SCIPIO

Component-Based Development & Open Distributed Processing

Author: Richard Veryard
Version: August 12th 1998

richard@veryard.com
http://www.veryard.com

For more information about SCIPIO, please contact the SCIPIO Consortium.

info@scipio.org
http://www.scipio.org

Preface

Purpose of document

This document provides an overview of recent trends in IT. It shows how CBD is the practical realization of OO and Distributed Processing. It provides a technological background for SCIPIO.

The document is intended for a non-technical audience. It attempts to make simple statements about the technologies under discussion, rather than providing detailed technical explanations and justifications.

Questions and exercises

This material has been developed for training purposes. The reader is invited to engage actively with the material. To this end, questions and exercises are interspersed with the text.

Q Do you want to enrich your understanding of the SCIPIO method by answering the questions as you along?

Q Do you want to test your understanding of the SCIPIO method by answering all the questions after you've read the whole document?

Acknowledgements

Some of this material was presented at a Unicom CBD Seminar in October 1997.

Overview

- Both Information Engineering (IE) and Object Orientation (OO) are powerful paradigms for systems development. But both have reached their limits.
- Traditional reuse is attractive, but difficult and expensive.
- Component-Based Development (CBD) offers a breakthrough in systems development. Reuse is achieved through defined interfaces. This achieves reuse more effectively than IE module reuse or OO class libraries.
- CBD has a good story on legacy reuse and on migration/transition.
- CBD provides key to distributed processing, and provides technical flexibility as well as business flexibility.
- Industry support for CBD is growing rapidly. This includes support from most of the major players in the computer industry, including: Microsoft, IBM, Sun, Rational, Select, Sterling, and many others.
- Meanwhile, there is growing user interest in CBD. This is seen in the formation of industry bodies such as the CBD Forum.

Introduction

How does IT support business?
And why does business support IT?

One day, a hot air balloon came down low over a field. The balloonist called down to a man in the field: "Excuse me, can you tell me where I am?" The man pondered for a while, and then replied: "You are in a hot air balloon, 15 metres above a field."

"You must be an Information Technologist." said the balloonist. "Your information was technically correct, but of no use whatever. And you took your time answering me as well."

"You must be a manager." replied the man, "You don't know where you are, you don't know where you're going, and asking me hasn't helped you a bit, but it's now all my fault."

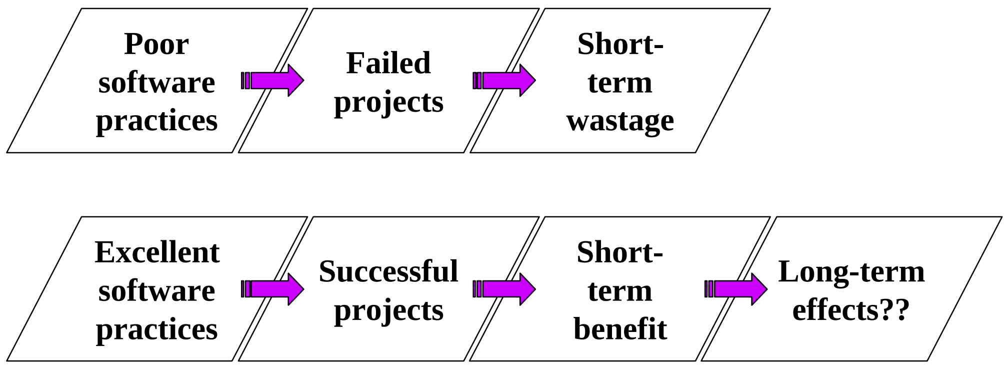
Information Technology failures are commonplace. Statistics can be obtained from many sources, showing alarmingly high rates of software wastage and project cancellation. Furthermore, many case studies have been described in which IT systems have had a disastrous impact on the business. An IT system that cost less than a million dollars to design and implement may contain an error that will cost the company tens of millions to put right. In extreme cases, the survival of the whole business will be put at risk.

Q How many IT failures have you read about in the papers?

Q How many IT failures have you personally experienced? Did any of these reach the papers?

Some cynical commentators have suggested that, if there is any statistical correlation at all between IT expenditure and business profitability, it may be simply because profitable companies have more money to waste on IT.

Many writers attribute all this wastage to poor software practices. Let us suppose they are right, and focus the question on successful IT projects. In the best-case scenario, how can IT be beneficial to business?



A successful IT project enhances the adaptation of the business to its environment. But adaptation to a specific situation usually compromises adaptability to future situations. For example, each more sophisticated marketing strategy may increase the interdependence between a business and its market, making it more difficult for the business to develop in new markets. As a company develops ever more flexible and powerful techniques of direct mail marketing, and embeds these techniques in its IT systems, it reduces its ability to use any other marketing channel.

Furthermore, as some management gurus have pointed out, success can be treacherous. As IT pervades a business organization, the organization has an ever-increasing investment - both financial and cultural - in an emerging configuration of formalized systems, which IT both enables and encourages. This formalization always omits something important. Furthermore, the identity of the organization is bound up with its systems. Thus the very systems that allow the business to survive in the short term may impede its survival in the medium or long term.

Technological advances claim to make IT systems more flexible.

Component-Based Development (CBD) can improve software quality and productivity.

Component-Based Development offers:

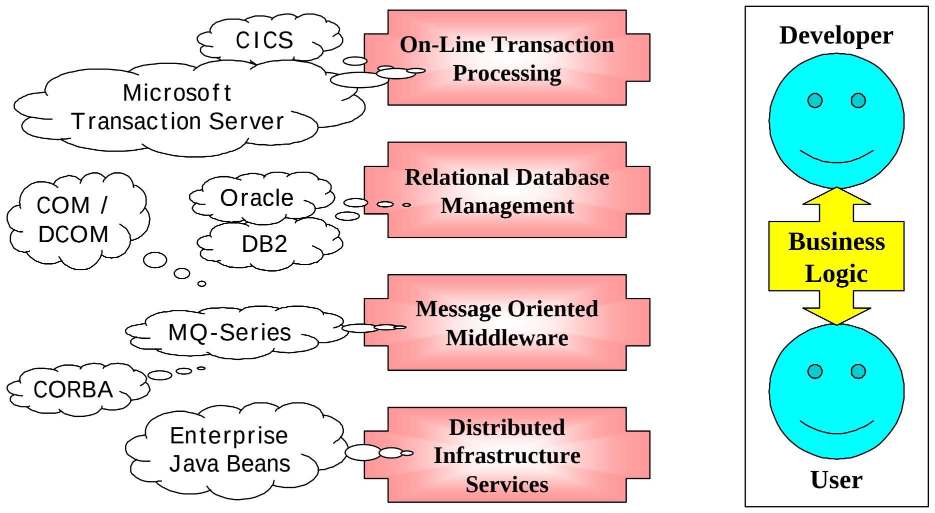
- "Plug and play"
- Assemble systems from available components.
- Greater choice of component services.
- Ability to reconfigure system and/or swap components to fit changing requirements.

Open Distributed Processing (ODP) supports a business drive for greater flexibility.

Open Distributed Processing offers:

- Ability to hide location.
- Greater choice of platforms and infrastructures.
- Object orientation (e.g. CORBA)
- Distributed databases and client/server
- Interoperability of heterogeneous systems
- Federated systems

Middleware hides technological complexity from developer and user.


CICS
Microsoft Transaction Server
On-Line Transaction Processing
COM / DCOM
Oracle
DB2
Relational Database Management
MQ-Series
Message Oriented Middleware
CORBA
Enterprise Java Beans
Distributed Infrastructure Services
Developer
Business Logic
User

At any point in the history of computing, there has been a set of technical issues and difficulties faced by the developers of application software. From time to time, some of these issues are taken away from the application developers, and automated into software devices that sit between the application program and the computer. This class of devices includes operating systems, programming language compilers and interpreters, file management and database management systems, messaging systems and other infrastructure services. Fourth-generation languages and CASE tools may also be seen as belonging to this category. I use the term **middleware** to denote all such devices.[1]

Ever since the launch of COBOL in the early 1960s, such devices have been presented as liberating. They may either permit the application developer to focus on the business logic and other aspects of the users' requirements, without unnecessary technical distractions, or they may permit so-called end-users to develop their own software applications. This has led to some writers forecasting the end of the application programmer.[2]

But while some technical complexities are taken away from the application developer, new technical opportunities inevitably introduce new technical complexities. Some people within Information Technology see this rapidly growing complexity as a personal and professional threat: the complexity seems to them to be escalating to infinity, or at least to a point where only a super-elite of programmers in specialist software factories would be able to master the technical issues.

But if we take a long view, we can see that the set of technical issues faced by the application developer has neither escalated to infinity nor dwindled to nothing, but remained about the same size over a long period.

Availability

The trend of technology is towards 'infinite' availability.[3]

[1] Some writers prefer to use the term middleware in a more restricted sense.

[2] James Martin and Ed Yourden are among the pundits who have forecast the end of the application programmer.

[3] Borgmann refers to this as the Device Paradigm.

Ideal		Reality
Total satisfaction	*Instant gratification*	More rapid Faster response
	Wherever I want it	More widely available Distributed, global New markets, new customers, new technical platforms
	Whenever I want it	More often available 24 hours, 7 days, minimum downtime
Total quality	*Zero difficulty* *Zero defects* *Zero risk*	Easier to use Reduced skill barriers, faster to get started Improved quality Greater reliability, predictability, security, …

These trends are exemplified by both Component-Based Development (CBD) and Open Distributed Processing (ODP). These technologies significantly increase the availability of computing power, although they do not (and perhaps never can) live up to the infinite expectations of some of their adherents.

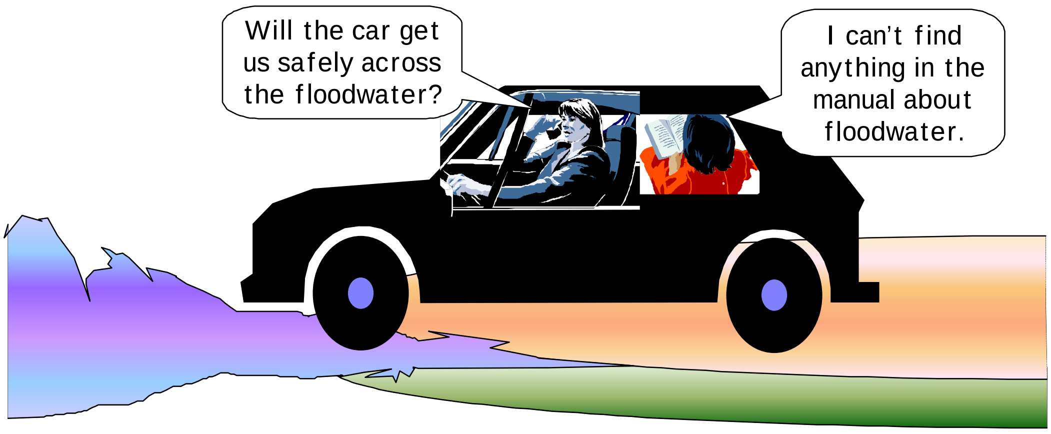
Recent technology trends support the current business agenda.

Old technology focus	**New business focus**
▪ Use IT to create ‘strategic’ applications ▪ Simplify and integrate the business processes	▪ Object technology & component-based development ▪ Workflow management & open distributed processing ▪ Internet/intranet/extranet
New business agenda	**Technology support for new business agenda**
▪ Total Quality Management ▪ Organizational Learning ▪ Customer Relationships ▪ Knowledge Management ▪ Strategic Alignment	▪ Object modelling ▪ Interface management

For more about the new business agenda, see the companion document in this series on **Business Change and Process Improvement**.

Reuse

A good component satisfies unexpected requirements.


Will the car get us safely across the floodwater?
I can't find anything in the manual about floodwater.

It is a rainy night. Jan is driving home, turns a corner, finds the road flooded. The car has been designed and tested for a range of driving conditions , but can it drive through shallow water? There is no reference in the driver's manual to floodwater.

Jan actually has two requirements

1. for a vehicle that can get them safely through the water hazard

2. for information as to whether (and with what modifications) the actual car is capable of satisfying the first requirement

Note the following points:

Jan does not want to be stranded in the middle of the floodwater, and will not attempt to drive through it unless confident of getting through safely. Therefore, requirement (i) is not satisfied unless requirement (ii) is also satisfied.

Would an inability to drive through floodwater count as poor quality? Jan's requirements were not stated when buying the car.

Jan's two requirements are interdependent. Therefore quality evaluation must apply to the car and the driver's manual together, not one independently of the other.

Q Think of some situations where this notion of quality would be relevant to software components.

As consumers, we can often choose between separate components or integrated solutions.

For example, when buying a home computer, hifi or video or camera equipment. Or when making decisions about personal finance and investment. Some people prefer to buy whole packages of products and services from the same supplier, while others prefer to combine things from different suppliers.

Q When else have you been faced with this kind of decision? Do you have a recent example?

How do you choose between separate components and an integrated solution? You might take some of the following factors into consideration:

- Greater control
- Reliance on other people's expertise
- Convenience
- Ease of repair and upgrade
- Cost of purchase and repair

Q Are there any other factors you would consider?

Q Under what circumstances do you prefer separate components? Under what circumstances do you prefer an integrated solution?

Q Now apply this thinking specifically to software. Under what circumstances would you choose an integrated software solution? Under what circumstances would you prefer to procure components separately?

There are many good reasons for software reuse.

Improves software product

Makes information systems more consistent.

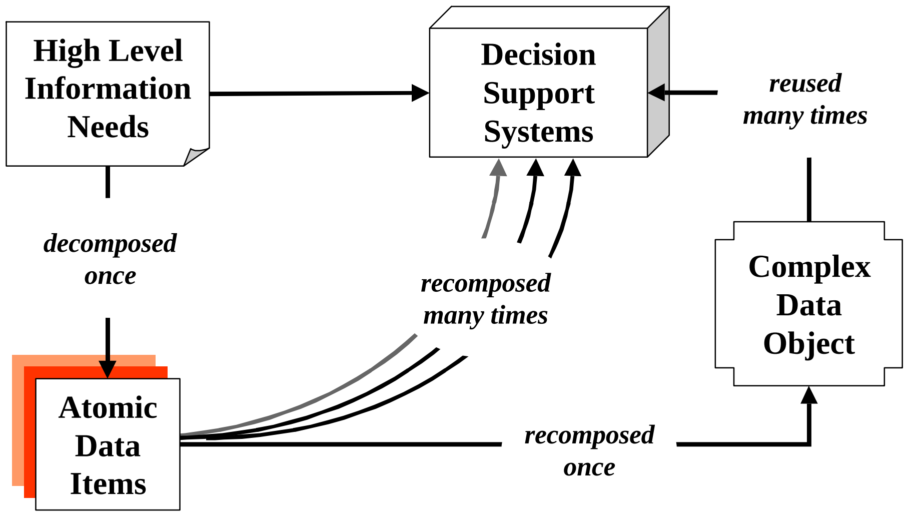


Figure 1: Reuse of Complex Data Objects

Top-down methodologies such as Information Engineering derived an understanding of the data requirements (at least in part) from high-level management information needs. The data requirements were decomposed into atomic data items: entity types, relationships and attributes.

But every time a developer or user wanted to construct a system to support these high-level information needs, she was obliged to compose data enquiries from the atomic data items. This was not only time-consuming but often error-prone. Management decisions could then be based on comparisons between inconsistently calculated data.

Q Can you provide some examples of this?

By defining a complex data object as in Figure 1, and making this available for use within management information and decision support systems, the composition and calculation of management data can be made much more consistent.

With component-based development, the complex data object is implemented as a component, which hides the composition of the object from the atomic data items.

Note that the complex data object could be supported by relational database management systems (RDBMS), by using relational views or subschemata. However, some RDBMS-based CASE tools did not provide support for these RDBMS features.

Enables more efficient testing.

For more discussion on this point, see the companion document in this series on **Software Component Quality and Testing**.

Improves whole product

Allows reuse of end-user skills, training & documentation.

> **Q** Can you provide some examples of this?

Supports improvements to business process

Rapid deployment of business best practices.

This is illustrated in the companion document in this series on **Business Change and Process Improvement**.

Improves software development process

Decouples development of separate components.

Faster build from existing components.

Improves software maintenance process

Facilitates replacement and upgrade of components.

To manage reuse, we need good metrics.

- ❖ What is desired level of reuse?
- ❖ What is current level of reuse?
- ❖ How to improve the level of reuse?
- ❖ What is improvement worth?
- ❖ How much to invest in reuse?
- ❖ How to manage the improvement?

It all depends what you mean by software reuse.

There are three possible ways of defining software reuse.

Static reuse can be defined in terms of the number of source code references to my component, or the number of software items that refer to my component.

Static reuse generates application benefit:

- ➢ Faster development

- ➢ Easier maintenance

Deployment reuse can be defined in terms of the number of consumers with access to services provided directly or indirectly by my component.

Dynamic reuse can be defined in terms of the frequency of execution of my component.

Business benefit comes from high levels of deployment reuse and dynamic reuse. This can often be achieved without high levels of static reuse. However, a lot of software engineering is focused on static reuse.

Reuse won’t happen unless you manage it properly.

Opportunities created by CBD

Software supply chain gets more complex.

- ❖ Extra ways of paying for software development and use.
- ❖ Extra demands for quality assurance.

New organization structures.

- ❖ Component factory.
- ❖ Reuse across organizational boundaries.

Prerequisites for CBD

More roles include:

- ❖ Component developer
- ❖ Component librarian
- ❖ Component broker / trader
- ❖ User of component service

Requires management support.

Component-Based Development

What is CBD?

Component-Based Development offers a radically new approach to the design, construction, implementation and evolution of software applications. Software applications are assembled from components from a variety of sources; the components themselves may be written in several different programming languages and run on several different platforms.

At first sight, Component-Based Development might seem to be little more than a fashionable new label for some traditional software ideas: modular programming and subroutine libraries. Even in the 1960s, these ideas promised high levels of software reuse (although this was rarely achieved). But to the extent that CBD is a genuine innovation, this is to be found in its approach to legacy systems: some of the most significant potential cost-savings associated with CBD involve extracting (or 'mining') components from existing code. It is perhaps this element of CBD that arouses the greatest scepticism, and offers the greatest potential rewards.

Components

A component is something that can be deployed as a black box. It has an external specification, which is independent of its internal mechanisms.

Components inherit much of the characteristics of objects in the OO paradigm. But the component notion goes much further than the OO object notion. OO reuse usually means reuse of class libraries in a particular OO programming language or environment. You have to be conversant with SmallTalk or Java, to be able to reuse a SmallTalk or Java class. You can reuse a component without even knowing which programming language or platform it uses internally.

The same specification may be implemented in several different ways. Furthermore, as the specification is a description of the behaviour of a component, and the behaviour may be described in several ways, the same component may satisfy many different specifications.[4]

[4] Some support systems do not recognize the possibility for a component to satisfy many specifications. This is a serious limitation to the reuse opportunities provided by these systems..

Interfaces

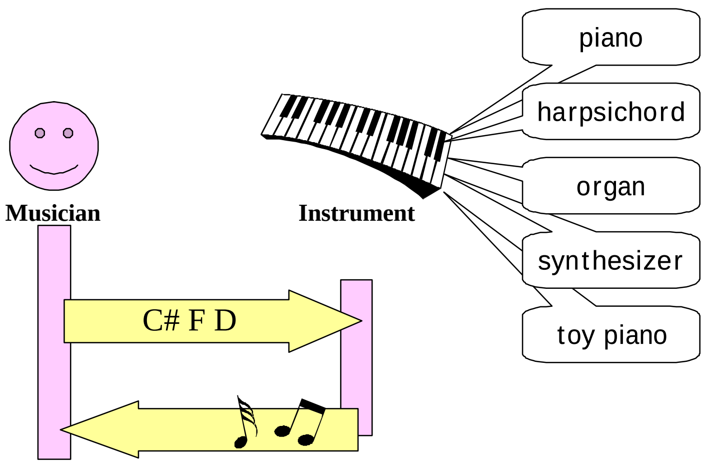


Figure 2: Musical keyboard as standard interface

A musical keyboard provides an interesting example of a standard interface. When I press a particular sequence of keys, I expect an instrument to respond with a particular sequence of notes. Many different instruments provide the same interface: pianos, harpsichords, organs and synthesizers. Some music software has a virtual implementation of the same interface: it provides a visual display of the same pattern of keys, but these keys are pressed with mouse clicks rather than directly with the fingers.

But there are also important differences between the various implementations of this interface. Some instruments are powered by electricity: this means that a complete specification of the interface includes the precondition <*instrument is switched on*>. On a piano, the keys also control the volume and duration of the note. On an organ, the keys control the duration of the note but the volume is controlled elsewhere. On a harpsichord or a toy piano, you get the same short note however long you hold down the key. On some toy pianos, you only get one note at a time however many keys you press.

Thus the piano satisfies one interface specification that is common to other keyboard instruments, but it also satisfies another interface specification that is unique to pianos.

Q How would you specify the interface so that only the piano was a valid implementation of the interface?

Q How would you specify the interface so that all the instruments mentioned were valid implementations?

What kinds of components are there?

Source

- External component libraries
- Inhouse component infrastructure
- Mining components from legacy systems

Granularity

- Small components (e.g. Active-X controls)
- Medium components (so-called "business objects")
- Large components (e.g. applications packages, legacy subsystems).

What kind of component processes are there?

The discussion of component-based development often concentrates on the creation of components and their assembly into applications, but there are several other important processes to consider.

Creation

Building new components, (re)using available stuff where possible.

Assembly

Building applications from components.

Encapsulation

When is a component not a component? One of the most important characteristics of a component is said to be something called encapsulation. What exactly is encapsulation?

Encapsulation is not an attribute of a lump of 'code' that makes it into a component. It is a characteristic of a relationship between a lump of 'code' and a person or role. This or that control is accessible to the end-user, these workings are not directly accessible to the end-user.

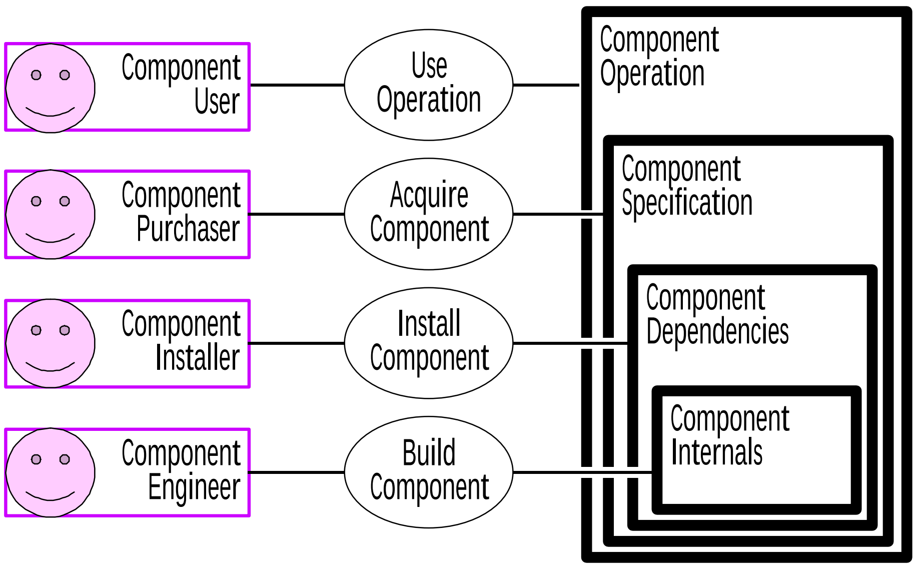


Figure 3: Four Levels of Encapsulation

Figure 3 shows four levels of encapsulation.

1. The unit of use is the operation. The user of a component (which may be human or software) only needs to know the specification of the operation(s) being used. The user doesn't even need to know that several operations are actually performed by a single component.

2. The unit of delivery is the component. The purchaser of a component needs to identify and acquire whole components.

3. If a component calls other components, then the availability of these other components is a precondition for a successful installation of the component. The installer, therefore, needs to know what calls are made to other components.

4. Most of the people who interact with a component can be satisfied with the above. The only person that needs to look inside it is the software engineer, who is charged with building, inspecting or modifying the internals of a component.

Publication

The publication of a component is the act of making it available to a defined public. There may be nested publics - thus a component may be published internally, or to beta customers, before it is published to the entire marketplace.

Alternatively, a beta version of a component may be made freely available for a limited time period, and then withdrawn, to encourage people to pay for the commercial release.

Publication usually involves some form of publicity. This is part of the publication process, and not an optional extra.

Dissemination

The word 'dissemination' originally refers to the process by which a plant spreads its seeds. Some plants launch their seeds into the wind or pop them into the air, others use animals or birds to carry their seeds great distances. Some seeds fall on stony ground, some seeds lie dormant in dry sands for many seasons until the rains come.

Similarly, some components may be tossed around by the random winds of the Internet, while others may be safely carried to fertile application ground. The original developer of the component cannot know - but may try to influence - how and when and where and by whom and for what purpose the component may be used.

Matching

A would-be component user is looking for a component. Perhaps with a definite specification; perhaps only with a rough idea of what is needed. Meanwhile the component publisher is looking to maximize the use of a given component. To focus either on the searching process or on the publication process is to see only half the picture. We should rather think of this as a matching process, in which the user and publisher collaborate. This collaboration may be initiated either by the user (demand-pull) or by the publisher (supply-push).

CBD Trends

Increasing automation and maturity of tools

Existing CASE and OO vendors are starting to move into the CBD marketplace, and there are some exciting new players as well. Use of the current tools still demands considerable manual effort. Shells and interfaces often have to be built by hand, and the matching of component requirements is poorly supported. Over the next couple of years, we can expect the tools to become more powerful and easy to use, covering a greater range of the CBD processes. This will enable still greater productivity and scaleability.

Greater differentiation of software work

Each advance in software engineering makes it more difficult for the software engineer to remain a true generalist. Some software engineers will become specialist component engineers, deploying formal methods to create and verify robust and widely reusable software components. Others will concentrate on satisfying local business requirements, designing context-sensitive and user-friendly interfaces.

From component scarcity to component abundance

A lot of the discussion of component-based development today is based on the premise that good components are scarce. A lot of the techniques are based on this mindset of component scarcity. However, if CBD takes off, we will quickly move into a world of component abundance. Indeed, we may even get into a world of component pollution, in which it becomes an antisocial act to overpublish redundant components.

Increasing variety while decreasing variation

Standard components can be configured in countless ways. For a business using CBD effectively, this establishes the possibility of mass customization, whereby the business can respond flexibly to the needs of individual customers, without losing economies of scale and scope.

From component to commodity

An interesting historical analogy of componentization can be found in the early history of the bicycle. Bicycles were manufactured in Coventry, using components from several other industrial towns and cities. See Figure 4.

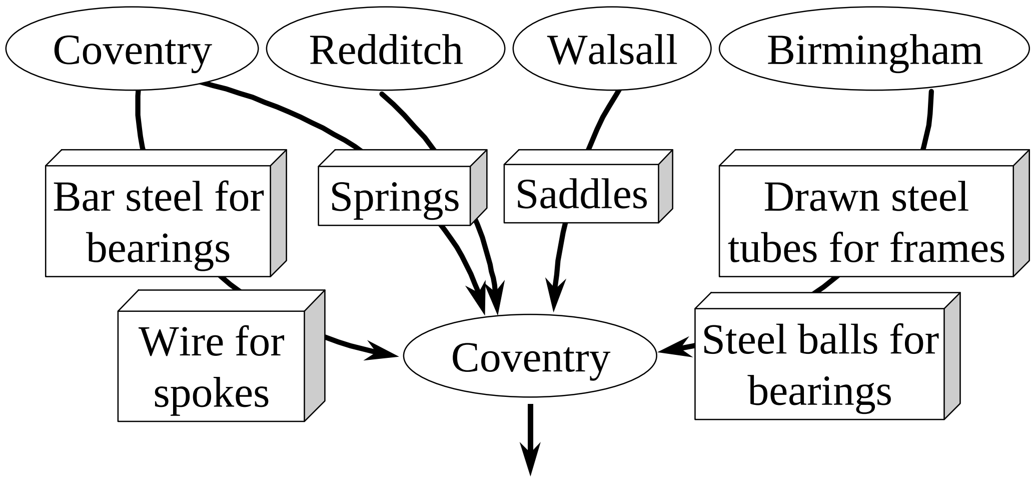


Figure 4: Bicycle components in the 1870s.[5]

‘Since the beginning of the cycle industry, local blacksmiths and mechanics have participated in constructing small numbers of bicycles to order. … Standardization … had two opposite effects on industry: it further enhanced mass production, and it strengthened the position of those small workshops. A considerable number of local bicycle makers could offer a “home-made” product to the residents of their small village at a price somewhat lower than that of factory-produced bicycles because of their lower overhead costs. Some large companies had specialized in the manufacture of standardized components, delivering them to both bicycle factories and local workshops. Thus three classes of machine could be distinguished. First, there were the mass-produced bicycles made by bicycle factories. Only the largest of these factories manufactured all components themselves; most of them had contracted out the manufacture of saddles, tires, and the like. Second, there were bicycles made by local workshops, constructed from proprietary components made by specialized forms. And the third class of bicycles, made by special departments of factories as well as by small workshops, was known as “de luxe” machines, produced without much regard for costs.’[6]

Q Can you identify similar trends in software? Do you have any real evidence that this is already happening? Where would you expect these trends to lead?

Another interesting feature of this historical example is that this industry structure, which was established in the 1870s for the bicycle industry, subsequently provided the capability to manufacture and service motor cars. Thus the early motor car industry reused the factories and workshops, inter-firm relationships, engineering and commercial skills, and even some of the specific components and tools, that had been developed for the bicycle industry some decades before.

Benefits of CBD

General features

- Separation of specification from implementation.

[5] Source: Bijker, p 35

[6] Bijker p 96

- Tightly contained improvements - permits local autonomy.
- Design with interfaces - offers wider reuse.
- Generic components (“frameworks”) can be designed around business relationships, not around specific tasks.
- Greater flexibility of control. Componentized data structures can be more flexible with respect to changing business information needs.

How can CBD improve the business process?

If a software application is assembled from components, then it should be easy to reconfigure the components to support desired changes in the business process.

Business processes may be improved in three ways:

Simplification.	Removing one or more steps from an unnecessarily complicated process, or reducing unnecessary variety in the process.	This can often by achieved by replacing components to produce stepwise improvements.
Integration.	Joining two or more previously unconnected or uncoordinated processes into a larger process.	This can often by achieved by adding components to create new links.
Transformation.	Creating a radically new process.	This can often be achieved by disassembling the components, and putting them back together in a new way.

How can CBD improve the software application?

There are several aspects of the quality of a software application.[7] We can describe how CBD contributes to each of these characteristics.

Functionality	Use of pre-existing components allows faster delivery of greater functionality. (See also Figure 5 below.)
Maintainability	The modular structure of a component-based solution allows individual components to be replaced easily.
Usability	Use of standard components supports commonality of GUI. CBD also supports desk-top integration, which gives the user a single view of heterogeneous data.

[7] This breakdown is based on ISO 9126.

Efficiency	Performance bottlenecks can be identified, and the need for performance tuning can then usually be localized in a small number of performance-critical components. Components can be internally optimized to improve performance, without affecting their specification; components can be moved between platforms to improve performance, without affecting the functionality or usability of the application.
Reliability	Given a formal and complete specification of a component, the reliability of a component comes down to the simple question: does the component provide a correct and complete implementation of its specification. The reliability of the application as a whole is a separate issue, but is clearly enhanced when the application is constructed from reliable components.
Portability	The specification of a component is platform-independent. So a component can be quickly rebuilt for a new platform, without affecting any other component. (With model-based development tools, this will often require no alteration to the internal design, merely a regeneration of the executable portion.)

In support of the functionality argument, Figure 5 shows how, in a typical situation, a traditional application package only covers some of the user requirements, whereas a number of fine-grained components may be able to come much closer to covering the whole set of requirements.

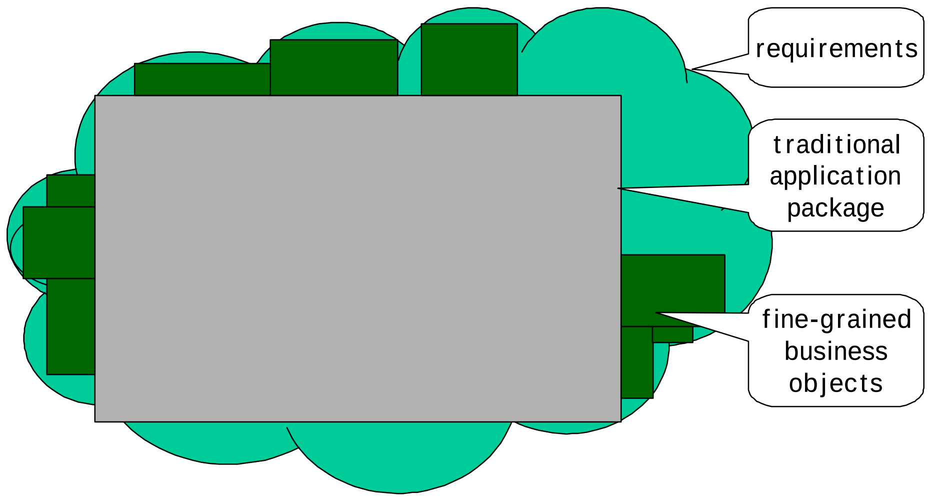

requirements
traditional application package
fine-grained business objects

Figure 5: How components improve functionality.

How can CBD solve problems of Year 2000 and EMU?

Legacy code is often compared to spaghetti, but lovers of Italian food will recognize that pizza provides a much better analogy. You try to cut out a wedge of pizza, but it remains connected to the rest of the pizza by innumerable strands of elastic cheese.

In conversion projects such as Year 2000 or EMU, the task is to get the entire legacy portfolio compliant with a specific new requirement, such as four-digit dates or a change of currency. The challenge is to break this task into manageable chunks.

A chunk of legacy system makes a component. Each chunk needs to be tested and implemented separately. Clean interfaces need to be defined between the chunks, so that converted chunks can interoperate with unconverted chunks. Management will be reassured when they see a growing number of converted chunks successfully tested and incorporated into production systems. As each chunk of legacy system is converted, it is provided with two parallel interfaces: one for communicating with unconverted chunks, and one for communicating with converted chunks.

The alternative is a very high-risk strategy: big bang conversion, where the entire legacy portfolio is converted from non-compliance to compliance in a single overnight integration test and implementation.

Component-based development is therefore the only low-risk strategy for Year 2000 and EMU conversion projects.

How can CBD improve the software process?

- Time to deliver
- Cost to deliver
- Risk of delivery projects

Industry support for CBD

Why is CBD counter-intuitive?

Suppose you were explaining pottery to a Martian. You'd have to explain what a pot was, and what it was for. Pots are used for a variety of purposes, including the storage and serving of liquids (such as wine and oil), the display of cut flowers, and cookery. You'd have to say that one of the main characteristics of a pot was its ability to hold water.

Then you'd start to describe the process: The potter scoops up some wet clay from the bottom of a river; she slaps it around, spins it and shapes it, before sticking it in an oven. 'Hold on', says the Martian, 'Do you really expect me to believe that a potter, however skilled, can make a waterproof pot out of wet clay?'

And when you put it like that, of course, it **is** hard to believe. If we hadn't seen it done with our own eyes, we'd start to doubt it possible.

Component-Based Development may seem equally implausible. The software engineer scoops up some wet COBOL from a legacy system, slaps it around a bit, wraps a few shells around it, and voila! - a fully interoperable, CORBA-compliant business object.

Component Issues and Prerequisites

Requirements	How do we know what we want to do?
Architecture	How do we balance structure with flexibility?

Investment	Who pays for an organization to get into component-based development? Not only are new tools and skills required, but the acquisition and verification of the components themselves may incur an upfront cost. Either this comes from the budget of a single development project - in which case it will be expected to payback within this project. Or it comes from some central infrastructure budget - in which case there are other management issues to address.
Procurement	To what extent do traditional software procurement processes conflict with CBD?
Security	To verify the security of your information and systems, you may need to look inside the components you are using, and it may be important to know their physical location. According to CBD purists, we aren't supposed to be able to do this.
Feature Interaction	How do I know that two components I want to use might conflict in ways I don't yet know about? (This is a problem already widely studied in the telecoms world, and is now starting to become relevant for software engineers.)
Testing	A component may be expected to satisfy many different requirements, in many different contexts. The original developer of the component may be unaware of the uses to which the component may be put. So how can it be adequately tested? For a more detailed discussion of this issue, see the SCIPIO document on **Component Quality and Testing**.
Configuration Management	Many organizations aren't very good at keeping track of a few large applications. How are they going to keep track of a vast number of much smaller components?
Tools	What tools do I really need? And how do I make effective use of them?

Standards

Component-based development relies on a commonly accepted set of interface standards. Such standards are emerging.

Vendor support

There are several tool vendors who have been actively pushing the concept of Component-Based Development for some time, before it started to get fashionable. This list includes: Microsoft, IBM, Select Software Tools, Sterling Software and SuperNova.

Then there are tool vendors who have announced support for the concept only after it has become fashionable. Such belated support can be interpreted in various ways. In some cases, it may mean that the concepts were already fully incorporated into the product architecture, but the marketing department has only now decided to draw attention to this. (This is a perfectly legitimate strategy: not every vendor has the marketing resources to educate the public in an entirely new concept.) In other cases, it may mean that existing products and documentation have been hurriedly (and perhaps superficially) retrofitted to enable the firm to jump onto the bandwagon.

Finally, there are several application package vendors, who are yielding to customer pressure to deliver their packages as suites of components with published interfaces, to allow greater flexibility to the users of these packages.

User interest and adoption

User interest in Component-Based Development is evidenced by the CBD Forum, which has attracted membership from a large number of large user organizations.

Much of the early work has been carried out within the Finance sector, especially Insurance.

Open Distributed Processing

ODP brings additional technological opportunities

Object orientation	▪ trading / broking ▪ various forms of transparency
Distributed databases	▪ geographical ▪ multiple ownership
Client / server	▪ two-tier ▪ three-tier
Heterogeneous systems	
Interoperability	▪ within organization ▪ between organizations
Federated systems	

ODP supports a business drive for greater flexibility.

Strategic Flexibility	▪ outsourcing ▪ inter-firm cooperation ▪ 'Liberation Management' (Tom Peters)
Organizational Flexibility	▪ decentralization ▪ transient organization
Technological Flexibility	▪ portability ▪ capability ▪ …bility

Open Distributed Component-Based Systems do not conform to traditional assumptions.

Beyond two-valued logic

In a closed non-distributed system, all enquiries have a definite answer. For example, either the customer record is found, or it is not found.

The behaviour of a component of such a system can therefore be specified using simple two-valued logic. If condition is TRUE then action or outcome is X; if condition is FALSE then action or outcome is Y.

In a distributed system, an enquiry may not have a definite answer. One simple reason for this may be that a remote server containing customer data is currently unavailable.

To specify the behaviour of a component of a distributed system, we may need to use three-valued logic. If condition is TRUE then action or outcome is X; if condition is FALSE then action or outcome is Y; and if condition is UNKNOWN then action or outcome is Z.

(A logically equivalent alternative to using three-valued logic explicitly is to define lots of exception conditions. But this tends to make the specifications more complicated.)

In an open distributed system, there are many other reasons why the results of an enquiry may be indeterminate. For example, an Internet search using the same search engine with the same parameters may not yield the same results twice.

Beyond traditional testing

Besides making the specification of components more complex, the indeterminacy referred to in the previous section makes it much harder to test and debug systems and components, as it often proves impossible to replicate anomalous results.

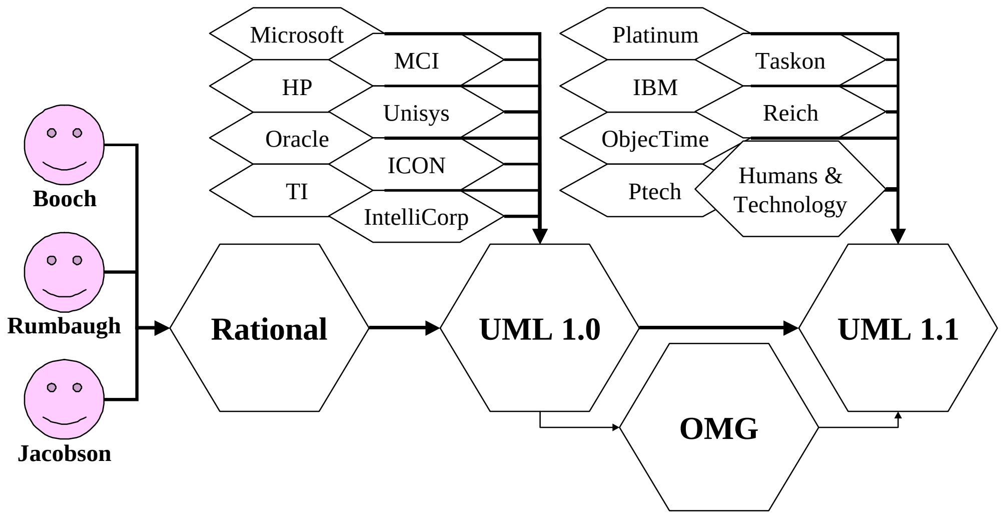
Furthermore, ODP challenges the traditional double negative of testing: if a component fails to fail a test, then it must satisfy requirements.

Unified Modelling Language (UML)

The Unified Modelling Language (UML) represents a common set of concepts and notations for object modelling.

De Facto Standard

The Unified Modelling Language (UML) was developed by a consortium led by Rational, in response to a requirement from the Object Management Group (OMG). Even before its acceptance by the OMG, it was becoming a de facto standard.



Growing Tool Support

Apart from Rational's own modelling tool, Rose, there are several other modelling tools that support the UML. The CBD Forum has assembled the following list of vendors committed to UML.

Advanced Software Technologies	CASEwise	Cayenne Software	Evergreen
ICON Computing	i-LOGIX	IntelliCorp	INTERSOLV
Logic Works	One Meaning	PLATINUM	Popkin
Rational	SELECT Software Tools	Siemens-Nixdorf	Softlab
Sterling Software	Sybase	Vision Software Tools	Visio

Source: CBD Forum [April 1998]

UML defines a large set of concepts and diagrams.

Display the boundary of a system	Use Case Diagrams
Illustrate the boundary of a system	Collaboration Diagrams Sequence Diagrams
Represent the static structure of a system	Class Diagrams
Model the behaviour of a system	State Transition Diagrams
Reveal the physical implementation architecture	Component Diagrams Deployment Diagrams
Extend your functionality	Stereotypes Packages

Source: Rational Corporation

There are some limitations of UML as a business modelling language.

Assumes that scope is unproblematic	Use cases model the behaviour of 'The System' BUT what is inside and what is outside?
Assumes that behaviour can always be assigned to specific actors within a collaboration	We may not want to subdivide joint actions.
Cannot model intentionality, except through behaviour	Can define obligations, but not responsibilities

There are some limitations of UML as a component modelling language.

UML pays lip service to components, but there is little support for the more challenging aspects of component-based development.

Similarly, UML pays lip service to open distributed processing, but there is little support for the more challenging aspects of the ODP reference model.

UML lacks a process.

The folks that produced UML are working on a process, but there is considerable doubt whether this process will be adequate.

Summary

This document has outlined the technology trends and opportunities associated with component-based development.

A defined process is required to exploit these opportunities. This is the purpose of the SCIPIO method, which is described in more detail elsewhere.

References

Other SCIPIO documents

Business Change and Process Improvement.

Software Component Quality and Testing.

Other references

Albert Borgmann, Technology and the Character of Contemporary Life. Chicago University Press, 1984.

Wiebe E. Bijker, Of Bicycles, Bakelites and Bulbs: Toward a Theory of Sociotechnical Change. Cambridge MA & London UK: MIT Press, 1995.

Butler Group CBD Forum. http://www.butlerforums.com/cbd/

ISO 9126: Software Product Quality.

ISO 10746: Reference Model for Open Distributed Processing.

Unified Modelling Language (UML). http://www.rational.com/