all.

- *Efficiency*: The execution speed of applications or application parts written in efficient system programming languages [25], like C or C++, is typically superior to higher level languages, like scripting languages. Higher level languages, in turn, are easier maintainable, and provide language constructs that ease adaptions. Runtime critical parts should be (or sometimes must be) written in system programming languages.

- *Integration of Several Object Systems*: In order to use third-party software, the object and component concepts of these technologies have to be integrated or adapted. Especially for key technologies, like middleware approaches, databases, transaction monitors, etc. the concepts of several technologies that are used have be integrated. E.g. usually in an enterprise context not all requirements are fulfilled by one middleware product [9]. Distributed object systems, application servers, transaction monitors, relational and object-oriented databases, etc. have to work in concert. For integration one OBJECT SYSTEM LAYER can be chosen by the company (which may be the object system of one of the technologies or a different one).

  Often foreign OBJECT SYSTEM LAYERS are even used in an object-oriented language. E.g. the CORBA or COM object systems are not exactly the same as the object systems of object-oriented languages, like C++ or Java, in which they are used. An OBJECT SYSTEM LAYER can provide an abstraction over the different object systems. But as a liability an additional OBJECT SYSTEM LAYERS also adds complexity to the system and it takes time to design, implement, and maintain it.

- *Adoption of Enhanced Object-Oriented Techniques*: Often an application uses an object system, like C++, Java, or Object Cobol, that only implements standard object-oriented techniques. These do not provide powerful adaptation and interception techniques, reflection and introspection, role concepts, or support for implementation of object-oriented design patterns (as in [19]) and framework parts. In these and similar cases the used object system can be combined with an OBJECT SYSTEM LAYER that implements such techniques.

- *Usage of Object-Oriented Design Concepts with Other Paradigms*: Most object-oriented software systems are not only designed with the object-oriented paradigm, but are combined with multiple other paradigms, like the functional, imperative, or logical. If other paradigms are more suitable for the implementation of a system part or other forces, like legacy integration, impose that parts of the application are implemented in another paradigm, but the main part of the system is object-oriented, the other paradigms have to be integrated with the object-oriented paradigm. Often other paradigms are only mapped onto objects instead of real integration, as for instance when a logical Prolog interpreter is wrapped by an object system.

- *Company's Politics and/or Customer Demands*: Company's politics may restrict developers to a certain set of programming languages, technologies, etc. Customers may demand that certain "new" or unproven technologies should not be used. An OBJECT SYSTEM LAYER that is wrapped behind well-known, reliable, or trusted technologies can be the only way to introduce certain techniques for such "political" reasons.

Solution

Build or use an object system as a language extension in the target language, and then implement the design on top of this OBJECT SYSTEM LAYER. Provide a well-defined interface to components that are non-object-oriented or implemented in other object systems. Make these components accessible through the OBJECT SYSTEM LAYER, and then the components can be treated as black-boxes. The OBJECT SYSTEM LAYER acts as a layer of indirection for applying changes centrally.

Architecture Overview

There are several implementation variants of the OBJECT SYSTEM LAYER pattern. In Figure 2 a conceptual, high-level architecture is shown with architectural collaborators that can similarily be found in the most instantiations of the pattern. However, often OBJECT SYSTEM LAYERS do not have a distinct implementation component, but just rely on a programming/design convention.

- A *Base Language*, like C or C++, is extended with an object system.

- A *Base Language Component* implements a reusable system part as a black-box component in the base language.

- The *Object System Layer* is a special base language component that implements an object system in the base language. It comprises two sorts of objects:

  - *Implementation Objects* implement the object-oriented parts of the application in the OBJECT SYSTEM LAYER.

– *Wrapper Objects* are COMPONENT WRAPPERS that wrap base language components behind an object interface, so that they can be used from within the object system.

The *Object System Layer Implementation* contains the intrinsic implementation of the object system, as for instance object and class implementations, and a MESSAGE REDIRECTOR (see the Design and Implementation section for more details).

- *Outboard Paradigm Wrappers* are base language components that incorporate components implementing outboard paradigms, such as the relational or logical paradigm.
- The *Main Program* may be a shallow interpreter main loop, but it can also be a large application that embeds the object system.

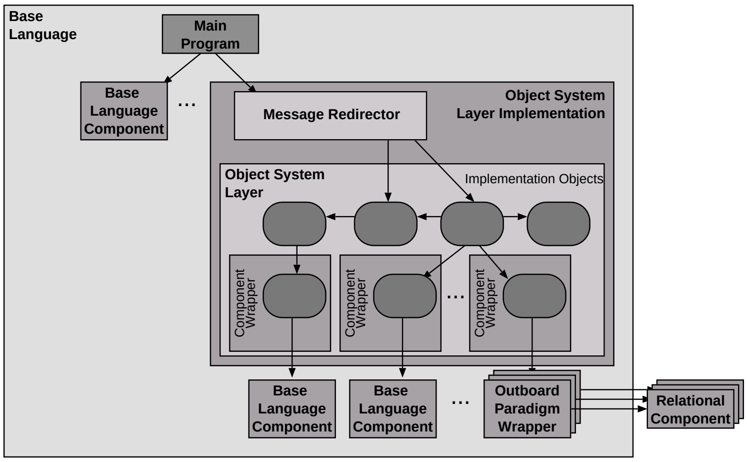

Figure 2. OBJECT SYSTEM LAYER **– Architecture**

Usually, the message dispatch from the using main program (or from a using component) to the implementation objects is handled by the MESSAGE REDIRECTOR pattern [7]. The pattern maps symbolic (i.e. usually string-based) calls to object/method implementations in the OBJECT SYSTEM LAYER. However, the usage of MESSAGE REDIRECTOR is optional, implementations objects can also be called directly.

When several components have to be integrated into the object system, they usually should be integrated in a unique way, despite different paradigms, object systems, programming languages, etc. This problem is resolved by the COMPONENT WRAPPER pattern [8] which can be used as part-of the OBJECT SYSTEM LAYER.

Design and Implementation

Class-Based Design and Implementation

In this section we discuss a simple, hypothetical internal design of an OBJECT SYSTEM LAYER. In the presented small example, as in Figure 3, a dynamic object system with objects, classes, class-objects, inheritance, and object-/class-nesting is built from a set of classes. Such a system can, for instance, be implemented as an extension of an existing scripting language, such as Tcl, or as an extension to an object-oriented language, such as C++.

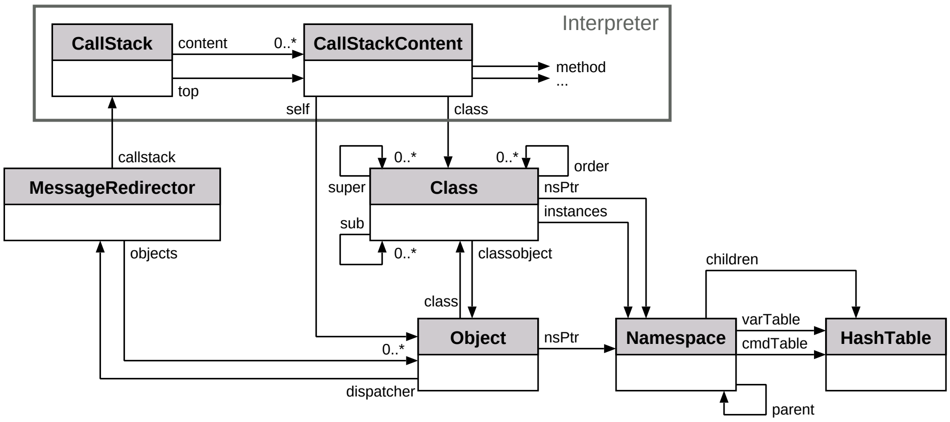

Figure 3. Object System Design (Excerpt)

The `Object` class maintains a reference to its class, and each `Class` knows its class-object. Moreover, classes store their instances in a hash table. Each class and each object in the OBJECT SYSTEM LAYER has a namespace, where it stores its local variables and methods (the object system is object-specific, thus it allows objects to carry object-specific methods). Each method implementation is referenced by a function pointer. The inheritance hierarchy is maintained by a list of super-classes and sub-classes on the `Class` class. The linearized inheritance order is maintained for fast and unambiguous computation of the inheritance hierarchy.

A call stack is stored with an interpreter and maintains the runtime call stack information of the object system. Besides several other information, the call stack stores the current object, class, and method (all can be introspected at runtime). On the foundation of the object system an interpreter evaluates the code call by call. The interpreter part is optional, and often an existing implementation can be reused, because it is already implemented by a scripting language implementation. Each object-oriented message is mapped by a central MESSAGE REDIRECTOR to the actual implementations in the OBJECT SYSTEM LAYER. The mapping of symbolic messages to implementations can be done by the following (simplified) algorithm:

```
int ObjDispatch (objectName, methodName, arguments, ...) {
  if (<callstack->top is not located in inheritance hierarchy>)
    method = <find methodName on current object>;
  if (<method not found>) {
    class = <search class hierarchy for methodName>;
    if (<class found>) method = <get method from class>;
  }
  if (<method found>) {
    result = <call method>;
    if (<result> == ERROR) <handle the error or raise runtime error>;
  } else {
    <raise 'proc not found' error>;
    result = ERROR;
  }
  return result;
}
```

Firstly, we try to find an object-specific method on the current object. If it is not found, we try to find the method in the inheritance hierarchy. The call stack carries the information which method of which class was executed before. From this class we search the linearized class order until we find another method with the given method name. If no method is found, an error is raised. Otherwise the method information is determined, and the method is invoked. The function for method invocation enters the information for the call stack. If the result is an error, it is handled or returned as a runtime error accordingly. Otherwise the result is returned.

This object dispatch mechanisms, implemented in the MESSAGE REDIRECTOR, can be used as a central place to introduce high-level language constructs that rely on message exchanges. E.g. the interception mechanisms filter [19] and per-object mixin [18] of the language XOTCL check at the beginning of the dispatch function for every message whether it has to be intercepted or not. If a filter or per-object mixin is registered, the message is redirected to the interceptor that can handle the message arbitrarily.

Document Archive System Example Resolved

In Figure 4 the architecture of the document archive system with an embedded OBJECT SYSTEM LAYER in form of the scripting language XOTCL is shown. The C/C++ interface libraries of the CORBA ORB implementation, the two databases, and the proprietary jukebox access protocol implementation are mapped to TCL commands in C extensions of TCL. In a set of XOTCL scripts we give these extensions object-oriented WRAPPER FACADES [28] that encapsulate the TCL commands in a set of interacting objects. On top of these scripts, we write various XOTCL scripts which are used in hot spots [26] of the design. The document archive clients and servers themselves are C/C++ components which embed XOTCL as a C library. These components are at the same time able to use C/C++ legacy parts that are not yet (or will never be) migrated to XOTCL scripts.

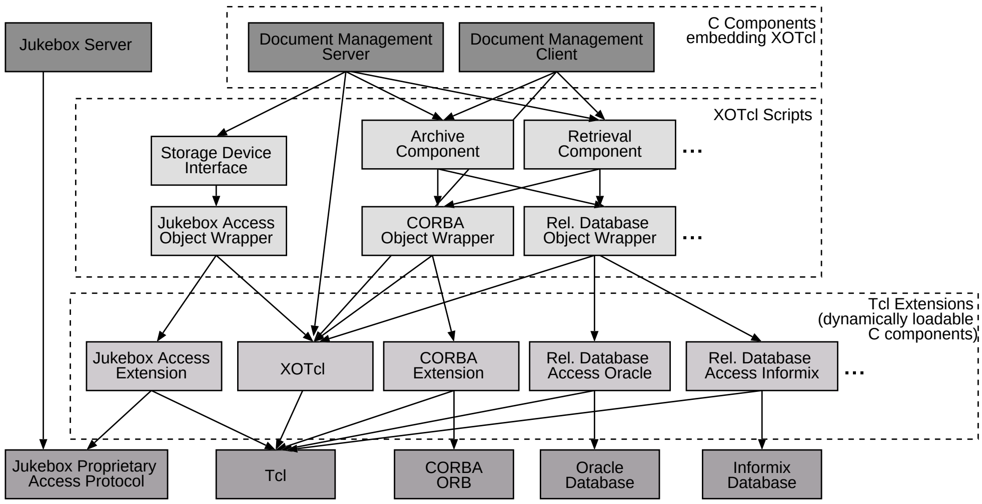

Figure 4. Document Management System with an Embedded OBJECT SYSTEM LAYER **(**XOTCL**)**

There are several advantages of the new architecture. The communication subsystem was incrementally migrated from the proprietary protocol to an implementation based on the standard CORBA. The database connection becomes independent from the used database management system. The OBJECT SYSTEM LAYER can be used as an integration base for various involved technologies, extensions, other OBJECT SYSTEM LAYERS, and for connections to other systems, like B2B systems, SAP R/3, etc. If the system is accordingly structured into components of the OBJECT SYSTEM LAYER, development and deployment processes may be simplified. Storage devices, archive server, and retrieval server are shielded by unique interfaces that stronger decouple clients from their internal implementation. Introduction of new storage technologies or cache strategies can be performed without interfering with clients.

The abstraction into COMPONENT WRAPPERS [8] for DBMS and communication access subsystems allows us to use the adaptive and reflective environment of the scripting language for adaptations to new products or new versions of the products. XOTCL offers us an integrating component concept for the various object systems involved in this application. Frequently changing parts of the system can be kept in scripts, so that they can be rapidly changed, while stable and performance sensitive parts are used as C/C++ components. All clients of the DBMS and communication access subsystems rely on the central COMPONENT WRAPPER abstractions, and changes in the products or to a new product do not affect the clients at all. These changes can be introduced in the wrappers, written in the scripting language, which supports powerful means for adaptation, like the interception mechanisms of XOTCL, and offers powerful string manipulation commands of TCL (e.g. useful for adaptation between the two different SQL dialects).

Disadvantages are sometimes reduced performance, because of interpretation/byte-code compilation, additional indirection, and method lookup. Furthermore, the described techniques forces the development organization to have experts in all the involved languages (here: TCL, XOTCL, C, and C++). This may result in costs for additional training to learn the object-oriented abstractions/language constructs, and/or to learn the conventions how to emulate object-orientation in a non-object-oriented language. Some might see the additional training as a benefit of the pattern.

Consequences

+ The flexibility of the appliction can be dramantically improved, since the OBJECT SYSTEM LAYER allows us to introduce many flexibility hooks and to implement high-level language constructs, such as interception, reflection, and adaptation techniques.

+ "High-level" language constructs, like runtime class creation, meta-classes, dynamic object-class relationships, and dynamic class-class relationships can be "language supported" by the pattern in languages that do not offer such functionality natively.

+ Classes can become runtime entities and can be manipulated with the same ease as objects.

+ The complexity of the application can be dramatically reduced, because techniques, like dynamic classes, per-object mixins, filters, and class objects, enable us to avoid unnecessary class definitions. Thus we avoid an explosion of the number of necessary subclasses.

– Complexity of the application can also rise, if the client has to maintain the OBJECT SYSTEM LAYER. Then issues like garbage collection, object destruction, relationship dynamics, reflection, etc. have to be programmed by hand. But this problem can be avoided by using an existing OBJECT SYSTEM LAYER as a library.

– Performance can be decreased through additional indirections and through flexibility hooks. However, performance problems can be avoided to a high degree by carefully designing the OBJECT SYSTEM LAYER. I.e. offering too many flexibility options that are not required by the application case is a serious design flaw.

– The OBJECT SYSTEM LAYERS conventions and interfaces have to learned by the developers. If the OBJECT SYSTEM LAYERS is a whole scripting language, a new language has to learned.

Pattern Variants

- *Object-Oriented Scripting Language*: Scripting languages are based on a two-level concept of components written in efficient system languages (like C) and components written in the scripting language [25]. The scripts glue various components together and are used in parts of the application that tend to change often. Often they come with language constructs, like dynamics, introspection, etc. that ease the glueing process and adaptations. This pattern variant is normally implemented with an explicit MESSAGE REDIRECTOR that interacts with the interpreter of the base language. There are several different object-oriented scripting languages that mainly enhance existing scripting languages with different forms of object-orientation:
  - XOTCL [21] is a TCL extension and implements powerful message interception techniques, dynamic object aggregations, nested classes, assertions, and several other high-level language constructs on top of the dynamic and introspective environment of OTCL. XOTCL (like all TCL extensions) can be easily embedded in C applications and can therefore also be used as a library implementing an object system for C programs.
  - [incr Tcl] is another object-oriented extension of TCL. It is also implemented in C and can also be used as a object library in C, but it only implements object-orientation in the style of C++.
  - Python is an object-oriented language that implements basic object-oriented means and also provides a C API. Like the TCL variants it is extensible by and embed-able in C/C++ applications.
  - Perl provides an object-oriented encapsulation mechanism as well, but is more famous for its string processing facilities and its operating system interfaces.
- *Library Implementing an Object System*: Non-object-oriented languages can be enhanced by a library that implements an object system on top of the non-object-oriented language. This pattern variant uses various implementation techniques, that are characterized by passing a state of the object system to the functions implementing the object-oriented methods. Two common techniques are:
  - A simple technique is to associate functions with objects by a convention. E.g. the first argument of each method in the object system can be interpreted as the object ID of the current object. Similarly, enhanced features, like inheritance, construction/destruction, or data hiding can be implemented by such conventions. This style requires a certain amount of discipline of all involved programmers.

- In C a function pointer can be associated with a structure representing the object. General classes implement more sophisticated behavior, like object creation/destruction, inheritance, etc. An example implementation may have the following form:

```
typedef struct Class {
  /* classes' state */
  HashTable* instances;
  ...
  /* classes' methods */
  int (*createInstance) (char* name);
  int (*destroyInstance) (char* name);
  int (*superclass) (char* name);
  ...
} Class;
```

  The task of the object system in the library is to define the general classes and/or objects, define the basic behavior of the object system on top of these classes/objects, and give initialization routines for the object system.

- *Enhancement of Object-Oriented System Programming Language*: Often even programming languages that have an object concept lack certain desired features. Then the usage of an OBJECT SYSTEM LAYER can add these feature. E.g. C++ does not offer reflective abilities. The REFLECTION pattern [1] enhances non-reflective languages with reflection. The PROTOTYPE-BASED OBJECT SYSTEM pattern [23] adds dynamic slots and methods to an object system. The TYPE OBJECT pattern [12] adds meta-level objects that contain type information. OBJECT SYSTEM LAYER combines such patterns in a distinct layer that is often loadable as a component. This variant has the drawback that the additional OBJECT SYSTEM LAYER adds complexity to the system and that it takes resources to design, implement, and maintain it.

- *Object System of a Key Technology*: Many systems, designed/implemented in functional or imperative style, have a partly object-oriented structure, simply because they use a certain key technology, like a database or a middleware, that imposes an object-oriented structuring of the system. E.g. if a large C application is combined with CORBA, the developers can adopt an object-oriented structuring (though it is not necessary), if they design the code for implementation of the methods declared in the IDL in an object-oriented way. At least the system gets an object-oriented interface.

Some instances of the pattern are combinations of the variants, e.g. XOTCL [21] is an object-oriented scripting language, a library implementing an object system, and it includes components for integration of key technologies.

Known Uses

Object-oriented scripting languages and libraries implementing an object systems are widely used. Here we just give a few examples of systems that explicitly use them for purposes as described in the Forces section.

- Some known uses of object-oriented scripting languages are:

  - The presented document archive system is a system which was reengineered with an OBJECT SYSTEM LAYER as basis for a piecemeal, component-oriented way of reengineering (see [10] for details).
  - xoComm [20] is a highly flexible and adaptive web server implementation that uses XOTCL to integrate the system's components, to access low-level network functionalities with object-oriented abstractions, and to provide flexibility/extensibility hooks.
  - NeoWebScript [17] is a server-based interactive programming environment for HTML code in web pages. It's interface uses OTcl's object system to allow the user to define classes for generating blocks of hypertext flexibly. A query class is used as an interface to integrate a database backend.
  - The Network Simulator (NS) [30] supports network simulation including TCP, routing, multicast, network emulation, and animation. It allows flexible configuration using scripts in OTcl.
  - The object system in the Graphics Notation (Gn) implementation [15] is implemented using OTcl. Every Gn class is an OTcl class with a number of subcommands implemented with C callbacks (in reusable blackbox components).
  - libsrm [27] is a framework for reliable multicast transport that uses an OTcl API wrapping a C interface for flexible access/configuration of the protocol.
  - Zope [31] is a popular application server that uses Python for integration of the involved paradigms and for flexible/adaptive development for web applications.
  - Caldera, a prominent Linux distribution, is developing the Caldera Open Administration System (COAS) [2], to provide a comprehensive and coherent framework for implementation of system administration mechanisms. COAS provides a plug-in framework for modules which are written in Python or C++ (or both). COAS integrates the Python object system with C++ by an embedded Python interpreter in order to benefit from C++'s efficiency and Python's flexibility at the same time.
  - In [29] the usage of scripting languages for the flight software of the mars pathfinder mission is discussed. The project uses the object-oriented scripting language [incr Tcl].

- Three popular libraries implementing an OBJECT SYSTEM LAYER are:

  - The *X Toolkit* [24] is a C library that handles the management of widgets (graphical objects for development of user interfaces) by exploration of a set of C `structs` and functions which are associated with these `structs` by their first argument. A general (intrinsic) widget management handles widget creation/destruction, setting of attributes, interoperability, etc. This layer is an OBJECT SYSTEM LAYER for the language C. Widget sets, like Motif or Athena, are built on top of the intrinsic library. The X Toolkit abstracts from the low-level details of the underlying XLib (that implements basic windowing functionalities). The Xt intrinsic library distinguishes between objects and classes. New widget classes can be sub-classed from existing widgets. Widgets may have attributes (called resources), associated methods, and can exploit the event system of X11 by means of callback methods. Each widget has an inner state which is readable through its methods.

- *libwww* is the reference implementation of the W3C for common Web protocols, like HTTP/1.1. It is a highly modular, general-purpose web API written in C. It uses similar techniques as the X Toolkit to provide object-oriented abstractions in C. The libwww style guide [22] gives examples how the object-oriented concepts of construction/destruction, data hiding, namespaces, `this` pointer, and inheritance should be emulated in libwww.
- *KA9Q NOS* [13] is a general TCP/IP implementation that is originally designed for packet radio transmition. It uses similar object-oriented techniques as the libwww.

- The integration of various middleware standards, like CORBA or DCOM, with non-object-oriented languages, like C or Tcl, or the object systems imposed by transaction monitors, like Tuxedo, are just a few examples of key technologies implementing an OBJECT SYSTEM LAYER. These technologies are used in countless projects, and they force the applications to use their object systems/models to a certain degree. Often a distinct OBJECT SYSTEM LAYER is used for integration.

Related Patterns

MESSAGE REDIRECTORS [7] are often part of the OBJECT SYSTEM LAYER pattern, since object calls must be mapped from symbolic calls to implementations in the OBJECT SYSTEM LAYERS. Sometimes their functionality is scattered over the code. COMPONENT WRAPPERS [8] are often used in OBJECT SYSTEM LAYERS for component integration.

There is a set of patterns which implement partial OBJECT SYSTEM LAYERS for limited purposes. E.g. the PROTOTYPE-BASED OBJECT SYSTEM pattern [23] implements cloneable objects with slots and methods to introduce modifiability and flexibility for variables and methods into object-oriented languages, like C++ or Java. In the same style reflective abilities are attached to object-oriented languages (with a distinction into meta- and base-level) by the REFLECTION pattern [1]. The style of division into base- and meta-level is similar to the CLOS meta-object protocol [14].

The technique to attach special objects carrying meta-level information to other objects, is explored by David Hay in several data model patterns [11], like PRODUCTS that are embodied in PRODUCT TYPES. Both are classed later on into PRODUCT CATEGORIES. Similar techniques are used in Martin Fowler's Analysis Patterns [4], e.g. for ACCOUNTABILITY and PARTY TYPES (here: the division into two levels is done with an operational and a base level). The TYPE OBJECT pattern [12] generalizes this approach of dividing the object system into implementational base-level and a meta-level that carries meta-information, like type or other reflective information.

OBJECT SYSTEM LAYER induces the usage of several other object-oriented (general-purpose) design patterns that fulfill the integration of different system parts. OBJECT SYSTEM LAYERS provide only a foundation for integration, they do not integrate system parts themselves. WRAPPER FACADES [28] are used to give procedural system parts an object-oriented interface. Since one main purpose of OBJECT SYSTEM LAYERS is to be an indirection layer, patterns can be applied that use the message dispatch within the OBJECT SYSTEM LAYER as a flexibility hook. ADAPTERS [5] are used to flexibly adapt to various different implementations of the same service (e.g. adaption to another database interface in the example). Central FACTORIES [5] abstract over object creation process, and let us introduce changes in creations, like object sharing through FLYWEIGHTS. FACADES [5] are used to shield sub-systems from direct access, thus avoiding strong coupling between clients and sub-system.

See Also

Meyer [16] discusses common techniques for object system implementation in languages without an object system. He compares the languages' abilities to implement the underlying concept of abstract data types. Various languages offer concepts that enable encapsulation of modules, like Ada (package), Modula-2 (module), or CLU (cluster). These modules are free associations of program elements, e.g. for abstract data type implementation. In C `structs` can be used for data structure encapsulation, while a set of embedded function pointers reference the implementation of behavior. Each instance of a class references its type. The type is a special run-time class structure that contains pointers to the method of the class. All these techniques are very low-level implementation techniques and require a lot of efforts in order to keep away from violating the concepts. A library that handles these low-level issues helps to avoid these problems and to automate advanced features, like inheritance, relations, interceptors, etc.

In [6] various design and language concepts are compared regarding the implementation of component concepts, encapsulation aspects, and lifecycle issues. The discussion incorporate imperative languages, like Modula-2 and Ada, functional approaches, like ML and Z, and object-oriented languages, like Smalltalk, Eiffel, and CLOS. In [3] Cox describes the design of the language Objective C and compares to other object-oriented approaches. Furthermore, a discussion of C techniques for implementation of object-oriented concepts can be found.

In [22] the techniques for implementing the concepts of construction/destruction, data hiding, name spaces, `this` pointer, and inheritance in the libwww are discussed. Construction/destruction is emulated in form of two functions `objectName_new` and `objectName_delete`. Object data is protected in libwww by declaring a structure in a header file, but not defining it. Member functions of the class – giving the object a namespace induced by its class – are called explicitly in libwww, e.g. `ClassName_memberFunction`. The `this` pointer is achieved by understanding the first parameter to any method as the object ID. Inheritance is mostly handled through explicit pointer casting and a first element in classes, called `isa`. Similar techniques of the X Toolkit are described in [24].

Acknowledgements

We would like to thank Ralph E. Johnson and Andreas Rüping for their helpful comments during the shepherding process. Additionally we would like to thank the members of the EuroPlop 2000 writers workshop for their constructive feedback.

References

[1] F. Buschmann. Reflection. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, pages 271–294. Addison-Wesley, 1996.

[2] Caldera, Inc. COAS: Caldera open administration system. http://www.coas.org, 2000.

[3] B. Cox and A. Novabilsky. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1990.

[4] M. Fowler. *Analysis Patterns*. Addison-Wesley, 1997.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[6] M. Goedicke. *On the Structure of Software Description Languages: A Component Oriented View*. Habilitation Thesis, Research Report 473/1993, University of Dortmund, 1992.
[7] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. to appear., 2001.
[8] M. Goedicke, G. Neumann, and U. Zdun. A pattern language for introduction of flexiblity into black-box component architectures. to appear., 2001.
[9] M. Goedicke and U. Zdun. A key technology evaluation case study: Applying a new middleware architecture on the enterprise scale. In *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*, Davis, USA, Nov 2000.
[10] M. Goedicke and U. Zdun. Piecemeal migration of a document archive system with an architectural pattern language. In *5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Lisbon, Portugal, Mar 2001.
[11] D. Hay. *Data Model Patterns – Conventions of Thought*. Dorset House Publishing, 1997.
[12] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
[13] P. Karn. The KA9Q NOS TCL/IP package. http://people.qualcomm.com/karn/code/ka9qnos/, 1996.
[14] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
[15] M. D. McCool. Gn – a graphics notation: Introduction, specification, and implementation. http://www.cgl.uwaterloo.ca/ mmccool/gn.HTML/gn.html, 2000.
[16] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
[17] NeoSoft. neoWebScript. http://www.neosoft.com:6969/neowebscript/, 2000.
[18] 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.
[19] 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, May 1999.
[20] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. *World Wide Web Journal*, 3(1), 2000.
[21] G. Neumann and U. Zdun. XOTCL, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, February 2000.
[22] H. F. Nielse. C programming style in libwww. http://www.w3.org/Library/User/Style/, 1998.
[23] J. Noble. Prototype-based object system. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.
[24] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual*. OReilly & Associates, Inc., 1993.
[25] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
[26] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press Books. Addison-Wesley, 1995.
[27] S. Raman and Y. Chawathe. libsrm: A framework for reliable multicast transport. http://www-mash.CS.Berkeley.EDU/mash/software/srm2.0/, 2000.
[28] D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.
[29] D. Smyth. Tcl and concurrent object-oriented flight software: Tcl on mars. In *Proc. of 2nd Tcl/Tk Workshop*, June 1994.
[30] UCB Multicast Network Research Group. Network simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/ns.html, 2000.
[31] Zope: Z object publishing environment. http://www.zope.org, 1999.