Understanding Business Requirements in terms of Components

Author: Richard Veryard
Version: October 14th1998

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

Describes the techniques used by SCIPIO for analysing business requirements.

Should be read in conjunction with one or more documented case studies for SCIPIO.

Acknowledgements

This material was developed for presentation at the CBD Seminar in Birmingham, October 1998. Thanks to David Sprott and Lawrence Wilkes of the Butler CBD Forum for their encouragement.

Introduction

Component-Based Development: New solution to old problems?

Technology writers often present CBD as the latest silver bullet in a long line of silver bullets. The supposed benefits of CBD - application flexibility, development productivity - have been promised many times before.

Furthermore, the business problems addressed by CBD don't seem very different from the business problems addressed by previous methods, such as Information Engineering. Indeed, some tools and methods companies continue to use the same case study material (notably various forms of Video Store) in their training courses; this reinforces the impression that CBD offers nothing substantially new to the business world, and is merely an internal IT matter.

But CBD is more than a marginal improvement to previous software techniques. It has specific relevance to the demands placed on IT by business, and enables us to address a whole range of new business problems.

Parallel development of business and IT

There are some important structural parallels between recent IT trends and recent business trends.

During the 1990s, the IT world has been slowly moving towards open distributed processing. Meanwhile, the business world is moving towards open distributed business - the so-called virtual organization. Many writers acknowledge this, and there is considerable discussion of specific aspects of this trend, such as Internet and e-commerce. Modern business organizations are best viewed horizontally, in terms of the internal and external relationships and their associated transaction costs, rather than as hierarchical command structures. However, the general lessons for software development are not always drawn out.

Changing requirements

Software writers almost always mention the increasing rate of change in the business and technological environment, but this is not analysed further. The business environment in particular is merely regarded as a random generator of new software requirements, and there are no attempts to identify any patterns or trends of business change. The response is to improve software architectures and the software process, supposedly to make software more responsive to random requirements change.

But although specific business changes cannot always be foreseen, there are some important patterns. If we can understand the kinds of things that are going on in the business world, we may be able to anticipate, or even promote, business change.

Business Change

Walmart transformed its business process by rearranging the process elements.

A well-known example of process change is provided by the US retail chain Walmart, which transformed its business relationships as follows.

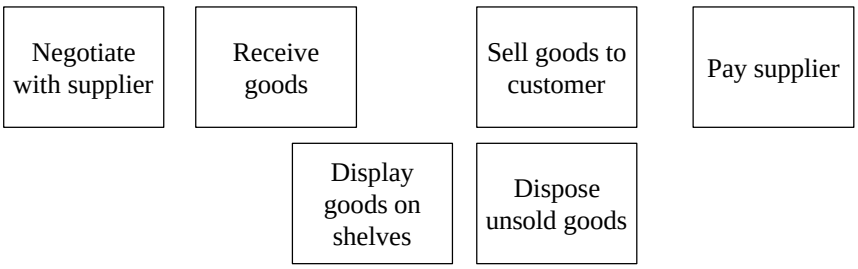
Goods were received from suppliers and displayed on the shelves in the retail store. But instead of the goods belonging to the retail organization, ownership remained with the suppliers. Walmart did not pay the supplier until the goods had been purchased by a customer.

Q What are the benefits of this change to Walmart?

Q Are there any benefits to suppliers? Why would suppliers agree to bear the costs and risks of retail inventory?

Before	**After**
❖ We pay for goods when we receive them from suppliers.	❖ We pay for goods when a customer buys them.
❖ If noone buys them, it's our problem.	❖ If noone buys them, it's the supplier's problem.

Note that this transformed process involves the same process steps but in a different sequence.



Q Show the difference between the Before and the After by drawing two process flow diagrams.

It is reported that the IT costs of this transformation were relatively small, as the applications were already well-designed and well-integrated, and could therefore be substantially reused to support the new business requirements.

There are three modes of business process change.

As we have seen, Walmart's process change can be regarded as a transformation.

Typically, simplification comes before integration ("don't pave the cow-paths") and integration before transformation. Early writings on business process reengineering concentrated on simplification.

<table>
<tr><th>Simplification</th><th>Integration</th><th>Transformation</th></tr>
<tr><td>❖ Removing redundant tasks
❖ Removing unwanted variety / complexity</td><td>❖ Single entry to multiple services
❖ Reducing interaction distances between process steps
❖ Connecting separate processes</td><td>❖ Altered business relationships
❖ Reconfigured business</td></tr>
<tr><td colspan="2">reduced cycle time</td><td></td></tr>
<tr><td></td><td colspan="2">improved customer satisfaction
& business excellence</td></tr>
</table>

In this document, we shall consider an example of process integration.

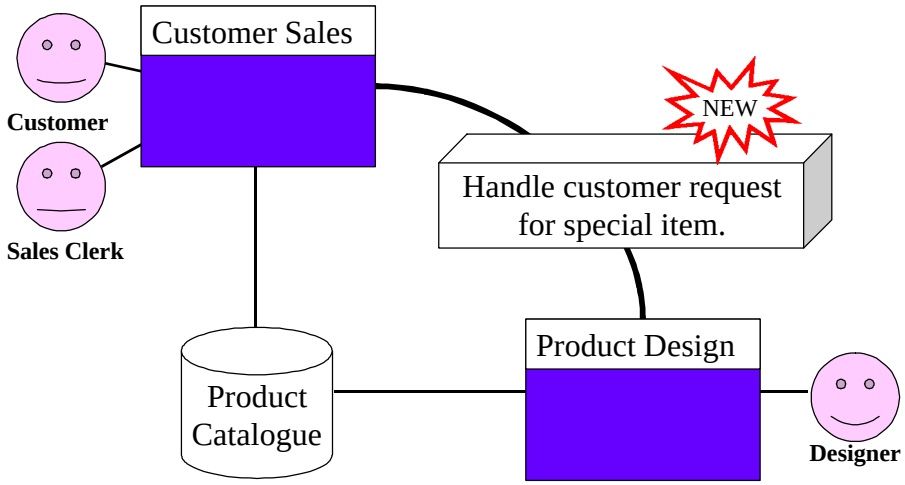
Components

We can use components to improve links between existing business processes.

A mail order company has two separate business processes. PRODUCT DESIGN creates a product catalogue, which is distributed to customers. CUSTOMER SALES receives phone calls from customers and creates sales orders.

Sometimes customers request product variations that are not included in the catalogue. In the current business system, these requests are politely rejected by the sales clerk, and there is no mechanism for informing the product designers. This represents lost business opportunity.

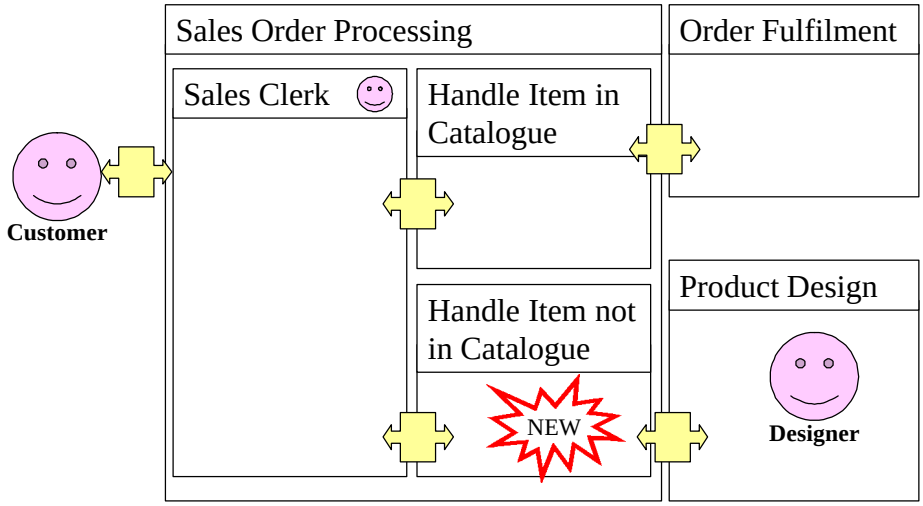
If we could plug-in a link between the customer sales process and the product design process, we might be able to handle (at least some of) these opportunities, and generate more business.



This link can be regarded as a component of the business[1]. We call it a component, although it is not itself a software component, because we intend it to have some of the properties associated with software components, and to have clearly defined behaviour and interfaces. And we are going to use some software components, together with some well-defined pieces of human activity, to implement this component.

[1] We'd like to use the term **business component** to denote a component of the business. However, many software writers use the term to denote a business-oriented software component. To avoid confusion, we refer to a well-defined chunk of business activity with component-like characteristics as a **component of the business**.

The behaviour of the whole system changes with the introduction of a new component.



We want to define the requirements for the new software components; and we also want to define changes to job descriptions. Thus we need to specify how the overall behaviour will change, and what additional behaviours are required.

There is then a high-level design task to decide which of the new behaviours will be carried out by software components, and which of them will be done by human roles: the sales clerk and the designer.

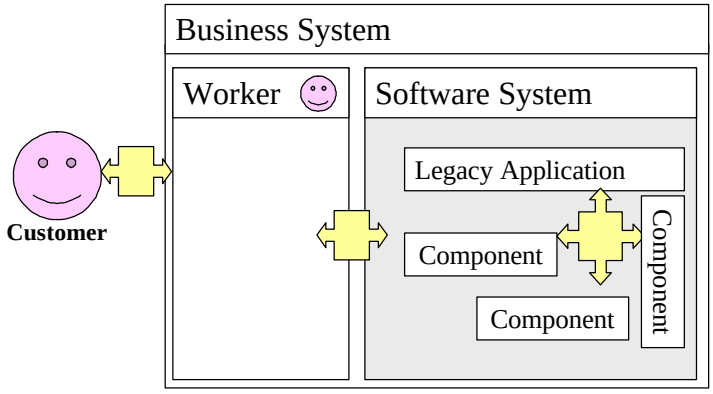
Using object modelling techniques, we can use class diagrams to specify the required behaviour of each role and each software component, as well as interaction diagrams to explore the interactions between them. Interaction diagrams are essential if we wish to analyse the emergent properties of the whole system.

Systems and Use Cases

Systems are composed of cooperating subsystems. This may include human roles as well as software.

Some object-oriented methods start requirements analysis with use cases, while others argue that it is better to do some other kind of modelling (such as business process modelling) first. SCIPIO is in the latter camp.

We view the overall business system as a collaboration between chunks of business activity; some of these may be performed by software components and some by human roles. Our business models concentrate on these collaborations.



The diagram shows several nested levels of system.

- The customer interacts with the business system.
- Within the business system, the worker interacts with the software system.
- Within the software system, the legacy application interacts with three new software components.

At any level, the external behaviour of the system or component can be defined as a set of use cases, which indicate the ways the system or component can interact with things in its immediate environment. A use case defines a binary interaction between a system and a human role (the "user" of the system).[2] By extension, the use case notation can also be used to define a binary interaction between two software artefacts: client and server.

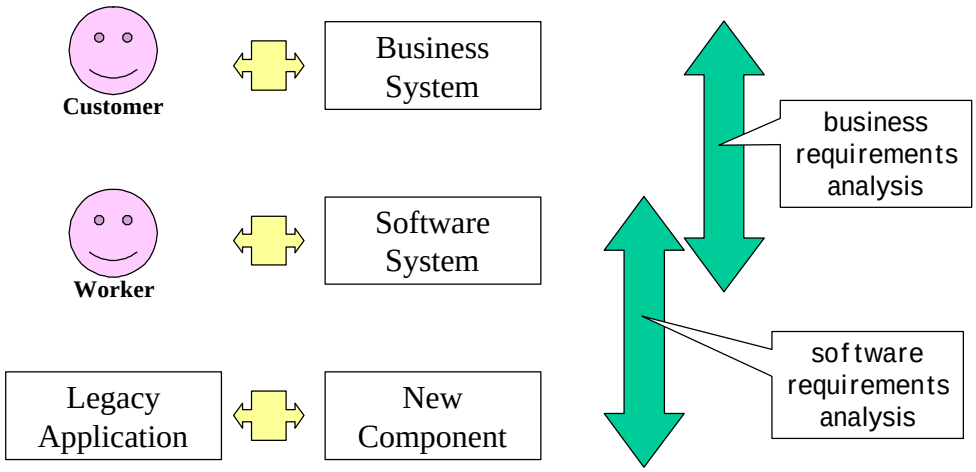
[2] Note that our model contains both binary interactions and n-ary interactions. In SCIPIO, we model these as **exchanges**. In other methods, such as Catalysis, these are modelled as **joint actions**. UML lacks an equivalent concept.

It is important to maintain a clear distinction between the behaviour of the business system and the behaviour of the software system. This may mean producing two distinct sets of use cases, and documenting the relationships between them.

Use Cases describe external behaviour of systems. We need to analyse nested levels of behaviour.

Business requirements analysis looks at the relationship between the required behaviour of the business system and the software system.

Software requirements analysis looks at the relationship between the required behaviour of the software system and its components.



It is possible to start a software project by specifying the use cases for the software system. In some circumstances this can reduce project duration and cost. But if the business context in which the system is to operate is left implicit and unanalysed, this has four possible consequences.

- Difficulty agreeing system requirements and priorities with users - **extended project timescale**.
- Difficulty for users to understand how to use the software effectively - **reduced system usability**.
- Missed opportunities for business process improvement - **reduced business benefits**.
- Software is less robust / flexible in the face of future business change - **reduced system maintainability**.

However, even if business requirements analysis is not done upfront, it is always possible - and often useful - to do it later on.

Projects address different levels of system.

This raises the practical question: who is going to do this work?

Most IT work, whether inhouse or outsourced, is structured as projects. Projects may be scoped according to different notions of system.

- On some projects, the behaviour of the business system as a whole is open to being reengineered. This may involve simplification, integration or transformation.

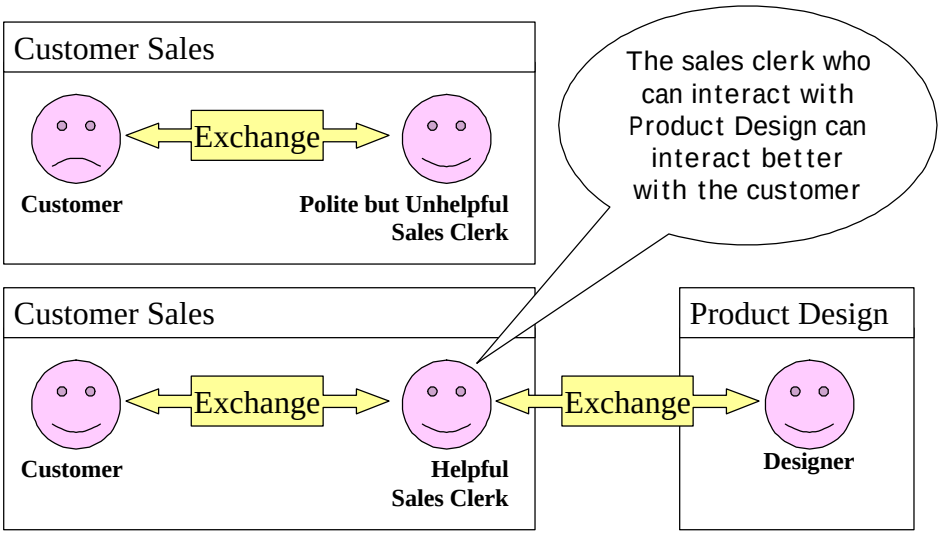
- On some projects, the behaviour of the business system is fixed or pre-defined, and the analysis and design concentrates on the required behaviour of the software system.

- On some projects, the focus is on building new components.

IT-based projects often find it difficult to gain access to the business change issues. This is a particular problem for external IT suppliers, such as software houses and systems integrators, whose project terms of reference are constrained by contractual arrangements. But in many organizations, it is difficult even for internal staff to establish management ownership or sponsorship of the business change issues.

Interactions

We focus our analysis on improving interactions.



Interactions at all levels can be improved. This includes interactions internal to a system, as well as interactions between the system and its environment. SCIPIO places primary emphasis on improving business relationships, according to a defined business strategy.

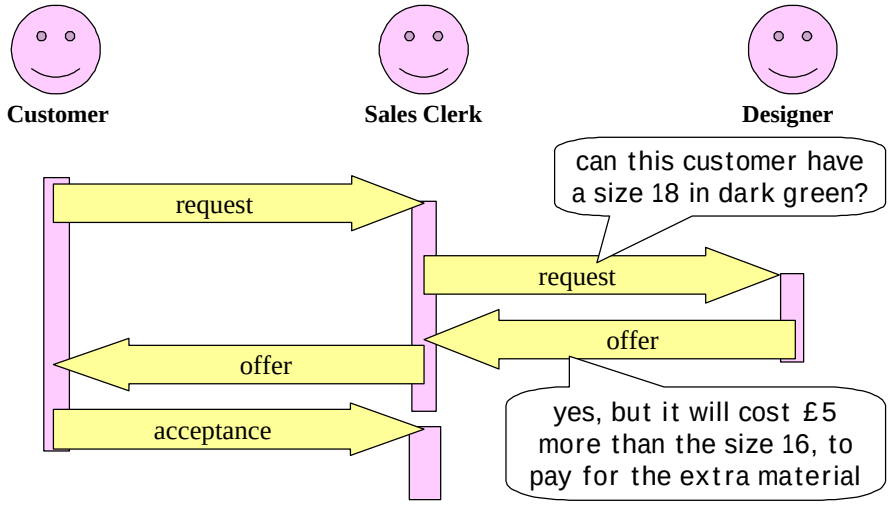
Usually we want to improve interactions by reducing interaction distances - making interactions easier, quicker, cheaper and more reliable. However, in some cases we want to maintain or increase interaction distance - making certain classes of interaction more difficult, for reasons of security or autonomy. This involves a device known variously as a **Chinese Wall** (at business system level) or **Firewall** (at software level).

This raises the question: improvement for whom? There may well be some stakeholders whose intentions are frustrated by the Chinese Wall, and whose transaction costs are increased. It is always important to identify potentially hostile stakeholders, such as fraudsters and any others whose intentions conflict with your business rules or objectives. Furthermore, it may be a business decision to improve some relationships at the expense of others. These business decisions should be explicit rather than inadvertent.

An exchange is designed as a system of messages.

We view a business system as a set of exchanges between agents (human roles or software artefacts).

Each exchange can be regarded as an aggregation of messages, over a range of scenarios.



We may show a specific sequence of messages using a simplified form of UML notation as above. It is often useful to annotate the diagram with specific instances of messages, as shown, to make them more meaningful to business users.

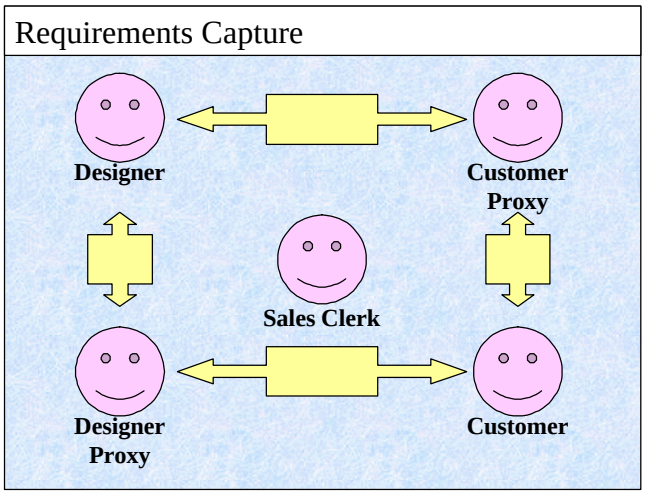
Business concepts can be tested at low cost and risk.

In this example, the business change is a business experiment. At first, we have no way of knowing how much extra revenue we might get. So we don't want to make a large investment commitment.

Component-based thinking helps here. We can install a "paper and string" mechanism to test the business concept. Perhaps this might involve an email message to the design department, generated automatically from the sales application. This mechanism may be unreliable, slow, and unable to handle large volumes, but it may be good enough to test the business feasibility of the concept. We can then replace this mechanism with a properly engineered component, once the business case has been verified. The improved mechanism can be plugged in without affecting the existing components, provided that the interfaces remain the same. In particular, the sales clerk does not have to be retrained.

Generalized interactions can make excellent reusable components.

Many of the relationships in this example can be regarded as forms of **proxy**.



- The software acts as proxy for the designer.
- The software acts as proxy for the sales clerk.
- The sales clerk acts as proxy for the designer.
- The sales clerk acts as proxy for the customer.

We might imagine an ideal world in which the customer always talks directly with the product designer. This would mimimize the interaction distance between them.

However, in the real world, customers may only get personal attention from designers on an exceptional basis. (If the request is very interesting, or the customer is very rich.) The rest of the time, the customer talks to the sales clerk, who represents (stands proxy for) the designer. Conversely, when communicating with the designer, the sales clerk represents (stands proxy for) the customer. We want these proxy relationships to be as effective and efficient as possible.

Furthermore, the sales clerk probably doesn't even have instant access to a product designer. Instead, the sales clerk communicates with a software component that represents the designer, which we may call the DESIGNER PROXY. This software knows (or learns) the questions that the designer will want to ask in a given situation, collects the relevant information, and channels it to a designer (perhaps using workflow management software or message-oriented middleware).

Using proxies improves the flexibility of the solution at all levels. The designer may be located anywhere in the world, in any timezone, and may be employed by a separate company, and this is transparent both to the sales clerk and to the customer.

Proxy behaviour may be implemented as a generalized component. We may also use **frameworks** to implement such abstract relationships[3].

[3] For more information on frameworks, see Catalysis.

Component thinking helps business change.

It is commonly argued that component thinking brings value to software development.

In this paper, I have tried to show how component thinking extends into the business requirements domain. We can regard clusters of business activity as components, regardless of the extent to which they may be performed by people or software.

There are several advantages of component thinking at the business requirements level. We can identify three in particular.

Public interfaces make business more flexible and scaleable[4].

Each organization unit performs defined services.

Contracts and service level agreements are based on clear division of responsibilities.

If we think of organization units as "components", we can use component-style modelling to define the services they provide. Each organization or unit has a clearly defined interface, and its performance can be measured in terms of a service level agreement monitored across the interface.

Stepwise implementation supports evolutionary change.

Changes can be made in one organization unit at a time, without affecting other units.

Business concepts can be tested quickly and cheaply

- *"paper and string" mechanisms*
- *low volumes*

Valid business concepts can be supported by robust software components.

Changes to a business can be made in modular fashion. Instead of disrupting the whole organization/process at the same time, we can make local changes quickly and cheaply, while holding the interfaces between units, provided that the overall architecture of the organization supports this. This makes it possible to implement genuine business prototypes, which test whether a new business concept actually works in a live business environment, before engaging in large-scale software development.

Seamless development aligns business change with system change.

Common approach for business analysis and systems analysis..

If we want to use component-based software development, there is a natural and seamless progression from business requirements analysis to software requirements analysis.

---

[4] This sometimes known by the rather ugly neologism: **rightsizing**.

References

SCIPIO. For more details on SCIPIO, please see the SCIPIO website at http://www.scipio.org/

Among other things, the SCIPIO website offers a more detailed version of the mail order case study, as well as a full development process framework.

Catalysis. For more details, please see the Trireme website at http://www.trireme.com/