ObjectiveView

The Object and Component Journal for Software Professionals

Welcome to issue 3 of ObjectiveView - in this issue:

Object/Component Architecture Series…

For: System Architects, Project Mangers, Technical Staff:

BEA's M3 Object TP Framework

Combining Object and TP technology…

Paul Crerand on the BEA M3 Object Transaction Monitor - Page 2

Development Process Series…

For: Project Mangers and Technical Staff:

Extreme Programming - XP

A lightweight development process…?

Yonat Sharon summarises Kent Beck's Extreme Programming Process - Page 10

Object/Component Architecture Series…

For: System Architects, Software Designers, Software Developers:

Design Patterns in IBM's San Fransisco

Applying design patterns for business benefit…

Brent Carlson on design patterns using the San Fransisco framework - Page 28

Development Process Series…

For: Project Mangers and Technical Staff:

XP - Cutting Through the Hype…

An unacceptable development process ….?

Authors Doug Rosenberg/Kendall Scott's counter to the XP camp- Page 16

Introducing Technology Series…

For: All levels of staff:

Introduction to Use Case Analysis in UML

Capturing requirements using use cases….

Author Robert Martin introduces use case analysis - Page 23

Introducing Technology Series…

For: All levels of staff:

Introduction to Component Based Development

Can components solve all the worlds problems…?

Michael Barnes introduces components and CBD - Page 36

Editorial/Production	Sponsored by	Sponsored by
see www.ratio.co.uk for back copies	Object Mentor www.objectmentor.com	Iconix www.iconixsw.com

Editor: Mark Collins-Cope

Free subscriptions by email: objective.view@ratio.co.uk by fax: (0)181 579 9200

Training/consultancy sales (at Ratio): (0)181 579 7900 - Rennie Garcia

Subjective Views… Object Architecture Series

The BEA M3 Object TP Framework

Paul Crerand of BEA gives an in-depth overview of how the BEA M3 supports CORBA within a transaction processing environment

Introduction

The BEA Systems M3 Object Transaction Manager (OTM) provides one of the key architectural components and operations that give robust scalability and performance to distributed client and server applications within a CORBA object model. Support for EJB Containers is being added for release in the summer of 1999, but is not discussed in this article. The BEA M3 TP Framework is analogous to an EJB Container. The M3 system adds a rich set of convenience objects that make it easy to program the most basic functions of a CORBA application, including:

- Generating and returning object references to the client application
- Instantiating, activating, managing, and deactivating the CORBA objects that implement your application's business logic
- Connecting the CORBA objects managed by your application to CORBA components, such as the Portable Object Adapter (POA)
- Starting up, shutting down, and allocating resources needed by the server application
- Scoping and managing transactional objects and operations
- Implementing a security model
- Scaling the application so that it can accommodate thousands of client applications, hundreds of server processes, and millions of objects

This article discusses these issues.

CORBA Objects

A CORBA object is a virtual entity in the sense that it does not exist on its own, but rather is brought to life when, using the reference to that CORBA object, the client application requests an operation on that object. The reference to the CORBA object is called an object reference. The object reference is the only means by which a CORBA object can be addressed and manipulated in an M3 system

When the client or server application issues a request on an object via an object reference, the M3 server applications instantiate the object specified by the object reference, if the object is not already active in memory. The data that makes up a CORBA object may have its origin as a record in a database. The record in the database is the persistent, or durable, state of the object. This record becomes accessible via a CORBA object in an M3 domain when the following sequence has occurred:

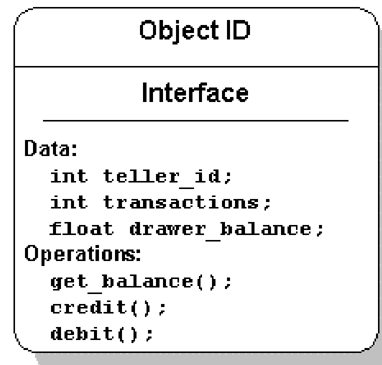
1.The server application's factory creates a reference for the object. The object reference includes information about how to locate the record in the database.

2.Using the object reference created by the factory, the client application issues a request on the object.

3.The M3 server application invokes the Server::create_servant() operation, which exists in the Server object. This causes the object's operations to be read into memory.

4.The M3 domain invokes the activate_object() operation on the object, which causes the record containing state to be read into memory.

Components of a CORBA Object



The Object ID

The object ID (OID) associates an object with its state, such as a database record, and identifies the instance of the object. When a factory creates an object reference, the factory assigns an OID that may be based on parameters that are passed to the factory in the request for the object reference.

The Object Interface

The object's interface, described in the application's OMG IDL file, identifies the set of data and operations that can be performed on an object. One distinguishing characteristic of a CORBA object is the run-time separation of the interface definition from its data and operations. In a CORBA system, a CORBA object's interface definition may exist in a component called the Interface Repository. The data and operations are specified by the interface definition, but the data and operations exist in the server application process when the object is activated.

The Object's Data

The object's data includes all of the information that is specific to an object class or an object instance. You can encapsulate the object's data in an efficient way, such as by combining the object's data in a structure to which you can get access by means of an attribute. Attributes are a conventional way to differentiate the object's data from its operations.

The Object's Operations

The object's operations are the set of routines that can perform work using the object data. In a CORBA system, the body of code you write for an object's operations is sometimes called the object implementation. When you create an M3 client/server application, one of the steps you take is to compile the application's OMG IDL file. The OMG IDL file contains statements that describe the application's interfaces and the operations that can be performed on those interfaces.

One of the several files produced by the IDL compiler is the implementation file. The implementation file is where you write the code that implements an object; that is, this file contains the business logic of the operations for a given interface. The M3 system implements an interface as a CORBA object. This is where the notion of a servant comes in. A servant is an instance of the object class; that is, a servant is an instance of the method code your wrote for each operation in the implementation file.

When the M3 client and server applications are running, and a client request arrives in the server application for an object that is not active, that is, the object is not in memory, the following events occur:

1.The M3 system invokes the Server::create_servant() operation on the Server object, if no servant is currently available for the needed object. The code that you write for the Server::create_servant() operation instantiates the servant needed for the request. Your code can use the interface name, which is passed as a parameter to the Server::create_servant() operation, to determine the type of servant that the M3 domain creates. The servant that the M3 domain creates is a specific servant object instance (it is not a CORBA object), and this servant contains an executable version of the operations you wrote earlier that implement the CORBA object needed for the request.

2.The M3 domain passes control to the servant, and optionally invokes the servant's activate_object() operation. Invoking the activate_object() operation gives life to the CORBA object, as follows:

a. You write the code for the activate_object() operation. The parameter to the activate_object() operation is the string value of the object ID for the object to be activated.
b. You initialize the CORBA object's data, which may involve reading state data from durable storage, such as from a record in a database.
c. The servant's operations become bound to the data, and the combination of those operations and the data establish the activated CORBA object.

Note: A servant is not a CORBA object. In fact, the servant is represented as a language object. The server performs operations on an object via its servant.

How Object Invocation Works

Since CORBA objects are meant to function in a distributed environment, OMG has defined an architecture for how object invocation works. A CORBA object can be invoked in one of two ways:

- By means of generated client stubs and skeletons, referred to as either static or stub-style invocation
- By means of the dynamic invocation interface, referred to as dynamic invocation, or the DII

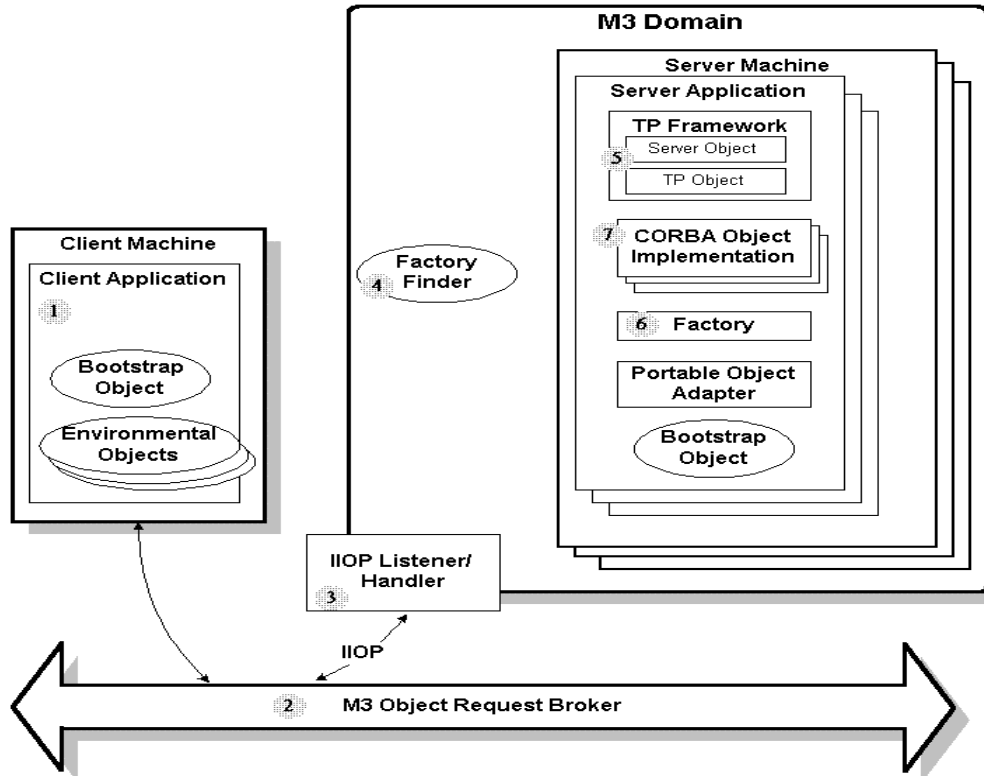
BEA M3 supports both the static and the dynamic invocation interface. This section only describes stub-style invocation, which is simpler to use than dynamic invocation.

When you compile your application's OMG IDL file, one file that the compiler generates is a source file called the client stub. The client stub maps OMG IDL operation definitions for an object type to the operations in the server application that the M3 system invokes to satisfy a request. The client stub contains code generated during the client application build process that is used in sending the request to the server application. Another file produced by the IDL compiler is the skeleton. The skeleton contains code used for operation invocations on each interface specified in the OMG IDL file. The skeleton is a map that points to the appropriate code in the CORBA object implementation that can satisfy the client request. The skeleton is connected to both the object implementation and the M3 Object Request Broker.

When a client application sends a request, the request is implemented as an operation on the client stub. When the client stub receives the request, the client stub sends the request to the ORB, which then sends the request through the M3 system to the skeleton.

The Basic Invocation Process

The following figure shows a high-level view of the M3 system architecture. The numbered call-outs in this figure match the numbered descriptions that follow.



Run Time Components in an M3 Application

1: Client Application

The invocation process starts in the client application. An end user gives input to the interface of an application. The client application translates the user input into a request for a service. A request contains all the parameters and data needed for the server application to perform the service. The M3 system supports multiple client application types including CORBA C++, CORBA Java and ActiveX (via the BEA M3 Application Builder COM-CORBA bridge tool-set, supplied with the M3 product), and also the BEA WebLogic Java Application Server.

Note the following components shown in the above Figure for the client machine:

- Bootstrap object. This object establishes communication between a client application and an M3 domain. It also has object references to the other environmental objects in the M3 domain.
- Environmental object. The client machine, and the M3 domain, have a number of environmental objects, including:
  - SecurityCurrent, which provides a means to perform client application authentication and authorisation so that the client can access the objects it is authorised for.
  - TransactionCurrent, which connects the client application to the M3 transaction subsystem, wherein the client can perform operations within the context of a transaction.

2: M3 Object Request Broker

A client request then goes to the M3 Object Request Broker (ORB), which is the primary mechanism that the M3 system uses to enable the separation of client and server applications. The ORB is responsible for all the mechanisms required to find the implementation that can satisfy the request, to prepare an object's implementation to receive the request, and to communicate the data that makes up the request.

When the client application sends a request to the M3 domain, the ORB performs several functions, such as validating each request and its arguments to ensure that the client application supplied all the required arguments. The POA prepares an object's implementation to receive the request and communicates the data in the request. In this way, the ORB provides a concept called location transparency, meaning that the client does not need to know where the target object is located nor anything about the underlying platform on which the object exists.

Typically, the ORB on the client machine writes the data associated with the request into a standard form. The M3 ORB receives this data and converts it into the format appropriate for the machine on which the M3 server application is running. When the server application sends data back to the client application, the M3 ORB marshals the data back in to its standard form and sends it back to the client ORB. The ORB, which spans the client and the server machines, performs the data marshalling.

3: IIOP Listener/Handler

The IIOP Listener/Handler is a process that receives the client request, sent using the Internet Inter-ORB Protocol (IIOP), and delivers that request to the appropriate server application. The IIOP Listener/Handler serves as a communications concentrator, providing a critical scalability feature. The IIOP Listener/Handler removes the burden of maintaining client connections from the server application. As the number of client applications increases dramatically, the server application still needs to maintain only one connection.

4: FactoryFinder Object

M3 client applications get object references to CORBA objects from factories in the M3 domain. To use factories, the client application must be able to locate the factories it needs. The M3 system provides the FactoryFinder object for this purpose. When server applications register their factories with the FactoryFinder object, any client application connected to the M3 domain can locate that factory via the FactoryFinder object. Client applications get an object reference to the FactoryFinder object via the local Bootstrap object.

5: M3 TP Framework

When the client request arrives in the server application, the TP Framework takes over to ensure that the execution of the request takes place in a co-ordinated, predictable manner. The TP Framework is not a single object, but is rather a collection of objects that work together to manage the CORBA objects that contain and implement your data and business logic. If a client request arrives that requires an object that is not currently active and in-memory in the server application, the TP Framework co-ordinates all the operations that are required to instantiate the object.

6: Factory

When the client application needs an initial object reference to a CORBA object managed by your server application, the client application typically gets that object reference from an object known as a factory. A factory is any user-written CORBA object that, as one of its operations, creates object references to other CORBA objects. As a server application programmer, you use factories as the primary means by which clients can get access to the CORBA objects implemented in your server applications.

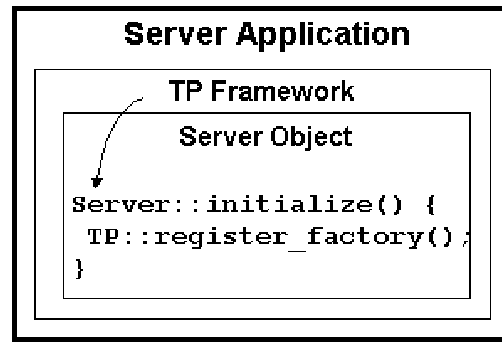
7: CORBA Object Implementation

The final destination of a client request is the CORBA object implementation, which contains the code and data needed to satisfy the client request. The CORBA objects for which you have defined interfaces in OMG IDL contain the business logic and data for your M3 client and server applications. All client requests involve invoking an operation on the CORBA object. The instantiation and management of the CORBA object is handled by the server application.

The Object Activation Process

Step 1: The server application is initialised.

The TP Framework invokes the Server::initialize() operation in the Server object to initialise the server application.



Note: The code lines shown in the figures presented in this section are pseudo-code only.

During the initialisation process, the Server object does the following:

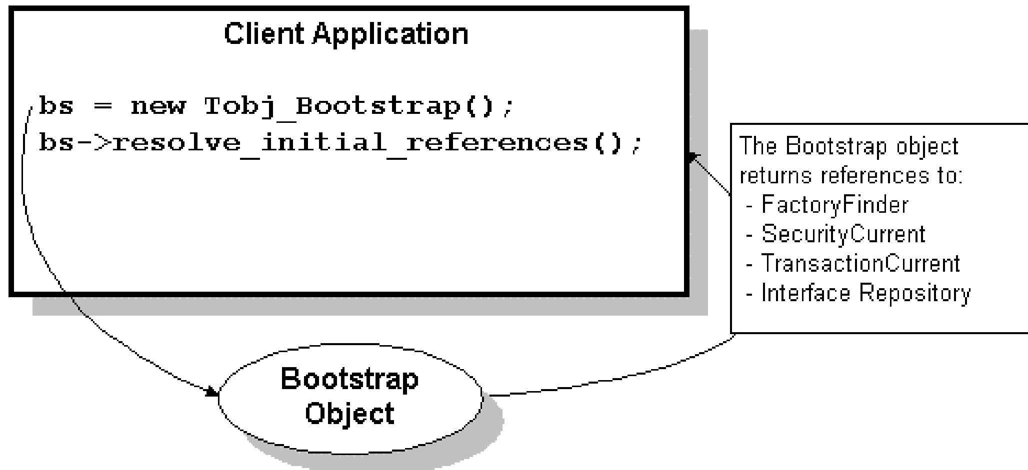
1. Gets the Bootstrap object and a reference to the FactoryFinder object.
2. Registers any factories with the FactoryFinder object. Note that there may be more than one factory.
3. Optionally get an object reference to the ORB.
4. Performs any process-wide initialization.

The Server object is the only user-written component of the TP Framework.

Step 2: The client application is initialised.

This is the process in which the client application uses its local Bootstrap object to obtain initial references to the other objects it needs.

Visit www.ratio.co.uk for links on object technology, additional articles and back copies of ObjectiveView (issues 1 and 2)



Step 3: The client application authenticates itself to the M3 domain.

After the client application has obtained initial references, it is typically ready to send a request to the M3 domain. If the M3 domain has a security model in effect, the client application needs to authenticate itself to the M3 domain before it can invoke any operations in the server application. To authenticate itself to the M3 domain, the client application:

1.Uses the Bootstrap object to obtain a reference to the SecurityCurrent object.
2.Invokes the logon operation of the PrincipalAuthenticator object, which is part of the SecurityCurrent object.

Step 4: The client application obtains a reference to the object needed to execute its business logic.

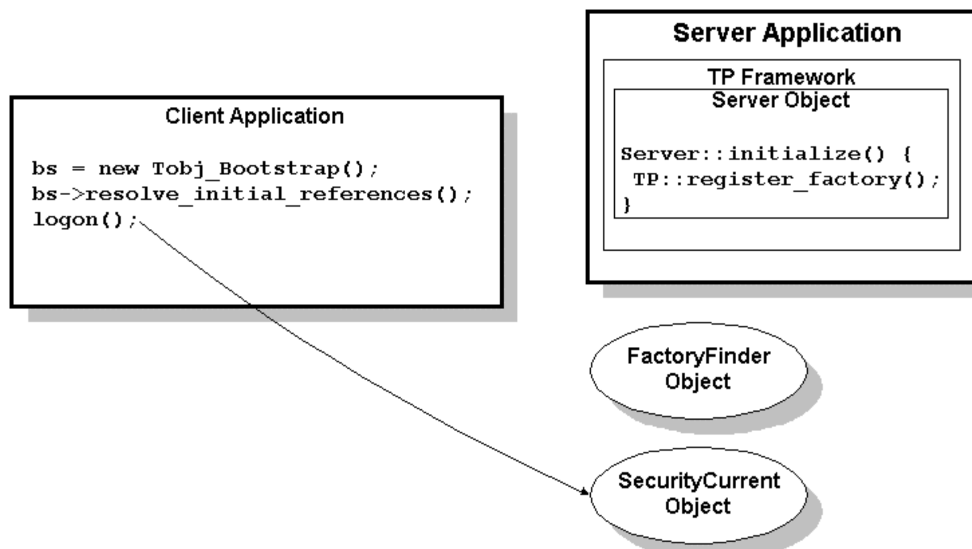
After the client application has authenticated itself to the M3 domain, the client application needs to obtain a reference to the object in the server application it needs so that the client application can execute its business logic.

Step 5: The client application invokes an operation on the CORBA object.

Using the reference to the CORBA object that the factory has returned to the client application, the client application invokes an operation on the object.

The M3 TP Framework

With conventional CORBA ORBs, implementing a server application requires you to write code that:



- Manages objects, bringing them into memory when needed, flushing them from memory when they are no longer needed, managing reading and writing of data for persistent objects, and managing events associated with the objects
- Initializes the server application, connects the server application to all of the resources required by the application, and shuts down the server application when the server application terminates
- Activates and deactivates implementations of objects
- Implements the server application's main dispatching loop

All of this code is in addition to the code that implements the application's business logic.

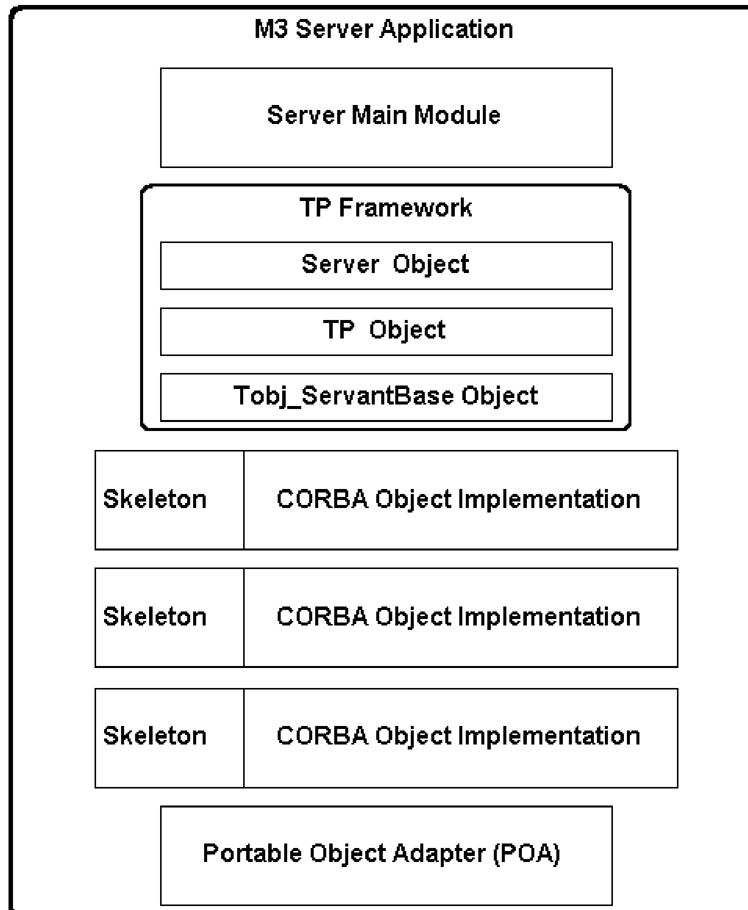
The TP Framework performs all of the functions described in the preceding section, and simplifies the following tasks:

- Initializing the server application and executing startup and shutdown routines
- Creating object references
- Registering and unregistering object factories
- Managing objects and object state
- Tying your server application to M3 resources, such as the Bootstrap object, the TransactionCurrent object, the transaction coordinator, and the FactoryFinder object.
- Getting and initializing the M3 ORB
- Performing object housekeeping

The M3 TP Framework defines a set of objects suited for building robust transaction processing applications using CORBA. The TP Framework sits between the Portable Object Adapter (POA) and the M3 server application, and the TP Framework integrates transactions and state management into the M3 server application. The TP Framework is intended to make the M3 server application programmer's life easier. It does so by providing a simple, easy-to-use application interface, and by automating the other tasks that you must deal with in building the server application. You are responsible only for plugging in your business logic and overriding default actions provided by the TP Framework.

Structure of the TP Framework

The figure below shows a basic view of the components in an M3 server application. Note the TP Framework, which is shown as a part of the server application.



The TP Framework plays a central role in the execution of the server application. The TP Framework has operations in it that the Server object, which you write, can invoke. The TP Framework also provides a number of services automatically to both the server application and the M3 system that are not visible to either the Server object, the servants, or the client application. The TP Framework consists logically of three parts:

- The main program
- A set of convenience objects which provide services
- An additional set of internal objects, not visible to the application programmer, that interact with the ORB, the POA, and the M3 system to invoke operations on your objects as needed and in the proper sequence.

TP Framework Main Program

The TP Framework main program provides a number of services to the server application including:

- Starting the server application.
- Connecting the ORB and the POA to the server application.
- Invoking the Server::initialize() operation in the Server object.
- Notifying objects when events such as server shutdown occur.
- Shutting down the server application, including invoking the Server object's Server::release() operation, and performing basic object housekeeping tasks.

TP Framework Convenience Objects

Server

The server provides the following operations that the TP Framework main program invokes at appropriate times during the execution of the server application:

Server::create_servant()
Server::initialize()
Server::release()

TP

The TP object is provided for you and provides a set of operations that servants can use during the execution of the application's business logic for performing tasks such as creating object references, registering and unregistering factories, and other operations, including:

TP::bootstrap()
TP::create_object_reference()
TP::deactivateEnable()
TP::get_object_reference()
TP::orb()
TP::register_factory()
TP::unregister_factory()
TP::userlog()

Tobj_ServantBase

This is a class from which your servants inherit. The Tobj_ServantBase object provides the following two virtual operations that you can override as part of your servant definition:

activate_object()
deactivate_object()

The Tobj_ServantBase object is provided for you. The TP Framework invokes these operations at appropriate times when processing client requests.

Additional Objects

The TP Framework has a set of internal objects that interact with the POA when a client request is received. These objects invoke operations on the Server object and in your servant (specifically, the activate_object() and deactivate_object() operations) at the appropriate times. These objects also deal with some of the details of transactions, and are not visible to the M3 application programmer.

Summary

In order for distributed object systems to deliver the promise they hold for enterprise computing, it is imperative that they be married with the benefits of transaction processing technology, the production-proven heart of today's mission-critical enterprise systems. When distributed objects and transaction processing technology is brought together, a new class of middleware, called Object Transaction Monitors (OTMs) is born. It is in the world of OTM technology that distributed object systems truly come into their own. BEA's M3 is such a product, formed from the integration of BEA's TUXEDO (The market leading TP Manager) with BEA's ObjectBroker (One of the two founding CORBA ORBs). In the world of enterprise systems, where it can take at least five years for systems to become production-proven, this facet delivers a real business advantage.

BEA M3 was first released on controlled availability in October 1997, and then on general availability in July 1998, and is the first enterprise class OTM on the market. The current release is M3 v2.2.

BEA M3 is a multi-platform, CORBA 2.2 compliant, POA based OTM, with both C++ and Java language bindings utilising IORs, and supports IIOP for connectivity from other CORBA ORBs as well as COM and Web-based environments. With its TP Framework and environmental objects enabling ease of development, especially in the areas of state management and transaction management, M3 provides a CORBA environment that is analogous to EJBs, support for which will be in M3 v3.0 scheduled for release summer 1999. An SNMP Agent is also available to enable application management from enterprise frameworks such as OpenView, TME and UniCenter etc. Integration with modelling

environments such as Rational Rose is also available from BEA, who are a RoseLink Partner. Support for M3 is also provided from many other ISVs including Ardent, Dynasty, Informix, Prolifics, RogueWave, Seague, Select, and Symantec for example.

With its support for CORBA, Enterprise Java Beans and TUXEDO services, all with COM clients, BEA M3 is a platform for deploying component based applications which are reliable, available, scalable and manageable.

Paul Crerand can be contacted on 'paul.crerand@beasys.com'.

Join BEA, Sun and Symantec at their "Spotlight on Java - Piecing together business & technology" seminar series taking place around Europe in 1999. Puzzled by how Enterprise JavaBeans (EJB) can sharpen your competitive edge? Or having realised that EJB is the right choice, do you need to know how to build powerful EJB Web-based Applications? The Java Platform - Piecing Together Business and Technology is a two-track event aimed at ensuring that you are well-positioned to take advantage of the latest EJB technology. For more details visit: http://www.beasys.com.

TOOLS EUROPE '99

"Objects, Components, Agents"

Palais des Congrès • Nancy, France

June 7-10, 1999

29th International Conference & Exhibition

www.tools.com/europe

Conference Chair: Jean-Pierre Finance, Henri Poincaré University, France
Programmme Chair: Richard Mitchell, University of Brighton, UK
Tutorials Chair: Alan Cameron Wills, TriReme International Ltd, UK
Workshops & Panels Chair: Jan Bosch, University of Karlskrona/Ronneby, Sweden
Conference Series Chair: Bertrand Meyer, Interactive Software Engineering, USA

Keynote Presentations by…

Jim Coplien, Bell Laboratories, USA
Erich Gamma, Object Technology International, Switzerland
Jean-Paul Figer, CAP Gemini, France
Bertrand Meyer, Interactive Software Engineering, USA
Trygve Reenskaug, Numerica Taskon AS, Norway

TOOLS CONFERENCES

Technology of Object-Oriented Languages and Systems

Object Development Process Series … Head to Head on XP

Extreme Programming

A Lightweight OO Development Process

Yonat Sharon summarises Kent Beck's Extreme Programming,- using his own words

Introduction

Extreme Programming (XP) is the name that Kent Beck (kentbeck@csi.com) has given to a lightweight development process he has been evolving over the years. This article contains excerpts from many of his posts to otug@rational.com (the object technology email discussion group)

Motivation for Extreme Programming

I observed that people didn't enjoy, and didn't actually use the feedback mechanisms that they read about-synchronized documentation, big testing processes administered by a separate group, extensive and fixed requirements. So I decided to look for feedback mechanisms that

- people enjoyed, so they would be likely to adopt them,
- had short-term and long-term benefits, so people would tend to stick to them even under pressure,
- would be executable by programmers with ordinary skills, so my potential audience was as large as possible and,
- had good synergistic effects, so we can pay the cost of the fewest possible loops

Enough philosophy, here are the feedback loops, how they slow the process, their short and long term value, and their most important synergies:

The Planning Game

Collecting User Stories

Before you code, you play the planning game. The requirements are in the form of User Stories, which you can think of as just enough of a use case to estimate from and set priorities from. My experience of customers using stories is that they love them. They can clearly see the tradeoffs they have available, and they understand what choices they can and can't make.

Each story translates into one or more functional test cases, which you review with the customer at the end of the iteration that delivers the story [An iteration is 1-4 weeks worth of stories]. The test cases can be written in any of a number of forms that are easily readable (and if you're smart, easily writeable) by the customer-directly reading spreadsheets, using a parser generator to create a special purpose language, writing an even simpler language that translates directly into test-related objects.

[...]

My experience with the Planning Game is that it works wonderfully at conceptualization. You get 50-100 cards on the table and the customers can see the entire system at a glance. They can see it from many different perspectives just by moving the cards around. As soon as you have story estimates and a project speed, the customers can make tradeoffs about what to do early and what to do late and how various proposed releases relate to concrete dates. And the time and money required to get stories and estimates is miniscule compared to what will eventually be spent on the system.

The stories are written by the customers with feedback from the programmers, so they are automatically in "business language".

Estimation

The strategy of estimation is:

- Be concrete. If you don't know anything about a story or task, go write enough code so you know something about the story or task.
- If you can, compare a task or story to something that has gone before, that is concrete, also. But don't commit to anything on speculation. [...]
- No imposed estimates. Whoever is responsible for a story or task gets to estimate. If the customer doesn't like the estimate, they can change the story. The team is responsible for delivering stories, so the team does collective estimates (everybody) estimates some stories, but they switch around pairs as they explore so everybody knows a little of everything about what is being estimated). Estimates for the tasks in the iteration plan are only done after folks have signed up for the tasks.
- Feedback. Always compare actuals and estimates. Otherwise you won't get any better. This is tricky, because you can't punish someone if they really miss an estimate. If they ask for help as soon as they know they are in trouble, and they show they are learning, as a coach you have to pat them on the back.
- Re-estimation. You periodically re-estimate all the stories left in the current release, which gives you quick feedback on your original estimates and gives Business better data on which to base their decisions.

Scheduling

There are two levels of scheduling in XP:

- The commitment schedule is the smallest, most valuable bundle of stories that makes business sense. These are chosen from the pile of all the stories the customer has written, after the stories have been estimated by the programmers and the team has measured their overall productivity. So, we might have stories for a word processor:
  - Basic word processing - 4
  - Paragraph styles - 2
  - Printing - 4
  - Spell checking - 2
  - Outliner - 3
  - Inline drawing - 6

(the real stories would be accompanied by a couple of sentences). The estimates are assigned by the programmers, either through prototyping or by analogy with previous stories.

Before you begin production development, you might spend 10-20% of the expected time to first release coming up with the stories, estimates, and measurement of team speed. (While you prototype, you measure the ratio of your estimates to make each prototype to the calendar- that gives you the speed). So, let's say the team measured its ratio of ideal time to calendar time at 3, and there are 4 programmers on the team. That means that each week they can produce 4/3 ideal weeks per calendar week. With three week iterations, they can produce 4 units of stories per iteration.

If the customer has to have all the features above, you just hold your nose and do the math- 21 ideal weeks @ 4 ideal weeks/iteration = 5 1/4 iterations or 16 calendar weeks. "It can't be four months, we have to be done with engineering in two months."

Okay, we can do it that way, too. Two months, call it three iterations, gives the customer a budget of 12 ideal weeks. Which 12 weeks worth of stories do they want?
[...]
XP quickly puts the most valuable stories into production, then follows up with releases as frequent as deployment economics allow. So, you can give an answer to the question "How long will all of this take," but if the answer is more than a few months out, you know the requirements will change.

[...]
To estimate, the customers have to be confident that they have more than enough stories for the first release, and that they have covered the most valuable stories, and the programmers have to have concrete experience with the stories so they can estimate with confidence.

Planning is continuous

XP discards the notion of complete plans you can stick with. The best you can hope for is that everybody communicates everything they know at the moment, and when the situation changes, everybody reacts in the best possible way. Everybody believes in the original commitment schedule- the customers believe that it contains the most valuable stories, the programmers believe that working at their best they can make the estimates stick. But the plan is bound to change. Expect change, deal with change openly, embrace change.

For more information on the planning game:
[http://c2.com/cgi/wiki?ExtremePlanning
http://c2.com/cgi/wiki?PlanningGame
http://c2.com/cgi/wiki?UserStory]

Requirements

Dealing with changing requirements

Where I live the customers don't know what they want, they specify mutually exclusive requirements, they change their minds as soon as they see the first system, they argue among themselves about they mean and what is most important. Where I live technology is constantly changing so what is the best design for a system is never obvious a priori, and it is certain to change over time.

One solution to this situation is to use a detailed requirements process to create a document that doesn't have these nasty properties, and to get the customer to sign the document so they can't complain when the system comes out. Then produce a detailed design describing how the system will be built.

Another solution is to accept that the requirements will change weekly (in some places daily), and to build a process that can accommodate that rate of change and still be predictable, low risk, and fun to execute. The process also has to be able to rapidly evolve the design of the system in any direction it wants to go. You can't be surprised by the direction of the design, because you didn't expect any particular direction in the first place.

The latter solution works much better for me than the former.
[...]

[XP says] to thoroughly, completely discard the notion of complete up-front requirements gathering. Pete McBreen <mcbreenp@cadvision.com> made the insightful observation that the amount of requirements definition you need in order to estimate and set priorities is far, far less than what you need to code. XP requirements gathering is complete in the sense that you look at everything the customer knows the system will need to do, but each requirement is only examined deeply enough to make confident guesses about level of effort. Sometimes this goes right through to implementing it, but as the team's skill at estimating grows, they can make excellent estimates on sketchy information.

The advantage of this approach is that it dramatically reduces the business risk. If you can reduce the interval where you are guessing about what you can make the system do and what will be valuable about it, you are

exposed to fewer outside events invalidating the whole premise of the system.
[...]

So, what if I as a business person had to choose between two development styles?

- We will come back in 18 months with a complete, concise, consistent description of the software needed for the brokerage. Then we can tell you exactly what we think it will take to implement these requirements.
- We will implement the most pressing business needs in the first four months. During that time you will be able to change the direction of development radically every three weeks. Then we will split the team into two and tackle the next two most pressing problems, still with steering every three weeks.

The second style provides many more options, more places to add business value, and simultaneously reduces the risk that no software will get done at all.

Documenting Requirements

You could never trust the developers to correctly remember the requirements. That's why you insist on writing them on little scraps of paper (index cards). At the beginning of an iteration, each of the stories for that iteration has to turn into something a programmer can implement from.

This is all that is really necessary, although most people express their fears by elaborating the cards into a database (Notes or Access), Excel, or some damned project management program. Using cards as the archival form of the requirements falls under the heading of "this is so simply you won't believe it could possibly work". But it is so wonderful to make a review with upper management and just show them the cards. "Remember two months ago when you were here? Here is the stack we had done then. Here is the stack we had to do before the release. In the last two months here is what we have done."

The managers just can't resist. They pick up the cards, leaf through them, maybe ask a few questions (sometimes disturbingly insightful questions), and nod sagely. And it's no more misleading (and probably a lot less misleading) than showing them a Pert chart.

Architecture (System Metaphors)

The degree to which XP does big, overall design is in choosing an overall metaphor or set of metaphors for the operation of the system. For example, the C3 project works on the metaphor of manufacturing- time and money parts come into the system, are placed in bins, are read by stations, transformed, and placed in other bins. Another example is the LifeTech system, an insurance contract management system. It is interesting because it overlays complementary metaphors- double entry bookkeeping for recording events, versioned business objects, and an overall task/tool metaphor for traceability of changes to business objects.

The system is built around one or a small set of cooperating metaphors, from which class, method, variables, and basic responsibilities are derived. You can't just go off inventing names on your own. The short term benefit is that everyone is confident that they understand the first things to be done. The long term benefit is that there is a force that tends to unify the design, and to make the system easier for new team members to understand. The metaphor keeps the team confident in Simple Design, because how the design should be extended next is usually clear.

More information:
[http://c2.com/cgi/wiki?SystemMetaphor]

Communicating the overall design of the system comes from:

- listening to a CRC overview of how the metaphor translates into objects
- pair programming new member with experienced member
- reading test cases
- reading code

I arrived at "system metaphor" as the necessary and sufficient amount of overall design after having seen too many systems with beautiful architectural diagrams but no real unifying concept. This led me to conclude that what people interpret as architecture, sub-system decomposition, and overall design were not sufficient to keep developers aligned. Choosing a metaphor can be done fairly quickly (a few weeks of active exploration should suffice), and admits to evolution as the programmers and the customers learn.

Design

Simple Design (or Precisely Enough Design)-

The right design for the system at any moment is the design that
1) runs all the tests,
2) says everything worth saying once,
3) says everything only once,
4) within these constraints contains the fewest possible classes and methods.

You can't leave the code until it is in this state. That is, you take away everything you can until you can't take away anything more without violating one of the first three constraints. In the short term, simple design helps by making sure that the programmers grapples the most pressing problem. In the long run, simple design ensures that there is less to communicate, less to test, less to refactor.

More information:
[http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork]

Countering the Hacking Argument

During iteration planning, all the work for the iteration is broken down into tasks. Programmers sign up for tasks, then estimate them in ideal programming days. Tasks more than 3-4 days are further subdivided, because otherwise the estimates are too risky.

So, in implementing a task, the programmer just starts coding without aforethought. Well, not quite. First, the programmer finds a partner.They may have a particular partner in mind, because of specialized knowledge or the need for training, or they might just shout out "who has a couple of hours?"

Now they can jump right in without aforethought. Not quite. First they have to discuss the task with the customer. They might pull out CRC cards, or ask for functional test cases, or supporting documentation.

Okay, discussion is over, now the "no aforethought" can begin. But first, they have to write the first test case. Of course, in writing the first test case, they have to precisely explore both the expected behavior of the test case, and how it will be reflected in the code. Is it a new method? A new object? How does it fit with the other objects and messages in the system? Should the existing interfaces be refactored so the new interface fits in more smoothly, symmetrically, communicatively? If so, they refactor first. Then they write the test case. And run it, just in case.

Refactoring

Now it's hacking time. But first a little review of the existing implementation. The partners imagine the implementation of the new test case. If it is ugly or involves duplicate code, the try to imagine how to refactor the existing implementation so the new implementation fits in nicely. If they can imagine such a refactoring, they do it.

You can't just leave duplicate or uncommunicative code around. The short term value is that you program faster, and you feel better because your understanding and the system as seldom far out of sync. The long term value is that reusable components emerge from this process, further speeding development. Refactoring makes good the bet of Simple Design.

More information: [http://c2.com/cgi/wiki?ReFactor & http://c2.com/cgi/wiki?RefactorMercilessly]

Coding

The Hacking Phase

Okay, now it really is hacking time. They make the test case run. No more of this aforethought business. Hack hack hack. This phase usually takes 1-5 minutes. While they are slinging code around with wild abandon, they may imagine new test cases, or possible refactorings. If so, they note them on a to do list.

Now they reflect on what they did. If they discovered a simplification while they were "no aforethoughting", they refactor.

This process continues until all the test cases they can imagine all run. Then they integrate their changes with the rest of the system and run the global unit test suite. When it runs at 100% (usually the first time), they release their changes. I never think more, both before and after coding, than when I am in the flow of this process. So I'm not worried about accusations that XP involves not thinking. It certainly isn't true from my perspective. And people who have actually tried it agree.

Collective Code Ownership

If you run across some code that could be improved, you have to stop and improve it, no matter what. The short term benefit is that your code is cleaner. The long term benefit is that the whole system gets better all the time, and everyone tends to be familiar with most of the system. Collective Code Ownership makes refactoring work better by exposing the team to more opportunities for big refactorings.

More information:
[http://c2.com/cgi/wiki?CollectiveCodeOwnership]

Coding Standards

You can write code any way you want, just not on the team. Everybody chooses class names and variable names in the same style. They format code in exactly the same way. There isn't a short term benefit to coding standards that I can think of. Longer term, coding standards that are chosen for communication help new people learn the system, and as the standards become habit, they improve productivity. Pair programming works much more smoothly with coding standards, as do collective code ownership and refactoring.

More information:
[http://c2.com/cgi/wiki?FormalStandards]

Pair Programming

This is the master feedback loop that ensures that all the other feedback loops stay in place. Any myopic manager can tell you with certainty that pair programming must be slower. Over the course of days and weeks, however, the effects of pairing dramatically reduce the overall project risk. And it is just plain fun to have someone to talk to. The pairs shift around a lot (two, three, four times a day), so any important information is soon known by everyone on the team.

[http://c2.com/cgi/wiki?ProgrammingInPairs]

Integration & Testing

Unit testing-

You can't release until the unit tests are 100%. The short term value is that you program faster over hour

and greater time scales, the code is higher quality, and there is much less stress. Over the longer term, the unit tests catch integration errors and regressions, and they communicate the intent of the design independent of implementation details. The unit tests enable refactoring, they drive the simple design, they make collective code ownership safe, and act as a conversation piece enhance pair programming.

More information: [http://c2.com/cgi/wiki?UnitTests]

You can't have certainty. Is there a reasonable amount of work that you can do that lets you act like you have certainty? For the code I write, the answer is yes. I write a unit test before writing or revising any method that is at all complicated. I run all of the unit tests all the time to be sure nothing is broken. Then I can act like I have certainty that my software functions to the best of my understanding, and that I have pushed my understanding as far as I can.

The cool thing about this strategy is it makes great sense short term and long term. Short term I program faster overall because I force myself to think about interfaces before thinking about implementation. I am also more confident, less stressed, and I can more easily explain what I am doing to my partner.

Long term the tests dramatically reduce the chance that someone will harm the code. The tests communicate much of the information that would otherwise be recorded in documentation that would have to be separately updated (or not). The writing of the tests tends to simplify the design, since it's easier to test a simple design than a complex one. And the presence of the tests tends to reduce over-engineering, since you only implement what you need for tests.

Oh, and just in case the programmers missed something in their unit tests, the customers are writing functional tests at the same time. When a defect slips through unit testing and is caught in functional testing (or production), the programmers learn how to write better unit tests. Over time, less and less slips through.

A real testing guru would sneer at this level of testing. That doesn't bother me so much. What I know for certain is that I write better code faster and have more fun when I test. So I test.

Continuous Integration

Code additions and changes are integrated with the baseline after a few hours, a day at most. You can't just leap from task to task. When a task is done, you wait your turn integrating, then you load your changes on top of the current baseline (resolving any conflicts), and running the tests. If you have broken any tests, you must fix them before releasing. If you can't fix them, you discard your code and start over. In the short term, when the code base is small, the system stays in very tight sync. In the long term, you never encounter integration problems, because you have dealt with them daily, even hourly, over the life of the project. Continuous integration makes collective code ownership and refactoring possible without overwhelming numbers of conflicting changes, and the end of an integration makes a natural point to switch partners.

More information:
[http://c2.com/cgi/wiki?ContinuousIntegration]

Functional testing [black box testing]-

You can't continue development until the functional test scores are acceptable to the customer. The short term value is that the programmers know when they are done with the functionality, and the customers are confident that the system works. Longer term, the functional tests prevent regressions and communicate important historical information from a customer perspective. The functional tests back up the unit tests to ensure quality and improve the unit testing process.

More information:
[http://c2.com/cgi/wiki?FunctionalTests]

On Customer Site

You can't just take your understanding of requirements and design and implement with them for a month. Instead, you are in hourly contact with a customer who can resolve ambiguities, set priorities, set scope, and provide test scenarios. In the short term, you learn much more about the system by being in such close contact. In the longer term, the customer can steer the team, subtly and radically, with much more confidence and understanding because such a close working relationship is built. The Planning Game requires an on-site customer to complete requirements as they are about to be built, and functional testing works much better if the author of the tests is available for frequent consultation.

Culture and Environment

Open Workspace

The best XP workspace is a large bull-pen with small individual cubbies around the walls, and tables with fast machines in the center, set up for pair programming. No one can go off and hack for hours in this environment. That solo flow that I got addicted to simply isn't possible. However, in the short term it is much easier to get help if you need help just by calling across the room. In the long term the team benefits from the intense communication. The open workspace helps pair programming work, and the communication aids all the practices.

Forty Hour Week

Go home at 5. Have a nice weekend. Once or twice a year, you can work overtime for a week, but the need for a second week of overtime in a row is a clear signal that something else is wrong with the project. Over the very short term, this will definitely feel slower. But over a few weeks, and certainly over the course of months, the team's productivity will be higher and the risk will be lower. Over the long term, XP's reliance on

oral history demands a certain level of stability in the staff, and 40 hour weeks go a long way towards keeping people happy and balanced. Rested programmers are more likely to find valuable refactorings, to think of that one more test that breaks the system, to be able to handle the intense inter-personal interaction on the team.

Less Stress!

These are all the practices I regularly teach. Contexts with less stress will require fewer of the loops to maintain control. [...]

Is XP Always Appropriate

There are certainly contexts that will require more practices- ISO certification, FDA auditing requirements, complicated concurrency problems requiring careful review, complicated logic that no one can figure out how to segment. However, the above works for my customers.

Looking at the above, it is still hard to imagine how XP can fly, not because it so out of control, but because it is so completely controlled. I think the reason it works is because none of the practices require a PhD to execute, and they are all fun, or at least stress relieving, and they all contribute, directly and obviously, to the system.
[...]

The result of this process [...] is clean, tight, communicative code that is flexible where it needs to be, but without an ounce of fat. And it has a test suite that allows dramatic changes to be put in with confidence, years after the system was originally built.

Summary

XP values analysis and design so much that on a 12 month project, the team will spend no less than 12 months on analysis and design (and testing and integration and coding). It's not "are these kinds of questions addressed", but rather when and how. XP does it throughout and in the presence of the production code.
[...]

The percentages- 40-65% A&D, 5% coding, 30-55% test and support. I think I agree with them 100%. It's just that I can measure those proportions over pretty much any given day for every programmer on the team.

[Ed note: Ralph Johnson <johnson@cs.uiuc.edu> said that in XP the software life-cycle is: Analysis, Test, Code, Design.]

More Information

Ronald E. Jeffries, Extreme Programming Practices:http://www.armaties.com/Practices/Practices.html
Many XP Discussions on http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap

This article was edited by Yonat Sharon and Mark Collins-Cope based on information sent by Kent Beck to the otug@rational.com discussion group. Whilst we have made every effort to reflect our understanding of Kent's views on XP, this article was not structured by Kent himself.

CBD*i* Forum Meeting
DELIVERING BUSINESS INTEGRATION 99
1st to 3rd June 1999, Boston, Massachusetts, USA
23rd to 25th June 199, London, England

Butler Group's Forum for Component-Based Development & Integration
http://www.butlerforums.com/cbdforum/events/june99.htm

Do you need to understand: How other major organizations are successfully integrating business applications using component approaches? How successful companies have organized and made the change to components? New approaches to component integration architectures, processes and technologies?

For more information contact Phil Storrow Tel: +44 (0)1244 570955 E-mail: phil.storrow@butlergroup.co.uk

Object Development Process Series … Head to Head on XP

XP: Cutting Through the Hype

Doug Rosenberg and Kendall Scott put the counter arguments to XP.

Introduction

Extreme Programming (XP) offers a number of useful ideas. We believe there is significant value in the strong emphasis on testing, and several of their guiding principles could prove to be very useful in the evolution of software development processes. However, dubious claims, a distinct lack of scalability, catchy but questionable slogans, and Xtremist posturing tend to obscure the good stuff.

Doug has had extensive conversations on the subject within OTUG. Some of these have been useful; others have been silly; some involve hard questions that have yet to be answered. Kendall has spent a healthy amount of time perusing the material at the Wiki site on the Web. Most of the following exploration of the structure and ritual of XP is based on those dialogues and that research.

Dubious Claims

Kent Beck and his primary followers have positioned XP as a "lightweight" process that produces results of higher quality, and on a more timely basis, than those associated with more detailed processes. (XPers tend to refer to these using terms such as Big Design. We use this term throughout this article for that purpose as well.) There's a rather gaping hole at the center of this "process," though: analysis has basically been tossed.

Let's look at some of the claims that XP acolytes make to justify the decision to simply skip Big Design's "cumbersome" elements of analysis.

Requirements Change Weekly

According to XPers, processes that dictate at least some upfront analysis of requirements have two major flaws:

- Big Design seeks to control change, by obtaining early and generally once-for-all signoff on requirements and architecture. Most actual Big Design methods have detailed structures for responding to the inevitable external changes that will occur, but these mechanisms are often seen as somewhat secondary to the process.
- Big Design sees any uncertainty as a danger. If we don't know something, we can't reason about it.

This sentence from a note that Doug received, in connection with OTUG, speaks volumes: "You say 'capture requirements' up-front, but frankly that really scares me, because I smell 'waterfall' from a mile away."

What's more, the feeling in the XP camp is that customers have little idea what they want going into a project. One true believer even declared that "the customers can't possibly really know what they need at the beginning." And even when they do start figuring their requirements out, they insist on changing them at least weekly, and, in some cases, daily. As a result, there's simply no point in trying to pin the requirements down before jumping into coding.

We agree that requirements analysis and capture is fraught with difficulty and uncertainty. But Doug has worked on plenty of projects, in a variety of industries, where his customers somehow managed to avoid being quite so indecisive. No, you don't have to capture every single requirement before you start building the system, but we insist that a certain level of upfront analysis will save you lots of time down the road.

Sometimes it helps to force the customers to refine their understanding and figure out what they want before launching into production coding. Customers expect some discipline from the development team; it's entirely reasonable to ask the customers to show some discipline as well. If the customers really have no idea of what they want until they see something running, prototypes (built with actual source code!) are a wonderful technique for helping them to gain a clearer understanding of what it is that they want. It's the job of a good analyst to work with the customer and help them refine that understanding.

There are certainly going to be situations where there are changes in customer requirements after coding starts. We suggest that whenever possible, these changes get handled in the next build cycle. When that's not possible, you should do the best you can. Several XP techniques are probably quite valuable in "doing the best you can."

Another correspondent asked Doug, "What happens if you find that late in the game, the customer's requirements have been misinterpreted and correction requires fundamental changes in design? The real questions are, how often does this happen, and how hard do we work upfront to prevent this from happening? And this is where the much-derided "analysis" process pays for itself many times over. We do analysis to prevent this from happening.

Use Cases Are Too Complicated

XP recommends that you capture requirements within "user stories." This is in contrast to use cases; the general feeling among XPers is that use cases can only frame user requirements at exhaustive level of details,

on long forms that are onerous to fill out. A quote from the Wiki site conveys the horror that the XP group feels toward use cases: "My purpose is to maintain a balance of political power between business and development. Use cases as I have seen them used are complex and formal enough that business doesn't want to touch them. This leads to development asking all the questions and writing down all the answers and taking responsibility for the resulting corpus. Business is reduced to sitting on the other side of the table and pointing."

Here's the way we see it.

A use case is a sequence of actions that an actor performs within a system to achieve a particular goal. There's nothing particularly complex, or overly formal, about that definition.

A use case is most effectively stated from the perspective of the user as a present-tense verb phrase in active voice. This indicates that the user needs to have a clear understanding of the sequence of actions that he or she is supposed to follow, which certainly implies that the customer is actively involved in the detailing of use cases. If that understanding isn't there, then the use case is too complicated, and it needs to be refined—you know, just like code cries out for refactoring.

In our experience, "analysis paralysis" can happen for at least three reasons connected to use case modeling:

- The team spends weeks building elaborate, elegant use case models that they can't design from.
- The team spins its wheels worrying about whether to use any or all of the UML constructs "includes," "extends," and/or "uses."
- The team wastes time with long and involved use case templates. (Interestingly, we seem to be somewhat in agreement with the XPers on this point).

XP acolytes are apparently only familiar with the kind of use case modeling that leads to analysis paralysis. We say that if you derive use cases in conjunction with rapid prototyping, and/or mine use cases from legacy documentation (in particular, user manuals), while you stay highly focused on your customer at all times, you have an excellent chance to figure out what you need to do before you start doing it, and the result will be significantly less grief down the road.

The Cost-to-Fix Curve Is Flat

Kent Beck and has claimed that the cost of changing a bug in code is the same as that of changing a design bug, because, well, the code **is** the design. We address the fundamental silliness of this concept later in this article, but in the meantime, we'd like to point out that since the Xtremists have bounced analysis from the picture, it's circular reasoning.

The reasoning is as follows. If we don't ever do analysis and design before code, then nobody can claim that it's cheaper to fix errors before we start coding—because there is no such thing as "before we start coding." Presto, the cost-to-fix defect curve has gone from logarithmic to linear.

Let us restate this slightly:

We can view software development as a continuous sequence of writing code, testing code, ripping up code, retesting code, rewriting code, retesting code, and so forth. It's all the same activity, with bits of analysis and design blended in along the way. Therefore, the cost to make a change is linear no matter when you discover the need to make the change. This kind of circular reasoning sounds to us like...well, like somebody's selling something.

In direct contrast to these circular arguments is our experience that many of the errors that cripple projects are of the form "I was assuming it was red (or maybe entitlement), and you were assuming it was blue (or maybe deduction)." These unstated assumptions migrate into code and manifest themselves as insidious bugs. When they migrate into the architecture itself, we're looking at major rip-out and rewrite (and re-test) activities.

We don't believe any claims that anything makes this free. Rip-up and rewrite is never free. If rip-up and rewrite is "normal," that still doesn't make it free, nor cheap. It might make the curve flatter in a bizarre kind of way, but it's still not linear. We still haven't seen anything that disproves the research done by Barry Boehm, which shows that the cost of fixing a defect increases exponentially as you move from analysis towards a released product. Repeated assertions that the curve is magically flat do not a fact make.

A Distinct Lack of Scalability

Let's see why it's unlikely that XP can scale up and be successful in connection with a project where those conditions aren't likely to be present.

Pair Programming

Within the ideal XP project, developers do their work in pairs. And not static pairs, either: people might shift among partners several times a day, based on what needs to be done and who knows what. The idea, of course, is based on the maxim that "two heads are better than one." As such, it's a terrific principle to try to adhere to.

But what happens when you can't have all of your developers in one place for any meaningful period of time in the "bullpen" (assuming one is available)? What do you do about the fact that there are lots of developers in this world who aren't exactly blessed with the kind of strong communication and social skills that are clearly necessary within the XP scheme? And how do you hold anyone accountable, when everyone takes credit for everything?

As we discuss in the book, the best approach involves implementating a set of checks and balances among

senior and junior people; for instance, we recommend that less experienced people review the sequence diagrams that the developers with significant OO expertise produce. Pair programming reflects this idea to an extent, but it also assumes that there'll always be one-to-one matchups available, and that's simply not realistic in many contexts.

Hm—maybe if it was called Pair Analysis, Design, and Programming, we'd be more comfortable….

Not that there's anything inherently evil about programming in pairs—there may very well be some benefits to be gained—but it looks to us like pair programming is an attempt to compensate for the lack of analysis by forcing two people to look at each line of code as it's written. This is analogous to forcing kindergarten kids to hold hands in pairs when they walk to the cafeteria because you can't trust them not to wander off in a random direction individually.

Looking at it another way: just as sure as the two of us paired together in chess would lose to Garry Kasparov every time, neither the buddy system nor "exhaustive testing" can compensate for a lack of upfront planning. After all, Garry plans his strategy very carefully.

Index Cards

The XP clan advocates heavy usage of Class-Responsibility-Collaboration (CRC) cards. Now, these cards are regarded far and wide as highly useful--but on a relatively small scale. They're certainly not designed to capture extended detail, and as those of us who did research papers before we discovered the Internet know all too well, they're a bear to keep track of. And how are you supposed to pass index cards along to developers from a distance?

The chief architect for one of our current clients spends lots of time working with the other programmers. In fact, he spends so much time that he doesn't have enough time for the new stuff that he needs to do, making him a fairly willing participant in getting some of his design documented. Unfortunately, this company has developers on two coasts, so the whole "everybody in a big room sharing index cards" idea kind of falls apart. Models that can be E-mailed back and forth will work a lot better for them. And, we believe that this client will be quite successful at using modeling to do a better job on their next generation of products, and that they will be able to significantly reduce their defect rate as well by providing more visibility into the architecture and design.

We've seen very few development efforts that don't have some fairly serious issues to deal with. This is especially true in fast-growing companies where the scale of the new projects is much greater than the scale of the projects that got them off the ground. We suggest that using a good visual modeling tool puts a development effort on significantly stronger footing than does a bunch of loose index cards floating around.

Lack of Accountability

XP adherents insist that Big Design tends to only let people who, in theory, have specific knowledge and skills do certain tasks (you know, the team has to include a Class Librarian and an Architect and a Technical Writer, and so forth). To this, XP says "everybody does everything," in direct response to the perception that "Big Design takes advantage of the fact that some individuals are better-qualified to make some design decisions than others." But are we now to believe that developers will all of a sudden become enthusiastic about having to actually write things down in ways that the new guy can understand right off the bat? (And what's wrong with having good designers do the design, anyway?)

Actually, as we'll explore later in this article, XP proponents proclaim that properly refactored code is more or less self-documenting, and that all other forms of documentation are too unreliable to merit existence. This is part of their justification for focusing so heavily on "oral tradition," and they think that keeping things so very lean enables all key information to be maintained in one place. After all, outside of the code and all of those index cards, there isn't going to be anything on paper, right? But it's simply not practical to think this is going to work for any project with more than a few developers in a room together. Yes, the number of communication channels increases exponentially as people join a project, and yes, we advocate a fairly minimalist approach in our book, but the XP way is just too minimal to work for most projects.

An XP Alternative: "Smart Programmers (SP)"

In general, most of our clients have very little interest in solutions that don't scale. However, if small-scale solutions are what grab you, here's one that Doug has some experience with, which for the purposes of this article we'll call "SP" for Smart Programmers.

This is very useful if you're bootstrapping a software company with no venture capital. It's is based on the principle that 5% (or less) of all programmers do 95% (or more) of the work, and that these are the only programmers worth hiring.

The essence of SP is as follows:

- Hire only the top 5% performers that are personally known to you, or referred by a top 5%er that you trust.
- Have a chief architect plan the software architecture very carefully, such that there is minimal coupling between work units (say, UML packages).
- Have the chief architect match the work units to the specific skill sets and experience of the SP staff.
- Have all of your SP programmers work from home, where they will not be disturbed, unless they explicitly request to work in a common facility.
- Limit status reports to half a page, sent via E-mail, once a week.

- Have your SPers bring in incremental builds for testing when they decide they are ready.
- Restrict communication between SP developers to E-mail and phone, on an as-needed basis. **Never** bring all of the programmers into the same room at the same time except during integration testing, and then only those whose code is directly involved.

Now, you can build exceptional amounts of software with extremely small teams in this manner. Doug’s experience indicates productivity levels much higher than those cited for XP on the Chrysler payroll project.

Plus, the productivity rates will be much higher than those generated by XP. However, try to have the chief architect skip analysis and upfront planning, or make a bad hiring decision, and you'll have a disaster on your hands. We never bothered to formalize, or even name, the “SP” technique because it doesn’t scale and there’s no margin for error. It looks to us like XP might also posess these characteristics.

Catchy But Questionable Slogans

If you want punchy slogans that are instantly memorable, XP is right up your alley. After all, as its disciples have made clear by now, “software is too...hard to spend time on things that don't matter,” like topics that defy easy sloganeering. Let's look at some of the key phrases being bandied about as rallying cries for XPers the world over.

There Are Only Four Important Things About Software

This is the big one, the raison d’etre for XP’s existence. What are these four things? Coding, Testing, Listening, and Design (formerly Refactoring).

Oh, and don’t forget: “That’s all there is to software. Anyone who tells you different is selling something.”

As we’ve already pointed out, “analysis” didn’t make the cut. (This may have something to with something that Mr. Beck said on OTUG: “I call analysis by its true name: lying.”) More importantly, though, the cognoscenti know that there are exactly seven important things about software. (We’re not going to share them here; buy our book and find out what they are, if you're not already Enlightened.)

Seriously, though, to declare with such vehemence that these four elements of software development are the Only Ones That Matter is extreme beyond realism, even for something called Extreme Programming.

The Source Code Is the Design

We confess to getting an enormous kick out of this one, not least because of the fun we had writing “Use cases are not requirements” and the like in our book. In the face of Wiki remarks like “When you are refactoring, you are doing detailed design,” it looks like we’re going to have to do another book that has at least one chapter specifically aimed at refuting this kind of nonsense.

Refactoring is a fascinating subject, and we’re looking forward to Martin Fowler's book. We’re sure it will offer many useful techniques for improving one’s code. But design is design, and coding is coding, and it’s silly to think that they’re one and the same. It’s bad enough that the UML Gods (well, two out of three, anyway) basically ignored preliminary design; now we have to deal with Xtremist fanatics twisting the idea of detailed design beyond all recognition.

Every time we see the argument that we like to summarize as “it’s so cheap to change the code that it doesn’t matter if we screw up the design,” we get a sick feeling in our stomach. This is what we’ve come up with in trying to figure out what causes this feeling and to verbalize it.

Within XP, the “what” description is captured as user stories. Our concern is how the project team proceeds across the gap between the “what” and the “how.” It would seem that one of the reasons that the increments in XP are so tiny is that, for each increment, the gap between “what” and “how” must be crossed not only to the design level, which is hard enough, but to the code level as well. This means that design errors and coding errors will be intermixed.

Time will inevitably be spent fixing coding errors in order to get the program running and testable, and the testing will then expose design errors. After the design is changed, all that energy and time spent fixing the coding errors and testing the fixes in that part of the program will be wasted, because that code has now been ripped out and replaced. There will be coding errors in the new code as well, and the process gets repeated. This is deemed to be OK, because programmers are “having fun” programming.

It's not that you can’t get there this way—but to us, it seems like there’s got to be a lot of wasted energy, and time.

If we don’t try rigorously to do the best job that we can, **all** the time (and this would include **before** we write code), we work sloppily. If we decide that it’s OK to work sloppily, as long as we work fast (“oh, it’s OK, we can just fix this later”), we get crap. Software is not magically different from other endeavors in this aspect, no matter how good a refactoring browser you have.

As far as verifying that the design is correct before you start coding, you have the best brains on the project review it, preferably at both a preliminary level (to make sure that nobody is shooting off full speed in the wrong direction) and again at a detailed level. After that, you code and get feedback via testing.

Do the Simplest Thing That Could Possibly Work

This slogan says that you should simplify, simplify, simplify. This is always a good thing to do when it

comes to avoiding the tendency to build stuff you don't really need. However, we're concerned that this is likely to have the effect of encouraging developers to slap Scotch tape, bubble gum, and rubber-bands onto a piece of code that was evolved with not enough attention paid to defining a solid architecture.

We had a recent experience where we inhaled some "as-built" Java code through a reverse engineering tool and found methods with names like *bugFix37*. Unfortunately, whoever invented that method was doing the simplest thing that could possibly work, in a quite literal sense. (We can just hear the echoes of his buddy programmer, gleefully saying "Hey, we're supposed to do the simplest thing that can possibly work, right?") For the record, the project in question was not built with XP, but it **was** built without formal analysis and design.

It doesn't help that another guiding principle for XP followers goes like this: if you think you might not need it, you don't need it. This kind of thing goes a long way toward encouraging the kind of insular thinking which results in software that doesn't solve problems that anyone outside of the bullpen thinks are worth solving. This quote from Ron Jeffries reinforces that idea: "[A]developer really needs to dig in until she has developed her own internal sense of quality. I don't expect mine to match yours; I do hope we are in touch enough with them that we can talk about them and use our different senses to choose a solution for today that satisfies both our internal metrics." Sadly, since everything is invisible in XP except code, hope is about all we're left with in terms of quality assurance.

We still think that the quality of a system is best reflected by measures such as coupling (the looser, the better), cohesion (the tighter, the better), and completeness, with things like weighted methods per class and number of children thrown in for good measure. See Chapters 5 and 8 of our book for more information about these kinds of metrics.

Xtremist Posturing

On the one hand, XPers like to have extended debates about whether "courage" or "aggressiveness" more correctly states their position as far as dealing with those pesky people who pay the bills.

On the other hand, XPers tend to wander up into the ozone and say things like this: "The Extreme Programming Masters possess understanding so profound they cannot be understood."

XP rhetoric strikes us as combining a fairly high degree of arrogance and stabs at mysticism that verge on the truly odd.

Ready, Fire, Aim

Just as the Unified Process describes four phases (iteration, elaboration, construction, and transition) that are performed within each iteration of a project, XP talks about "stories, testing, coding, and design." In strong contrast to the Big Design that the Unified Process symbolizes, the Xtreme approach offers the "nanoincrement," which is basically an increment no longer than a week, and perhaps as short as a day. Faster, faster, faster!

We have nothing against finding ways to accelerate the process of software development--as long as those ways don't amount to "ready, fire, aim."

By refusing to spend any meaningful amount of time figuring out what do to before starting to "produce," and insisting that doing great work on an ongoing basis "in the small" will somehow magically result in the whole system being correct when the smoke clears, XPers seem to think they've found the silver bullet to counter Big Design's supposed tendency to get bogged down in detail.

This is a remarkably short-sighted approach. We can reduce the risk of failure with analysis and design to very acceptable levels without throwing away the baby with the bathwater. Going to code early is not risk-free; the risks are simply of a different nature (in some cases, perhaps more subtle). We should try to minimize overall risk levels on a project, not just some of the risks.

When Doug first started teaching OOAD about 7 years ago, the most common failure points in the training workshops always occurred during the effort to move from a requirements level (use case) view of the system behavior into a detailed design level (sequence/collaboration diagram) view of that behavior. Trying to make this leap--we call it the leap across the "what-how" gap in the book--from a pure requirements view to a detailed design view is extremely difficult. What people invariably found is that by adding in the "preliminary design" view of the dynamic model (robustness analysis), they're consistently able to get teams past this transition point.

We can still leverage the benefits of incremental development (perhaps with slightly larger increments), rigorous testing, and other techniques that may prove useful, such as pair-programming. In this way, we are guaranteed that we not only will we always have a functioning program, but that we've taken our best shot at getting it "right the first time" and minimized the need for labor-intensive rework and retest along the way.

Documentation Is Useless

A true Xtremist believes that the code is not only the design, but also the documentation.

"Merciless refactoring" (another key XP buzz phrase) will always, always, **always** result in code that's perfectly clear to even the most inexperienced of developers. One disciple coined the term Extreme Clarity, which he defined as "[t]he property of a program such that the source code exhibits everything about the system that anyone needs to know, immediately to hand, clear at a glance, accurate to six places."

There's no argument from us that this isn't a desirable result. And as far as non-code documentation, isn't it true that it's impossible to keep it in synch with those fantastically fleet coding machines? And isn't it true that oral communication tends to work better than written on most projects? After all, "[a]ll documentation is to be distrusted as out of date and subjectively biased." This sounds to us like an irrational fear of documentation.

Actually, in the context of a development effort, a competent technical writer who knows something about code will generally be able to keep up with the developers and analysts and so forth. Also, effective oral communication is of critical importance, but that doesn't excuse not writing stuff down--for outsiders, for new guys, for future reference, for reasons that maybe aren't real obvious at the moment. As far as subjective bias, well, all writing has some kind of bias, but software documentation is supposed to convey what **is**, **not** what **could be**, and if it doesn't, that's not the fault of the medium.

The absence of anything other than oral communication means that the people who write the code have to be around to answer questions. Historically, this has been used as a "job security" mechanism countless times over the last half century. In all too many cases, the guy who has the documentation in his head (or the team that has the documentation in their collective heads) is gone **before** the maintenance programmers show up.

Here's a story from Doug.

"I had a programmer working for me, on our dataflow diagram editor. His schedules kept slipping. We met to talk about it. He said--and this is an exact quote that I remember quite clearly: "You can't make me go any faster, you can't get someone to help me, and you can't fire me." He was confident in saying this, because all the design documentation was in his head. After I got over being shocked at hearing this, I fired him the next day. It was one of the best decisions I ever made. I replaced him with a guy who wrote great code, and documented what he did."

Anyway, recording the diagrams for posterity is **not** the fundamentally important thing. **Reviewing** the diagrams--and thereby the design--before coding **is** of fundamental importance. If you can have the senior engineers on a team review (and correct) sequence diagrams for all the scenarios that are going to be coded, and you have verified that the architecture (the static model) supports all those scenarios before you code, you're going to have a relatively easy time with code and test, and you're going to have minimized the amount of rip-up, re-code, and re-test that's going to be done.

And then there's this, from Mr. Beck: "Most people express their fears by elaborating the [CRC] cards into a database (Notes or Access), Excel, or some damned project management program." So now we see that Xpers look at customers in terms of uncertainty (with regard to their requirements) and fear (as far as keeping track of what's going on). We await the XPites' manifestation of "doubt" so XP can assume ownership of the legendary FUD triad, previously associated with Microsoft.

Ask the Code

We'll be happy to concede the point that programming still has a degree of "art" to go with the elements of "science" on which we prefer to focus. But some of the statements that Xtremists make tend to make non-believers like us think that XP is something of a cult of personality. For example:

- "Restructur[e] the system based on what it is telling you about how it wants to be structured."
- "Code wants to be simple."
- "The system is riding me much more than I am riding the system."
- "A *code smell* is a hint that something has gone wrong somewhere in your code. Use the smell to track down the problem."
- "Ask the code."

The idea, apparently, is that once code comes into existence, it assumes some mystical powers of communication, and it's incumbent upon its creator to listen, listen carefully, to the voice...you are getting verrrry sleepy...ah, sorry.

The bottom line is that all that matters to Kent Beck is the code, the whole code, and nothing but the code. Design represents "visions," analysis represents "visions of visions," and methodology as a whole represents "visions of visions of visions," and "[t]he cases where…visions came true only exacerbate the problem." As long as the developers' thoughts are simple—in other words, as long as they can just write their code without all that silly upfront stuff—they'll work "without stress, without fear, with joy and love for [their] task and [them]selves."

Kool-Aid, anyone?

Conclusion

XPers like to rail about how nothing else works. For instance, when Doug started pointing out, on OTUG, that upfront analysis really is a good thing, he got more than one E-mail asking "how much code do you write, anyway?" The implication that he doesn't write enough code to qualify as a "real programmer," in the eyes of Xtremists, holds about as much water as the idea that he doesn't understand the UML's "extends" construct (readers of our book, and faithful OTUGgers, know about that nonsense already), but that's not the point. The point is that the advocates of XP have a nasty tendency to insult those of us who prefer less extreme, but still highly workable, approaches to building quality software--and that's too bad, because there's some good stuff in there.

Perhaps some day we'll see a more reasoned and balanced version of the process. In the meantime, we concur with the guy who said on Wiki that "One very attractive feature of Extreme Programming is that it

tries to ride, instead of quelling, the deepest of urges in any true programmer, namely the urge to hack."

In the meantime, we'd like to close this article with a question.

Suppose you decide to try XP on your project, and six months later, you decide that the approach isn't meeting your expectations. Nothing is written down—there are no architecture or design documents and the only knowledge of the project is in the heads of the developers. ***What do you do?***

Doug Rosenburg can be contact on dougr@iconixsw.com. Kendall Scott can be contacted on SoftDocWiz@aol.com

UK recruitment bulletin from

Permanent & Contract

London	C++ Developers/Architects	£45,000/£50+ph

C++ on Unix with object database and banking experience for prestigious financial institution. Versant preferable. OOA/OOD and knowledge CORBA. Working in risk department on architecture or development. Rational Rose and UML a bonus

Permanent

Cambridge	C/C++/Visual C++ Analyst Programmers	£17-30,000

High profile GIS company needs developers. Either C, C++ Visual C++ skills. No GIS required. Preferably from scientific background. Must be graduate in numerate subject. Java, HTML skills a bonus. Team working in either: development, customer services, sales support.

Cambridge	C++ Analyst Programmers	to £30,000

C++ analyst programming role. Ability with Artificial Intelligence and biology a bonus. Strong familiarity with Object Oriented Programming. Working on 3D entertainment simulation

South	OO Consultants	£35K+£10K bonus

Opportunities for both post and pre-sales consultants. Clients based in banks, telecoms and other blue chip companies. Acting as technical OO guru. Account management basis. C++ on Unix and any object database is highly desirable.

South	C Analyst Programmers	to £35,000

C analyst programmers for ERP software house. Product like SAP. OO environment. Experience in business or commercial software rather than telecoms or control. Secure company and career path.

South	Internet Developer	to £38,000

Java, HTML. System design and interface. People management skills and web skills would be great. Working in media company. Developing complex, commercial applications for the internet. Experience with C, Unix, NT, RDBMS, SQL & OO also required.

Many positions working in Games companies for developers, producers and artists, with and without industry experience.

Looking for a UK based new permanent or contract opportunity?

Please call Tracy on **(+44) (0)181 579 7900** if you have skills in any of the following: C, C++, Visual C++, Delphi, Java, Visual Basic, Ada, Eiffel, Smalltalk, Versant, Objectstore, ODBMS, CORBA, COM/DCOM, Rational Rose, UML, OOA/OOD.

Introducing Object Technology Series

An Introduction to UML Use Case Diagrams

Robert Martin gives an overview of UML Use Case diagrams

Introduction

An important part of the Unified Modeling Language (UML) is the facilities for drawing use case diagrams. Use cases are used during the analysis phase of a project to identify and partition system functionality.

They separate the system into *actors* and *use cases*.Actors represent roles that can are played by users of the system. Those users can be humans, other computers, pieces of hardware, or even other software systems. The only criterion is that they must be external to the part of the system being partitioned into use cases. They must supply stimuli to that part of the system, and the must receive outputs from it.

Use cases describe the behavior of the system when one of these actors sends one particular stimulus. This behavior is described textually. It describes the nature of the stimulus that triggers the use case; the inputs from and outputs to other actors, and the behaviors that convert the inputs to the outputs. The text of the use case also usually describes everything that can go wrong during the course of the specified behavior, and what remedial action the system will take.

Some Examples

For example, consider a point of sale system. One of the actors is the customer and another is the sales clerk. Here is just one use case from this system:

Use Case 1: Sales Clerk checks out an item

1. Customer sets item on counter.
2. Sales clerk swipes UPC reader across UPC code on item
3. System looks up UPC code in database procuring item description and price
4. System emits audible beep.
5. System announces item description and price over voice output.
6. System adds price and item type to current invoice.
7. System adds price to correct tax subtotal

Error case 1: UPC code unreadable

If after step 2, the UPC code was invalid or was not properly read, emit an audible ‘bonk’ sound.
Error case 2: No item in database

If after step 3 no database entry is found for the UPC flash the ‘manual entry’ button on the terminal. Accept key entry of price and tax code from Sales Clerk. Set Item description to “Unknown item”. Go to step 4.

Clearly a point of sale system has many more use cases than this. Indeed, for a complex system, the number can reach into the thousands. The complete functionality of the system can be described in use cases like this. This makes use cases a powerful analysis tool.

Drawing Use Case Diagrams.

Figure 1 shows what the above use case might look like in UML schematic form. The use case itself is draws as an oval. The actors are drawns as little stick figures. The actors are connected to the use case with lines.

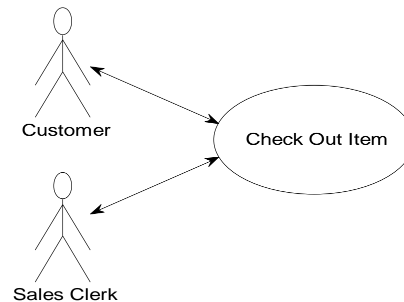


Figure 1 - A use case diagram

Clearly this is a pretty simple notation; but there is more to it. Use and actor icons can be assembled into

large “system boundary diagrams”. Such diagrams show all the use cases in a system surrounded by a rectangle. Outside the rectangle are all the actors of the system, and they are tied to their use cases with lines. The box represents the system boundary; i.e. it shows all the use cases that are inside a particular system. Everything inside the box is part of the system. Everything outside is external to the system. See Figure 2.

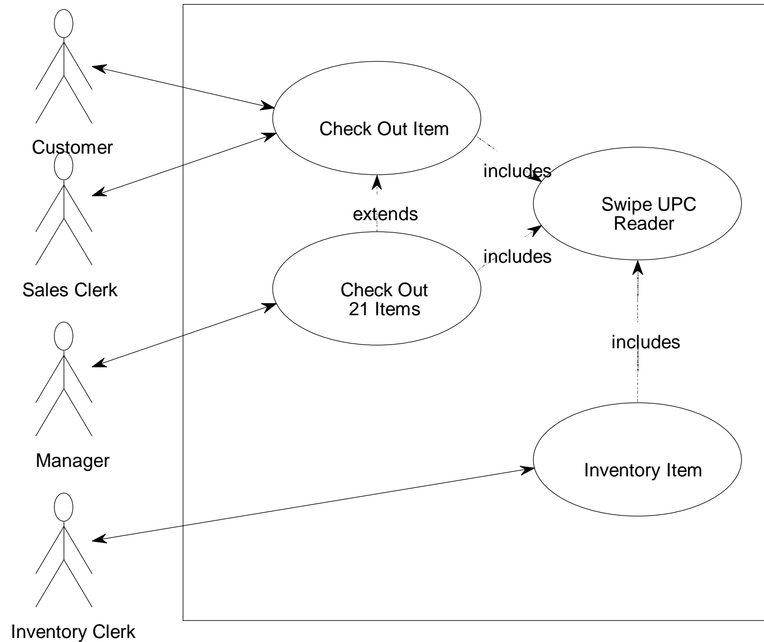


Figure 2 - A more complex use case diagram

Extends and includes

Aside from showing the system boundary, a few more actors, and a few more use cases; this diagram shows some relationships between the use cases. These relationships are «includes» and «extends». The includes relationship is the simplest of the two. Note that it appears twice in Figure 2, once from Check Out Item to Swipe UPC Reader. The other is from Inventory Item to Swipe UPC Reader. The Swipe UPC Reader use case describes the behavior of the actor and the system when the actor swipes the UPC reader across the bar codes on a product.

As we saw in our first use case, the Swipe UPC Reader behavior needs to occur within the Check Out Item description. However, since inventory clerks also swipe bar codes while counting objects on the shelves, this same behavior must be part of the Inventory Item use case. Rather than write the description for this behavior twice; we can employ the «includes» relationship to show that it belongs in both use cases. Having done this we might change the description of the Check Out Item use case as follows:

Use Case 1: Sales Clerk checks out an item

1. Customer sets item on counter.
2. «includes» Swipe UPC Reader.
3. System looks up UPC code in database procuring item description and price
4. System emits audible beep.
5. System announces item description and price over voice output.
6. System adds price and item type to current invoice.
7. System adds price to correct tax subtotal

So, the «includes» relationship is very much like a function call or a subroutine. The use case being used in this fashion is called an *abstract use case* because it cannot exist on its own but must be used by other uses cases.

The other interesting relationship is the «extends» relationship between Check Out Item and Check Out "21" item. In many stores, sales clerks under the age of 21 are not allowed to check out liquor. When an underage sales clerk sees a liquor item, the clerk shouts "21" over the P.A. system. Soon a manager walks up and swipes UPC code on the item. This represents a change to the use case that could be addressed in one of two ways.

First, we could add 'if' statements to the Check Out Item use cases such that it looked like this:

Use Case 1: Sales Clerk checks out an item

1. Customer sets item on counter.
2. If item is liquor
2.1. Call "21" over P.A. system
2.2. Wait for manager.
2.3. Manager «includes» Swipe UPC Reader
2.4. Goto step 4
3. «includes» Swipe UPC Reader.
4. System looks up UPC code in database procuring item description and price
5. System emits audible beep.
6. System announces item description and price over voice output.
7. System adds price and item type to current invoice.
8. System adds price to correct tax subtotal

Although this works fine, it has a severe disadvantage. Remember the Open Closed Principle (OCP)? t states that we don't want to create software that has to be modified when the requirements change. Rather, in well designed software, a change in requirements should cause of to add *new* code rather than change old working code. The same rules apply to the functional specifications of use cases. When the requirements change we want to add new use cases, not change old existing use cases.

Thus, rather than add the if statement in the use case, we use the «extends» relationship. This relationship allows us to specify a new use cases that contains commands for overriding and modifying the extended use case. Thus the Check out "21" item use case in Figure 2 overrides and extends the Check Out Item"use case. The text for the Check Out Item use case might appear as follows:

Use Case 2: Check Out "21" Item

1. Replace Step 2 of Check Out Item with:
1.1. Call "21" over P.A. System
1.2. Wait for manager
1.3. Manager «includes» Swipe UPC Reader

This achieves our goal of allowing new features to be added to the use case model without needing to change existing use cases.

Extension Points

Notice that the Check Out "21" Item use cases mentions the "Check Out Item" use case directly. This is unfortunate. What if we wanted to extend several other similar use cases in the same way? We'd have to have as many extending use cases as extended use cases. And all the extending use cases would be nearly identical.

We can remedy this situation by adding extension points to the extended use cases. Extension points are simply symbolic names that identify positions in the extended use case that the extending use cases can call out. Thus, our two use cases might look like this:

Use Case 1: Sales Clerk checks out an item

1. Customer sets item on counter.
2. XP21: Sales clerk swipes UPC reader across UPC code on item
3. System looks up UPC code in database procuring item description and price
4. System emits audible beep.
5. System announces item description and price over voice output.
6. System adds price and item type to current invoice.
7. System adds price to correct tax subtotal

Use Case 2: Check Out "21" Item

2. Replace XP21 of extended use case with:
2.1. Call "21" over P.A. System
2.2. Wait for manager
2.3. Manager «includes» Swipe UPC Reader

Text Management.

One of the problems with document maintenance is that when a single requirement changes, it may affect many places within the text of the functional specification. Indeed, sometimes the amount of redundant information in a functional spec can be very high, causing significant maintenance problems. The goal of use cases and their relationships is to manage the textual descriptions within a functional specification, and thereby reduce the redundant information. By structuring the use cases and their relationships properly, you can create functional specifications that never need to be changed in more than one place. For very large projects, this can be a significant gain.

The Structure of a use case is not the structure of the software it represents.

It is tempting to look at the structure of a use case document as a precursor or progenitor of the structure of the software it represents. However, this is not likely to be the case. Use cases are structured to minimize textual redundancy. While this is a laudable goal, it has nothing to do with software design considerations. Thus, the structure of the use cases is not related to the structure of the resultant software. Use cases do not represent objects or classes in the eventual designs. The relationships between use cases do not forshadow relationships in the software design. The structure of the use cases and the structure of the software are unrelated.

Use Case Diagrams have low information content.

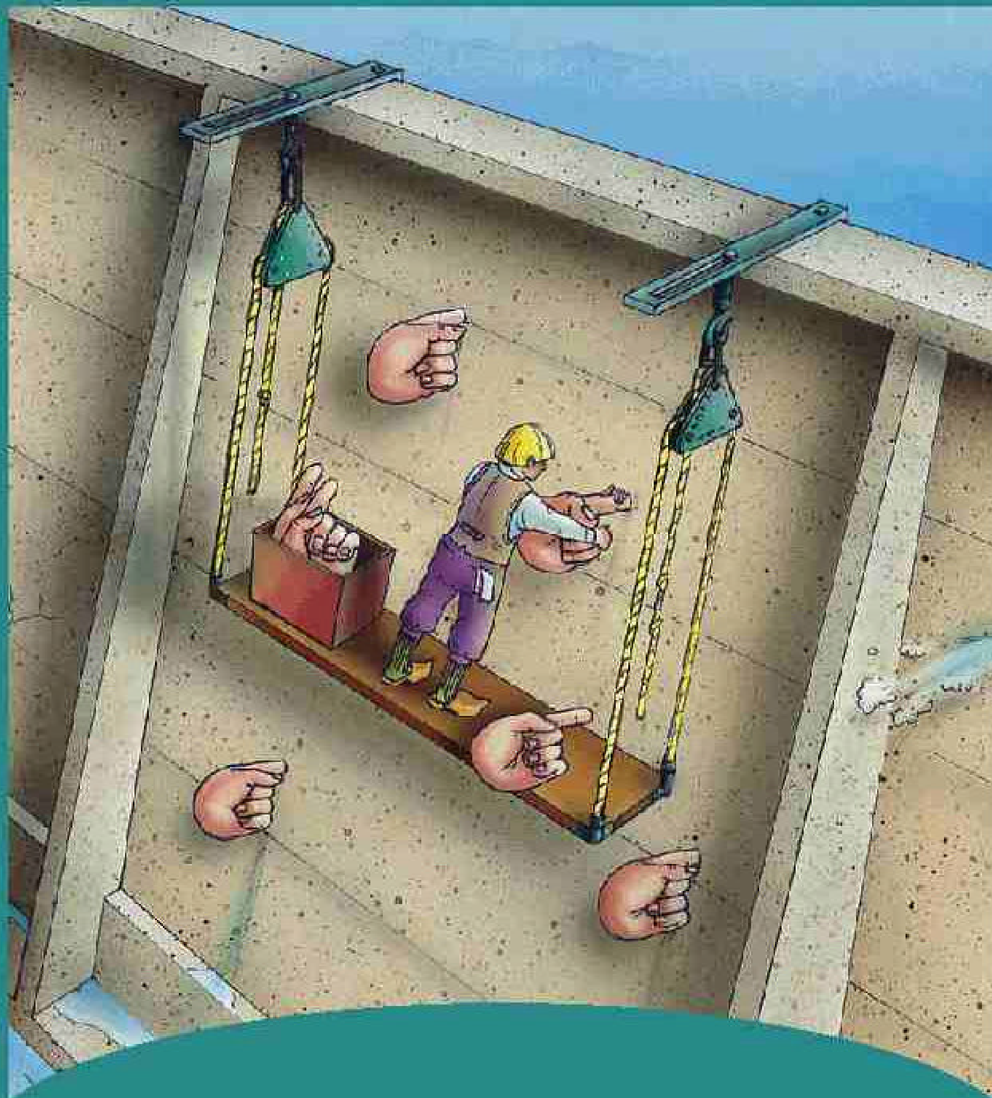
Use case diagrams don't tell you very much. They convey the structure of the use cases, but tell you very little about the text within them. As such, they are not particularly interesting documents when they are separated from their textual descriptions. At best the diagrams provide a nice roadmap of relationships so that readers can reconstruct the *whole* text of a given scenario by tracing through the «includes» and «extends» relationships inserting the text of the former, and modifying the text according to the latter.

Conclusion

Use cases are powerful tools for analysts to use when partitioning the functionality of a system. Use case relationships and the corresponding diagrams help analysts to structure use cases such that their textual descriptions contain a minimum of redundant information; thus making the whole text document much easier to maintain. But use cases are not design tools. They do not specify the structure of the eventual software, nor do they imply the existence of any classes or objects. They are purely functional descriptions written in a formalism that is completely separate from software design.

This article is an extremely condensed version of a chapter from Robert Martin's new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall.

This article was written by Robert C. Martin of Object Mentor Inc. Copyright (C) by Robert C. Martin and Object Mentor Inc. All rights reserved. Object Mentor Inc, 14619 N. Somerset Circle, Green Oaks, IL, 60048, USA
phone: 847.918.1004 fax:847.918.1023
email:rmartin@objectmentor.com web:www.oma.com



If all you are doing is plugging holes...

OBJECT MENTOR has a better way!

OBJECT MENTOR INCORPORATED

Training • Mentoring • Development

objectmentor.com **the** site for OOD

1 • 800 • 338 • 6716

Subjective Views… Object Architectures Series…

Design Patterns for Business Applications -- the IBM SanFrancisco Approach

Brent Carlson of IBM discusses the use of design patterns within SanFransisco.

The Goal: Effectively Modeling a Business

Any business application's primary objective should be to effectively model the structures and processes of the business which is using that application. An effective business application should capture the operational behaviour of the business as well as serve as a tool by which decision makers can assess and drive change into their business. Too often, however, existing business applications force users to fit their business processes to the static business model which the application implements. This mismatch results in wasted energy, unnecessary effort, and missed opportunities to improve business operations.

When properly used, object-oriented analysis and design techniques provide an opportunity to better define and implement flexible and maintainable business applications. IBM's SanFrancisco frameworks provide a solid basis upon which effective business applications can be built. By defining and implementing a well thought-out set of business components which address the common requirements of business applications, they give the application developer a major head start in developing an application that will truly meet the needs of its users. Let's discuss some of those common requirements and SanFrancisco's approach to providing solutions for those requirements.

Common Requirements of Business Applications

While business applications vary considerably in their details, many common requirements can be defined which every enterprise-strength business application must meet in order to be sufficiently flexible to the end user. Some of these key common requirements that all business applications share include:

Modelling a hierarchical business organisation (e.g., subsidiaries, holding companies, operational units), including the ability to selectively share or restrict access to business information throughout the hierarchy

Example: The company FoodWarehouse is a bulk distributor of food products to restaurants and repackagers. It divides its business into two operating units: frozen goods and bulk goods, maintaining separate product lists for each unit. However, each unit shares warehouse space in a number of the company's warehouses. In addition, the company defines a single financial chart of accounts which applies across both units.

Selectively caching balance information at varying levels of detail

Example: FoodWarehouse sells a variety of goods, some of which have expiry dates after which the goods cannot be sold. FoodWarehouse attempts to ensure that goods do not remain in stock beyond the expiry date by allowing order entry personnel to offer special discounts for goods with excess stock levels. As part of the support for this business policy, the application needs to be able to rapidly retrieve quantity information on a lot by lot basis for each product in stock.

Adjusting to varying business policies, not only on an installation by installation basis but also within a particular installation

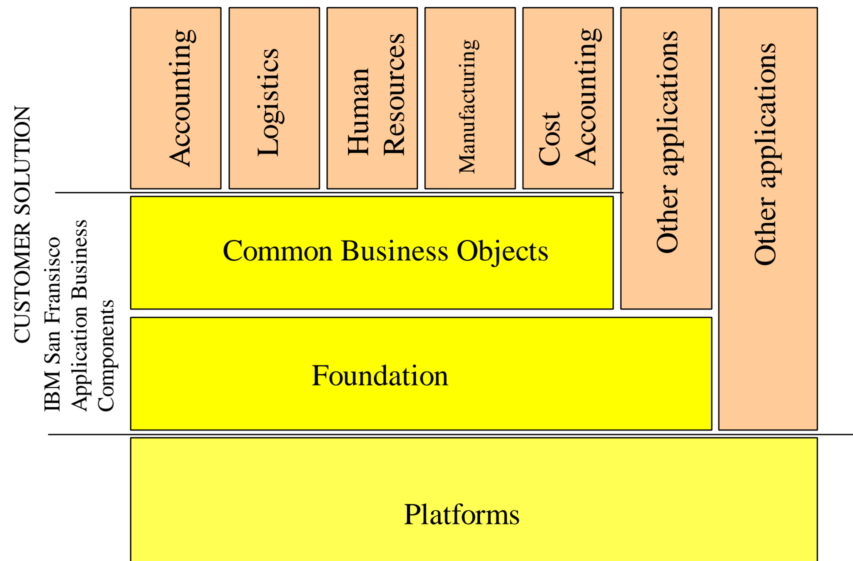
Example: Most warehouses owned by FoodWarehouse have the same configuration and thus can use the same put-away policies when storing incoming goods. However, FoodWarehouse also leases some warehouse space which is configured differently and requires different put-away policies. In addition, some products require special put-away policies to ensure proper inventory rotation.

While this list of requirements is certainly not exhaustive, it serves to point out the type of flexibility that a well-designed business application must provide. Proper use of design patterns can play a role in providing such flexibility. We'll be discussing some of the specific design patterns used within the SanFrancisco frameworks to build an effective foundation upon which application developers can build flexible business applications. But first, let's go over what the SanFrancisco toolkit provides to a developer.

The Context: a Brief Overview of SanFrancisco

SanFrancisco is a set of cross platform, object oriented, commercial application frameworks specifically designed for a distributed environment -- the first ever written in the Java development language -- that can be used to build business-critical multi-platform applications. It is composed of three layers: the Foundation layer, the Common Business Object (CBO) layer, and the Core Business Process (CBP) layer. Application developers can build on top of any of the

three layers depending upon the particular needs of their application.



The Foundation Layer

SanFrancisco's Foundation layer provides an object/programming model by which business object classes can be defined and implemented and a set of object services which the business objects built using the Foundation object model rely on.

SanFrancisco's object model defines a set of base classes from all SanFrancisco business object classes are derived. These base classes encompass both business objects which have independent lifecycles and which are uniquely identifiable (Entity objects), and those whose lifecycle depends upon the business object in which they are contained (Dependent objects). The object model also provides for a specialised class of Dependent objects that encapsulate atomic units of work associated with a single business object or spanning across multiple business objects (Command objects).

The object services provided by SanFrancisco are those which would be expected from any object application server product: object lifecycle (creation/deletion), transaction, locking, ORB, and naming support among others are part of the Foundation package.

The Common Business Objects Layer

The Common Business Objects layer of SanFrancisco provides three distinct functions. First, as the name implies, it provides a set of business objects which are common across all applications. Business objects such as Company, BusinessPartner, Currency, and NumberSeries all fall into this category.

Second, SanFrancisco CBOs provide a set of common financial objects which serve as interfaces to the financial components of a business application (i.e., the general ledger and accounts receivable/accounts payable ledgers). These interfaces allow other components of the application to reflect financial changes resulting from business processes into the financial components with a minimal amount of knowledge about those financial components. These common financial objects also provide a sophisticated mapping function which enables domain-specific information (e.g., products, warehouses, customers) to be mapped to the financial account information without exposing that account information throughout the non-financial components.

Finally, the SanFrancisco CBO layer provides a number of generalized mechanisms which are effectively partially implemented solutions to recurring application problems. These generalized mechanisms are meant to be extended and customized for each use, and are in effect, frameworks within the larger SanFrancisco framework. We'll be touching on some of these generalized mechanisms later in this article.

The Core Business Process Layer

The Core Business Process layer of SanFrancisco is composed of numerous components, each of which addresses the core business objects and processes for a particular business domain. Each core business process provides only those processes and objects

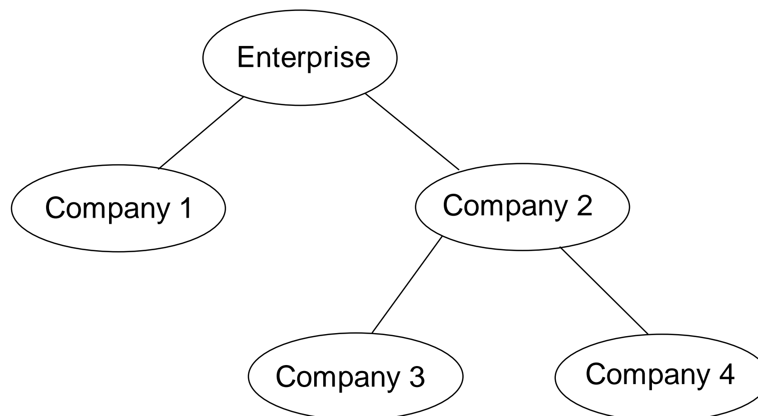
which apply to every application within a particular domain. Application providers extend these core business processes to add additional function particular to their set of application requirements and to customize the objects and processes provided by the framework to meet their application's needs. The SanFrancisco frameworks currently support General Ledger, Accounts Receivable/Accounts Payable, Warehouse Management, and Order Management Core Business Processes.

The Solution: How SanFrancisco meets Business Application Requirements through Design Patterns

Let's take a look at how SanFrancisco meets the business application requirements introduced in the FoodWarehouse example:

Requirement 1: Modeling a Hierarchical Business Organization

Before an application can model a business enterprise, it must be able to represent the internal structure of that enterprise. As you would expect, SanFrancisco allows business application developers to define subsidiaries, operating units, departments, and other business organizational units. In fact, SanFrancisco gives special treatment to these organizational business objects. SanFrancisco does this by providing a Company object hierarchy which serves as the "spine" of the framework/application. Company objects within the hierarchy form a tree structure that is anchored by a single Enterprise object (a specialization of Company). For example, a business enterprise with two subsidiaries, one of which is broken up into two different operating units, could be represented by the following object hierarchy:

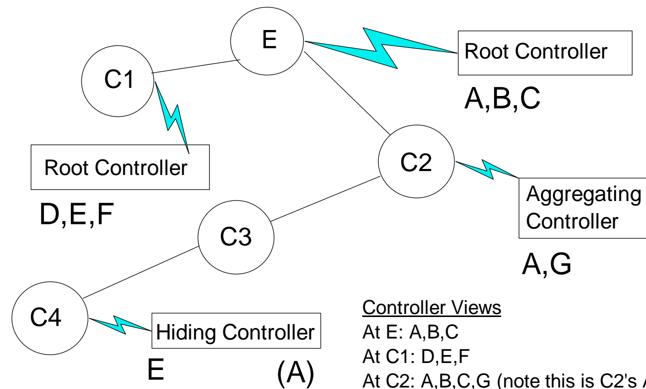


SanFrancisco also defines an active company which serves as the application's operating context. Company objects hold information such as intelligent collections of "top-level" business objects (called Controllers -- more about these in a couple of paragraphs) and default information and behavior which must be readily accessible by the application.

What does it mean to be a "top-level" business object? Within SanFrancisco, top-level business objects are those whose natural owner from the domain perspective is the Company. Some examples of top-level business objects in the SanFrancisco frameworks are BusinessPartner, Product, Warehouse, and Bank. In contrast, business objects such as SerialNumber (owned by Product) and StockLocation (owned by LocationControlledWarehouse) are not top-level business objects because they have a natural business object owner as defined by the domain.

Now that SanFrancisco has effectively modeled the business hierarchy, how does it selectively share or restrict access to business information within that hierarchy? SanFrancisco defines the Controller design pattern to meet this business need. Controllers are essentially smart collections of specific business object types. These collections provide views of their business objects from the Company context where they reside. SanFrancisco provides separate Controllers for each of the top-level business objects BusinessPartner, Product, Warehouse, and Bank mentioned previously.

SanFrancisco defines three distinct Controller types: root, aggregating, and hiding. A root Controller can exist at any level within the Company hierarchy. Root Controllers are independent; in other words, they have no dependencies on any other parent Controllers within the hierarchy. On the other hand, aggregating Controllers are dependent on parent Controllers for their basic contents but are capable of adding additional contents or overriding their parents' contents. Hiding Controllers are specializations of aggregating Controllers that can also selectively hide their parents' contents. Take a look at the following configuration example:



Company E (the Enterprise object) has a root Controller which contains objects A, B, and C. Company C1 introduces its own root Controller. This prevents any user running in C1's context from seeing objects A, B, and C -- only objects D, E, and F are visible. Company C2 provides an aggregating Controller which both overrides an object (object A) and introduces an additional object (object G). Company C3 does not have a Controller of its own so delegates its Controller behavior to its parent company C2. Company C4 introduces a hiding Controller. This Controller both overrides an object (object E) and hides another object (object A) from being visible.

How do aggregating and hiding Controller objects provide these composite views? They rely on a specialized implementation of the Chain of Responsibility pattern that is provided by the Foundation base class DynamicEntity. DynamicEntity implements a set of interfaces which are used to attach dynamic attributes (called properties) to specific object instances. The underlying implementation supports delegation of property retrieval from one DynamicEntity to another -- the Chain of Responsibility pattern. SanFrancisco's Company class extends DynamicEntity and thus inherits this chained property support, establishing its parent Company as the successor object. By convention, SanFrancisco attaches all Controller instances to a particular Company instance by property using a well-known property naming convention. An aggregating or hiding Controller instance retrieves its parent Controller by property using its well-known name, thus relying on the underlying Chain of Responsibility provided by Company. These well-known property names also make it possible to retrieve the correct Controller from a particular Company object without knowing where Controller objects have been placed in the Company hierarchy or what type of Controllers have been placed in the hierarchy.

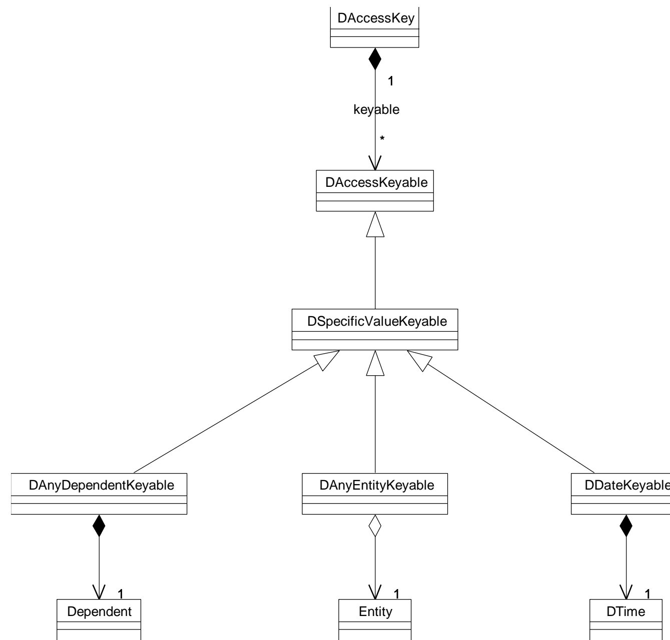
Because the Controller design pattern is such a core concept, SanFrancisco provides special code generation support for it. Most Controllers can be fully generated from their design model representation -- a major productivity improvement for developers.

Requirement 2: Selectively Caching Balance Information

Information like inventory levels, outstanding customer credit amounts, and financial account balances are the raw materials that feed the processes used to run a business. Because many of these business processes are response-time critical, business applications often need to be able to selectively cache summary balance information to improve application responsiveness. For example, SanFrancisco tracks inventory levels by product, warehouse, location, lot, serial number, stock type, and unit of measure. This allows SanFrancisco based applications to build the necessary financial and logistical mechanisms to manage a business. Remember, however, in our example that FoodWarehouse also needs to be able to rapidly retrieve inventory levels for some products on a lot by lot basis, ignoring many of the other tracking levels provided by the framework.

SanFrancisco addresses requirements like these at two levels. At the Common Business Object level, SanFrancisco provides generalized mechanisms to build composite keys out of arbitrary combinations of business objects, and to use those keys to build up sets of cached balances against the criteria described by those keys. At the Core Business Process level, various domain-specific business objects specialize these generalized mechanisms to meet particular business requirements.

Let's first discuss the generalized mechanisms. These "frameworks within a framework" provide partial implementations of solutions to common business problems. In the case of composite keys, the Common Business Object layer of SanFrancisco provides classes which represent both the composite key and the components of that key. The composite key class (DAccessKey) is a base class which is expected to be subclassed to be made domain-specific. The key component classes ("keyable" classes), on the other hand, are concrete classes which can be used as is. The DAccessKey class and a sampling of keyable classes are shown below.



As can be seen from the above diagram, DAccessKey holds onto a collection of DAccessKeyables. This diagram also shows three concrete keyable classes, two of which are generic and one of which deals with a specialized business object, the Dependent subclass DTime. DAnyDependentKeyable and DAnyEntityKeyable are capable of encapsulating any Dependent or any Entity object, respectively, so that the encapsulated object can be used as part of a composite key. DDateKeyable, on the other hand, is designed to encapsulate only a DTime object and treat that DTime object as a date during key comparison. These are just a few examples of keyable classes provided by the Common Business Object layer.

Likewise, the Common Business Object layer also provides a generic concrete class which is used to maintain cached balance sets. Each entry in this set is keyed by a specific DAccessKey instance with a different combination of keyable values. The Common Business Object layer also defines another generic concrete class which manages a collection of cached balance sets, each of which is configured to cache against a different combination of keyable values.

So far, this discussion has been rather abstract -- let’s tie it to our FoodWarehouse example. Remember that SanFrancisco keeps track of product inventory levels by product, warehouse, location, lot, serial number, stock type, and unit of measure. For our FoodWarehouse example, the order entry person needs to be able to quickly retrieve the inventory level for any product based on lot. Thus, we need to set up a cached balance set which builds its keys (DAccessKeys) taking into account specific values for only lot and unit of measure. DAnyEntityKeyables would be used to encapsulate the pertinent Lot and UnitOfMeasure objects, and other placeholder keyable objects (not discussed above) would be used to indicate that the other tracking attributes are not of interest.

However, up to this point, we’re still dealing with generic objects, which can be confusing and error-prone to code against. How does SanFrancisco convert these generic objects into objects which have domain meaning? The warehouse management core business process defines a DProductBalanceAccessKey class which extends from DAccessKey and introduces domain-specific methods (for example, getLot(), setLot(Lot), getUnitOfMeasure(), setUnitOfMeasure(UnitOfMeasure)). The implementations of these methods understand which keyable subclass to use when managing their associated business objects, and also know the correct position of the keyable within the collection of keyables held by DAccessKey. Warehouse Management also defines a ProductBalances class which encapsulates the cached balance set collection class provided by the Common Business Object layer. This ProductBalances class delegates all cached balance responsibility to the encapsulated class, but directly manages the creation and updating of Inventory objects to represent fully qualified inventory levels as needed.

The warehouse management core business process benefits greatly from the partially implemented composite key and cached balance design patterns provided by the Common Business Objects layer. These CBO classes provide most of the needed implementation, leaving the warehouse management classes responsible for domain specialization (in our example, domain-specific interfaces for Lot and UnitOfMeasure and domain processing such as Inventory object creation and updating). Of course, this reduces both the testing and maintenance effort required by warehouse management since most of the implementation comes from the CBO classes.

Requirement 3: Supporting Flexible Business Policies

One of the most likely ways that business applications are customized is in the area of business policies for various processes. End users of business applications expect to be able to change the way an application behaves based on their internal business policies, rules and laws applicable to various jurisdictions the business operates in, generally accepted business principles unique to their line of business, and many other reasons. For some applications, such customization is very painful, requiring major rewrites of many application modules and the accompanying instability and maintenance nightmares that such changes introduce. Other applications are designed in a heavily parameterized manner, so that the end user can set various "knobs and buttons" on the application to get it to behave in a manner that approximates their requirements. The tuning process for such applications can often be rather lengthy, however, and end users still may not be able to adapt the application to some of their more esoteric requirements.

How does SanFrancisco deal with this challenging customization environment? Throughout the development process for CBO and CBP features, the SanFrancisco team relies on an extended team of domain experts from numerous application development companies to identify areas which are highly volatile and likely to be customized. SanFrancisco then isolates the business algorithms for each of these points in the framework using the Strategy pattern or a SanFrancisco-unique extension of the Strategy pattern. (Because the term "strategy" doesn't have much meaning to a typical domain expert, SanFrancisco refers to the Strategy pattern as the Policy pattern in its documentation and throughout the remainder of this article.)

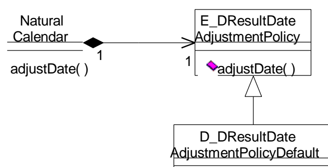
SanFrancisco always provides a default Policy subclass for every policy introduced by the framework so that the framework is operational "out of the box". Sometimes, these default policies implement a typical algorithm for a business application. In many cases, however, framework default policy implementations are simple algorithms which a typical application would not use.

In certain circumstances, SanFrancisco provides a set of complete policy implementations. For example, the warehouse management CBP provides a set of costing policies which fully implement various costing algorithms such as standard, average, latest inbound, last in first out (LIFO) and first in first out (FIFO). In general, SanFrancisco provides multiple policy implementations when a meaningful set of algorithms can be defined that meet most application requirements.

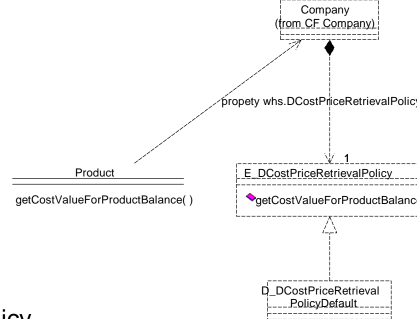
Whenever SanFrancisco introduces a policy, the development team takes special care to ensure that the interface is defined broadly enough to cover the vast majority of business situations. Typically the domain experts on the team begin the interface definition process by exploring a number of different business algorithms to varying levels of detail, then working with the object oriented designer responsible for the policy to define its interface based on the requirements of those algorithms. SanFrancisco policy interfaces also always accept as an additional parameter the business object which has domain responsibility for the method implemented by the policy. This gives concrete policy implementers a great deal of flexibility, as they can retrieve attributes from the passed-in business object and navigate through its contained objects to retrieve additional inputs to their business algorithms.

SanFrancisco Policy Scoping Mechanisms

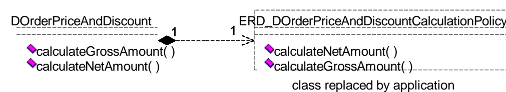
Object-specific Policy



Company-wide Policy



Application-wide Policy



Many twists to the standard Policy pattern occur in a real-life business application environment. Consider the context that a particular business policy should be customized in -- should it be possible to vary the behavior on a business object by business object basis, or should the behavior be variable only on a per company basis (perhaps because of legal requirements) or even across the entire application? Examples of all three levels of variability occur in the SanFrancisco core business processes, and SanFrancisco defines a consistent way to design and implement policy classes for each level.

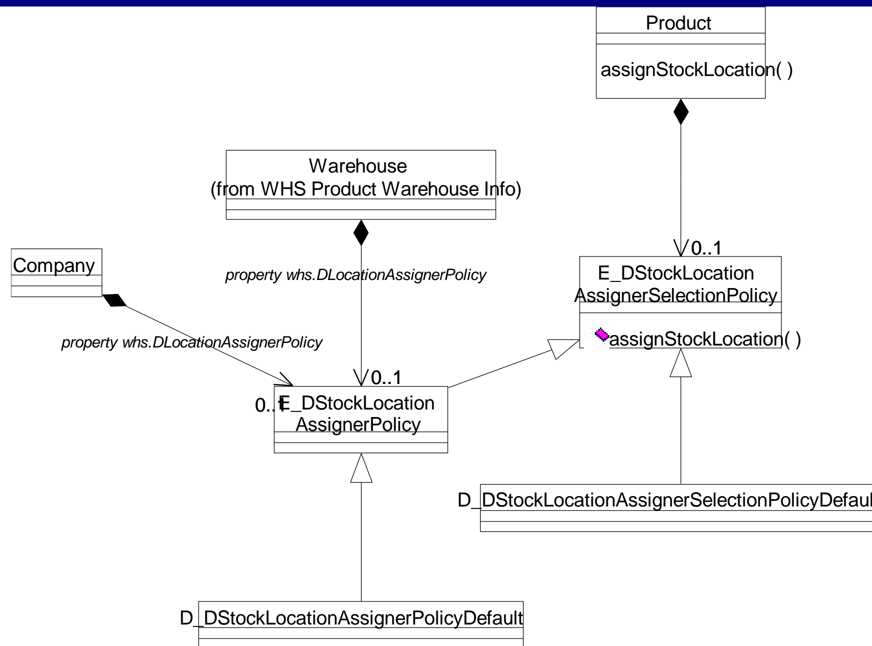
As can be seen from the above figure, SanFrancisco implements each level of policy scoping differently. When a policy is scoped to a specific business object, then the mechanisms defined by the standard Strategy pattern apply. Company-scoped policies take advantage of the dynamic property support implemented by Company (via its base class DynamicEntity) to ensure that all business objects within the scope of a particular company use a single policy instance. Application-wide policies take advantage of the Abstract Factory mechanisms implemented by the Foundation layer to allow application developers to replace the standard policy implementation provided by the framework with an implementation tailored to their specific application (or even on an installation by installation basis).

SanFrancisco Extended Policy Pattern Example -- Chain of Responsibility Driven Policy

The SanFrancisco team has also found that the standard Policy pattern is inadequate for certain business application requirements. As a result, the SanFrancisco framework has defined two extended Policy patterns: the Chain of Responsibility Driven Policy pattern and the Token Driven Policy pattern. Let's discuss the Chain of Responsibility Driven Policy pattern in more detail.

Domain algorithms often need to vary based on the objects being processed as well as on the object doing the processing. In our FoodWarehouse example, most warehouses can use a standard location assignment algorithm, but some warehouses require a specialized algorithm, and some products need to be able to override the standard algorithm. The Chain of Responsibility Driven Policy pattern meets these requirements by supporting the ability to selectively override Policy behavior and also to define which objects being processed can override Policy behavior.

The pattern defines two related class hierarchies: a standard Policy class hierarchy and a Selection Policy class hierarchy. Both the Policy and the Selection Policy class hierarchies are composed of a base and a default concrete class. As would be expected, the default concrete class in the Policy hierarchy implements a framework-provided domain algorithm. The default concrete class in the Selection Policy class hierarchy implements an encapsulated chain of responsibility to search for a Policy instance by property through various domain objects which are being processed, ending at the active Company. The Company object is configured to hold the default Policy algorithm for the method as a dynamic property.



As you can see in our particular example, the Product business object has domain responsibility for the method assignStockLocation() and thus holds as an attribute an instance of DstockLocationAssignerSelectionPolicy. Notice that the base policy class DStockLocationAssignerPolicy inherits from the Selection Policy base class. You might think this structure seems a bit odd; however, if you view a Policy subclass as a specialized form of Selection Policy which always selects itself, then this inheritance hierarchy makes sense. Such a hierarchy gives the end user a great deal of flexibility when configuring various business objects. Configuring a business object with a Selection Policy instance increases application flexibility at a small cost to performance, while configuring an object with a Policy establishes a particular domain behavior and limits flexibility but with improved performance.

Again returning to our FoodWarehouse example, those products which have special put-away requirements will be configured with a specialized Policy subclass which supports those requirements. The remaining products will be configured with a Selection Policy subclass whose algorithm first checks the warehouse to see if an overriding policy has been established, then goes to the context company to get the default algorithm to use. Of course, in general the application provider can and will define multiple concrete subclasses of both the Policy and Selection Policy to provide different business algorithms and chains of responsibility. In some cases, these subclasses will be provided as part of the standard application, and in other cases, specialized subclasses will be written on site to meet the needs of a particular installation.

The Power of Design Patterns in a Framework

SanFrancisco thoroughly integrates design patterns throughout its various layers. Such patterns are the primary means by which an effective framework describes its architecture and design parameters. This helps to focus the application developer's attention towards the areas of designed-in variability within the greater scope of the framework's architecture. As so aptly described by Gamma, Helm, Johnson, and Vlissides in their influential book Design Patterns, "A framework is a set of cooperating classes that make up a reusable design for a specific class of software. The framework dictates the architecture of the application. A framework predefines design parameters so that the application designer can concentrate on the specifics of the application." The SanFrancisco development team strongly believes in the power of design patterns to make application developers more effective.

As has been shown by our examples, specific design patterns used by SanFrancisco do not stand alone. Each layer of SanFrancisco introduces flexibility and extensibility into the framework by partially or completely implementing various design patterns. The patterns introduced by lower layers of SanFrancisco tend to be mechanism based. Many of these patterns are specific implementations of well-defined industry standard patterns such as Command, Chain of Responsibility, and Abstract Factory. The higher layers of SanFrancisco provide context to such patterns by incorporating them into higher-level design patterns that meet specific business application requirements. The power of SanFrancisco becomes evident through the use and interplay of these various design patterns throughout the layers of the framework. Such interplay between patterns within a framework often results in "the sum being greater than the parts", such as the Controller pattern relying on the chained property behavior provided by Company, and the merging of Strategy and Chain of Responsibility patterns (again built on dynamic properties) in the Chain of Responsibility Driven Policy pattern.

Design patterns are also important to SanFrancisco for another reason. Design patterns describe the interactions between various business objects in a generalized and consistent way so that an application developer can quickly learn how to use and extend the framework. This concept of design reuse in effect creates a higher-level design language which serves to elevate the thoughts of the application developer, allowing that developer to concentrate on the requirements of the application being developed. This is analogous to the step forward in software development that occurred when compiled programming languages such as FORTRAN were defined. These languages elevated programmers above the assembly language mechanisms (for example, how to implement a do loop), allowing them to think solely in terms of what those mechanisms should be used for instead of also being concerned with the mechanics of implementing the mechanisms. Design patterns do the same for SanFrancisco, especially because SanFrancisco focuses not only on consistently using design patterns throughout the framework but also on providing consistent documentation wherever those patterns are used. This helps application developers to quickly identify design pattern usage and step beyond the underlying framework mechanism to the domain problem being solved, resulting in more time spent on solving real business problems and less time spent dealing with low-level mechanisms.

Visit www.ratio.co.uk for links on object technology, additional articles and back copies of ObjectiveView (issues 1 and 2)

EMail objective.view@ratio.co.uk to subscribe to ObjectiveView for email (Word or PDF) or hardcopy delivery

Hardcopy available for UK, Eire and Northern Europe only.

Introducing Object/Component Technology Series…

Introduction to Components

Component-based development is a critical part of the maturing process of developing and managing distributed applications. Virtually every vendor is touting components as the best technique to address a wide spectrum of problems, from building graphical user interfaces to integrating large-scale, enterprise-wide applications. In this report, Michael Barnes of Hurwitz Group provides an introduction to components and component-based development.

Introduction

Hurwitz Group has long encouraged modular application development that leverages previous IT investments, including software and training. We believe IT organizations can leverage components to successfully build modular, responsive applications. Component-based development has the potential to improve developer productivity, speed the development process, and result in the deployment of more flexible applications with a far higher level of quality. This modular approach to application development can also facilitate the practical reuse of existing legacy applications, including logic, data, and application services. Before these benefits can be realized, however, IT organizations must understand the implications of components and how their implementation will affect not only new development projects but also existing system infrastructures, the application development and maintenance process, and staff training requirements.

Defining Components

Hurwitz Group divides the component world into two major categories: large-grained and fine-grained. Fine-grained components provide only a small piece of functionality and must work together with other fine-grained components to deliver a complete, recognizable function. For example, the individual elements that make up a graphical user interface screen — a button, a list box, or a dialog box — are fine-grained components. (This view of fine-grained components maps directly to the more traditional definition and implementation of objects.) Multiple fine-grained components can be elements of a large-grained component.

Unlike fine-grained components, large-grained components encompass a complete piece of business functionality. An example might be a general ledger, an asset depreciation component, a letter of credit, a manufacturing work-in-process tracking module, or even an entire legacy application, such as an inventory tracking application. In short, large-grained components perform a complete, discrete business function.

Hurwitz Group believes large-grained components have the potential to deliver greater productivity to developers than fined-grained components. This is primarily due to the higher level of abstraction large-grained components provide. By focusing on the component interface and not the inner workings of a component, application assemblers are able to build systems based on a thorough understanding of the business processes these systems will support. This is in contrast to strict object-oriented (OO) development in which developers must have an understanding of both the business processes and the inner workings of fine-grained components (or objects), including a detailed understanding of the underlying technology, the proper method of implementing the components, and the complex dependencies among various components.

Components Versus Objects — Multiple Interfaces Versus Multiple Inheritance

In addition to their level of abstraction, a crucial difference between objects and components revolves around inheritance. Objects support inheritance from parent objects. When an inherited attribute is changed in the parent object, the change ripples through all the child objects that contain the inherited attribute.

While inheritance is a powerful feature, it can also cause serious complications that result from the inherent dependencies it creates. These dependencies are a major reason why OO development has failed to live up to the hype and deliver on the promise of increased reuse. Applications that incorporate objects with strict inheritance models are extremely complex and difficult to model and test. In addition, OO development with strict inheritance requires organizations to impose an unrealistic level of discipline on developers to improve productivity based on reuse. Without this discipline, reuse falters because the applications and the inherent dependencies within them are too complex for developers to have a clear understanding of all the objects. If this is the case, it is difficult, if not impossible, for developers to recognize the objects' potential value in other applications or in different contexts within the same application.

In contrast to the multiple inheritance model of objects, components are characterized by multiple interfaces. For example, consider a customer management component. A sales automation system leveraging this component needs an interface that allows access to the customer name, contact history, phone number, and credit limit. At the same time, a shipping system using the same component only needs an interface that supports a limited amount of information exchange, essentially a name and address. By developing components with multiple interfaces, IT organizations can simplify the development process and facilitate reuse by supplying different departments and/or developers with only the information, services, or both, that they need to leverage.

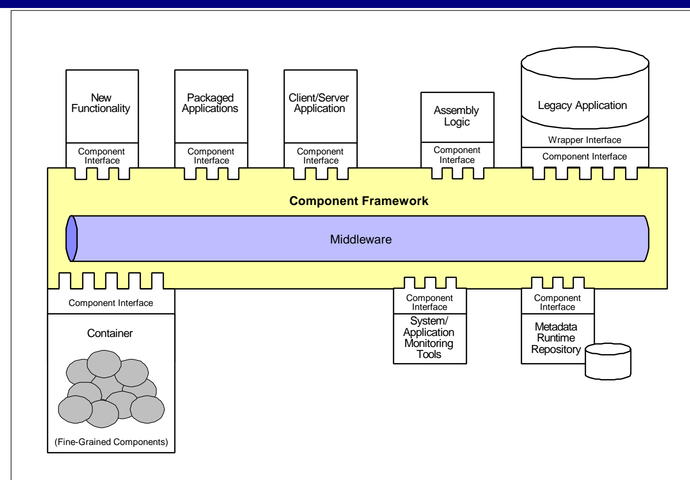
Components effectively eliminate the problem of dependencies related to object inheritance. Instead, component interfaces act as the "contract" between the component and the application. The application has no view inside the component beyond the exposed interface. This provides IT organizations with the flexibility to update components while maintaining only the interface and behavior of the components. Organizations no longer need to be concerned with all the dependencies in an application that may lead to conflicts or software defects when one piece of the application is changed.

Exposed component interfaces also allow IT organizations to more effectively leverage previous IT investments. This is true for legacy code or even complete applications that were not originally intended to be reusable components. By wrapping this code or application to hide its inner workings, organizations can expose only the interface that different groups require.

Requirements of a Component-Based Architecture

The following section provides a detailed description of the key elements required in a component-based architecture (see figure below). IT organizations that implement this architecture will have the flexibility to add specific functionality and/or products as they are needed

.



A distributed, component-based architecture

A component-based architecture must include the following:

Multiple interfaces. Multiple interfaces allow the framework to support the variety of components and component models that developers need to simplify the delivery of a full range of business functionality to end users.

Repository. This is required to store and manage the components. It contains critical information — metadata — about the component and the interfaces. To add a new component, developers simply add the component's metadata and interface definitions to the repository. In addition to critical runtime support and component storage, repositories also provide a central location for managing and coordinating the development process. This includes requirement definition, application development, QA testing, and application deployment.

Middleware. This supports the translation that will be necessary among the various programming languages, platforms, and component models. An organization's components will be heterogeneous, built using different languages, run on different platforms, and based on different component interoperability models. Middleware also provides a single, coherent event model and method invocation technique. The event model describes how events are exchanged within an application; the method invocation describes how a method (or service) within a component is used. Middleware addresses one of the largest risk areas in large-scale development: integrating all the pieces of an application and using the same "language" to achieve this integration. Middleware becomes that de

facto language and provides a common definition that reduces the complexity of developing within a component architecture.

Coordination logic. This rationalizes and controls the activities of all the components that an application requires to perform its work. Hurwitz Group believes that a component architecture will achieve this significant task via a coordination logic service. Using any of a number of approaches — 3GL, 4GL, business rules, or, more likely, a workflow facility — the coordination logic service will manage long sequences of events. Hurwitz Group believes workflow facilities offer the strongest potential for coordination because they can best handle the very long (and often convoluted) sequences of events that characterize business processes. However, the key to the success of this approach is large-grained, abstract components with interfaces defined in terms of business functions (e.g., methods like "Create Order" or "Add Customer"), not IT functions.

Systems and applications management capabilities. This is handled through a component interface, which can provide the management instrumentation, monitoring, and reporting capabilities not routinely available in most existing code. Ideally, the management facility will also provide functionality for assessing the performance impact on IT systems when different components are invoked. This functionality is necessary to provide a characterization or detailed understanding of component-based systems. This understanding is achieved by isolating and measuring the performance of individual components within that system.

Existing code wrapped into components. This capability allows developers to incorporate legacy code, existing client/server applications, and even packaged applications as components. Wrapping turns existing code into a black box. Its functionality is invoked only through the defined interface. Developers expose the application's functionality through the component interface. A variety of techniques are available to wrap legacy applications and expose an interface, including using an existing API, using "screen scraper" technology, or interfacing to legacy middleware such as CICS.

Implementing a Component-Based Architecture

Once organizations understand the appropriate level of abstraction required within the components they are building, these organizations are faced with the challenge of building a component-based architecture. The key to successfully implementing this architecture lies in creating the modular framework that will enable all the different pieces to work together. Ultimately, an organization may be dealing with hundreds, if not thousands, of reusable components, including various middleware services, coordination services, and system management services. In addition to all the sound development practices IT organizations should normally pursue, Hurwitz Group has identified three areas that require particular attention in the development of component-based architectures: modeling, configuration management, and testing.

Modeling. Data modeling is a standard part of most large-scale development projects. Increasingly, organizations are also creating object models. In the component world, however, two additional models must be created, the interface model and the component interaction model. The interface model will resemble an object model but without the inheritance aspects. The interaction model will more closely resemble a set of workflow definitions and rules that characterize how the system accomplishes its business functions. Once these two models are complete, developers can leverage their existing development tools and staff expertise to generate or write the necessary code.

Configuration management. This is the process of coordinating component usage throughout the entire application development effort. Unlike standard client/server applications being built today, a component application will consist of many independent applications assembled together into the component framework. Critical issues include the ability to easily version components and manage multiple instances with different versions of the same component. Consider the upgrade of a packaged application. For a period of time, both versions may need to exist while the functionality and data are migrated between versions. In a component-based environment, configuration management will be essential for ensuring availability and scalability.

Testing. IT organizations must also address the interaction of components within an application and the complexity and highly distributed nature of these component applications. Simple function and unit testing is not sufficient in the component world. Individual components should be thoroughly unit tested and debugged before they are declared ready for use in component-based development. In addition, developers need to test the end-to-end functioning of the application, which will entail multiple components working together, often across multiple platforms. Finally, to ensure successful deployment, developers need to test for scalability and load handling, and identify bottlenecks as usage increases.

Through attention to modeling, configuration management, and testing, developers can build a component architecture robust enough to handle critical business computing. Each component brings business functionality. When coordinated with other components, this functionality delivers the systems support that IT organizations will require to deliver applications that meet user requirements and support a company's business processes.

Conclusion

Components are not new, they are simply the embodiment of modular programming techniques. Since Microsoft first introduced the VBX component for Visual Basic, developers have used components to add sophisticated functionality to their GUI client applications. However, components can do much more than enrich a GUI. Hurwitz Group views components as the means through which organizations can rapidly respond to changes in business processes and meet the

demands of end users. This is achieved through high levels of reuse and modularization. To realize this payoff, organizations need to clearly understand the requirements for successfully implementing a component architecture. These requirements include technology and product support as well as improved support for managing the entire application development process.

This article is reprinted with permission from Hurwitz Group, Inc. Copyright 1999 Hurwitz Group, Inc. Hurwitz Group may be contacted as follows: Hurwitz Group, Inc. 111 Speen Street, Framingham, MA 01701, 508.872.3344 http://www.hurwitz.com, email: info@hurwitz.com.

ICONIX

www.iconixsw.com

ICONIX multimedia tutorials on object technology are in widespread use in over 40 countries.

Visit our web-site for detailed information, to see screenshots and order online through our secure server!

NEW! **Complete CORBA (6 CD set)**
http://www.iconixsw.com/CompleteCORBA.html

NEW! **ComprehensiveCOM (6 CD set)**
http://www.iconixsw.com/ComprehensiveCOM.html

Unified/UML Series (4 CD set)
http://www.iconixsw.com/Unified.html

OT for Managers Series (2 CD set)
http://www.iconixsw.com/OT4Managers.html

Check out the widely acclaimed new book by ICONIX President Doug Rosenberg and Kendall Scott: **Use Case Driven Object Modeling with UML - A Practical Approach**
http://www.iconixsw.com/UMLBook.html

And don't forget our onsite UML training workshops.

http://www.iconixsw.com/UMLTraining.html

Call us at (+1) 310-458-0092 for more info.

Training From Ratio Group

Excellence in Object and Component Training

Object Oriented Analysis and Design Using UML

5 Days - Hands On - Intensive

Background

The application of OOA/D techniques has substantial benefits in reducing system development risks and improving the quality of object oriented software developments. UML (the Unified Modelling Language) has emerged as the de-facto standard for OO modelling techniques. In this intensive hands-on workshop, you will gain practical experience of the major techniques of UML. The workshop culminates in-group project work during which you undertake a small but complete analysis and design project.

Summary

This course will give you a practical understanding of the major techniques of the UML object oriented analysis and design notation, and how these techniques can be applied to improve quality of productivity during the analysis and design of computer systems.

Object Oriented Analysis and Design - Advanced Modules

Available on request as part of the above course

- RSI Approach to Use Case Analysis (as seen in C++ Report/Patterns '98)
- Component Based Development using RSI, Patterns and UML
- Achieving software re-use in your organisation
- Customer Specific Group Project
- Implementing UML in C++
- Implementing UML over Relational Databases
- Implementing UML in Java

Object Oriented Programming in C++

5 Days - Hands On - Intensive

Background

C++ is one of the fastest growing programming languages of today, bringing the benefits of object-oriented development to the much used C programming language. This course leaves student with a practical knowledge of how to implement object oriented principles.

Summary

This course will leave students with a firm understanding of the C++ language and its underlying object-oriented principles. Attendance on the course will enable participants to make an immediate and productive contribution to project work.

Object Oriented Programming in Java

5 Days - Hands On - Intensive

Background

Although only a few years old, Java has recently emerged as a major develop-ment language for cross-platform and internet based applications. Its platform independent architecture has made it an ideal object oriented development language for both stand-alone and internet (browser applet) based development.

Summary

This course will give you a practical understanding of the major features of the Java development environment and language, both in the context of web applets, and in the context of stand-alone applications. Students will leave the course able to start productive work immediately.

Visit *www.ratio.co.uk* or call Rennie Garcia on +44 (0)181 579 7900 for more information.