ObjectiveView

The Object and Component Journal for Software Professionals

Claude Monet 1840-1926 "Poplars from Marsh"

Plus:

Designing Distributed Object Systems

Product Line Architectures

The OPEN Process Specification

Published by

RATIO

OO consultancy – training – tools – recruitment

see www.ratio.co.uk for back copies

Sponsored by

ObjectMentor Inc.

www.objectmentor.com

ObjectiveView

The Object and Component Journal for Software Professionals

CONTENTS

IN THE NEXT ISSUE OF OV...

- ✓ **Interview with Ivar Jacobson**
- ✓ **Clemens Szyperski:** ***"Recent Insights into Components"***
- ✓ **Bertrand Meyer on Component-Based Development**
- ✓ **Ralph Johnson:** ***"The Dynamic Object-model Architecture"***

CONTACTS

Editor
Mark Collins-Cope
markcc@ratio.co.uk

Production
Karen Ouellette
karen@ratio.co.uk

Free subscription
objective.view@ratio.co.uk
(with 'subscribe' as subject)

Circulation / Sponsorship Enquiries
objective.view@ratio.co.uk
Tel: +44 (0)20 8579 7900

R A T I O I S P R O U D T O S P O N S O R

TOOLS
Europe 2000

"Enterprise Architecture - Patterns - Components"
Mont Saint-Michel / Saint-Malo Normandy/Brittany, France

5-8 JUNE 2000

Visit http://www.toolsconferences.com/europe
for programme and registration details

TOOLS… the major series of international conferences entirely devoted to Objects

Introducing Technology Series

A Profile of XML

Richard Vaughan with an introduction to XML…

Introduction

The Mexican poet Octavio Paz once commented 'the differences between the spoken or written language and the other ones – plastic or musical – are very profound, but not to such an extent that they make us forget that essentially they are all language: expressive systems which possess significant power.'

Nowhere more than in computing do expressive systems abound, our most recent bouncing baby being the eXtensible Markup Language or XML. In fact Junior is now a strapping toddler of two years of age and given his explosive growth rate one wonders just how big he will become in adulthood. However, the computing industry is also guilty of putting much old wine in many new bottles. There are, therefore, other crucial questions, for example what kind of child *is* XML and what does it actually want *to be* when it does grow up?

This article examines XML, its origins, syntax and its relationship to software systems.

Some Background

Throughout this decade, the growth of the Internet and the success of the Web has seen an increasing demand for a more powerful version of HTML that can serve as a universal data interchange standard. For a variety of reasons it was not possible to extend HTML itself, not least because it is a presentation-oriented schema. The search therefore began for an alternative.

Standard Generalized Markup Language (SGML) was originally considered but was rejected because of various inherent problems such as the challenge of developing suitable parser technologies. It was therefore apparent that a new language was needed and thus XML (a subset of SGML) was born. In fact, the specification was developed very quickly over a period of only eighteen months. This was mostly because of the high demand for a universal format, however it was also in an attempt to prevent the standard from becoming clogged with lots of extra 'goodies', most of which appear on only a few people's wish lists.

Note however that although XML can be used for transmitting information across a network (and has a very big future in terms of the Internet) it is *not*, by definition, an Internet issue. In fact, it can just as easily be used in a non-networked environment. When an application receives some data, this can come from a secondary storage medium such as disk or tape as easily as from a network link. Developing this principle, the information does not even have to come from 'outside' the machine at all. Applications can in fact use XML to communicate between themselves at run time using operating system services such as pipes and shared memory arenas.

Given that XML is not fundamentally an Internet issue, it is therefore not a 'web language'. It is possible to present XML data in a browser by means of stylesheets, however this issue is peripheral to the core standard. XML is therefore not a direct replacement for, or extension of HTML, although the future of HTML is now a moot point.

What's the Deal?

The pivotal issue is that XML is a universal file format and therein lies the source of the commotion. The principle of divide and conquer has proved time and again to facilitate systems development because it allows us to deal with problems in manageable chunks. In fact, the softer the links between its components, the more robust a system will be as a whole. Indeed, the 'decoupling' theme is the prime mover in the current trend towards component-based development.

XML therefore enables us to soften the links between system components, thereby allowing us to protect and capitalize upon our development investments. XML can do this in two ways. Spatially speaking, an application can communicate with another physically separate application without *advance* knowledge of that application (hence the fuss about XML and the Net). Temporally speaking, an application can write data without prior knowledge of the future applications that may read it.

XML and the Application

This decoupling is accomplished by sending descriptions of the information being communicated *along* with the information itself. In other words the data stream also contains metadata or 'mark-up' and the price paid for this approach is that applications

must incorporate some form of parser to separate mark-up from data.

The simplest form of parser is the event driven variety whereby an appropriate procedure is called for each type of mark-up construct encountered. This is effected in C/C++ by means of pointers to functions. The second class of parser is the tree-based form where an internal data structure known as a 'document tree' (essentially an object hierarchy) is generated during analysis. Once the tree is built, the application is free to navigate the structure, reading and updating it as it chooses. If need be, the tree can then be written out again, as XML, to a file or network link.

Note that a number of third party parser technologies are available to developers. Examples are Microsoft's COM based component, Vivid Creation's library and James Clark's Expat.

Let us now explore some XML markup and see how the syntax and grammar operate.

Some Syntax

A complete XML script is called a document and although this is treated as a single *logical* object, *physically* a document can be composed of many separate files or 'entities'. An XML document must therefore be composed of at least one top-level file called the 'document entity' and any files separate to this are called ' external entities'. The document entity must contain a single top-level 'element', the general form for which is as follows:

```
<StartTag>

   Data


</EndTag>
```

Here **<Tag>** tells the application that some marked-up data follows and the **</Tag>** indicates the end of that data. Note that **Tag** will normally be some useful label for the enclosed data. This syntax operates such that elements can be nested within each other. For example:

```
<Tag>

   <NestedTag>
   Data
   </NestedTag>

   <NestedTag>
   Data
   </NestedTag>

</Tag>
```

Which allows complex composite data to be represented. The second aspect to element syntax is that the start tag can carry 'attributes'. For example:

```
<Tag Attribute1 = "Data",
Attribute2 = "Data">
```

In essence, this is no different to the first general example we saw in that an element is stated as *containing* various data and this is one of the more confusing aspects of XML. In a grammar such as C, block enclosure and nesting is the only containment model available, which makes for a considerably simpler syntax.

In addition, 'empty' element tags are possible which consist of a start tag but no corresponding end tag. These take the following general form:

```
<Tag/>
```

Note the trailing slash indicating that this tag stands alone. Empty element tags can possess attributes and in fact this is their only real purpose as there is no other way for them to 'enclose' data.

Let's now see an example of the above general forms in action:

```
<CompactDisc Type        = "Music"
             Title       = "Shifting Images"
             Band        = "Stampede">

   <Track    Title       = "Huckleberry Finn"
             RunningTime = "4.33"
             Copyright   = "G Shelter"/>

   <Track    Title       = "Lemon Luminance"
             RunningTime = "4.19"
             Copyright   = "A  Livingboy"/>

   <Track    Title       = "The Seeing Room"
             RunningTime = "10.56"
             Copyright   = "N  Kiwit"/>
```

```
</CompactDisc>
```

This piece of XML describes a Compact Disc. The attributes in the most enclosing start-tag state that it is a music CD and detail the name of the album and the band that recorded it. Our hypothetical CD also contains three tracks and the empty element tags representing these possess attributes describing the title, running time and author.

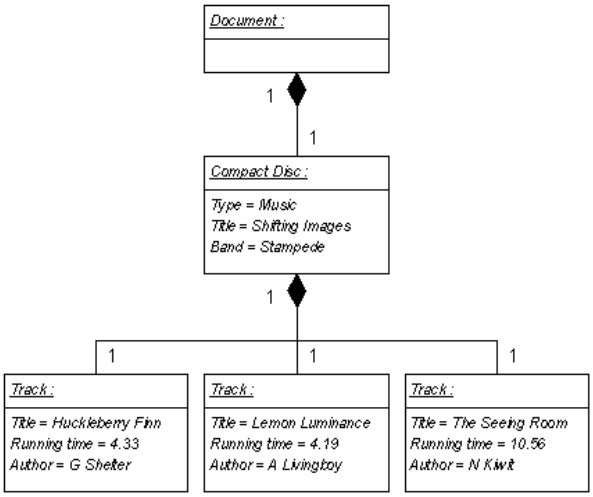


Figure 1. Document tree that would be generated from this document (in UML notation)

Constraining Markup

The above XML script is an example of what is called a 'well formed' document. This means that it obeys all the rules such that start and end tags are properly nested etc. However, there are no constraints placed on this document. The CompactDisc element could easily contain an element describing the price of bread, which would of course be inappropriate.

To assert the required structure and content of marked-up content one must include a Data Type Declaration or DTD. During processing the parser checks the document's element content and structure against the constraints stated in the DTD. If the elements conform to the DTDs type declarations then the document is said to be valid.

In addition to a DTD, documents can contain an 'XML Declaration' in the document entity and a 'Text Declaration' in external entities. These can be used to state what version of XML is being used (although only version 1.0 currently exists), the kind of character encoding scheme used and so on.

Note that DTDs, implemented as external entities, provide the substrate for the XML vertical market data formats that are becoming increasingly available. Here an organization can define and publish a data interchange standard for a given domain thus making a lingua franca for that domain available to all interested parties. Examples include ChemML for defining documents that describe chemical and molecular information, DocBook for paper publications and MathML for mathematical equations and data. There are also a slew of business-oriented data formats such as FinXML, OFX, Biztalk and cXML.

Further Markup

XML goes much further than the above example however. It is possible to embed comments within entities (files) just as one would in a traditional programming language. There are some differences however in that comment text can be made available to the application that is parsing the document and there are certain restrictions on the placing of comments within an entity.

It is also possible to create predefined entities within the DTD, which can be referenced in marked-up content rather than spelt out explicitly. This facility is very similar to the #define preprocessor directive in C/C++. Note that these predefined character

sequences can be contained within entities that are separate to the main document entity and in this case the 'entity reference' mechanism operates very much like the #include preprocessor directive in C/C++. Further to this, it is possible for the DTD to be held in part or whole in a separate file just as one would keep all C/C++ type declarations and prototypes in separate headers.

XML also supports the concept of conditional sections which operate in much the same way as would 'comment out' a section of code in a program source file. Note however that this and the entity reference mechanism explained above do not enjoy the flexibilities we are used to in C/C++.

Of course there are times when we wish to use mark-up characters such as <, > and & in their literal form rather than as signals to the parser. To cater for this, XML allows these characters to be escaped by means of a 'character reference' mechanism. In addition, XML defines the concept of non-parsed character data or CDATA sections. These are areas of content that the parser ignores completely thereby allowing markup characters to be used literally without recourse to character references. The purpose of CDATA sections is to enable sequences of raw binary data to be carried within a document thereby enabling an application to send any kind of data such as images or sound.

Implications

There are however a number of implications in using a markup solution to data interchange. Firstly, because XML documents are human readable this means that they can also be created by hand using a simple text editor program. This is in marked contrast to the use of proprietary file formats, which often yield machine-readable data only.

However, although proprietary formats are generally not human readable they can be considerably faster to read and write and can therefore mean faster applications. This is because the 'meaning' in the data is implicit in its sequential byte ordering, as opposed to being explicitly stated by bulky metadata. Similarly the 'understanding' of the data is integrated into the application code itself, obviating the need for a separate parsing stage between the information and the application's internal data structures.

In addition, marked-up data means larger files and longer data streams. These therefore take up more disk space and take longer to transmit across a network connection. In addition, an XML application will often be larger and slower because of the parsing technologies involved. Indeed, this is the classic time and space tradeoff we must accept every time we choose to decouple systems components still further. I.e. performance gets slower and software gets bigger.

Finally, there are the human resource costs to consider in a move to XML. Programmers need to understand the syntax to be able to work with DTDs and also need experience in working with parser interfaces in order to produce shippable code.

As pointed out earlier however, the true significance of XML is that disparate applications can talk to each other. Using XML, an application can write and read data to and from other unknown applications. Moreover, should one really need speed of transmission, it is quite possible to mix the proprietary and universal file format approaches by wrapping binary data in CDATA sections.

Further to this, XML text can be compressed before transmission, which ameliorates the bandwidth consideration significantly. Note that various schemes have been suggested for making XML more terse, usually at the risk of errors, however none of these have been much more effective than using compression techniques. In essence, XML's design favours readability and robustness over space considerations.

Conclusion

In many ways the advent of XML can therefore be seen as part of a trend towards greater unification, which has a parallel in the rise of (UML) and this can only be for the general good.

However, as a formal grammar or 'expressive system', XML simply does not measure up. It is fraught with restrictions and duplications and generally lacks the flexibility that we enjoy in languages such as C++. For example, there are two entity reference mechanisms, DTDs use 'parameter entities' and content markup uses 'general entities'. There are also restrictions in entity reference syntax regarding external entities. In addition, one can use either attributes or element nesting to the same effect and this causes considerable confusion in XML neophytes.

Furthermore, comments can be used to signal sections of a document that must be ignored. Yet CDATA sections do the very same thing, while conditional sections can also be employed to the same ends but only in *external* entities. Why not have a single mechanism for 'blindfolding' the parser? This would make markup semantics easier to understand and parsers would thus be easier to write, more reliable and faster in operation. To signal the difference between a comment and run of

binary data one could use some form of 'comment header'.

Currently, a number of peripheral standards are under development such as XLL (XML Linking Language), XPath, XPointer, XSL (XML Stylesheet Language) and XQL (XML Query Language). Amongst these are XML schemas, which are a proposed alternative to Document Type Declarations. One reason that these are being promoted is because DTD syntax is inextensible. That is to say that additional markup properties cannot be expressed beyond what the lexicon allows. But wait a minute, are we not talking about the *eXtensible* Markup Language...?

To be a complete heretic, one can argue that a close cousin of C not a subset of SGML should have been developed. Learning XML would then parallel the transition from C++ to Java where the syntax is so similar that an experienced C++ programmer can be up to speed with Java in a matter of days. Instead of having to digest a set of new (and somewhat arcane) semantics, programmers would be able to read and write XML syntax within hours. Moreover, the grammar in which applications were coded (whether C++ or Java) would be near identical to the grammar written and read by those applications. Now that would be *real* unification.
Muddying the waters still further, an initiative has recently been launched to develop a cut down version of XML called Simple Markup Language or SML. Proponents point out that most developers are using only a subset of the full specification and to standardize on that would therefore result in a language that was easier to learn and implement. The proposed SML specification would be the same as XML but with no attributes, processing instructions, DTDs, CDATA sections and so on.

In conclusion, it would seem that Junior is a Curates Egg and very much in a state of flux.
Moreover, we have been here before with Java and have seen discrepancies in virtual machine implementations and proprietary extensions to the specification. XML is already showing similar blemishes with, for example, discrepancies between parsers. An overarching solution can only work if the standard is adhered to by all and to the letter.

The reality however is that given its potential and the industry's response, individuals and organizations alike cannot afford to ignore XML. Indeed, it would appear that, despite the confused picture, companies are already investigating and implementing the XML approach because they fear being left behind. It may have its shortcomings however given the backing of the heavyweights and its potential for the Internet; XML (in whatever form it grows into) is not going to go away.

Richard Vaughan is a consultant and trainer with Ratio Group Ltd., specialising in C++, Patterns and XML. Richard can be contacted at 'richard.vaughan@ratio.co.uk'. Ratio Group can be contacted on +44 (0)20 8579 7900.

P U B L I C S C H E D U L E C O U R S E

RATIO

We Know the Object of…

XML for Software Developers

A Four-Day Hands-On Course
by Richard Vaughan

2 Dates in London, UK

6 – 9 March 2000 · 17 – 20 July 2000

This course will give you a sound theoretical understanding of XML and its related specifications, while providing practical experience in implementing and applying XML within applications. It covers a range of tools, technologies and approaches essential for managing the data interchange requirements of a distributed computer environment.

For more information on this course, contact Ratio on +44 (0)20 8579 7900 or by email at bookings@ratio.co.uk

Please note: class size is limited, so book early!

Object Design Series

Designing Distributed Object Systems

Jason Garbis, author of Enterprise CORBA, discusses the differences between designing stand-alone and distributed object systems…

Introduction

At first glance, it seems that a component-based application should be equally usable whether the components are contained within a single process (local), or distributed across multiple processes (remote). Ideally, we'd be able to migrate from a local component-based application to a remote one in a straightforward manner, perhaps even automatically with the help of a tool or two. However, this is not the case. There is a fundamental difference between local and remote applications, even if they are both component-based. Migrating from one to another requires a creative human element, and designing for one is different from designing for the other. We cannot blindly transfer the object model from one to the other.

That's the bad news. The good news is that designing for local and remote applications are nonetheless similar tasks, with a lot of overlap. In this article, we explore the differences and important things to be aware of. An illustrative way of doing this is to look at an example of a local component-based application, and examine how it would behave if it were magically converted to a remote application. However, before we can do this, we need establish some context for this task.

Let's define a component-based application as an application that makes use of one or more functionally distinct elements (components), each of which is encapsulated inside a well-defined interface. This is why, when we look at a well-designed component application, it is appealing and natural to make some of these functional components accessible remotely, or to centralize this functionality.

If we think about moving from a local, component-based application to a remote one, what we are really talking about here is the differences between a 2-tier architecture and an N-tier architecture. Actually, this is a simplification, since most well-designed applications are internally layered, and can therefore be thought of as having internal tiers, and not just being a 2-tier application. However, the essential difference is that these tiers will now be physically separated into distinct processes, perhaps on different hosts.

This may mean that the server elements will be servicing multiple clients concurrently, which raises a number of additional issues. Alternatively, each client process could have its own dedicated server process. This deployment pattern is only applicable under certain circumstances, and is less frequently used. In general, distributed applications tend to consist of server processes than service multiple concurrent clients.

Sample Application

Let's examine a sample component application, and see what happens if we simply distribute it across servers and hosts.

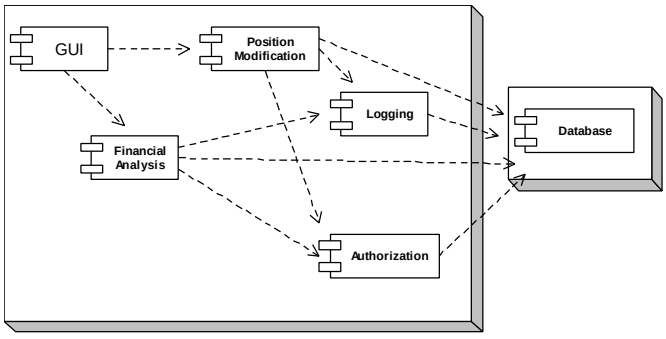


Figure 1. Elements of a simple financial services management application

Traders use the GUI to perform financial analysis (in the Financial Analysis component) and modify the holdings (positions) of their various portfolios in the Position Modification component. These two functional elements make use of the Authorization and Logging components. All the components (excepting the GUI) make use of the database as well. There are multiple traders, and therefore multiple client-side applications running concurrently. The database can handle this, because all access to it is performed within the end user's logon. The position manager's updates are transactional; again the database properly keeps these updates isolated until the client application either completes the transaction, or times out.

Because we did a good job designing our components, and chose a technology that is well-suited to remote as well as local access, we now decide to repackage and redeploy this application, with remote access to our components.

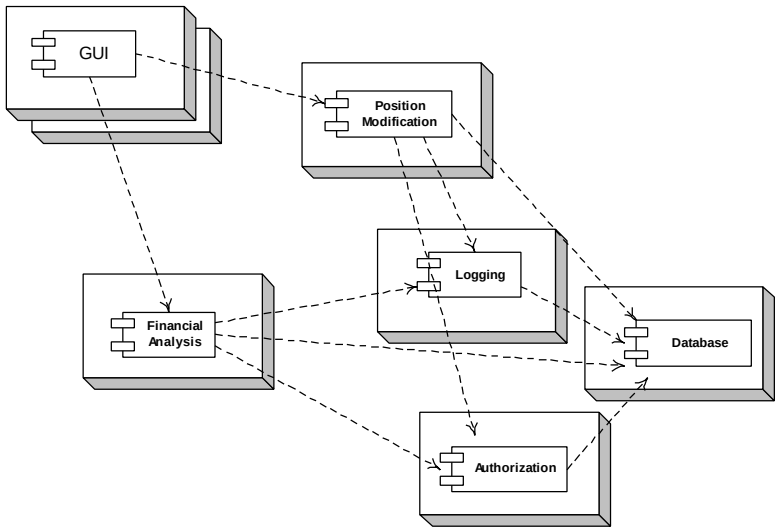


Figure 2. Possible deployment diagram

We now have multiple processing running concurrently in our system, with some form of communication between them. There are many implications of this new deployment: First, the remote components are now "farther away" from each other. This means that rather than making local (in-process) method invocations, all inter-component interaction must use some mechanism to communicate between processes or across hosts. Any such remote invocation will be much, much slower than a corresponding local invocation, so the performance of the system will be slower.

Second, client elements now have to somehow obtain a reference to each remote component at runtime, rather than relying on local references typically resolved at link time.

Third, we have broken up our local, relatively monolithic system into a number of distinct component communicating across a network. Therefore, there are many more things that can go wrong, and we must be able to detect and handle a failure of any of the elements of the system.

Fourth, some of the components are now shared by multiple client processes concurrently. This means that we have to address issues of concurrency, client session management, security, and transactions. Fifth, we may need to duplicate functionality or components in a distributed model. For instance, we may want to have both local and remote logging elements, in order to log greater detail locally than remotely, and to be able to continue logging in the event of a failure.

Wow! That certainly seems complex and dangerous, and may be scary enough to convince us not to develop distributed applications at all! Fortunately, of course, we not typically faced with migrating a local component application to a remote component application, but rather designing these separately. Because the design process for these two tasks is different, we can address all these issues in our distributed system model, rather than addressing them after the fact. That is, we cannot simply translate from a local object model to a remote model. To turn this statement on its head, designing for local access is different than designing for

remote access – it includes the additional step of turning a local component model into one suitable for remote access.

How can we begin to adapt our design process to address the different characteristics of local and remote systems? First, we need to recognize that our object model should properly reflect all our knowledge of our business domain. This seems like an obvious goal; the point is that we want to consider the intended distribution aspects of our system only after our knowledge of the business model is relatively complete. That is, we suggest an additional step in the design process, where we begin to model the system's distribution, and design the exposed component interfaces based on the business objects, the distribution architecture, and the use cases.

Approaches to Handling Distribution

We offer the following general approaches to the distribution step. Remember that we begin with a set of components suitable for local access, and need to modify this so that it results in a component that can be robustly and efficiently accessed remotely.

- Increase the granularity of exposed components
- Make judicious use of non-Object-Oriented elements, such as structures

Let's look at these in turn.

Increase the granularity of exposed components

The object model should contain as many elements as are needed to properly model your business domain. However, this can result in a fine-grained object model; that is a model with many "small" classes, with many attributes or methods exposed across the classes. This may make sense for local access, but may perform poorly for remote access, due to both the large number of remote method invocations that must be performed, as well as the large number of object references that must be obtained and managed by the client. By increasing component granularity, we mean consolidating local classes into a smaller number of remote components, and increasing the amount of work performed for each remote invocation. This has the effect of reducing the number of object references our clients must obtain and manage, as well as reducing the number of remote method invocations performed.

Of course, these tasks must be performed intelligently – ay, there's the rub. Our consolidated component should only be comprised of logically related local classes. Likewise, by deliberately reducing the number of methods exposed, we are changing the flavor of our system, and often offering service-level interfaces rather than attribute-level interfaces. The important thing is to make sure that this consolidation is done with full knowledge of our expected use cases.

Make judicious use of non-Object-Oriented elements

When designing components for remote access, it is important to realize that not everything should be an object. This may seem unappealing to purists, but in practice, it is often a necessary step in order to achieve acceptable performance and scalability. Our intention should be to look at how clients need to access the data associated with our objects, and based on this knowledge, present the data in efficient and concise manner. For instance, consider a class with 10 attributes. In a local application, it may make sense to expose this state via 10 getter and setter methods. For a remote application, such an element would be very inefficient to use, and would be better off representing the state in a single data structure with 10 fields, and only expose a single get and set method.

Another approach is to not expose business, but rather expose manager objects, which manipulate the data associated with the business objects. In this case, clients make invocations on the manager objects, and identify the target object by passing in an Object ID to the manager object. Let's look at an example of performing both these steps.

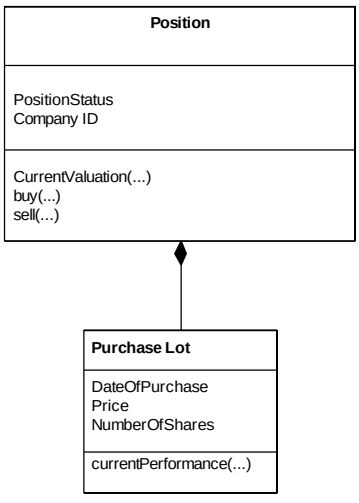


Figure 3. Example of object model performing both steps suggested above.

Both the `Position` and `PurchaseLot` elements are modeled as objects, each having some attributes and methods. In addition (not shown), we will need a `PositionFactory` to create `Position` objects. The effect of choosing this model is that everything is an object. This is conceptually easy to work with, and useful, for a local application.

Let's consider how this model would behave as a remote application. We map this to OMG IDL in a straightforward way, as follows:

```
// OMG IDL
// typedefs, enums, structs and exceptions omitted
interface PurchaseLot;   // forward declaration

typedef sequence<PurchaseLot>lotSeq;

interface Position
{
  attribute PositionStatus status;
  attribute string        companyID;

  Money currentValuation();
  PurchaseResult buy(…);
  SaleResult sell(…);
  lotSeq getLots(…);
}

interface PurchaseLot
{
  attribute Date   dateOfPurchase;
  attribute Money  purchasePrice;
  attribute long   numberOfShares;

  Performance currentPerformance(…);
}
```

These interfaces are rather fine-grained – in order to fully evaluate a position, clients have to obtain references to one `Position` object, and multiple `PurchaseLot` objects, and must make numerous remote invocations. If our use cases are such that this is a common occurrence, our GUI will likely perform slowly. Designing for remote access would lead us to a slightly different model, of coarser granularity. An example of this is shown below.

```
// OMG IDL
// some typedefs, enums, structs and exceptions omitted
struct PositionLot
{
      string lotIdentifier;
      Date   DateOfPurchase;
      Money  Price;
      long   NumberOfShares;
};

typedef sequence<PositionLot>seqPositionLots;

struct Position
{
  string          positionID;
  PositionStatus  status;
  string          companyID;
  Money           currentValue
  seqPositionLots lots;
};
```

```
interface PositionManager
{
  Position getPosition(in string positionID);
  PurchaseResult buy(in string positionID, …);
  SaleResult sell(in string positionID, in LotIDSequence lotsToSell, …);
}
```

With this model, client applications only have to obtain a single object reference – the `PositionManager`. Position information is exposed as a single data structure, which clients can efficiently retrieve in a single invocation on `getPosition()`. Each time a client makes an invocation on the `PositionManager`, it identifies the position to work with, by passing a `positionID` argument. This approach is a bit like a Remote Procedure Call (RPC), but has a number of advantages for remote systems. First, it tends to reduce the number of remote invocations made by clients. Second, it usually reduces the number of object references that clients must obtain. These two advantages are especially true for the common search-and-select-one use case, where clients perform a query and display the results so that the user can select one business object to perform further work with.

Note that this example shows both the coarsening of interfaces (by eliminating multiple interfaces in favor of a manager interface), and the introduction of data structures rather than object attributes. In many cases, we may choose to make use of only one of these approaches, and still obtain significant benefits. For example, rather than eliminating the `Position` interface, we may choose to keep it, and augment it with a corresponding `PositionInfo` data structure. Combining these will allow us to view the business object as data when it's most beneficial (such as in search-and-select, or update), but also treat it as an object when that perspective is beneficial (such as in invoking semantically meaningful business methods, or dealing with relationships between objects).

Going from Remote to Local

What about the reverse task – that of taking remote components and combining them into a single, local process? In general, this step can be performed without modifications to the deign, as long as the required infrastructure functionality is supported locally as well as remotely. Obviously, the platforms and languages must be compatible, in order to combine these components into a single process. In addition, a distributed application may rely on the containing server to supply some infrastructure elements, such as security or transactionality. If components are put into a single process, without a container, such features may be lost. This would be the case, for instance, with an Enterprise Java Bean. It relies on its container to enforce the transactionality and security declared by the bean developer. If a client were able to invoke directly on the bean implementation outside of a container, security and transactionality would be lost.

Conclusion

Designing a component application for remote access involves additional considerations than designing for local access. We suggest an additional step in the design process, where business objects are compared to the anticipated use cases, and modified to expose coarser interfaces and make use of non-object-oriented elements such as data structures. In addition, remotely accessed components must address new issues such as security (authorization and authentication), concurrency, partial failure, or transactions. These issues may have been nonexistent, or relied on implicit solutions in a local environment. In a distributed environment, they must be explicitly addressed. Despite the additional complexity, the benefits of distributed component applications are well-proven and well-worth the effort required. The guidelines outlined here should help you realize these benefits.

Jason Garbis is Principal Consultant with IONA Technologies, and is co-author of 'Enterprise CORBA', published in 1999 by Prentice Hall Professional Technical Reference.
Jason can be contacted at 'jason.garbis@iona.com'.

Object/Component Architecture Series

Product Line Architectures

Jan Bosch, author of the forthcoming book 'Design and Use of Industrial Software Architecture' discusses software product lines…

Introduction

Achieving reuse of software has been a long standing ambition of the software engineering industry. Every since the 1960's, the notion of constructing software systems by composing software components has been pursued in various ways. In the first proposals, components contained functions that could be reused by application programmers, whereas, with the emergence of procedural programming languages, the level was lifted to modules. Although modules could be of a considerable granularity, they were intended to be reused without adaptation which reduced their usefulness drastically. The subsequent paradigm, i.e. object-oriented programming, provided a solution to that problem by representing modules as classes that could be inherited and extended by subclasses. However, the granularity of classes is limited, which lead to the definition of object-oriented frameworks. A framework was traditionally intended to provide the basis for application development, but, during recent years, frameworks are used increasingly often as coarse-grained components.

Component-oriented programming has been an emerging trend during the 1990s. Most proposals to achieving component-based software development assume a market divided into component developers, component users and a market place. However, this proved to be too ambitious for most types of software. This has been identified during recent years and there has been a shift in focus from *world-wide* reuse to *company-wide* reuse of components. Parallel to this development, the importance of an explicit design and representation of the architecture of a software system for the fulfilment of the quality requirements of systems has become increasingly recognized.

The combination of the discussed concepts and developments lead to the definition of *software product lines*. A software product line consists of a product line architecture, a set of reusable components and a set of products derived from the shared assets. Among others, one of the important differences between traditional approaches to reuse and software product lines is that it is explicitly recognized that adopting a software product line approach has organizational, process, technology and business aspects, i.e. it affects the complete software development organization. The success from early adopters, including several large european industries, in the context of EU ESPRIT, i.e. ARES and PRAISE, or ITEA/Eureka, i.e. ESAPS, projects, and otherwise, has shown that this technology has the potential of creating pervasive reuse in real industrial contexts; something that traditional approaches to reuse did not succeed with.

The notion of software product lines is the topic of this article, which will only provide an overview over the concepts and issues associated with this approach. For a more extensive discussion, I refer you to an upcoming book from Addison-Wesley [Bosch 00]. In the remainder of this article, we first introduce the notion of software product lines and especially the phases and processes associated with the development and evolution with it. Then we discuss some organizational models for software product line based development and we describe some experiences from companies that have adopted this approach.

Software Product Lines

Software product lines present a highly promising approach to achieving reuse of software within an organization. In figure 1, the main components of a software product line are shown. The main component is the product-line architecture, i.e. the software architecture that captures the commonalities between the products in the product line, while supporting the differences between the various products. The second concept is the component set that contains implementations for the architectural components defined by the product line architecture. These components may be traditional code modules, but also object-oriented frameworks are frequently used. The third main asset in the software product line is represented by the products derived from the reusable assets. For each product, first a product-specific software architecture is derived from the product line architecture. Then, suitable components are selected from the component set and instantiated with product-specific information and, where necessary, code extensions. If necessary, product specific code is developed for the product requirements not supported by the reusable assets. Finally, the product architecture, the instantiated components and the product-specific code is integrated to form the final software product.

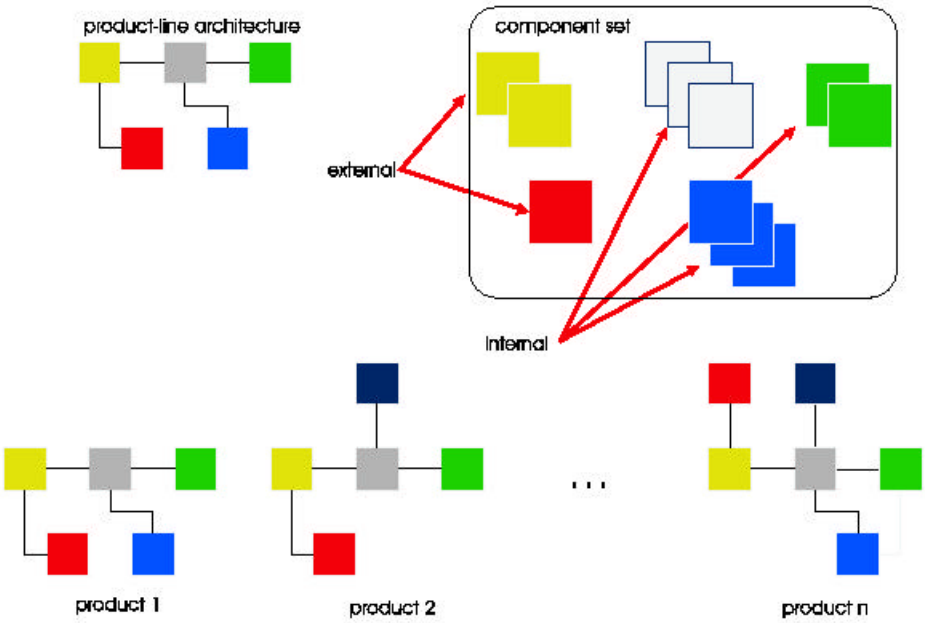


Figure 1. Overview of the main software product line assets

In the sections below, we discuss the development, deployment and evolution of a software product line in more detail.

Development

When the decision to initiate a product line has been taken, the first step is the development of a software architecture for the product line that supports the functional and quality requirements of the systems included in the family. This phase includes activities such as scoping, commonality and variability analysis, architectural design and verification.

Once the product line architecture has been designed, the subsequent phase is the development of the components that make up the common part of the system family. We discuss two types of components, i.e. traditional components as presented by contemporary literature, e.g.

[Szyperski 97], and object-oriented frameworks. In our cooperation projects with various industrial software development organizations, we have seen an increase in the use of object-oriented frameworks as components in product line architectures. One reason for this is the high level of configurability that is available within an object-oriented framework. This suits the notion of system-families very well, since it allows for the easy configuration of components for individual systems.

The development of systems based on the product line architecture and components could be viewed as the third phase in development. However, since the focus of this phase is on software architectures and reusable components, we discuss system development in the next part, i.e. deployment.

Designing a product line architecture

The design process of the product line architecture consists of six main steps, i.e. business case analysis, scoping, feature and system planning, product line architecture design, component requirement specification and verification. Below each of the steps is briefly summarized:

- **Business case analysis**: This analysis is concerned with establishing, at a sufficient level of certainty, that the product line approach represents a cost-effective and superior approach. In addition, the analysis provides data for deciding on a revolutionary or evolutionary path to convert from a product-based to a product line based organization of software development.
- **Scoping**: This activity determines the systems and the product features that are included in the product line. It may not be beneficial or even desirable to include all systems and features in the product line; especially not from the start.
- **System and feature planning**: The focus of this step is on identifying the characteristics of subsequent versions of the product line architecture. Since there will be continuous development of the product line architecture in terms of the features and systems it supports and incorporating new, but anticipated, features is generally considerably easier than

unanticipated features, it is important to develop a plan for feature incorporation.

- **Product line architecture design**: The main step in the process is, obviously, the design of the software architecture for the product line. In [Bosch 00], we present the Quality Attribute-oriented Software Architecture (QASAR) design method. QASAR consists of three main phases, i.e. functionality-based architecture design, architecture assessment and architecture transformation. The latter two phases are performed iteratively until the software architect is confident that especially the quality requirements are fulfilled. In addition, the software architecture design process should maximize the possibility to derive product architectures from the product line architecture.
- **Component requirement specification**: The software architecture dictates a set of components that implement the required behaviour. This activity is concerned with specifying the requirements for each of the components. The requirement specification defines the interfaces, the functionality, the quality attributes and the variability that the each component should support.
- **Verification**: Finally, before entering the next activity, i.e. component development, it is important to verify that the product line architecture supports the requirements specified by the stakeholders. This can either be established by a stakeholder meeting or by architecture assessment teams that perform an external evaluation.

Developing reusable components

The component development process is rather constrained due to the aspects, rules and constraints imposed by the software architecture and the, potential, presence of legacy code that may need to be incorporated. Components, in our definition, should support three types of interfaces, i.e. provided, required and configuration interfaces. The provided and required interfaces are intended for the interaction with other components. The provided interface presents the operations that can be invoked on the component, whereas the required interface specifies what components and operations the component requires access to for correct operation. The configuration interface is intended for use by the user of the component. When constructing an application or system based on reusable components, each component needs to be instantiated and configured. The configuration consists of defining, for each variation point supported by the component, what concrete variation should be used in this component instance. Each configuration interface provides access to a variation point and allows the software engineer to control it.

An important aspect of component development is to design the configuration interface. A number of configuration mechanisms are available to the software engineer, e.g. inheritance, extensions (especially providing hooks), configuration (selecting and arranging predeveloped pieces on functionality), template instantiation and generation (generating an instantiated component based on an input specification).

Finally, the companies that we work with generally use object-oriented frameworks as components in their software product line. The advantages of using object-oriented frameworks is that it allows for coarse-grained components and yet supports considerable variability for each instantiation of the framework. In [Bosch 00], we distinguish a number of different approaches to using object-oriented frameworks, i.e. the system-specific extension model, the standard specific extension model, the fine-grained extension model and the generator-based model.

Deployment

The software architecture and set of reusable component that are part of the product line are deployed during system development. The intention is that the effort required for the development of systems should be decreased drastically by using the architecture and component as a basis for the system. The instantiation of systems based on the reusable assets in a software product line consists of seven steps, i.e. requirement specification, system architecture derivation, family component selection and instantiation, development of product-specific components, system integration, system validation and, finally, system packaging and release. Below, a selection of these steps is discussed.

Software architecture derivation is concerned with configuring the product line architecture based on the system requirements, leading to a product software architecture. The configuration includes the addition and removal of components and relations in the architecture.

Component selection and instantiation is concerned with the product line components that will be used in the system under development. The instantiation may consist of two parts, i.e. straight-forward configuration of the component and the development of product-specific component extensions. Configuration includes activities such as parameter setting and defining input specifications to code generators. The development of product-specific component extensions occurs typically when using white-box framework as a component. In that case, the component architecture and its generic behaviour have been defined, but the

system-specific code needs to be added by the software engineers developing the actual system.

Finally, it may be necessary to develop product-specific software. The product line components do not necessarily implement all product requirements. The subset of the requirements that is not fulfilled by the product line must be implemented by product-specific software. This software is not part of the product line, but only included in the source code for the specific system. When using the evolutionary approach to developing a product line, the product- specific software may provide useful hints about likely useful extensions to the functionality supported by the product line.

Evolution

Once the first versions of the product line architecture, the set of components and the set of systems have been developed, the evolution of all these assets will become the primary activity. Evolution is, up to some extent, similar to development, but the presence of assets is a major complicating factor. The evolution is initiated by new requirements on existing systems and by new systems that need to be incorporated in the product line. Evolution caused by the new requirements takes place on all assets, i.e. architecture evolution, component evolution and system evolution. Architecture evolution is concerned with changes to the components that make up the product line architecture, changes to the relations between these components, etc. Component evolution is concerned with the incorporation of new and changed requirements on the component functionality which generally affects the component internals, but may also affect the component interface, which causes effects on the architectural level.

Product evolution may express itself in two ways. Traditionally, systems evolve through subsequent versions that incorporate new requirements. During recent years, a new type of system evolution can be identified, i.e. run-time evolution. Systems or products that have been shipped to customers can be upgraded with new components or new versions of existing components. However, each individual instance of the system may have its own configuration of older and new component versions. Run-time evolution is also referred to as dynamic architecture.

Organization

We have identified four organizational models for software product lines. Below, we discuss, based on our experiences, the applicability of the model, the advantages and disadvantages and an example of an organization that employs the particular model.

- **Development department**: In this model software development is concentrated in a single development department, no organizational specialization exists with either the software product line assets or the systems in the family. The model is especially suitable for smaller organizations. We have seen successful instances of this model up to 30 software engineers. The primary advantages are that it is simple and communication between staff members is easy, whereas the disadvantage is that the model does not scale to larger organizations.
- **Business units**: The second type of organizational model employs a specialization around the type of systems in the form of business units. The business units share the product line assets and evolution of these assets is performed by the unit that needs to incorporate new functionality in one of the assets to fulfil the requirements of the system or systems it is responsible for. Three alternatives exist, i.e. the unconstrained model, the asset responsibles model and the mixed responsibility model. The model is often used as the next model in growing organizations once the limits of the development department model are reached. Some of our industrial partners have successfully applied this model up to 100 software engineers. An advantage of the model is that it allows for effective sharing of assets between a set of organizational units. A disadvantage is that business units easily focus more on the concrete systems rather than on the reusable assets.
- **Domain engineering unit**: In this model, the domain engineering unit is responsible for the design, development and evolution of the reusable assets, i.e. the software architecture and the components that are make up the reusable part of the software product line. In addition, system engineering units are responsible for developing and evolving the systems built based on the product line assets. The two alternatives include the single domain engineering unit model and the multiple domain engineering units model. In the latter case, one unit is responsible for the product line architecture and others for the reusable software components. The model is widely scalable, from the boundaries where the business unit model reduces effectiveness up to several hundreds of software engineers. One advantage of this model is that it reduces communication from n-to-n in the business unit model to one-to-n between the domain engineering unit and the system engineering units. Second, the domain engineering unit focuses on developing general, reusable assets which addresses one of the problems with the aforementioned model,

i.e. too little focus on the reusable assets. One disadvantage is the difficulty of managing the requirements flow and the evolution of reusable assets in response to these new requirements. Since the domain engineering unit needs to balance the requirements of all system engineering units, this may negatively affect time-to-market for individual system engineering units.

- **Hierarchical domain engineering units**: In cases where an hierarchical product line has been necessary, also a hierarchy of domain units may be required. The domain engineering units that work with specialized product lines use the top-level assets as a basis to found their own product line upon. This model is applicable especially in large or very large organizations with a large variety of long-lived systems. The advantage of this model is that it provides an organizational model for effectively organizing large numbers of software engineers. One disadvantage is the administrative overhead that easily builds up, reducing the agility of the organization as a whole, which may affect competitiveness negatively.

Experiences

Finally, we have recognized a number of additional factors that influence the organizational model that is optimal in a particular situation. These factors include geographical distribution, project management maturity, organizational culture and the type of systems.

We have studied and cooperated with several companies that employ a software product line approach. We discuss experiences of these companies and the issues that have been encountered by the staff at these organizations. These experiences and issues have been organized into organizational, process and technology issues.

Organizational topics that need to be addressed include, among others, the increased amount of required background knowledge by software engineers, the lack of management support for long term goals, the questioned need for domain engineering units, the difficulty of selection the appropriate organizational model, the time-to-market pressure against the quality of the reusable assets and the lack of economic models.

Process issues that were identified by the companies involved in the case studies include the importance and difficulty of information distribution between business units, the difficulties associated with maintaining up to date and accurate documentation, effort estimation problems, especially when designing reusable assets, and the scoping of the software product line.

Several issues related to technology were identified, including the need for multiple versions of reusable assets, the increasing number of implicit dependencies between components during evolution, the difficulty of using components in new contexts, the lack of appropriate tools support, feature scoping, early intertwining of functionality and the lack of encapsulation boundaries and required interfaces.

Concluding, software product lines can and are successfully applied in software development organizations, ranging from small to large. These organisations that we studied are struggling with a number of difficult problems and challenging issues, but the general consensus is that a software product line approach is beneficial, if not crucial, for the continued success of the organisations.

Conclusion

Software product lines are present an approach to achieving pervasive, company-wide reuse of software assets. Different from earlier proposed approaches to achieving software reuse, software product lines are already successfully applied by a variety of software development organizations. The principles underlying software product lines do not only apply to companies developing products, but also to software consultancy companies and IT departments. The advantages of using software product lines include drastic decreases in software development and evolution cost, time-to-market and staff numbers. However, adopting a software product line approach must be a strategic decision because it affects the business model, the organization, the processes and the technology associated with software development.

In this article, we have presented an overview of the main processes and issues associated with software product lines. The main steps while adopting a software product line approach include the design of a software architecture for the product line, the development of the reusable components and the derivation of the products that are part of the family. Once the software product line has been initiated, the evolution of all the aforementioned assets becomes the main challenge. We have discussed a number of organizational models that can be adopted for software product lines. Finally, the experiences collected from a number of companies that have applied software product line principles for several years have been presented.

References

[Bosch 00] J. Bosch, *Design and Use of Industrial Software Architectures* (working title), Addison Wesley Longman (forthcoming), ISBN 0-201-67494-7, June 2000.

[Szyperski 97] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

Jan Bosch is Professor of Software Engineering at the University of Karlskrona/Ronneby in Sweden. He is author of 'Design and Use of Industrial Software Architectures' to be published June 2000
Jan can be contacted at 'Jan.Bosch@ipd.hk-r.se'

R A T I O I S P R O U D T O S P O N S O R

The 1st International Conference on

eXtreme Programming and the Flexible Processes in Software Engineering

XP2000

Cagliari, Sardinia, Italy
21 - 23 JUNE 2000

KEYNOTE PRESENTATIONS TO INCLUDE:

Kent Beck
Extreme Programming For Beginners

Martin Fowler
Refactoring Techniques

Ron Jeffries
User Stories and the Planning Game

Ralph Johnson
Framework Development and XP

Alistair Cockburn
Designing a Light Methodology

as well as…

WORKSHOPS • PANELS • TECHNICAL PAPER PRESENTATIONS
Led by Software Development Lab Responsibles of Daimler Chrysler, Ericsson, Hewlett-Packard, Motorola, Siemens, Sun Microsystems, etc.

Visit **http://numa.sern.enel.ucalgary.ca/extreme**
for full conference program and registration details

Object/Component Architecture Series

The Topsy Turvy World of UML

Hubert Matthews and Mark Collins-Cope discuss a visual metaphor mismatch that inhibits OO designers thinking architecturally.

Diagrams are a common way of writing down and communicating ideas. We use them for all sorts of things – road maps, wiring diagrams, UML class diagrams, etc. One common feature of diagrams is that all parties must understand the meaning of the symbols being used – blue roads represent motorways on most road maps, for instance. Communication is difficult without this shared knowledge, as one has to ask what a symbol means, how is it different from some other symbol, and so on. Thus at first we need some form of legend or rubric to help us decipher the symbols.

Once we have assimilated the symbols we can start to search for their meaning. This involves understanding how and why they are connected in that particular way. One great aid to comprehension is to have standard conventions – maps traditionally have North towards the top, circuit diagrams have inputs on the left and higher voltages at the top. This is what we're used to and it helps to orient us. Breaking these conventions slows us down and is usually counter-productive.

What happens, however, if two related diagramming conventions contradict each other? Let's look at UML class diagrams and architectural diagrams.

UML class diagrams, used to show the static structural aspects of object-oriented designs, traditionally have superclasses towards to the top of the page and subclasses drawn underneath them, as follows:

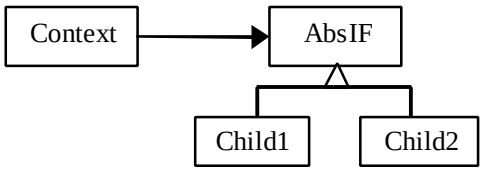


Figure 1. Class diagram showing inheritance relationship

The inheritance relationship in the class diagram (Figure 1) is sometimes referred to as *specialisation*. This is a clue to one of the conventions of the class diagram: more specialised things are shown towards the bottom of the diagram. The reasons for this convention are probably rooted in the way we talk about them: "the AbsIF is at a *higher* level of abstraction, its *sub*classes at a *lower* level."

Architectural diagrams, such as in Figure 2 below (an older notation), sometimes used to show the layering of the software in a system, show the most application specific layers towards the *top* of the diagram, and we talk about them accordingly: "this software is built *on top of* our persistence framework, which is in turn built *on top of* the *underlying* platform software…"

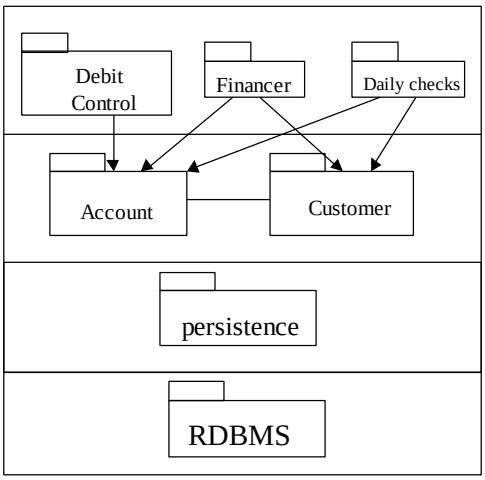


Figure 2. Layered Architecture

Now this would be a big "so what" if it were not for one important fact. *The two types of diagram show us different views (or levels of detail) of essentially the same thing: the structure of our software*. Let's try super-imposing a UML class diagram and an architectural layering diagram on top of each other (with thanks to the Gang of Four's Factory Method pattern for the example):

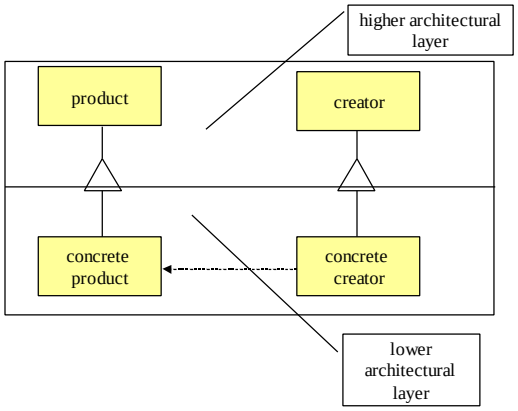


Figure 3. Super-imposed UML class diagram and architectural layering diagram

Something is clearly wrong here. We've got the abstract classes in the more application specific higher architectural layer - when in fact they should be in the *lower* layer - being more generic and re-usable than their concrete derivatives; and the software dependencies go upwards, rather than downwards as implied by the standard architectural convention.

Reworking the diagram, reversing the common UML convention, we get the following:

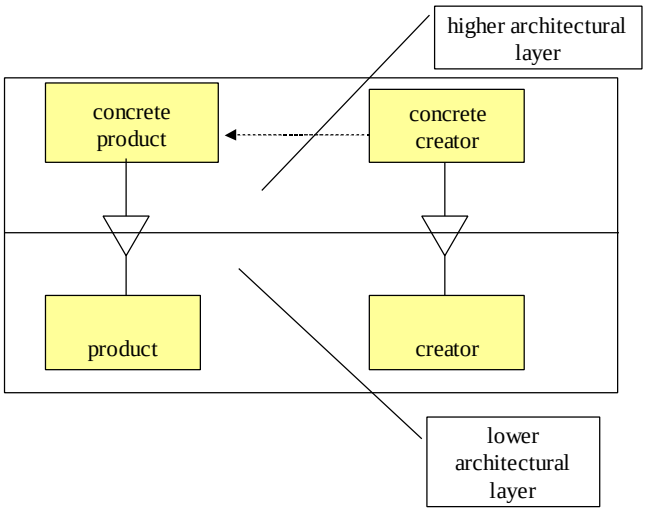


Figure 4. Figure 3 reworked, reversing the common UML convention

The two diagram types can now be seen to complement and reinforce each other perfectly. At the lower architectural level (perhaps a level indicating re-usable infrastructure) we have the abstract product and creator classes. At the higher, more application specific (and less re-usable) level, we have concrete realisations exploiting the infrastructure provided. It all makes perfect sense - and we may even have gained an additional clarity on why we were using the factory pattern in the first place!

So, two conventions, both pervasive in the industry, are clearly at direct odds with each other - causing a visual metaphor mismatch that we believe inhibits broad architectural thinking during design. Since adopting the *highest is most specific* convention when using class diagrams, we have found a lot of our design and architecture thinking has clicked rather neatly into place, visualising and organising architectural dependencies has become easier, and we've got a much clearer view on what it means - architecturally - for a component or package to be re-usable. Copernicus would have been proud :-)!

Mark Collins-Cope and Hubert Matthews undertake design and architecture consultancy for Ratio Group. They can be contacted on +44 (0)20 8579 7900 or by email at 'markcc@ratio.co.uk' or 'hubert@ratio.co.uk' .

The diagrams presented above are taken from Ratio's 2-day Component-Based Development using UML training course. Further details of this course can be obtained by emailing 'info@ratio.co.uk' or calling the contact number above.

Development Process Series

OPEN is the Objective

Author Brian Henderson-Sellers introduces the OPEN Development Process…

What is OPEN?

OPEN is a *third-generation, full lifecycle, process-focussed, object-oriented methodological approach* that is ideally suited for *component-based development* as well as object-oriented software development. OPEN stands for Object-oriented Process, Environment and Notation and is in the public domain. OPEN in fact defines a process framework that can be (and is meant to be) tailored to specific projects, specific organizations, specific skills sets and so on.

Let's take each of those italicized adjectives in my first sentence in turn and return to the framework issue at the end.

OPEN is an *object-oriented (OO) methodological approach*. Whilst there is some discussion about exactly what a software development methodology is or should be, it is generally agreed that it should encompass rules, suggestions, heuristics, guidelines etc. for building software systems. Indeed the number of elements and its structure can be quite complicated *(Figure 1)*.

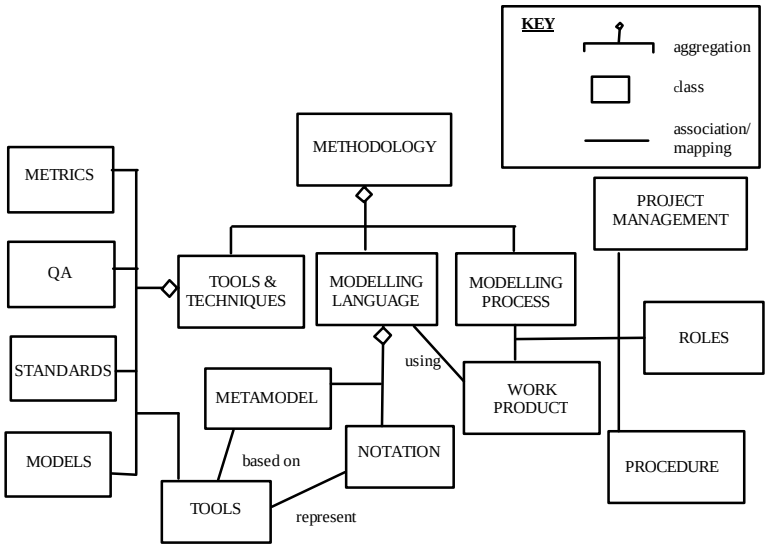


Figure 1. What is a method? *(redrawn from Graham et al., 1997)*

Many methodologies focus very much on the rules, tips and techniques for modelling but do not address the larger scale issues of people (roles in Figure 1), process, project management, quality assessment, metrics, standards and so on. A good methodology should do all of these.

Methodologies, often simply called methods, have been around a long time; but it is only since around 1990 or so that they have been available to support object-oriented development. An object-oriented approach relies on the notions of abstraction, strict modularization & information hiding and polymorphism. Requirements, design and code all use the same model of the "object" which encapsulates together state and behaviour with a tightly controlled interface. The initial emphasis is always on the "what" not the "how" within the development lifecycle. With this more holistic view of modelling and software design, it is beneficial to ask about the high-level responsibilities that an object has: responsibilities for doing, for knowing and for enforcing. In later design and coding, these are translated into methods of the classes. Whilst using a responsibility-focussed approach is found useful, object orientation can also give good results

using a data-driven approach or a use case driven method, depending on the particular problem and problem domain.

OPEN is a *third-generation OO methodology*. The first OO methodologies, created about 1990, were primarily tips and techniques rather than true methodologies. However, they are often referred to as first generation OO methodologies. They were constructed by individuals or small groups. They were used in pilot projects in industry and in teaching. By about 1994 two things were happening. Methodological writing increased so that the original methodology developers of 1990, along with an increasing number of others, started to read widely and incorporate good ideas from a wide variety of sources in their own methodologies. The methodologies began to look more and more like each other, supporting common ideas and common principles. Once published, these became known as second generation methodologies. Typical examples are Booch's 1994 approach, the SOMA approach and MOSES (the last two being direct precursors of OPEN). Despite the publication and teaching of these second-generation methodologies, it soon became clear to these methodologist authors that the software industry in general still seemed reticent to adopt their ideas. One reason was perceived to be the lack of support in the sense that second-generation methodologies were still under the control of only one person or, at best, a small group of people. Furthermore, CASE tool vendors found it difficult to support such a large number of methodologies. In addition to good tool support, for largescale industrial usage, an industrial user needs to be sure that, if one provider of support vanishes, there will remain alternative sources. This led to the active collaboration of methodologists in order to create third-generation methodologies. Begun in late 1994, OPEN is the premier example of a third-generation methodology, created by the collaboration of over 30 methodologists, researchers, tool vendors and practitioners worldwide.

OPEN is a *full lifecycle methodology*. Software can be considered to have a lifecycle from birth to death. The need for software can arise when business problems need solution. So the first step (the birth) occurs when a business problem is identified. This is a problem which must be clearly enunciated. And although a software solution is not mandated at this stage, for those problems that do lead to a software solution, the requirements engineering activity, which focusses on the elucidating the business problem, is clearly a vital part of the software lifecycle. Business decision making, requirements engineering and systems analysis are all "early lifecycle" activities. OPEN includes tasks and techniques which are useful in these early stages. Few other OO methodologies pay more than lip service to these more business-focussed issues. Yet in the real world, if technology (here object technology or OT) is to be relevant to commercial environments, an OO method must consider these early lifecycle issues and not just assume that the lifecycle begins with the handing over of a clearly and uniquely defined requirements definition to the software developer.

Similarly, a methodology should cover the late lifecycle activities. Whilst most are good at program design and coding, they tend to tail off in their coverage of issues such as deployment and user training and future enhancements/maintenance. It is just as important that a method addresses these issues, perhaps using testing metrics to do fault detection and usability studies to evaluate customer acceptance of the delivered product, for instance.

OPEN is a *process-focussed methodology*. Process is the key to good software development practices. It imposes order and rigour. A process, of any sort, tells you how to take certain steps in order to accomplish a specific task or goal - to get something done. Taking steps involves ordering those into some sequence because we live in a temporal universe and, as individuals, do not live concurrent lives. A process offers a repeatable and manageable underpinning to software development. It has been called "documented decision making" and equated to workflow description.

Different problems, architectures, people, organizations etc. need different process models - for example, waterfall, spiral and recursive/parallel; whether the project is the first of its kind or a variation on a theme; whether the focus is a one-off development or whether the creation of reusable assets is of major concern.

Some software development seems to occur in a very ad hoc fashion. When successes occur, the underlying reason is not obvious and there is no means to identify how to repeat the success. And conversely, when failures occur (as they inevitably will in an ad hoc development shop), there is no way of identifying how to fix the process and learn from the failure and avoid a repeat failure in the future.

A process of any sort lays down some guidelines to help developers set their own (personal and team) standards that they can follow. It is then possible for other personnel to temporarily or even permanently take over a role and for managers to control, monitor and evaluate how well the development is progressing towards completion. A process thus identifies activities that need to be done, probably recommends means by which to achieve these goals

and, most importantly, creates a sequence (or a set of sequences) which allows temporal planning.

Processes may be in an individual's head or may be written down as an organizational (or international) standard to be followed on each project. They may be large or small; "authoritarian" or flexible. Perhaps of greater importance is whether the process is sufficiently mature to provide for repetition. In other words, if I say I am applying a process, do I apply it the same way the second time and would a second person applying this same process have the same result as me? The different answers to these questions are ably captured in the five-level framework known as the Capability Maturity Model or CMM. This framework evaluates the maturity of an organization's processes. Level 2 organizations utilize processes (Level 1 does not) but these processes are not written down and therefore highly individualistic. CMM Level 3 is generally regarded as the lowest level in which the use of process even approaches being satisfactory (from a software engineering viewpoint). At Level 3, the process is institutionalized and used on each project. It is documented and is repeatable. When we talk about OO process, we generally assume that the organization undertaking the OO software development conforms to at least CMM Level 3. On the other hand, an organization at Level 1 or 2 could raise themselves to Level 3 by adoption of the types of OO process discussed here.

Iterative, Incremental and Parallel Lifecycle Processes

In an OO development environment, many of the traditional process and associated project management techniques are applicable. However, there is one major constraint that can alter this. This is the recognition by all OO developers, consultants and mentors that the process lifecycle for an OO development must be: (i) iterative (ii) incremental and (iii) parallel.

An iterative lifecycle is one which occurs several times. In contrast, the traditional waterfall lifecycle dictates that you follow a number of steps (often called phases) sequentially and once any given step is completed - it is never returned to. In an iterative lifecycle, there is often some sequentiality but, after steps are completed, they are often returned to for a further iteration. Iterations are thus "circular" - although this is no excuse for rapid prototyping and hacking. Iterations need to be planned and to go across all lifecycle stages (user requirements elicitation, analysis, design, code and test).

Incremental delivery is linked with the iterative approach to some degree in that an OO development should deliver products to the users incrementally, usually at the end of each iteration, possibly every few weeks. Incremental delivery keeps the customer in the loop, ensuring that they always have in their possession a delivered and running version which is at worst a few weeks old. They can thus give immediate feedback rather than waiting for a one-time delivery of the full system perhaps as much as three years after they first made the request for its development.

Finally, OO supports a parallel lifecycle in that the full software system awaiting development can be easily broken down into packages or subsystems. Because of the high degree of modularity supportable in an OO development, it is relatively easy to ensure that these several packages can be developed essentially independently of each other.

OPEN's Process Framework

As a third generation OO methodology, one of the prime purposes of developing OPEN was to provide a useful and usable "standard" software development process. OPEN has many elements: process, modelling, management, measurement and so on. There are two levels of process in OPEN: the Software Engineering Process (SEP) and the modelling process *(Figure 2)*

EXCELLENCE IN SALES AT RATIO GROUP

Our mission - to be the U.K. brand leader for Object-Oriented related services. To achieve this we need to take on more high calibre sales staff. Current vacancies include:

- **Training Sales Executives** to sell our OO related training products to both new and existing customers. Sales experience essential; exposure to OO or similar technologies highly desirable.
- **Recruitment Executives**. Some exposure to OO is desirable. Experience in IT recruitment is essential.

Positions are based in Ealing, West London. We'll pay a competitive base salary, a good OTE (£45 to £50K) based on realistic targets, and we have no earning cap on commission.

For further details, or to submit CVs please contact Kate Harper on +44 (0)20 8579 7900 or email her via kate@ratio.co.uk

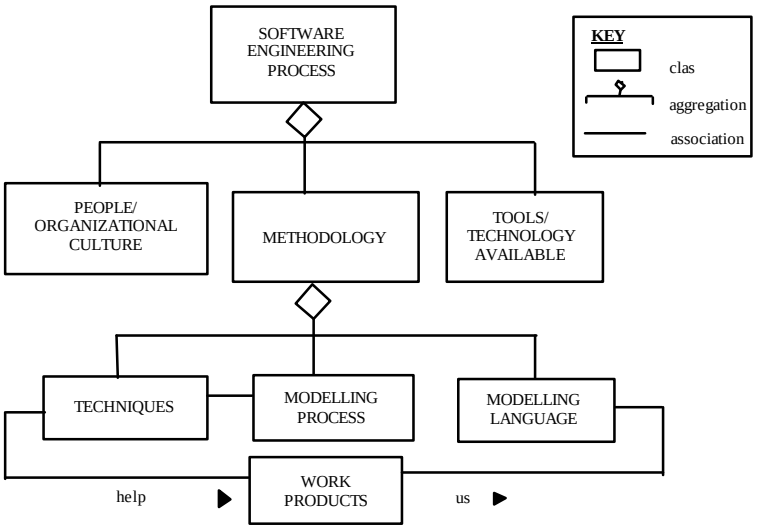


Figure 2. A software engineering process encompasses methodology, people/organizational culture and tools & technology available. In turn, methodology consists of, primarily, lifecycle process, techniques and representation.

The modelling process assists with identifying how things change with time and what work products should be created and when. It is a central part of the methodology, which is objective (i.e. independent of people) and can be written down. The modelling process is just one component of the methodology.

The second process element of OPEN is the Software Engineering Process or SEP which brings together the methodological element in the context of one or more individuals in the team, as well as taking into account organizational culture and organizational standards and the technology available. This is the "real" process in the sense that if this fails so does the project and ultimately the product. Whilst a modelling process can be codified, it relies on real people to make it effective. Different people have different skills sets and varying experience. The organizational culture also has a bearing here. Using a process that is very authoritarian in an organization that is very collegiate in its culture can be a disaster.

Similarly, the effect of available technology is evident. If the project mandates a high degree of traceability and version control then a tool that does code generation and reverse engineering may be called for - for instance, using a drawing tool that only supports data flow diagrams makes it difficult to design with an OO mindset.

There are thus a myriad of variables in any software development project: tools, programming languages, people, processes, quality goals, size and so on. It is not possible to use a one-size-fits-all SEP. Larger projects require more project management; smaller projects can compress the timescales of requirements analysis/design/code into days or even hours without the need for detailed project management. University projects have needs for intensive activities interspersed with relative lull. This is where the tailorability of OPEN comes to the fore.

Clearly, a single, "out-of-the-box", pre-tailored process is inadequate. What is needed is a process framework which establishes the overall architecture of the process while still permitting choice at the detailed level. Making those choices and constructing one specific OPEN-compliant process is called process tailoring or process engineering. It permits one process framework to be instantiated to create several project-specific and tailored processes.

When discussing a methodology, it is common to divide up the lifecycle which the methodology advocates into chunks. Traditionally these have been called phases. So we talk of the "analysis phase" or the "design phase". For a linear lifecycle, like the traditional waterfall model, this is fine. "Phase" sounds like you complete phase n before progressing on to phase $n+1$. You can then factor in milestones, work products, test criteria etc. quite nicely at the end of each of these several phases.

However, if we adopt a more flexible lifecycle model, such as the spiral, fountain or contract-driven lifecycle (OPEN will support any of these but favours a contract-driven lifecycle), then we are better able to support those difficult-to-manage aims

of iteration, incremental delivery and parallel development. These are agreed as being the optimal way of building OO software. Once we allow for iteration, we must permit the development team to move on from design to analysis i.e. apparently a "retrograde step". The word phase now seems inappropriate. One commonly used word (and one used in OPEN) is "Activity" *(Figure 3).*

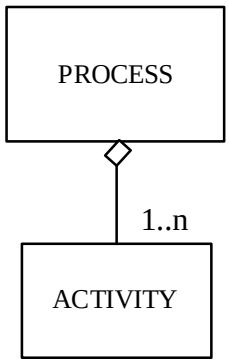


Figure 3. A Process can be modelled as being made up of a number of Activities.

Activities are, like phases, chunks focussed on doing things. In addition, they say "what" is to be done but not how it is to be done. On the other hand, they lose the temporal sequencing implication of phases so we now need to add back somewhere the notion of sequencing rules. In OPEN, when using a contract-driven lifecycle, these are added as pre- and post-condition on the activities thought of as objects. This applies the programming by contract ideas to the very description of the process and results in a process that is itself object-oriented.

Activities are generally largescale descriptions of what is to be done. They are longterm objectives but are difficult to manage because of their potential duration. To manage the "what", a finer discrimination is needed. In project management parlance, a Task is the smallest unit of work which can be evaluated as either complete or not complete. Tasks are thus smaller scale "jobs to be done" associated with each of the activities in the lifecycle. But tasks don't say "how" the jobs are to be done. This is the role of the Technique *(Figure 4).*

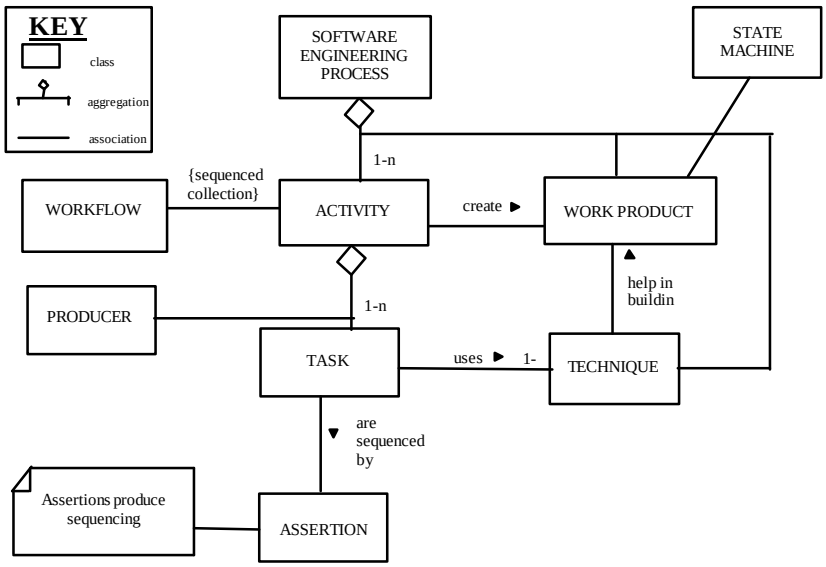


Figure 4. The SEP has many elements: including Activities, Tasks, Producers, Work Products and Techniques.

The technique describes in full detail how we might use object technology, object-oriented concepts and years of experience of users to accomplish the Tasks we have set ourselves. Since techniques are just ways of doing things, they can be thought of as the "tools of the trade". Just as the tools of the trade of a plumber include hammers, screwdrivers and wrenches, the tools of an object technologist include knowledge about the use of, for instance, CRC card modelling, aggregation modelling, OO team building [There are over 150 techniques documented in the book The OPEN Toolbox of Techniques so we won't list them all here.]. Similarly, the value of a plumber is the knowledge and experience in choosing which of the screwdrivers and wrenches are needed to solve any particular problem. Just so for the object technologist who needs to be able to choose the best

OO technique to accomplish the specific OPEN Task being worked on. Choosing the correct technique is largely a matter of experience, real or surrogate. The latter is encapsulated in the OPEN tailoring matrix shown in *Figure 5*.

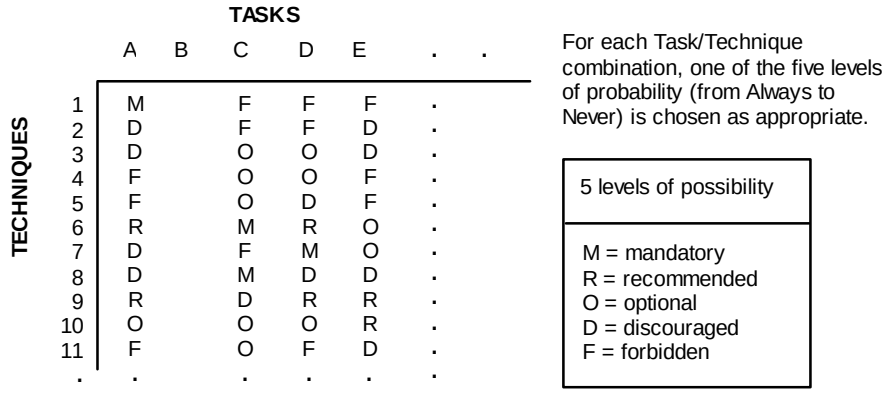


Figure 5. Matrix. *A core element of OPEN is a two-dimensional relationship between tasks and techniques. Each task may require one or several techniques in order to accomplish the stated goal of the task; and techniques may be applicable to several tasks. For each combination of task and technique, an assessment can be made of the likelihood of the occurrence of that combination. Some combinations can be identified as mandatory (M), others as recommended (R), some as being optional (O), some are discouraged (D) but may be used with care and other combinations that are strictly verboten (F = forbidden). Filling in the matrix values is an important part of the lifecycle tailoring Task in OPEN (adapted from Graham et al., 1997)*

This matrix [A similar matrix is used to link Activities and Tasks.] represents the reality that it is not just one Technique that is useful for each Task. Any chosen Technique may in fact be useful to help fulfil several Tasks. Conversely, any chosen Task is likely to need the use of more than just the one Technique. Part of this many to many linkage is because there are, in fact, many "duplicates" in OPEN's toolbox. For example, there are several techniques for finding objects. Some OO software developers start by a textual analysis, some use simulation, some use CRC cards and yet others prefer a use-case driven beginning to a software project. It is your choice.

Activities performed iteratively by means of a set of tasks result in work products. Work products are the documents, including software, that are produced either for internal inspection or for external evaluation and final delivery/use. In OPEN, these artefacts are delivered as part of the post-condition of the Activities. However, since the lifecycle is iterative and incremental, often the delivery is only partial (but planned that way). Delivery may be to other members of the team, to the manager or to an external party, such as the end-user/customer. It needs to be made clear to the recipient of each work product just what proportion of the final delivery is being made in such an incremental lifecycle. Thus work products are built up over several iterations but linked to the activities. They are not created by the activities directly but rather by the techniques used to realize the tasks of the activity. Any one work product can therefore be the result of the application of several techniques spread over several iterations.

As we have seen *(Figures 3 and 4)*, the overall architecture of OPEN is that of a number of Activities which are connected together in a flexible and tailorable way to form an OPEN process which is one specific instantiation from the OPEN process framework *(Figure 6)*.

Email us at objective.view@ratio.co.uk to subscribe to ObjectiveView for electronic or hardcopy delivery.
(type 'subscribe' in subject line)

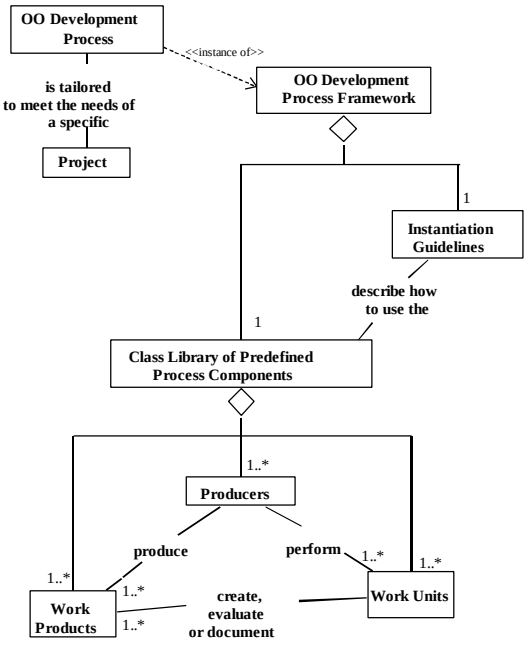


Figure 6. The structure of a development process framework
(diagram supplied by D.G. Firesmith, 1999)

The way these Activities are put together will lead to slightly different versions or instantiations. Each Activity is represented by an object in the process description *(Figure 7)* and these are connected together by lines representing potential transitions that the user(s) of the process can make (an example pertinent to MIS domains is shown here).

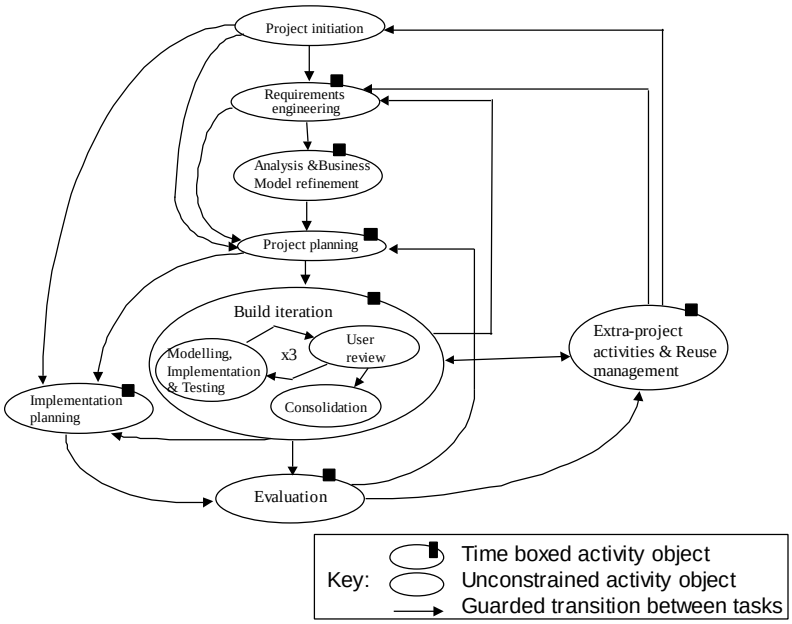


Figure 7. The contract-driven lifecycle for a single project in an MIS domain.

The development proceeds as suggested by these lines but only when the post-conditions of the current activity have been met. These should be specified clearly and should include testing criteria, document delivery, model building criteria etc. Once these have been satisfied, then the development team may move on to another Activity - again assuming that the pre-conditions of that next Activity have been achieved. These might be that certain work products are available, that a certain percentage of the system has been designed, that certain signoffs have been made etc.

The part of a methodology that then allows you, the developer, to deliver documentation and other artefacts, including the final code, must include a modelling language *(Figure 2)*. A modelling language is a metamodel plus a notation. The metamodel is essentially the set of rules that say what you can and cannot do with the language elements, which are themselves represented graphically or textually (the notation). One well-known modelling language is the Unified Modeling Language which was endorsed by the Object Management Group in late 1997. UML tends to be more pragmatic than some other modelling languages [Such as OML (OPEN Modelling Language) which is compatible with UML but offers some useful extensions.] and more aligned with hybrid approaches such as C++, Java and relational databases. Together, the metamodel and notation will be the tools you use to depict the results of modelling and coding as you build software. Understanding of the metamodel is the realm of methodologists and CASE tool builders; software developers do not generally need to see the metamodel itself. If the methodologists get the metamodel right and the CASE tool vendors implement it correctly, then you have access to fine tools to increase your quality and productivity.

Finally, we need to expand a little on the statement that OPEN is suited for *component-based development*. Development in the new millennium will incorporate distributed architectures almost by default. With the advent of the internet and industrial strength middleware to support distributed architectures, a good methodology has the responsibility of providing the detailed process of architecting and designing fully encapsulated components that can be deployed over a company's network or on the web. OPEN provides detailed support for such applications.

International Support for OPEN

Support for OPEN is in the form of an international group of researchers, consultants, CASE tool vendors and academics who are responsible for developing and maintaining OPEN. As of August 1999, there were 37 members worldwide including authors such as myself, Ian Graham, Don Firesmith, Meilir Page-Jones, Tony Simons and Houman Younessi. All material is public domain and is distributed via books and articles many of which are directly downloadable from the website at www.open.org.au or one of the mirror sites.

The Consortium itself does not market any products or services, although individual members may be connected with companies that do so. In fact, we prefer to work with third party companies worldwide who can offer local and continuing support. This can include support for training, mentoring, consulting and tool distribution. These sources are all advertised through the website. Contact them directly or ask the folks at Ratio for further details.

Summary

The beauty of the OPEN process is that it is not a straitjacket that lays down the law on what you shall and shan't do. Rather, OPEN is a process framework that can be tailored by individual organizations in a way that suits them. Tailoring requires choosing specific Activities, the way Activities are interlinked and Tasks appropriate for those Activities together with compatible and effective Techniques. All the elements from which to choose and tailor your own OPEN-compliant methodology are in the full texts on OPEN (see Suggested Further Reading section below).

Furthermore, while OPEN encompasses an iterative, incremental and parallel process, which should, of course, form the mainstay of any OO software development approach, it leaves it up to the users of the process to choose the lifecycle style (e.g. spiral, waterfall, fountain, contract-driven) and to express their work products in whatever modelling language they see fit and relevant to their development environment.

This tailored version of OPEN thus fits your company requirements "like a glove" whilst still being in accord with the overall OPEN "standard". Flexibility brings ownership - a major objective realized!

Suggested Further Reading on OPEN

Most of the papers on OPEN are available for free download from the OPEN website at http://www.open.org.au with mirrors in Europe and USA. Some sample chapters are also available of the books (see below).

- The OPEN Process Specification, Graham, I., Henderson-Sellers, B. and Younessi, H., 1997, Addison-Wesley, UK, 314pp
- The OPEN Toolbox of Techniques, 1998, Henderson-Sellers, B., Simons, A.J.H. and Younessi, H., Addison-Wesley, UK, 426pp + CD
- Documenting A Complete Java Application Using OPEN, 1998, Firesmith, D.G., Hendley, G., Krutsch, S. and Stowe, M., Addison-Wesley, UK, 404pp + CD
- OPEN Modeling Language (OML) Reference Manual, 1998, Firesmith, D., Henderson-Sellers, B. and Graham, I., Cambridge University Press, New York, USA, 271pp
- Requirements Engineering and Rapid Development. An Object-Oriented Approach, 1998, Graham, I., Addison-Wesley, UK, 271pp
- OPEN-ing Up UML: Modelling, 1999/2000, Henderson-Sellers, B. and Unhelkar, B., Addison-Wesley, UK (in press)
- The OPEN Process Framework. An Introduction, 1999/2000, Firesmith, D.G., Henderson-Sellers, B. and Unhelkar, B., Addison-Wesley, UK (in press)
- Object-Oriented Development Process Framework Specification, Firesmith, D.G. et al. (in preparation)

© 1999 Brian Henderson-Sellers

Brian Henderson-Sellers is Director of the Centre for Object Technology Applications and Research and Professor of Information Systems at University of Technology, Sydney (UTS). He is author of eight books on object technology and is well-known for his work in OO methodologies (MOSES, COMMA and OPEN) and in OO metrics. He is a frequent, invited speaker at international OT conferences.

U K R E C R U I T M E N T B U L L E T I N F R O M

RATIO

The most stimulating OO jobs in the UK!

Ratio continuously has vacancies for IT professionals with the following skills:

- Object-Oriented Analysis and Design
- Object-Oriented Architecture
- Object-Oriented Development in C++ and Java
- Object-Oriented Project Management
- CORBA/DCOM

For internal roles within Ratio or to join one of our prestigious external clients.

Both permanent (£40,000+) and contract (c.£1500/week) positions are available.

For more information regarding these opportunities, please call Ratio on +44 (0)20 8579 7900, or email us your CV at jobs@ratio.co.uk, or visit our web site at http://www.ratio.co.uk for more details.

URGENT!!!

PRE-SALES SUPPORT, OO, C++, JAVA, CUSTOMER FACING HAMPSHIRE (C.£40,000)

UML DEVELOPERS WITH REAL-TIME EXPERIENCE, HAMPSHIRE (C.£30,000)

Ratio… We Know The Object

Subjective Views Series

Interview with Robert C. Martin

eXtreme Programming

Mark Collins-Cope interviews Robert C. Martin, President of ObjectMentor, Object Guru, Editor of C++ Report and author of 'Designing Object-Oriented C++ Applications using the Booch Method' on his recent endorsement of XP.

General

[Mark Collins-Cope] Bob, you've recently endorsed Kent Beck's eXtreme Programming approach to software development. This may come as a surprise to some as you've previously been a big advocate of the use of UML as a design tool, or at least you've written extensively on it, what would you say to those who have been surprised in this way?

[Robert Martin] I would express my own surprise at their surprise ;-). XP and UML are not mutually exclusive. I am still a proponent of UML, and will continue to use it and write about it.

XP is a development process. It says nothing about UML whatsoever. Some folks have taken this to mean that XPers don't use UML to model their ideas. However, this is not the case. In XP, we do create models; and we do design our software. And we can use UML for those Designs.

XP does approach analysis and design a little differently, however. The diagrams produced by UML (or whatever modeling method you may be using) are, with very few exceptions, ignored by XP once they have yielded code. XP puts its value in the code, and in the use cases; and ignores the intermediate steps.

Process and Notation

[Mark] Process is a hot topic in the software development community at present. We have RUP, Open, Catalysis, XP etc., all of which seem to offer something of a contradictory view of the world. What is the average software developer or manager supposed to make of this? Is there such a thing as a right and a wrong approach to software development?

[Bob] Yes, there is. The right approach is the one that gets the job done with a minimum of fuss. Kent Beck makes an astute observation about processes. Process is about managing fear. We put processes in place because we are afraid. If our fear is large, we put big processes in place. If our fear is small, we put little or no process in place. The process that is right for a given team is the process that balances their fears against their ambitions.

XP is a process for the ambitious team that wants to get to market fast. XP manages fear by using *people methods* rather than paper methods. In XP, the fear of speed is mitigated by working in pairs, writing lots of tests, communicating with the customer on, at least, a daily basis.

[Mark] By managing fear I assume you mean fear of project failure, bad design, etc. Haven't these fears arisen for a reason - past failures for instance. In which case, aren't there good reasons for having well defined check points in the project lifecycle to spot these failures before they become too big to remedy?

[Bob] Checkpoints are a good thing. When something is good, XP turns the knob up to 10. That's why it's extreme. So we'll have checkpoints every few minutes. We'll run tests every few minutes to ensure the system still behaves properly. We'll reconsider the design every few minutes and refactor as necessary. We'll continually re-estimate our schedules and re-prioritize our plans so that we're never working on out-dated assumptions.

[Mark] Would you call XP a RAD technique? Why?

[Bob] No. RAD techniques deal with fear by assuming that the developers wish to be courageous to the point of foolishness. XP is not a high-risk process. It goes fast, but it also goes safe.

XP is dominated by tests. There are more lines of test code written in an XP project than there are lines of production code. The tests are run every time we make any kind of change. They tell us when we have broken something. In XP it is illegal to check in code that does not pass *all* the tests. Thus, the code never gets badly broken.

[Mark] UML now contains 8 notations. Do you think UML has got a bit too huge and all encompassing for its own good? Is the emergence of XP something of a backlash against this and more heavyweight process definitions? Is XP part of a larger pendulum swing from formality to informality?

UML's size is not a detriment. Developers are not honor-bound to use every bit of UML notation. Rather, we can use only that which we need, and ignore the rest. Therefore, I'm glad that UML is large, because it gives me a large set of tools to draw from.

I think lightweight processes like XP *are* part of a backlash against huge formal processes. Processes have been getting ever bigger and ever heavier for the last decade or more; and without an obvious increase in software quality to support the growth. It's as though the software industry has said "Gee, our big processes aren't giving us the benefits we need; I guess we have to make them still bigger".

Many of us have fought against the increasing corpulence of processes for years now. I think XP is a statement from those of us who have had enough of untamed process growth.

On the other hand, it would be a mistake to think that XP was a move away from either formality or discipline. XP is code-centric; and there is little that is more formal that code. XP is also highly disciplined. The XP practices have very narrow parameters. You can't just decide not to do them.

[Mark] So lack of process is not the reason for software failures that have motivated larger process. Where would you pin the blame then?

[Bob] The processes in use today focus upon paper rather than people. Paper can't think. Paper can't solve problems. Paper can't adapt to change. We can schedule all the reviews we want. We can produce all the analysis and design documents we want. We can coordinate and cross check and plan all we like. But as long as people are considered second order elements of the process, the process will fail.

Alistair Cockburn says it best. Process is a second order effect upon success. The first order is people. XP focuses upon people. It provides a framework within which people can communicate effectively. As Kent Beck says, XP is an attempt at making it OK to tell the truth.

[Mark] Ralph Johnson is quoted as saying (this is hearsay) that the XP process is analysis...test...code...design... would you say this is an accurate description of XP?

[Bob] No. It is, however, an adequate description of one development episode in XP. A development episode might last an hour.

An XP project is filled with thousands of little micro iterations. Each of which contains a component of analysis, test, code, design. The ordering is significant. Understanding comes first (i.e. analysis); which is mixed with some design as well. Then we write test cases that describe that understanding. The act of writing tests means that we must also have a design for the code in our minds. Then we write code that passes the test cases; and of course there is an element of design involved with writing the code. Finally we refactor the code to make it as simple and clean as possible. So there is an aspect of design in all four of the steps.

Software Architecture

[Mark] On the subject of the overall software architecture of a system (the package structure, architectural layering, etc.), you have always been a proponent of the application of good design principles (acyclic dependencies, interface segregation, etc.). Is the architecture of a system (as defined here) at risk with XP? The concept of a system metaphor doesn't seem, to me at least, to sit with this definition of architecture, what are your thoughts on this?

[Bob] Architecture is probably the single most important aspect of any software project. Without a good architecture, a system will degrade into a quivering glob of slime. Any process I use to develop software will be centred on producing the best possible architecture.

XP has an architecture step known as the System Metaphor. The concept of the metaphor is not the architecture in its full form. Rather it is the seed from which the crystalline structure of the architecture will grow.

Architecture can never be fully decided in an orgy of up front design. Architecture, like everything else in the software environment, must evolve. XP provides the iterative frequency, and the architecture focus to ensure that a strong architecture will evolve as the project grows.

There are rules in XP that force the developers to maintain and evolve the architecture. Rules of simplicity -- rules of communication -- rules of procedure. Developers are not allowed to check in a module if it is not in the cleanest, simplest, and most

flexible state that they can think of. Developers must always collapse duplicate code whenever it is found. Developers cannot work alone, but must always work in pairs, so that they can challenge each other on architectural issues.

[Mark] I'd agree with your sentiments regarding architecture as important, and that you're unlikely to get all aspects of an architecture right up front. But an up-front vision of the architecture system can't be a bad thing, can it?

[Bob] No, not at all. That's why we create a system metaphor in XP. It's also why every release begins with an exploration phase, and every iteration begins with reassessment of the current design and architecture.

Design

[Mark] 'Do the simplest think that could possibly work' is the design maxim of XP. Is it not true that with a little more forethought and a little less immediacy, the necessity to refactor existing code could be avoided? A related point is that of how 'holistic' an approach is adopted. By holistic I mean the opposite of 'piece by piece' - taking a wider view of functional requirements and trying to design a solution which covers, say, 5 rather than one requirement.

[Bob] It is possible that with "a little more forethought" we could reduce some refactoring. So why would we choose to refactor instead of apply forethought? Simply because refactoring is cheaper and more reliable.

I'm not saying that forethought isn't valuable – it certainly is. But forethought is speculative. And time spent on speculative ventures is much more expensive than time spent on sure ventures. Thus, I prefer the surety of refactoring to the speculation of long range forethought.

Thus, prior to each task, I will design that task; and make sure it fits into the current system architecture. I will write the tests. I will write the code that makes the tests pass. And then I will look again at the code and refactor it, in small steps, until I think its design is good. No long range speculation. Only short term surety.

Does this lead to revolutions? Certainly! There will be times when the refactoring approach gets caught in a local minimum. In such cases there is a better approach to the entire system, but it requires an incremental effort to push out of the local minimum and get into the more global minimum. Once such a need has been identified, XPers are unafraid to refactor into this new global minimum. They aren't afraid because:

1. They have the tests to prove at each step of the way that nothing has been broken.
2. They work in pairs, so that every step has the benefit of two minds.
3. They practice group ownership of the code, so they are all familiar with every bit of the existing code.

Could the change to the better design (the global minimum) been prevented with some forethought? Perhaps. How much forethought does it take to ensure that you have found it? What will you pay for it? Why should you pay this up front? The XP philosophy is: pay for what you need, when you need it; and not before.

[Mark] But if you know you're going to need a certain design structure to cover the next two requirements you're going to work on, wouldn't you just put in that extra bit of work up-front to cover them? And is it really cheaper to refactor than to design?

[Bob] Is your design structure right? How can you be sure if you haven't implemented one of the features yet? Are you sure it would be cheaper to implement what you think the design ought to be? Or would it be cheaper to implement the design when you need it and no sooner? XP chooses the latter course in the assumption that up front payment on speculation is, on average, more expensive than paying right now for just what you need right now. XP justifies this by assuming that refactoring in the presence of copious unit tests, and pair programming is very inexpensive.

[Mark] We all know that design is difficult. It's difficult to think about design, and certainly has less immediate reward than writing code. My personal experience, though, is that getting the basics of the software design right pays big dividends in the longer run (and by this I mean over a two to three months timescale). Is there not a danger that as XP has no 'design deliverables' per-se, that design is going to suffer?

[Bob] No, there is no danger at all. Indeed, design thrives in an XP environment. XP is all about design. We continuously drive the code into the best possible design that we can think of. We never say "We'll go back and fix that later."

We must come to terms with what design is. The design of a program is the shape of its code. The partitioning of the code into methods, classes, packages, etc; and the relationships that exist between those elements. We might represent these things in diagrams; but the diagrams are not the

design. They are just proxies for the design. The real design is in the code. XP does not value design proxies. XP values direct expression of the design in code. And XP values the best possible design for the code.

It places such a high value on this, that forces us to write tests that enable us to fearlessly refactor, just so that we can be free to move the code in the direction of what we perceive to be the best design.

[Mark] True, what we're working towards at the end of the day is to have our code in a shape that makes it easy to maintain, extend, etc. And true, we're likely to need to restructure - refactor - our code sometime to make this a reality. I think the essence of what worries me amongst others is that without an explicit 'design' phase you're just postponing decisions that you're going to have to make as some point, although I can see that the focus on refactoring means you may get there eventually.

[Bob]Again, XP assumes that paying now for what you might need later is more expensive than paying now for what you know you need right now. The risk with this approach is that there may be so much more rework than with a more up-front approach, that it overwhelms the cost of the up front design, and the cost of carrying all that extra unutilised design through the project lifecycle. XP assumes that rework in the presence of unit tests and pair programming is inexpensive, and that the cost of up front design, wrong guesses in that up front design, and carrying all the up front design elements in the software is expensive.

[Mark] Something UML (class and package diagrams) do offer is the ability to look at your software at a higher level of abstraction: putting the focus on high level design, etc. Is there not a danger that the lack of a design focus (in deliverable terms at least) in XP may mean that a new generation of programmers don't get to see the benefit high level thinking? Would you, for example, advocate XPers get OOA/D and UML training?

[Bob] To the last questions, yes, and yes. XPers should certainly know the principles of OOD, and should be able to use a design notation like UML. It is the proper application of the principles of OOD that keep the structure of the code flexible enough to be refactored. This is especially true in C++. Without strong use of the principles, C++ code will become so badly intertangled as to completely resist refactoring.

Am I concerned about the loss of high level thinking? Not at all. There is lots of high level thinking done in XP. It is done at the beginning of the project, the beginning of each release, the beginning of each iteration, and the beginning of each task. Moreover, the high level structure of the software is *exposed* by the code. In a well designed OO program, you can rip the low level details out of the program without changing, or even recompiling, the high level modules. One of the most important principles of OOD says simply that high level modules should have no dependencies on low level modules (The Dependency Inversion Principle).

[Mark] In what way can code become resistant to refactoring?

[Bob] If you make a single change to a module, and that forces you to recompile for an hour, that module is resistant to refactoring. In order to refactor you must be able to get quick turnarounds. What causes long recompiles? Improperly managed dependencies!

Using XP, engineers are *always* sensitive to turnaround time. If a change they make increases turnaround, they refactor until turnaround is fast again.

User Interface

[Mark] Where does user-interface design fit in with XP? Does XP has any impact on the quality of interface a system might have, be that positive or negative, and would you encourage a user-interface specialist to be involved in this?

[Bob] XP is a software development process. User interface specification is a different topic entirely. I would not have software engineers design the look and feel of a user interface unless that interface was going to be used by other software engineers. (And maybe even then I wouldn't have the engineers design it).

Visit Ratio's web site at http://www.ratio.co.uk for links on object technology, additional articles, and past issues of ObjectiveView.

P U B L I C M A S T E R C L A S S

We Know the Object of…

eXtreme Programming

A One-Day Seminar
by Robert C. Martin
President of ObjectMentor Inc.
Author of:
"Designing Object-Oriented C++ Applications using the Booch Method"
Editor of: C++ Report

20 June 2000
London, UK

"XP - You'll love it or hate it, but one thing's for sure, you NEED to know about it!"

For more information on this course, contact Ratio on +44 (0)20 8579 7900 or by email at bookings@ratio.co.uk

Please note: class size is limited, so book early!

Team and Personnel Issues

[Mark] Pair programming is one of the facets of XP that I find attractive. What benefits do you believe it offers? Do you believe the benefits outweigh the obvious costs, and how would you convince at IT manager of this?

[Bob] There have been several studies conducted on the topic of pair programming (See the work of Laurie Williams at the University of Utah, lwilliam@cs.utah.edu). These studies show that paring causes no loss of productivity at all, while significantly decreasing defect rate, code size, and job dissatisfaction.

I believe pair programming offers programmers a way to maintain stimulating relationships with other engineers. A way to share responsibility, overcome fears, combine ideas, and just have fun writing code. Convincing IT managers, who are unmoved by the weight of other evidence, is a matter of convincing them to try it on one project (along with the other XP practices).

[Mark] Does 'role' separation exist in an XP team (analyst, architectural authority, etc.) or can everyone do everything?

[Bob] Everyone does everything. Clearly some people will gravitate to certain kinds of jobs. But there are no impressive titles like "architect". Remember, too, that if one person has a skill, he will teach that skill to all his pair partners. And since pairings are kept very short in XP (e.g. four hours) everyone will be exposed to everyone else's expertise in very short order. Thus all the engineers influence each other greatly and learn from each other continuously.

[Mark] So if roles do exist it is because they occur naturally within the team, rather than because they are assigned?

[Bob] Yes. Of course there must be a manager who is responsible for providing resources to the team, and keeping distractions away from the team. This manager is also responsible for monitoring the process and trying to spot and correct problems. From time to time the manager may need a certain role filled. Rather than appointing someone to the role, the manager will present the need to the team, and let someone volunteer (or be elected).

[Mark] Some IT managers may feel that the emphasis on developer satisfaction may be at the expense of wider business objectives - deadlines, etc. ("we have to have this system by this date") How would you address such concerns?

[Bob] XP is focused on customer needs. The customer is the only source of requirements and priorities. There must be a customer (or a suitable proxy) on site with the developers on a full time basis. This customer bears the responsibility for product definition and schedule. XP empowers that customer to specify exactly what the team is going to be working on, and in what order.

On the other hand, the team is responsible for all estimates. The customer cannot load a release with more than the team agrees can be done. The customer can tell the developers what order to do things in, but now how long it will take them.

In the end, the customer gets the most important things done first. If a deadline is missed, it is only the least important things that are missing.

Moreover, XP continuously measures progress against deadlines. By the time a release is less then one third complete, it will be clear whether the velocity of the team is sufficient to meet the schedule. And if it appears that the schedule will be missed, the customer has the ability to remove scope from the release.

Finally, the measured velocity of the team is applied to all future estimates, such that the estimates become more and more accurate over time.

Reusable Artifacts

[Mark] There is still a clear driver in the software development industry to get re-use happening to cut down development costs and time to market (see recent interest in component based development). How does XP sit with such approaches, and how would you approach the issue of developing re-usable infrastructure (e.g. a persistence layer, an email component, etc.) in XP?

[Bob] There is one sure way to fail to produce a reusable framework; and that's to design it up front. This strategy has been repeated many many times, and the results are frequently the same; the reusable framework isn't very reusable.

The best strategy for creating reusable frameworks is to evolve them concurrently with at least three applications that use them. XP can certainly be used for this evolutionary process.

[Mark] Can't design up front work when the designer has substantial experience of the needs of the developers who will be using the component or framework they are developing.

[Bob] Frameworks are big investments. Yes, they *can* work if the designer has the necessary experience, but how much of a risk are you willing to take? Wouldn't you rather *know* that the framework will be effectively reusable?

Integration

[Mark] XPs emphasis on continual integration is interesting, and will certainly bring any 'integration errors' rather quickly to the attention of the team. But, some of the projects I've been involved in have millions of lines of code, and can take literally hours or even days for a complete compile and rebuild. In such an environment continuous integration may prove to be problematic. Is this just a case of the wrong project for XP?

[Bob] Possibly. However, when millions of lines of code require hours or days to compile, the problem is not one of sheer volume. Consider, there are 86,400 seconds in a day. If a 10-megaline program requires one day to compile, the compiler is chuntering through 115 lines of code per second. That's not particularly quick.

When dependencies are poorly managed, compile times are O(N*N). When dependencies are well managed, compile times drop to O(N log N). When compile times are inordinately long, there are dependency problems in the design. And those dependency problems are causing more heartache than just compile time.

Still the question remains. If integration is expensive, can XP be used? The trick would be to design the system such that integration is not painful. Using component design strategies, and infrastructures like CORBA, COM, and RMI become critical for this. In effect we take a huge

project and turn it into several dozen smaller projects with little or no integration overhead.

[Mark] With a conscious focus on up front design?

Of course! There is a tendency to think that just because XP defers a certain amount of up front design, that XP is anti-up front design. This is far from the case. XP simply defers those decisions that make economic sense to defer. Clearly a large component-based project would require at least enough up front design to get an idea of what the components were. Then each component could be developed by an XP team.

Testing

[Mark] XP has a strong focus on functional testing. I particularly like the idea of writing the tests before you write the code. What benefits do you think this offers?

[Bob] There are so many! A suite of tests gives you an anchor. You know immediately if any change you have made has broken something. This enables high rates of refactoring. A suite of tests gives you a supplemental document that describes the code being tested. If you want to know how to create a certain kind of object, there is a test case that shows you how. If you want to call a particular method, there is a test case that shows you how. And this document remains 100% accurate and very complete, regardless of how much the software changes over time.

The act of writing tests before you design is an act of design. Kent Beck calls this "Design by Testing". It forces you to think through the item you are about to test, from the point of view of a user. This is very valuable.

[Mark] Testing is, I believe, integrally related to the whole idea of refactoring. Are all tests and test results automated? What tools are typically used to assist in this (if any), and how would this type of testing work in a GUI intensive environment, where there is a requirement to drive the UI to get something to happen, and the results maybe appear on the screen?

[Bob] All tests are automated, so far as this is possible. There are some very nice tools that help with this. The XUnit family of test frameworks has proven to be very useful. See www.xprogramming.com for more information.

GUI testing can be done at the functional level simply by grabbing the widgets, stimulating them (faking a button press for example) and then reading their state. Look and feel tests are much harder to automate. I do them manually; but there are automatic strategies.

Documentation

[Mark] 'The code is the documentation.' This reminds me of the self-documenting code arguments when Pascal first came out. Can the code be the documentation, and how?

[Bob] There is little doubt that code is documentation. Code is a document. The question is whether it is a good vehicle for others to read and gain understanding of the software. The usual problem is that the software is so incredibly focussed upon details that there is simply no way to use it to get the big picture, and to derive the intent.

Sadly, most programs fail very badly at communicating intent. This, again, is a dependency management issue. When a single module deals with both high level and low level issues, it is not a well partitioned module.

Well written code is well separated code. Well separated code is easy to read because each module covers on and only one level of detail. High level modules remain high level. Low level modules remain low level. And all dependencies point from low levels to high levels. (Exactly the opposite of structured analysis and design).

Notice that this has nothing whatever to do with language. The readability of a program has *nothing* to do with the language. One must assume that the reader is familiar with the language and therefore the language is a non-issue. The issue is purely and simply one of structure and communication. If the code is well structured, and if names are well chosen for the functions and variables, then the code will be good at communicating intent.

This leaves the notion of a roadmap. XP has the concept of a metaphor. It is the job of the metaphor to act as the roadmap so that people understand the concepts and relationships within the system. XP does not demand that this metaphor be written down, so long as it is firmly entrenched in the programmers minds. Personally, I think writing it down is not such a bad idea; so long as it is kept very small; and is not any kind of a time sink.

The notion of oral documentation is scary to many people. However, with pair programming, and with people rapidly moving from pair to pair, the oral knowledge will rapidly diffuse through the organization, and any newcomer will learn it very rapidly. XPers are not afraid of oral documentation.

[Mark] One major nightmare all of us in software have to face is the issue of documentation getting out of sync. with the code. I presume this is the motivation behind 'the code is the documentation.'

[Bob] It is one, but it is a minor one. The real motivation is simply that design documents have high short-term value and low long term value. XP recognizes the short-term value but ignores them once they have been committed to code. There is no room in an XP project for keeping old design diagrams in sync with new code.

Some environments demand these documents for bureaucratic reasons. In such cases, the *customer* will demand the documents, will prioritize their creation, and will schedule them in a normal XP project. In XP, if the customer wants something, it gets prioritized and scheduled. When the time comes for it to be done, the developers do it. If that means documentation, then the documentation will be produced. It is not likely, however, that the developers will ever look at the documentation again.

[Mark] I've known some developers who can pick up a piece of code, trawl through it and get a useful grip on what's going on (I once worked on Unix system software, where this was a common approach). Personally, I always hated doing this, perhaps this was because the code was so awful in the first place - and as you said, didn't communicate intent at all - or perhaps I'm just not right type of developer for XP :-)? Is there a certain type of developer to whom XP is more applicable?

[Bob] The only constraint I can think of is that the developer must be comfortable working with others. Remember that in XP everything is done in pairs. Communication is one of the prime values of XP. So lone cowboys, or developers who can't work well with others, are not going to do well with XP.

[Mark] Given the increasing prevalence of tools that can reverse engineer say, a class diagram, from a piece of code would it not be possible at least at this low level to maintain an additional form of documentation? Is this is a trend you would like to see continue to higher levels of abstraction, e.g. generating package diagrams with dependencies from source code?

[Bob] The more tools the better. If I can *quickly and easily* use a tool to convert my code to UML diagrams, and then browse my code by navigating those diagrams, then I'm all for it. Together-J, and Visual Age for Java do things like that. It can be a very powerful way to view the code.

The point is, that it is tied directly to the code. There is no effort expended in creating the diagrams, no effort expended in keeping the diagrams in sync. The diagrams are just a convenient reflection of the code. And they are just as expendable as before, because they can be recreated on demand.

More general

[Mark] Scalability has been raised as a concern about XP .... is XP just intended for small teams? How big a team would you say can be effective using XP? and would you put a maximum limit on the size of an XP team?

[Bob] Our experience with XP is with small teams of a dozen or less. My personal belief is that it will scale quite well. But caution is indicated. It would be irresponsible to flash cut a 100-man team to XP at the moment. I would, however, have no qualms about easing a 30-man team into XP.

There is another issue here. Team size is not a good indicator of project complexity. A dozen people are probably enough to write 90% of the software projects in play today. Many, perhaps most, projects are simply overstaffed by a factor of five or ten. IMHO, teams need to be lean and aggressive.

*[Mark] Could you see XP being *misused*? What are the biggest 'gotchas' that new XP teams are likely to face?*

[Bob] Anything can be misused. One form of misuse is over-enthusiasm.

For the foreseeable future, new XP teams are going to be enthusiastic about trying XP. I think they need to temper their enthusiasm so that they don't expect more than they can deliver. XP is a process, not a miracle. It will help, perhaps a lot. But it's not the master stroke that's going to remedy the software crisis. After all, software is *hard*. Even with XP, there will still be schedule overruns, still be improperly set expectations, still be unreasonable demands, still be political manoeuvrings. XP is not the signpost of the millennium. (er...)

[Mark] Would you expect users of XP to customise it to their own circumstances? What typical customisations have you encountered? Are there any you would recommend personally, and in what circumstances?

[Bob] One of the demands that XP makes on its teams is that they continuously review the process and evolve it. If a practice gets in the way, change the practice. Geographic separation is a case where some of the XP practices have to be customized.

You have to get creative with pair programming. You can use cute tools like NetMeeting so that both parties see the same screen on their individual workstations, and each can take control of the program. You can use delayed pairing, such that one person writes the code and another reviews and refactors it.

One customization I have used, and that we are likely to see become very prevalent in the Java and C++ world, is the integration of dependency management principles into XP. Java, and especially C++, incur a high price for mismanaged dependencies. As I said before, a poor dependency structure can make refactoring intractable.

[Mark] What form is this integration likely to take?

[Bob] Extra refactoring rules. Currently we have the Once and Only Once rule, and rules related to readability. Integrating the dependency management rules will simply extend this list.

[Mark] There's something of a habit in the software development industry of new approaches being touted as the solution to the 'software crisis'. Cynics may see XP as just another passing fashion. How would you respond to that?

[Bob] I respect healthy scepticism. It counters the failings of over-enthusiasm. So, to the sceptics I say, thank you for keeping everybody honest.

I expect, however, that anyone who is vocal in their scepticism would also be putting lots of effort into verifying that their skepticism was more than just their own prejudice. I would be expecting them to take a serious look at the data and coming to reasoned conclusions. It is my belief that upon investigation most sceptics will lose *some* of their scepticism.

[Mark] To round off, how would you summarise projects that are or are not appropriate for XP?

[Bob] The project that is appropriate for XP is the project being developed by a team that wants to go fast, produce high quality software, work reasonable hours, and keep their customers happy. They have to like each other well enough to stay in close contact with each other for long periods of time. And they have to be disciplined enough to follow the rules, even when the pressure starts to mount.

[Mark] Bob, thank you very much…,

[Bob] You are quite welcome.

R A T I O I S P R O U D T O S P O N S O R

TOOLS
Europe 2000

"Enterprise Architecture - Patterns - Components"
Mont Saint-Michel / Saint-Malo Normandy / Brittany, France

5-8 JUNE 2000

KEYNOTE PRESENTATIONS TO INCLUDE:

Bertrand Meyer	**James O. Coplien**	**Ian Graham**
Inventor of Design by Contract	*Founder of the Patterns Movement*	*"Requirements Engineering, an OO Approach"*

as well as…

HANDS-ON TUTORIALS • WORKSHOPS • TECHNICAL PAPER PRESENTATIONS
DISCUSSION GROUPS • PRODUCT DEMONSTRATIONS

Visit **http://www.toolsconferences.com/europe**
for full programme and registration details

TOOLS… the major series of international conferences entirely devoted to Objects

Object Mentor, Inc. www.objectmentor.com

Better Software —
By lecture, demonstration and by example

Training • Mentoring • Development
for accelerated project success

Our goal at Object Mentor is to **improve the skills of your development team** within the constraints of your project's goals and deadlines. Your success is our success. We apply that philosophy with each customer, on each project. Through years of working with a diverse set of software development teams, we have developed a **flexible set of service offerings** that can be matched to your organization's current skill set, required skill set, and project budget and deadlines.

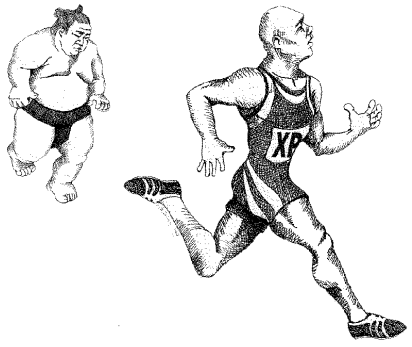
The process of developing a team's skill involves a number of steps, including formal instructor-led training, individual self-study and practice, and mentoring by experts to reinforce the skills being developed. Many organizations have software development experts, but these engineers are required to lead development projects and rarely have enough time to spend with other team members. Delays become inevitable while the team thrashes and struggles to learn the technology and apply it at the same time.

Object Mentor fills the gap. Our **training, mentoring, development, and on-line services** will develop, lead and support the team to project completion. Thrashing is minimized because your Object Mentor will lead the team forward. In the process, we **develop the team's skills**, leaving the team with the expertise to support the project on going and tackle increasingly complex software projects in the future.

Is a heavyweight process slowing you down?
OBJECT MENTOR has a better way . . .

TRAINING COURSES:

- **Extreme Programming (XP)**
- **Principles of OOAD with UML**
- **Advanced OOAD and Design Patterns**
- **Programming in Java: From UML to Code**
- **Object-Oriented Design in C++**
- **Object-Oriented Overview for Managers**



All courses are offered as **public courses** in the U.S.A and **on-site courses** throughout the world.

XP Immersion Training
with Kent Beck, Martin Fowler, Ron Jeffries, and Robert C. Martin

see www.objectmentor.com for details.

W E K N O W T H E O B J E C T O F

TRAINING

Excellence in Object and Component Training

The following courses are offered both in-house and on a regular public schedule basis.

Object-Oriented Analysis & Design using UML

This course gives you a practical understanding of the major techniques of the UML (Unified Modelling Language) object-oriented analysis and design notation, and how these techniques can be applied to improve quality and productivity during the analysis and design of computer systems.

What they thought…

"Thanks for this! Everybody is buzzing after the course. Thanks to you and your team for all of your efforts, particularly the lecturer, who has an excellent manner and just knows his stuff inside out."
Chris McDermott, Polk Ltd.

What they thought…

*"Patterns were particularly useful as were the hints & tips & tricks that were sprinkled throughout. It was also very useful to be shown *why* some of the techniques we use are good; up until now we've been choosing the techniques based on instinct."*
Phil Harris, Silicon Dreams

Component-Based Design using UML

This course gives you a firm understanding how to analyse and design extensible and customisable re-usable business (domain) oriented components, and how to assemble such components to create bespoke applications. The course has a clear focus on the architectural aspects of component-based design.

Object-Oriented Programming in C++

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.

What they thought…

"This has been a worthwhile exercise. The course was concise ... well focused via examples and practical sessions"
Course delegate, MTI Trading Systems

"Things were explained clearly, in simple terms and with relevant examples."
Course delegate, Primark

What they thought…

"I particularly liked the hands-on implementation of the Java language theory in an extendable example."
Graham Hoyle, Tetra Ltd.

"Really good course, well presented, well informed, lots of leads to wider ideas, etc."
Roy Turner, Silver Platter Information Ltd.

Object-Oriented Programming in Java

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.

Email info@ratio.co.uk or call Ratio Sales on +44 (0)20 8579 7900 for more information.