Chapter 1 Next Generation Software

Outline

In this chapter we outline some trends in the next generation of software development, and challenges that software development methods must address in order to support component-based development, open and distributed systems, adaptive business-driven solutions, and iterative development.

Section 1.1 sets the stage for the four main trends that we see following object technology. Section 1.2 describes the move to component-based development (CBD) and outlines what new issues CBD will raise. Components will inter-operate in open and distributed environments — Section 1.3 discusses how.

Component-based development will effectively support business driven solutions, in which flexible software systems enable the business to adapt more readily to its needs, as discussed in Section 1.4. And these new development practices will take place increasingly in a world of rapid, iterative development. Section 1.5 outlines some changes this will bring.

For components to succeed widely, they must be designed to be flexible and adaptable. Section 1.6 discusses building with such “pluggable” components.

Finally, Section 1.7 outlines the challenges software development methods face with these technology trends, and Section 1.8 places Catalysis and the rest of this book into perspective, based on these challenges.

1.1 *Trends for the next generation*

Technology change is accelerating.

Software development is a constantly evolving blend of engineering, science, art, legacy, and hype. From the earliest days of mainframe-based systems, through the evolution of client-server systems and the many politically-correct variants of "open" systems, to the object-oriented, component-based, and internet-enabled systems of today, the way we design and build software has evolved in sometimes dramatic, sometimes subtle ways. Software development, and information systems technology in general, is today going through an unprecedented period of dramatic change.

After objects — what?

Object technology has already made its mark on system modeling, design, and implementation. Objects, the most recent element of this change, will have a very significant influence on the evolution of software development over the next several years.

Benefits of object-oriented development

Certain benefits of object-oriented development have proven themselves over the past several years, such as:

- Polymorphism: the ability to have multiple implementations of the same interface, and to design and implement against an interface.
- Dynamic binding: deferring the binding of a service request to a particular implementation until run-time, based upon the object receiving the request.
- Incremental definition (inheritance): defining one class by inheriting from another and implementing only those parts that are different.

Shortfalls of "pure" object-oriented development

Traditional "pure" object-oriented development evolved bottom-up, driven by innovative programming languages and their enthusiastic proponents. All descriptions are structured around the programming language units of object classes and the services they provide. This view has led to several shortcomings:

- Many interesting interaction units involve multiple objects. Hence, the most useful components to re-use in design and implementation are not individual classes, but generic frameworks involving multiple classes. These multi-object units of design are often not given the attention they merit.
- The OOP "message-send" is not always a suitable way to model behavior, particularly at more abstract levels when modeling the problem domain or business, or when designing higher-level "connectors" between components.
- By using a single hierarchy of classes for both interface and implementation, interface decisions were not clearly separated from implementation ones, resulting in several unnecessary dependencies.
- Rampant usage of sub-classing results in excessive coupling from white-box reuse between subclass and superclass, partly caused by the inadequate separation of interfaces from implementations.
- Unfortunate language-specific dependencies crept into most object designs, even to the level of source code dependencies with header-files. This impeded wider-scale re-use, and made boundaries of language, process, and machine much more visible than necessary.
- Most design methods focused on the services an object provides, ignoring the services it requires of other objects and the notifications it can send them. This impedes easy assembly of larger components.

Towards open business-driven component-based systems.

Clearly, while providing some of the essential underpinnings of a better approach to software development, traditional object-orientation is not the whole answer. The most significant trends that are facing us today include:

- Component-based development — designing and implementing systems by assembling components, customizing or extending them as needed; and publishing components in a form that can be used to design and build others, based purely upon interface specifications.
- Open distributed systems — moving beyond the world of client-server into a world where the network *is* the system, adapting and evolving its form as well as content all the time. This requires standards for inter-operation, such as CORBA and COM, as well as standard infrastructure services.
- Adaptive business-driven solutions — software systems solve a business need, hence software development and evolution should be driven by the modeling and improvement of processes to support an adaptive business.
- Iterative development — increasingly, any development method must permit top-down as well as bottom-up and iterative development with incremental deliverables, while still separating different concerns (users requirements, architectures, code) as appropriate.

The next several sections in this chapter will discuss these trends in more detail, and the corresponding challenges to software development methods.

1.2 *CBD — Component-Based Development*

Component-based development is being touted as the next major shift in software development, impacting everything from the construction of user-interfaces, rapid application development, internet-enabling of legacy applications, and more.

1.2.1 The Move to Components

Component-based development is the buzz!

We are quickly moving into a world where we build software from existing components primarily by assembling and replacing interoperable parts. You pick components from a palette of choices. Each component exposes the set of properties and behaviors by which you can control it, and through which it will interact with you and with other components it is connected to. By "wiring" these exposed bits together, you rapidly assembly the functionality you need.

Components span a very wide range.

These components range from user-interface controls like list-boxes and hypertext-navigators, to infrastructure components and frameworks for networking or communication, to full blown business objects. You buy, unwrap, adapt, plug them into your system, and wire them to each other, to get instant new functionality, simply based on their published interfaces, without ever looking at their implementation. Companies are already seeing the benefits of adopting this approach.

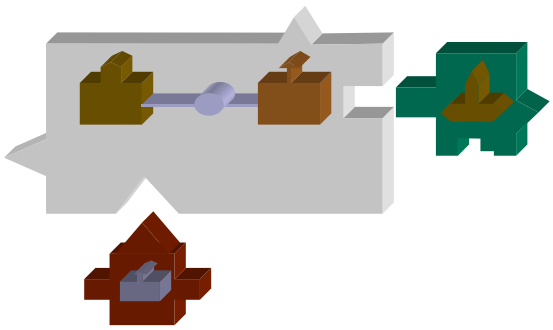


Figure 1: Component-based assembly could become pervasive

Software assembly makes an attractive vision.

We read increasingly about the promise of assembling full-blown "business-objects" and complete frameworks for scheduling, trading, customers, and orders. The potential improvements in implementation and test time and in product quality make this a very attractive vision.

Assembly may even happen in cyberspace!

We can conceive of the day — in the very near future — when components of our software systems are located in cyberspace by software brokers that dynamically match required services with those provided. These components download themselves to our machines, much like Java applets do today, negotiate capabilities and interfaces like fax machines, then connect to each other and inter-operate.

Standardization of infrastructure and "vertical" domain APIs helps

Java (through Java Beans and its enterprise cousin, *EJB*), Active-X, and Corba, already provide an infrastructure for component-level reuse. Much of the work of publishing component interfaces, connecting to components, and communication across machine

and language boundaries is done for you in standard ways. Moreover, this standardization is now making its way to "vertical" domains, such as insurance, banking, and telecommunication. When this becomes a reality, business components themselves will be defined and purchasable with standard interfaces.

1.2.2 What is new about Components?

Component defined

So what are components? And how do they differ from software units we have worked with previously, including objects? Lets start with a definition:

> Component — An independently deliverable unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole.

Most common uses of the word 'component' mean an implementation unit which can be composed with other implementation components. Others will be more like *"frameworks"*, a unit of implementation, modeling, design patterns, or specification, which is specifically designed to be generic and customizable.

Consider a mainframe application.

Consider a traditional monolithic host-based application; it was written assuming a dumb terminal at the other end, from which it received textual commands and to which it wrote screen displays. All the application logic and data were encapsulated within the host application. However, there were often no smaller units of encapsulation within that application, and all procedures shared the same global data. Evolving and modifying such systems are exceedingly difficult.

It takes *screen-scrapers* to compose such components.

Could such a host-based application be considered a component? Yes — but not a very elegant one. Its interfaces to the outside are screen outputs and terminal inputs, so the only way to *"compose"* such components is to write glue components that emulate terminals and screens, and appropriately decode this information to coordinate with another component.

Next came large-grained O.S. and database components

The arrival of more modern operating systems brought better services for coordinating across multiple applications — IPC, RPC, and for separating system services such as those of a database server. We now had large-grained components like the database, operating system, and individual applications. We even saw an early component architecture on UNIX, with the model of *pipes* and *filters*; each application consumed and produced streams of data, and the applications could be composed by simply connecting up the streams as needed. Elegant, but narrow and limited in applicability, since the only kind of *connector* was a pipe.

Object technology changed component granularity for good

The arrival of object-oriented languages like C++ and Smalltalk changed the granularity of components. It was no longer entire applications that communicated with each other, but fine-grained objects like buttons, list boxes (on a UI), products, orders, and line items. These objects encapsulated local state and data representation, and offered services to other objects via their published interfaces. Unfortunately, traditional object-oriented languages had a narrow focus, as discussed in Section 1.1.

OpenDoc and COM opened up components

The next generation of components appeared in the Macintosh environment as OpenDoc, and in Windows as OLE/COM. This generation of component technology was distinguished by the finer granularity of components its exposed. For example, the

Microsoft Word application would expose numerous “objects” including documents, paragraphs, tables, and words; a Visio application would expose drawings, shapes, and cells. Components were connected not just at the level of the entire application, but at the level of objects within an application. A business object such as an *NetworkElement* may now be connected to a *row* in a database, a *paragraph* in a Word document, and a *Shape* in a Visio drawing. Moreover, the nature of the *“connectors”* started to change, from being directly coded requests for services, to higher level concepts such as *properties* and *events*.

Workflow systems required new kinds of connectors

In parallel, interest in *workflow systems* was growing. The components used in these systems were larger grained units of business activity, and the *connectors* represents the flow of work products, such as transferring a travel requisition, or replication of an order for parallel processing in different activities.

Today’s components are easier to compose

Today’s component technology is characterized by CORBA and COM. Components can range from large-grained applications to fine-grained objects, connecting to each other via published interfaces regardless of language and machine boundaries, and even inter-operating across enterprise boundaries via internet technologies such as IIOP. The nature of the *“connectors”* between components can become more high-level than explicit service requests, so components get *wired* together in a more natural way, as suggested by the component ‘wiring diagram’ below.

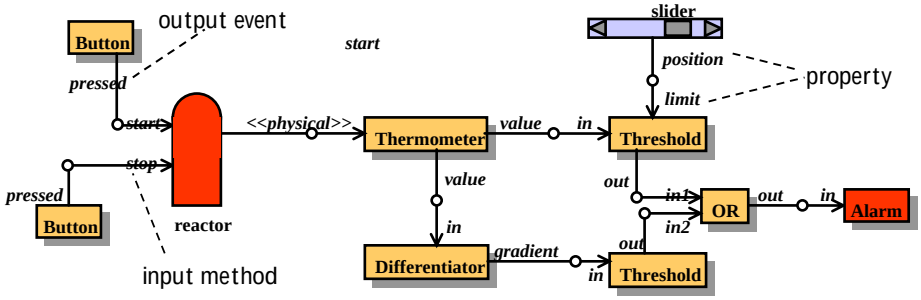


Figure 2: Different kinds of "connectors" in a component design

And are one step beyond traditional objects

Many components are not very different from large-grained objects, even if their implementation uses multiple classes. Most components are best implemented using object-oriented languages. Different forms of component *connectors* can be both modeled and implemented at a lower level using standard object techniques. But components do bring with them a improved focus for larger-scale software development:

- interface-centric design, rather than classes and inheritance
- standard technical infrastructure services, for naming, directory, transactions, etc.
- language and location transparency
- better composition mechanisms, such as properties and events

1.3 *Open Distributed Systems*

The world's networks are increasingly looking like one big computer. The internet is an "open distributed system", and one that will become increasingly sophisticated and central to most businesses. In this context, "open" means each component — whether a coarse-grained object or a framework of collaborating parts — may be called upon to connect to and work with components its designer never knew about; the form and content of the network of interacting components is constantly evolving.

Open componentware is especially important for the internet.

For example, a stock-trading system might include components that publish raw stock data, define various financial models that can be applied to evaluate different companies, apply stock data to selected financial models, and interact with trader components that buy and sell financial instruments. Such a system relies on definite interfaces between components (*StockTrader, QuotePublisher*), is intrinsically distributed, and is open and evolving over time.

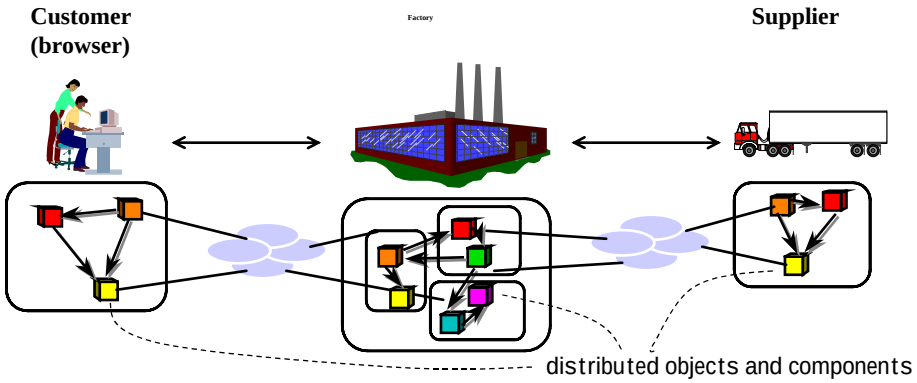
Consider an on-line stock-trading system.

The internet and intranet will provide further impetus to interoperable components, as they evolve from an information-sharing medium into a full application development platform. Application design and implementation are evolving from the desktop-centric client-server paradigm to a new class of network-centric applications. Based on technologies like Corba / IIOP, Java / RMI, and Corba, components will dynamically connect to, inter-operate with, and even extend, others that they have little prior knowledge of.

The intra- and internet need interoperable components.

This inter-operation crosses the traditional boundaries of an enterprise to realize new kinds of federated systems across a 'virtual enterprise', building new services that transparently integrate and customize services on different networks. In the classic *supply-chain* illustrated below, a factory's resource planning systems are connected to objects and components on the customer end, and to corresponding objects and components at its suppliers. The entire federation operates via the internet, using appropriate security measures.

The 'virtual enterprise' can be a reality today



We wish to use components to model, design, and build open object systems, whose form and function is extensible, consisting of multiple encapsulated components, and constantly reacting to stimulus. The parts, their interfaces and interactions, and rules

All these need open object and component systems.

for such interactions, do not preclude any compatible replacements, plug-ins, or extensions. Such changes may take place across systems or even dynamically within a single running system.

An object system a one whose run-time form is described by a structure of objects, and whose run-time behavior is described by the interactions between those objects and their effects on each other via their interfaces. Many objects will reflect problem domain concepts. An object design is one in which the structure of the system is based on concepts known to the user, reflecting the structure of objects in the problem domain.

1.4 *Adaptive Business Driven Solutions*

Computers reflect the business they support

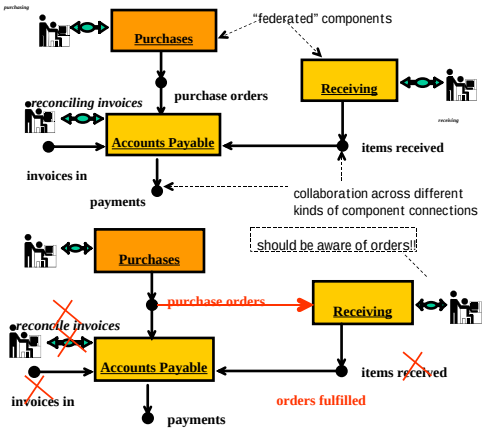
The relationship between open systems and businesses is simple. Departments in an organization have their own computing machinery, and so do individuals in each department. Each machine is supposed to support the activities of its owner. The structure and flow of business interactions between individuals and departments are reflected in the structure and flow of interactions between their machinery. In fact, enterprise software systems, almost by definition, should meet three basic criteria:

- support the business
- adapt rapidly with the business
- at all times be sufficiently functional and timely

Business change implies software change

What happens when the company is reorganized? Obviously the interactions between machines are reorganized in parallel. This is exactly what object technology has always been about: reflecting the essential structure of the business world in a manner which enables systems to adapt with the business they support.

The following example is adapted from [Mabey et al]. A classic business process for supplies might include activities for purchasing, receiving shipments, and reconciling shipments against purchase orders before issuing payment. This business was susceptible to "dumping" — unsolicited deliveries and subsequent invoicing. The software systems and their interactions reflect this business operation.



The operations of this business were streamlined by recognizing that only supplies that were ordered should be delivered and paid for. The software components and their connections must be adapted to this new process.

Not all changes at the business level are of such a grand scale, yet they all place similar demands upon the supporting software. For example, the ability of a business to introduce a new innovative pricing plan for phone calls, or a customer loyalty-based purchasing incentives, may be entirely constrained by the ability of the enterprise software systems to adapt to such a change.

Since most software requirements arise from some change, or opportunity for change, in the business -domain, they are best formulated in terms of the business model. Ideally, such business models should also form the basis of the software requirements and designs as well.

The same is true for non-business and smaller systems — they change too!

Not all systems have a “business” flavor to them, in the sense of a commercial activity that generates money. However, all systems, big or small, have to deal with evolution and change over time. These changes often come from the problem domain itself; hence, software should reflect the essential structure of its problem domain.

Continuity from problem domain to code

Hence software development must be driven by the modeling and improvement of processes to support an adaptive business. For a non-business system, the software must be based on a model of the problem domain itself. Moreover, it would be nice if similar principles and techniques are used to model the business (which, after all, increasingly consists of human roles interacting with major software-components), as are used to model and design the software components themselves.

1.4.1 Legacy Wrapping

Adapt legacy systems by “wrapping” them

In adapting to changing business needs our software systems must evolve, but they clearly cannot be discarded. “Legacy” systems must be adapted to fit into the overall architecture that is more component and object-based, perhaps utilizing CORBA or COM. Doing this requires “wrappers” — a software layer that adapts the legacy system and offers an object-like interface to the rest of the system. Wrappers range from very simple, where the entire legacy system appears as one large object, to more complex, where the software layer presents a virtual set of objects from within the legacy system.

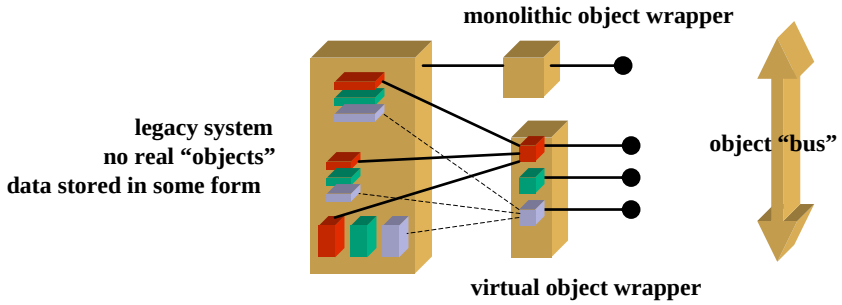


Figure 3: Wrapping legacy systems

Naturally, this poses a challenge for modeling techniques. We would like to describe the interfaces of a component regardless of whether it is implemented as a wrapper around a legacy system, or a newly built one.

1.5 *Iterative Development using Components*

Component-based development has further fueled the move towards rapid application development (RAD), with quick, iterative and incremental development of systems and applications.

Rapid iterative development proves its worth.

The trend towards rapid assembly directly complements changes in the software development process. The value of iterative development and incremental delivery of features is now widely recognized. Utilizing components and 4GL's, and working in Rapid Application Development (RAD) environments can shorten the cycle-time from requirements exploration to implementation.

It lets us converge on the right solution better

The reasons for iterative and incremental development are quite simple. Because businesses change constantly, and business needs are best understood by the business user, user involvement is an important part of software development. Often, not everything needed is known up front; nor is everything known up front truly needed! Frequent iterative and incremental delivery, if carefully planned, helps "re-vector" development effort appropriately, converging on a business solution with the right combination of function and timeliness.

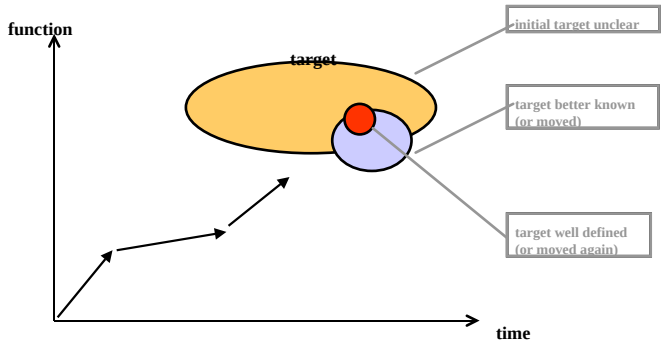


Figure 4: Iteration and requirements uncertainty

Separation of concerns is important even for iterative development.

With iterative development[1] it is still important to separate out problem domain or requirements issues from designs and code. The authors have seen more than one project in Visual Basic that made good initial progress in assembling an application, then rapidly deteriorated to where the implementation code became incomprehensible, yet it was the only available description of the application being built.

Changing code must be separate from more abstract models

In an iterative and / or incremental development lifecycle, some design decisions might be re-visited and revised more often, due to changing requirements or new knowledge about issues like performance and flexibility. Each such pass often generalizes and then re-specializes previous designs, e.g. to add new features or improve performance. If we are to succeed at iterative development without being reduced to working only at the code level, we will need very clear separation between the constantly changing code, and higher levels of design descriptions, to help us maintain our designs and models effectively.

1. The same discussion applies to any maintenance activity on an existing system

1.6 *Designing Pluggable and Generic Components*

We are increasingly building extensible frameworks.

To succeed with components, our components have to be flexible, so they can be adapted to use in different contexts. We are increasingly building frameworks for entire families of products and for wide ranges of customers, as opposed to one-time dedicated usage. Maintainability and extensibility have become dominant quality objectives, and the design challenges are correspondingly harder.

Components are units of design effort.

Building good software is about designing and plugging together components. A component is a piece of design-effort that makes sense as a unit — it can be designed, moved around, stored in a library, incorporated in a variety of designs, updated, or replaced; classes, functions, pieces of analysis, and patterns are all components in this general sense. If you don't design your software in well-defined components, then it will be inflexible and difficult to change. If you don't use previously-built components in your designs, then you're doomed to continually cover the same old ground every time you write a new application, repeating a lot of the same mistakes.

Components include implementations, designs, hardware, people, and roles at all scales.

Components can be small things doing simple programming tasks — keeping a list of items or representing a person's name; or they can be complete applications like a word-processor or a spreadsheet; or skeletal applications designed for extensibility, like an application framework; or they can be people in a workflow — operators, clerks, managers; or they can be pieces of hardware — a multiplexor or a robot arm. Components are designed from smaller components. (The components approach is sometimes called 'fractal'! — that is, scale-invariant.) If we're engineering a whole business process, our components will be roles of people and departments and perhaps some computer systems. If we're designing a computer system, the components will be pieces of hardware and software. When designing a software component, its parts should be simpler components – all the way down to the level of individual program statements. Design is a recursive process.

Good components are reusable.

In every case, we should hope to be able to use components that already exist; ideally, any new components we have to design ourselves should just represent those features that are entirely new. But for that ideal to happen, the components we put together must be adaptable. For example, a component that handles members of book-libraries is less useful than one that can be employed to handle any kind of membership. Designing a component well is not just a matter of making it work correctly, nor only about making it perform efficiently. It is also about generalizing it to be adaptable to a variety of purposes — not just the one to which it is about to be applied immediately.

Performance vs. generality

There is often a trade-off between performance and generality. In object-oriented design we emphasize features like basing the design on the business model, polymorphism and encapsulation — all in aid of genericity, but sometimes worsening run-time performance. A good object-oriented design is one which balances these needs appropriately. And if the relation between an optimized design and the business model is well-documented, the design will still be flexible, even if with a little more effort.

There is also a trade-off between genericity and design time. Making a generic component takes more effort. You get the investment back when it is eventually used in several other contexts; but the sacrifice of short-term deadlines for longer-term benefit is clearly a management issue that has to be understood and backed at all levels.

Generic vs. rapid design

This book is about the trade-offs and the technology of building from components. This isn't something we're unfamiliar with. Most designers assume some platform of existing parts — the windows system, a database, or the standard libraries that come with the compiler. But it isn't something that's universally done well or consistently — or as pervasively as we will advocate in this book.

This book shows how to build component

Well-designed components can be plugged together in two distinct ways, as shown in Figure 5:

..that can be plugged together.

- Large grained components are composed into larger applications, by writing code to utilize and co-ordinate their services into meaningful business transactions.
- Generic components themselves are customized for the job at hand, by 'plugging-in' small parts that provide application-specific versions of some parts.

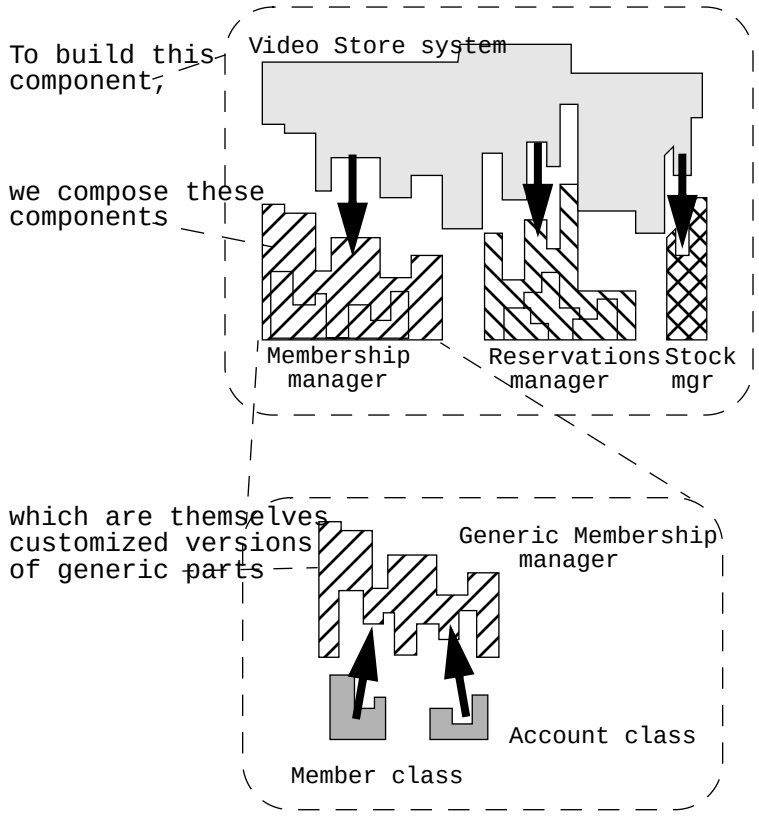


Figure 5: Generic components are customized, then composed

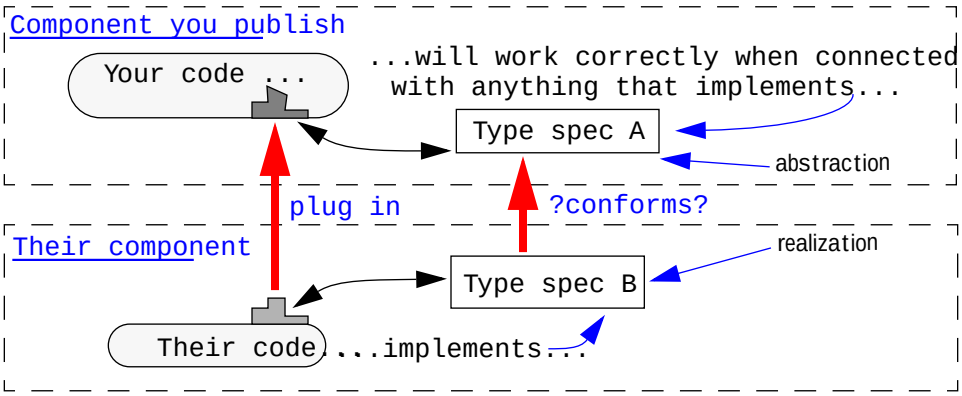
1.7 Component-Ware Challenges

Components must plug together at all levels

Like components, plugs are 'fractal': the approach is applicable on all scales. Whether we're talking about the interface to a little object that just counts up and down, or the system that runs your nuclear power station, the same basic principles apply. The component approach to design can only be used effectively if you have a way of knowing whether two components will work together properly.

This demands clear "contracts"

In component-based design, clear abstraction is essential as the means to specify the contract between a component and any others that may be used in conjunction with it. This is perhaps the biggest challenge to overcome before we can readily plug components into each other.



Big Question. When some third party puts these components together, will they work? What should you do to be sure they will work?

Figure 6: Components and plug-conformance

Conformance means meeting a contract

Conformance is an essential relationship between an abstract requirement and any realization. Documenting conformance means expressing the belief that a more specific requirement (anything down to a complete implementation) is correctly described by a more general abstraction — i.e. meeting a contract.

Component-assembly requires precise behavior descriptions.

Precise Behavior Descriptions. In order to be able to dynamically locate and use components suitable for some need, whether it be within a CASE tool environment or dynamically in cyberspace, we will need an explicit and machine-processable representation of the interfaces and behaviors involved. The behaviors will have to be specified in some mutually agreed-on form, and must quite precisely express the assumptions and guarantees that both users and implementors can make.

Re-usable components will be generic and extensible frameworks.

Designing Extensible Components and Framework. It is not realistic to expect non-trivial implementation components to suit the exact needs of every specific context in which it will be used. Hence, our component *implementations* must permit adaptation to customize them for different uses, and our component *interfaces* and *design models* must document such permitted extensions, while still accurately describing the behavioral constraints that must be guaranteed regardless of extensions.

Modeling of Frameworks. Designing frameworks which are re-usable and extensible across entire families of applications brings new challenges to modeling and methods. Such systems are typically built by polymorphic composition and extension of multiple smaller frameworks. We will need "open" modeling and specification techniques which can accommodate such extension, as well as ways to describe extension mechanisms, such as subclassing and composition, without requiring full access to and knowledge of the implementation.

Our modeling and specification techniques must be more "open".

Component Repositories. The ideas of assembling components should not merely apply to implementation activities, but also to all the related modeling, specification, and architectural design activities as well. Our component repositories should contain generic versions of models and architectural patterns that we compose and adapt to our needs. The composition and refinement of component models must be well defined.

Component repositories contain composible design models as well.

Legacy Component Models. Our modeling and design techniques should smoothly integrate legacy components as well i.e. systems that have not been built with object or component technology, but which will have some wrappers built to adapt them as needed. This means we will need an approach that describes the external interfaces of components clearly, yet abstractly enough to permit both legacy and new implementations.

Abstract descriptions even of legacy systems

This book talks about how to ensure the plug-compatibility of your components. We will see how to specify simple interfaces, then deal with more complex ones, and look at techniques for layering the complexity of an interface.

And that is what this book is about!

1.8 *Where Does Catalysis Fit In?*

This blurb summarizes Catalysis goals

The objectives of the Catalysis method are summarized in this single, buzzword-laden blurb. Taking apart the marketing jargon on this blurb, we have:

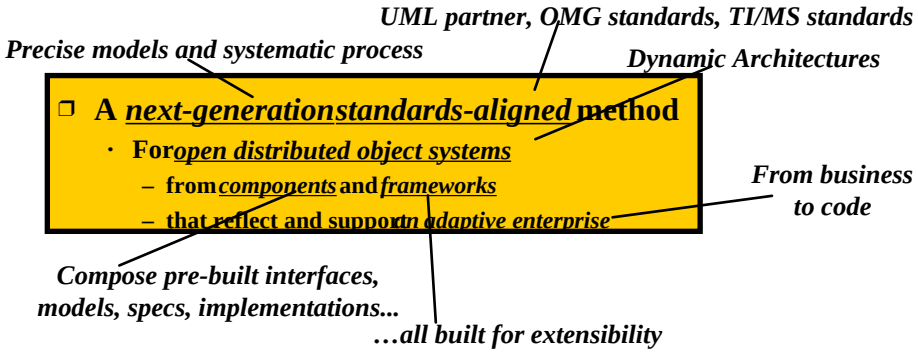


Figure 7: Method marketing blurb

- *next-generation*: Catalysis provides a systematic basis and process for the construction of precise models starting from requirements, for maintaining those models, for re-factoring them and extracting patterns, and for reverse-engineering from detailed description to abstract models.
- *standards-aligned*: Catalysis is based on, and has helped shape, standards in the object modeling world. Both authors have been involved in the OMG standards submissions for object modeling, and one has been a co-submitter in defining the Unified Modeling Language (UML 1.0, 1.1) with a consortium of companies, as standardized by the OMG in September 1997. Catalysis has been central to the component-specification standards defined by Texas Instruments and Microsoft, the CBD-96 standards from TI/Sterling, and the Sterling Cool:Cubes tool family.
- *open distributed object systems*: it is our goal is to support the modeling and construction of open distributed systems -- those whose form and function evolves over time, as components and services are added and removed from it.
- *components*: little, if any, modeling or implementation work should be done from scratch. If you draw two boxes and a line between them, chances are someone has done something very similar before, with an intent that is also very similar, if you only abstract away certain specifics. All work done in Catalysis can be based on composition of existing components, at the level of code, design patterns and architectures, and even requirements specification.
- *frameworks*: in particular, some of these components are built so they are easily adaptable and extensible. We call these components “frameworks”, generalizing somewhat the traditional definition of a framework as a collection of collaborating abstract classes.
- *adaptive enterprise*: and, we want to use these techniques from business to code and back. Catalysis provides novel support for abstraction and refinement, enabling true support for business-to-code models, and a strong foundation for use-case driven modeling and design.

Catalysis effectively addresses some of the problems facing the next generation of software development. The method was developed — at a time when some argue there is scarecely need for another method — with the specific goals of being:

This book tells how we met some of these goals.

- Simple: all of Catalysis can be reduced to utilizing three basic constructs at three levels of description. This simplicity belies its expressiveness; different diagrams can be used at each level, but their purpose and scope is clearly defined. For example, state diagrams are not a pre-defined specification device in the method; rather, they are a particular visual representation of attributes, invariants, and operation specifications. Even the parts of Catalysis that use rigorous notations deliberately use a combination of information and formal descriptions, and can be adopted in a "light" version on any project.

  While this book will spell out the rigor possible with Catalysis, which may seem intimidating to some, the general philosophy in applying it is one of 'just enough' rigor; there is a light-weight path through Catalysis.

- Sound: All the diagrams[1] used in a Catalysis development have a precise meaning, and there is little room for ambiguity in interpretation; even abstract descriptions convey precise meaning. This applies from the level of capturing user reqiurements and business rules, all the way to document interfaces of code components; abstraction combined with precision makes a nice combination.

- Systematic: the deliverables and their relationships to each other are very precisely defined, as are suggested techniques for developing and checking them. Catalysis also provides a clear document structure, and even detailed documentation templates that can be adapted depending on project needs and CASE tools used. This provides the foundation for a repeatable and predictable development process.

- Standard: diagrams used in Catalysis are based on the Unified Modeling Language, a standard set of constructs and notations for describing object and component systems. What Catalysis adds is a precise interpretation of these diagrams to greatly simplify their usage, and a development process that clearly separates concerns.

- Complete and consistent: all deliverables produced in a Catalysis development have clearly defined criteria for consistency with other artifacts, and for completeness against the more abstract requirements they fulfill. These can be used for automated checking, for design reviews and inspections, and for progress monitoring. The method and its techniques are applicable from business level to code.

- Traceable: The Catalysis concept of refinement provides concrete traceability between abstract requirements and the realizations of those requirements, from the level of business rules to code. Moreover, this is achieved without giving up the separation of concerns that is so important in team and component-based development.

- Scalable. Too many methods today offer either little bits and pieces ot techniques and notations without the development framework needed to scale to large projects, or else provide guidance for large project planning and lifecycle management without a sound technical basis underlying them. Catalysis aims to provide

1. Informal sketches and doodles will always be invaluable in their own way, but should typically be evolved into more precise descriptions.

a tailorable combination of process, techniques, and precision that can be adapted to different projects.

- Accessible: Catalysis has been used on projects of many sizes. Its current support includes training courses and mentoring, a growing collection of certified practitioners, method documentation, case studies, and increasing tool support from commercial tools.