CASE STUDY Liam O'Croder Mail Order Co

Author: Richard Veryard
Version: August 9th 1999

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

Illustrates SCIPIO as a method for specifying the requirements for software component development, in a context of business process improvement.

- The case study is intended to exercise the essential principles of SCIPIO.
- The case study is intended to illustrate several important techniques of SCIPIO.
- The case study illustrates how the main diagrams are used.

Questions and exercises

This case study has been developed for training purposes. The reader is invited to engage actively with the case study. 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 case study?

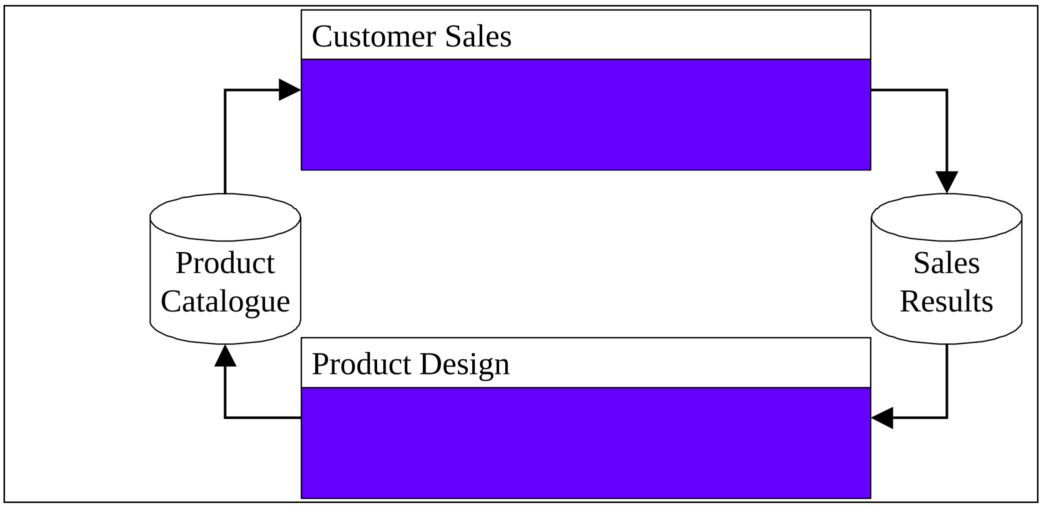
Acknowledgements

This example is adapted from a recent essay on IT trends. Deborah K. Lewis & Alexander Morrow, 'The Prairie School: The future of workgroup computing', in Derek Leebaert (ed) The Future of Software (MIT Press, 1995)

Thanks to Angela Hakim, David Iggulden, Ian Macdonald and Michael Mills, for their critical comments.

Business Process Introduction

Let's start with a business process loop that appears to be complete.



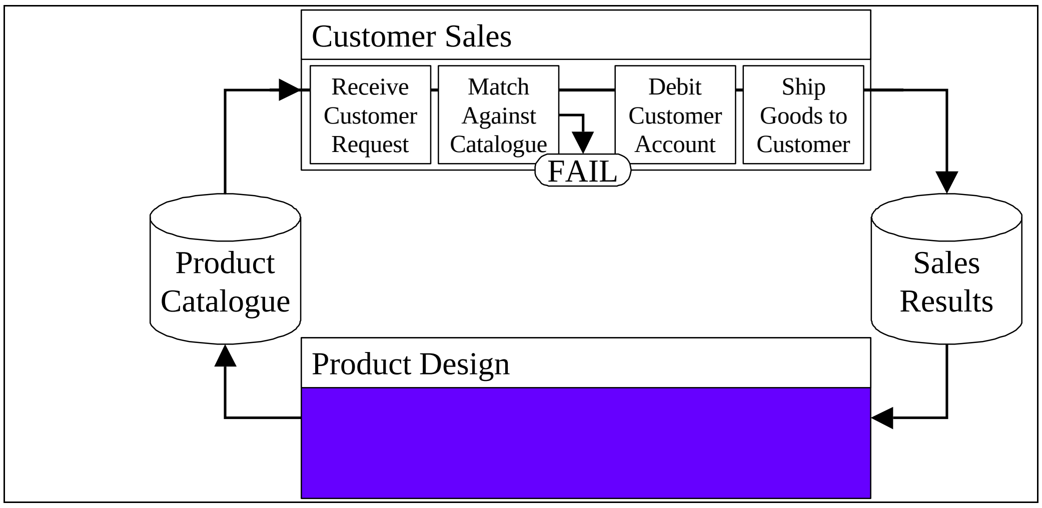
Imagine a mail-order company, whose customers place orders by phone.

The diagram shows two main subprocesses:

- Customer Sales takes and fulfils orders from customers against a pre-existing catalogue.
- Product Design periodically creates new versions of the catalogue, and is influenced by historical sales data (among other things).

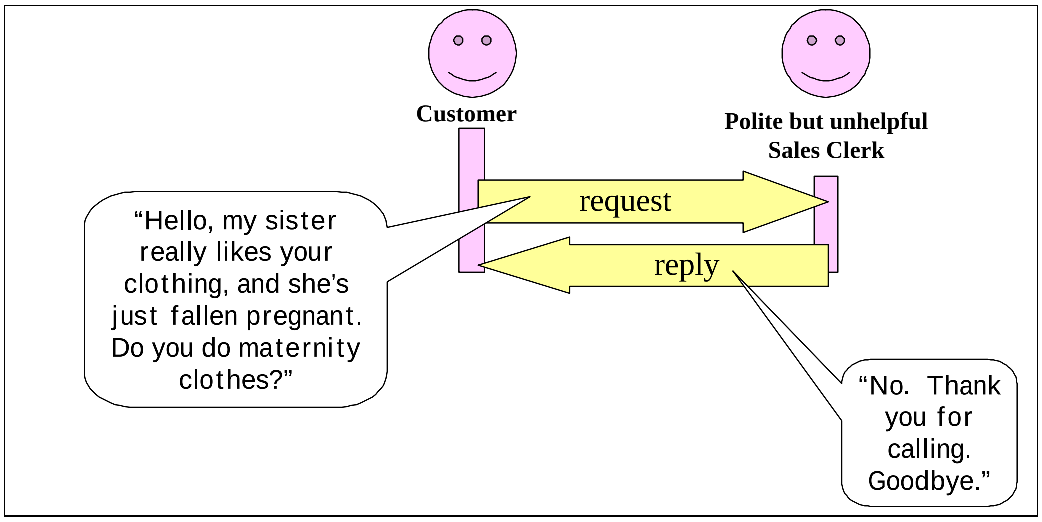
Let's ignore other processes, such as production and delivery, for the time being.

But when we zoom into Customer Sales, we find that there's a hole in the process.



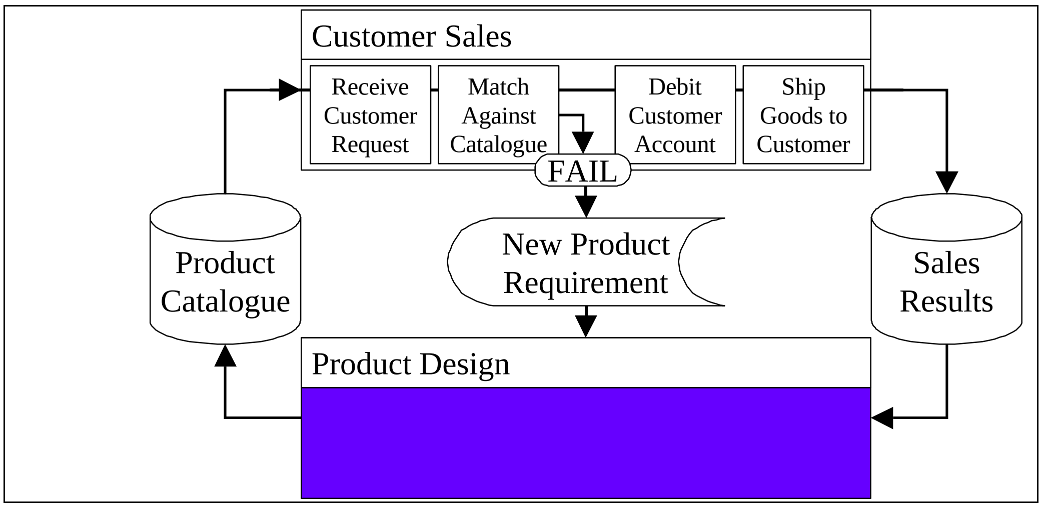
One problem with this process is that the sales results are asymmetrical. Successful sales are recorded, but unsuccessful ones are not. This means that the information fed back to the product design process is one-sided. The product designers discover the actual demand for the things that ARE in the catalogue, but do not discover the demand for anything that is NOT in the catalogue. They also don't find out about customers who go away when they are told the price or delivery date.

If the requested item is not in the catalogue, the customer goes away.


Customer
Polite but unhelpful Sales Clerk
request
reply
"Hello, my sister really likes your clothing, and she's just fallen pregnant. Do you do maternity clothes?"
"No. Thank you for calling. Goodbye."

And most importantly, if the requested item is not in the catalogue, that's the end of the conversation with the customer.

We can plug this hole by creating a new link between the two main subprocesses.



In this example, we are going to plug in a new business component, creating a new link between Customer Sales and Product Design. This increases the level of **process integration**. (This is one of the three main types of process improvement we have identified, the other two being **process simplification** and **radical process transformation**.)

To implement this business component, we shall have to install some additional software components, as well as altering the working practices of both the sales clerks and the product designers.

A traditional solution might be to install a component within the Customer Sales area to collect New Product Requirements, which the Product Designers can consider when issuing the next version of the Product Catalogue. (This is the kind of solution a traditional data-oriented methodology might have produced.

When I moved house recently, I wanted some telephone lines installed in the new house. I had some special requirements, which the telecommunications operator couldn't satisfy. It took me some time to explain to the sales clerk exactly what I wanted. When he finally understood my request, he admitted that it was a reasonable one, and promised to pass it onto the marketing department. Did this mean I could have what I wanted after all? Not this time, but maybe it will be possible by the next time I move house.

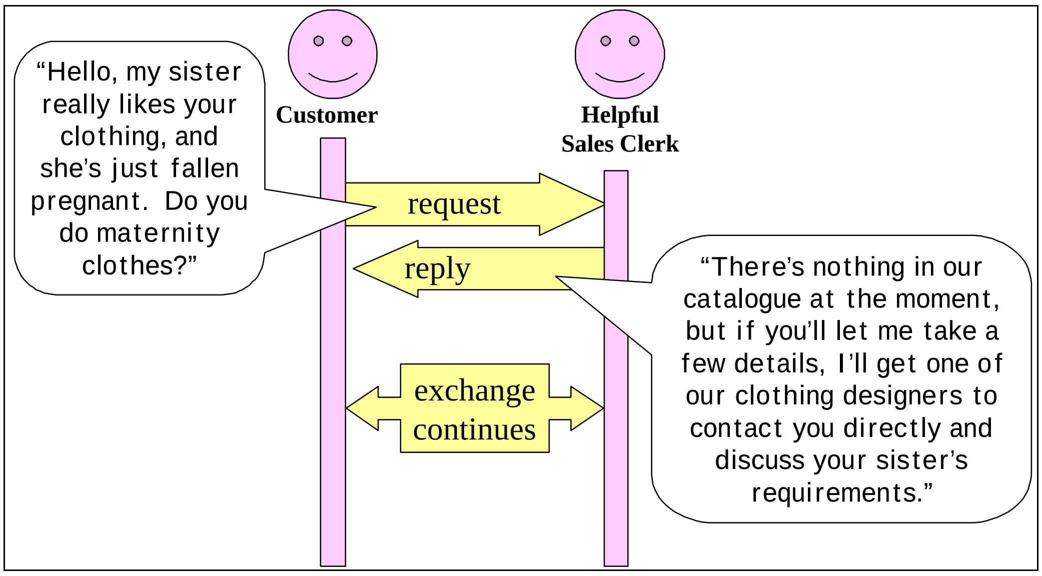
In other words, this solution might yield future (hypothetical) benefits for future (hypothetical) customers, but there is no incentive for the current (disappointed) customer to stay on the line and provide more detailed information about his requirement. In short, the solution doesn't enhance the relationship with the actual customer.

In this case study, we are going to look at a more ambitious solution, which provides a much faster response to the customer's requirements.

With the cooperation of the design department, perhaps we could sell things that are not in the catalogue.

If the sales clerk is not limited to selling things that are in the current version of the catalogue, then we may have a way to continue the conversation with the customer.

This reflects one of the key principles of SCIPIO, which is that we want to focus on improving business relationships - in this case with the customer.


"Hello, my sister really likes your clothing, and she's just fallen pregnant. Do you do maternity clothes?"
Customer
Helpful Sales Clerk
request
reply
"There's nothing in our catalogue at the moment, but if you'll let me take a few details, I'll get one of our clothing designers to contact you directly and discuss your sister's requirements."
exchange continues

This example may seem futuristic and implausible to many readers. But there is a current trend in many industries to find ways of adapting products and services to the needs of individual customers, without losing the economies of scale. This is known as **mass customization**[1].

Q Have you experienced anything similar? Can you see how this kind of added responsiveness could be relevant to your organization?

Q What are the practical limits to personalized products and services?

There are other holes in this process.

The reader might wish to consider these two processes in a wider context, and draw a larger diagram to show additional interfacing processes.

Q What other processes would you expect to interface with these two processes? What processes might trigger relevant events?

Q What are the possible sources of relevant information? What other inputs might be needed to perform these processes effectively? What processes might generate these inputs?

[1] For more information, see the Managing Change website at http://www.managingchange.com/

The purpose of this exercise is to give the reader an opportunity to explore this diagram and its use, by extending its scope. However, the case study itself will remain within the original scope, and we shall not develop these extensions further.

However, it should be noted that there are many possible inputs and triggers to Product Design. Rather than analysing each one separately, it might be worth developing a standard interface into Product Design, so that any new source of marketing information could be plugged in without fuss.

Q What would you need to know to design this interface? Sketch a design, making some reasonable assumptions.

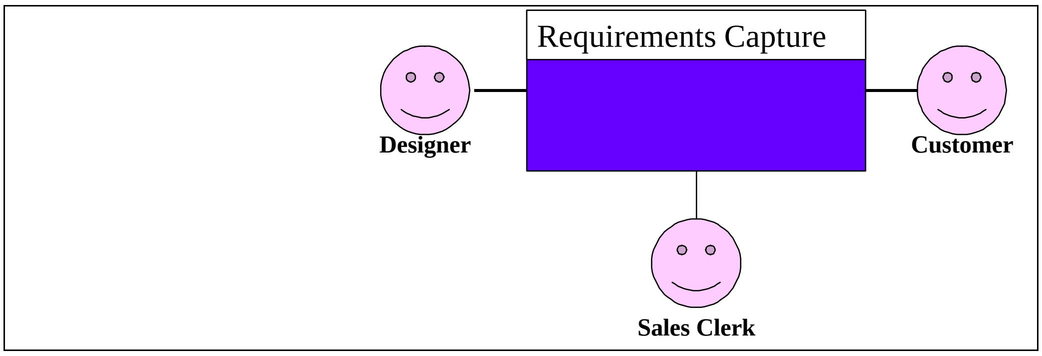
Modelling Business Relationships

We shall need to explore this example from several perspectives. For the purposes of this case study, we shall start by looking at the business relationships, both external and internal. We view a business system as a **collaboration** between several responsible agents.

In SCIPIO, we use these models to understand how **ownership** and **responsibility** are shared and distributed between many agents to perform one or more business processes.

In this case study, we are going to make changes to the business system at this level. In other projects, there may be no current desire or mandate to change the business system, but the computer system should still be designed with an understanding of the business system context. Among other benefits, this reduces the risk that future business changes may require extensive and expensive changes to the computer systems.

The customer, the sales clerk and the designer collaborate to capture requirements.



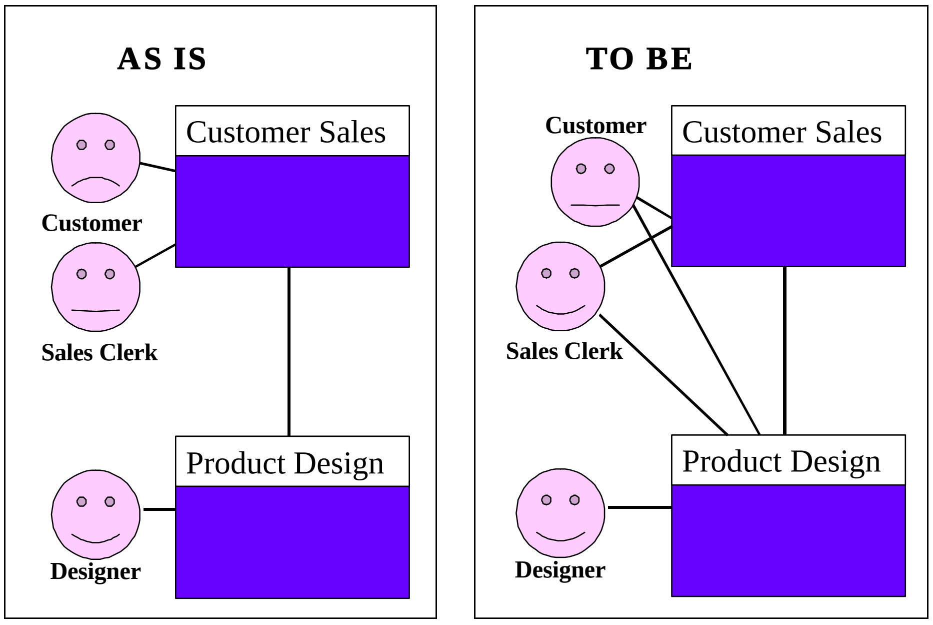
To plug the hole, we think in terms of a collaboration (or **joint action**) between the designer, the customer and the sales clerk.

There has always been such a collaboration, but the participation of the three collaborators has been rather remote. What we seek to do is to make this collaboration **closer**.

Thus this model is an **invariant** model - it shows something that has always been valid and will remain valid. We need to explore the differences between the present (remote) way this collaboration manifests itself and the desired (closer) manifestation. This is shown next.

The sales clerk and the frustrated customer become participants in Product Design.

To understand the nature of the change in the business relationships, we draw two models: an As-Is model of the present situation, and a To-Be model of a possible future situation.



We consider the complications that arise when the two processes belong to different organizations.

Modelling the business relationships also allows us to look at the issues of ownership and obligation. For example, we might consider the complications that arise when the customer sales process and the product design process 'belong' to separate organizations.

Who would own the design that emerged from this collaboration? What kind of contractual agreement would be required to formalize the collaboration between the two organizations? How would the collaboration be managed, perhaps jointly by two separate management systems?

Q Who owns the design that emerges from this collaboration?

Q What kind of contractual agreement is required to formalize the collaboration between the two organizations?

Q How is the collaboration to be managed?

Assuming these contractual and management issues are resolved, we then need to look at how this improved collaboration would be implemented. To do this, we look at a different view of the collaboration, showing the transactions or exchanges that are required to make the collaboration work.

Modelling Transactions and Exchanges

Now we are going to add some detail to the business relationship models, to show the transactions and exchanges that form each relationship.

We need to consider the sharing and distribution of **authority** and **control** across one or more business processes. Who is talking to whom about what? We also consider the distribution and exchange of knowledge and information relevant to the business process.

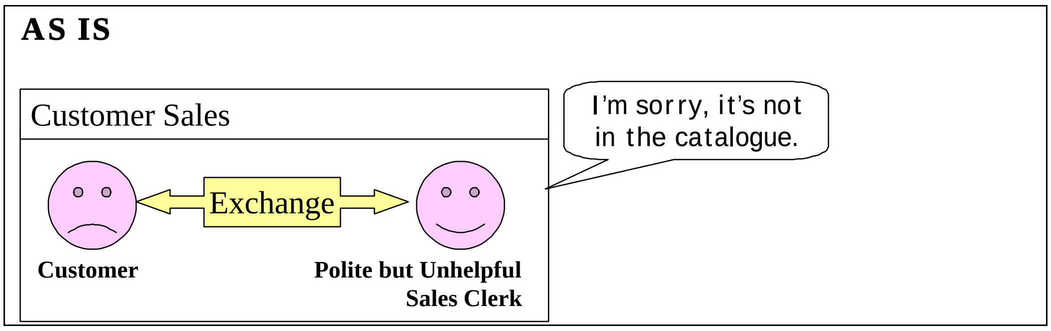
We focus our analysis on improving interactions.

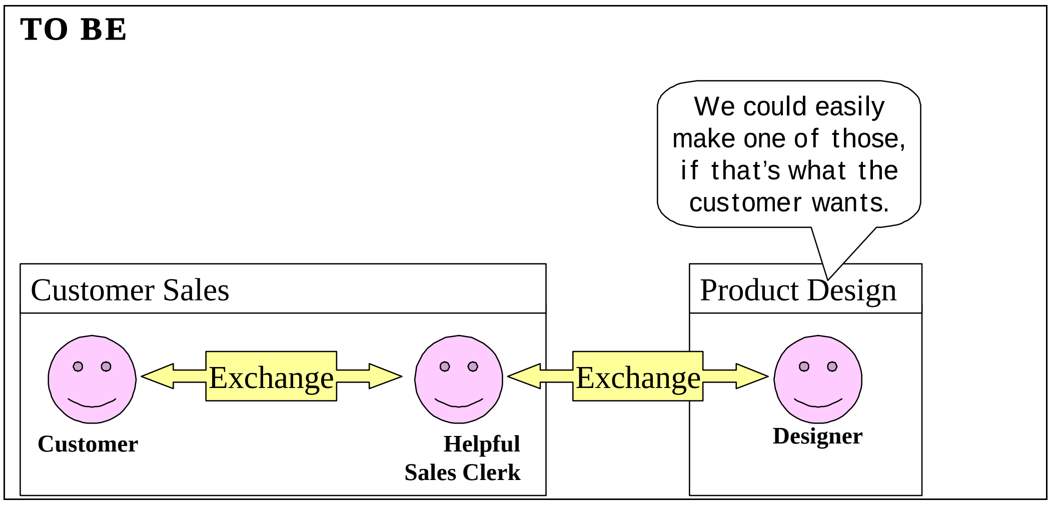
Interactions at all levels can be improved. This includes interactions internal to a system, as well as interactions between the system and its environment.

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).

The sales clerk who can interact with Product Design can interact better with the customer.

The **transaction** view concentrates on the need for an additional **exchange** - between the sales clerk and the designer. It shows how providing the sales clerk with access to this exchange effects a transition in the sales clerk herself (from unhelpful to helpful), at least as perceived by the customer.


AS IS
Customer Sales
Exchange
Customer
Polite but Unhelpful Sales Clerk
I'm sorry, it's not in the catalogue.

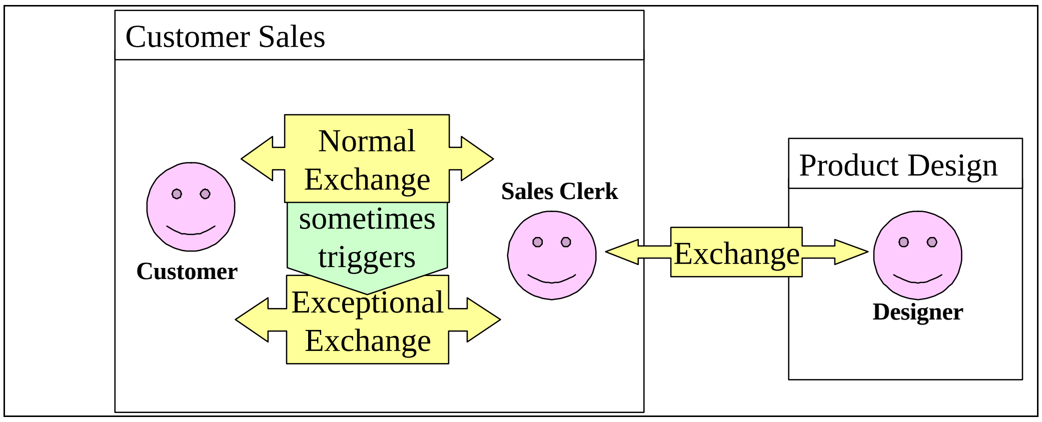


This shows a single exchange between the customer and the sales clerk, which has become more complex. An alternative way of showing the 'TO BE' situation is shown next.

After looking at this alternative, we'll then look at what's needed to implement the exchange between the sales clerk and the designer.

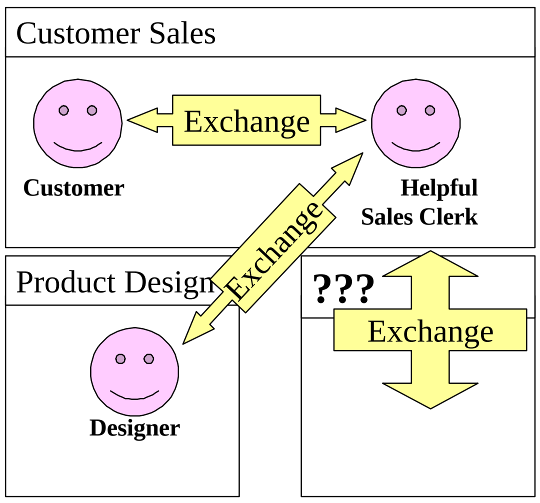
The Sales Clerk's interaction with the Product Designer may be regarded as exceptional.

An alternative way of showing the 'TO BE' situation would be to have the primary exchange (i.e. order-taking) invoking a supplementary exchange (i.e. requirements-taking) triggered by failure of the primary exchange. We leave it as an exercise for the reader to convince herself that this is logically equivalent.



Now we'll come on to look at what's needed to implement the exchange between the sales clerk and the designer.

We can consider other ways to enable the sales clerk to be more helpful.



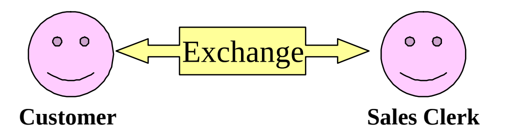
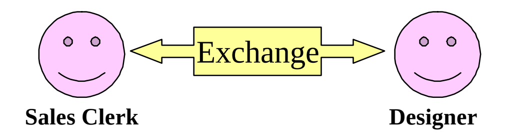
Q What else (other people, other objects) could we give the sales clerk access to?

Q What additional software components would create this access?

Let's note that providing the sales clerk with access to an exchange with the designer isn't the only way of effecting an improvement. Another possibility is to increase the sales clerk's power to analyse the customer's requirement in more detail and suggest product substitutions.

Each exchange is defined in terms of its purpose and outcome.

We need to **describe** the exchanges that currently exist (As-Is) and **specify** the exchanges that are required (To-Be). We define the logic of each exchange in terms of its preconditions and postconditions, as well as any invariant conditions. (The invariant specifies what conditions the exchange must preserve, or the things it must not do.)

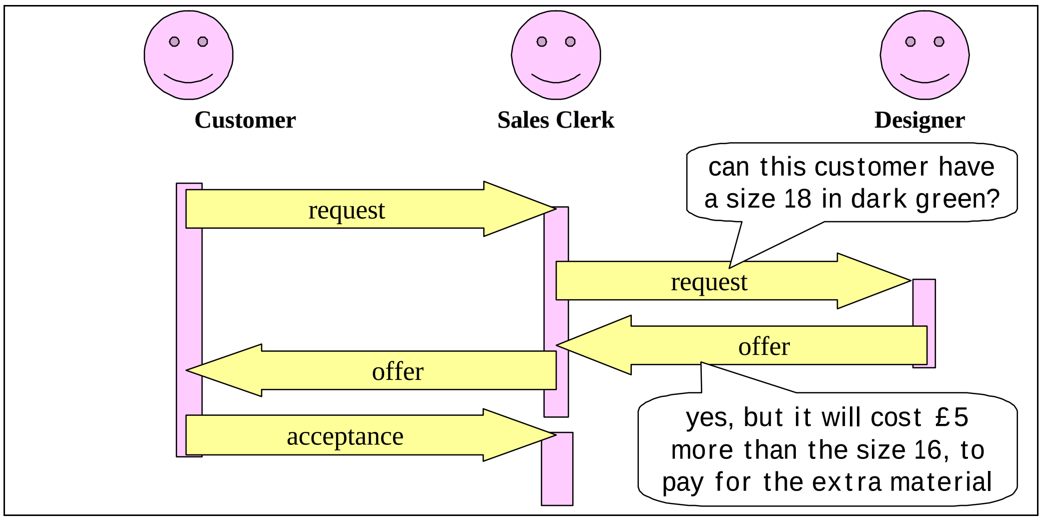
			
Name	Create sales order to match customer requirement.	*Name*	Determine response to customer requirement.
Logic	POST: sales order created OR customer goes away	*Logic*	PRE: customer requirement not in catalogue. POST: special offer made OR special offer refused.

Q Is there anything we need to specify as invariant conditions for these exchanges?

In the As-Is situation, there will often be a significant gap between intention and reality. However, this case study does not explore this difficulty.

Each exchange is designed as a system of messages.

For a specific scenario, an exchange can be decomposed into a specific sequence of messages.


Customer
Sales Clerk
Designer
can this customer have a size 18 in dark green?
request
request
offer
offer
acceptance
yes, but it will cost £5 more than the size 16, to pay for the extra material

Q Is anything missing from the scenario shown?

Q What information does the customer need to make a decision? Where does this information come from?

Additional messages are required to support other scenarios.

There are usually many different message diagrams for each exchange diagram, with different sequences of messages, each one representing a different scenario.

The above message diagram reflects the scenario where the designer says "Yes but" and the Customer says "Yes please".

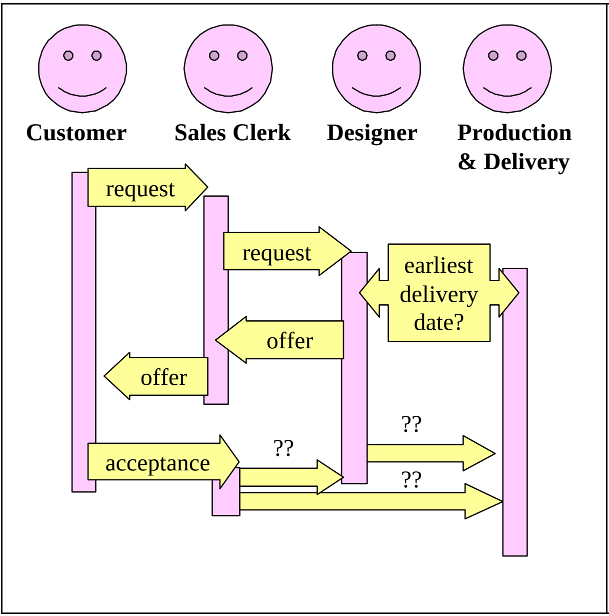
Q What other scenarios are there to consider?

For exchanges representing complex negotiations, there may be an indeterminate number of messages going backwards and forwards between the negotiating parties, before an agreement is reached. For such exchanges, the message diagram will need to be as complex as the negotiations themselves. The exchange diagram, however, shows such negotiations as a single exchange.

The interaction continues.

Now we explore the options for assigning responsibility.

The production and delivery operation must get some specification of what the customer has ordered.



Option 1: the sales clerk receives the specification from the designer and sends it to the production & delivery operation when the customer accepts the offer.

Option 2: the designer makes the specification available to the production & delivery operation. The sales clerk merely receives and passes a reference to this specification.

Option 3: when the customer accepts the offer, the sales clerk informs the designer, who passes the specification onto the production & delivery operation.

Which of these options do you think is more robust with respect to a variety of scenarios? What object **identifies** the transaction in each case?

Where does the responsibility lie for identifying the earliest delivery date? (We shall return to this question later.)

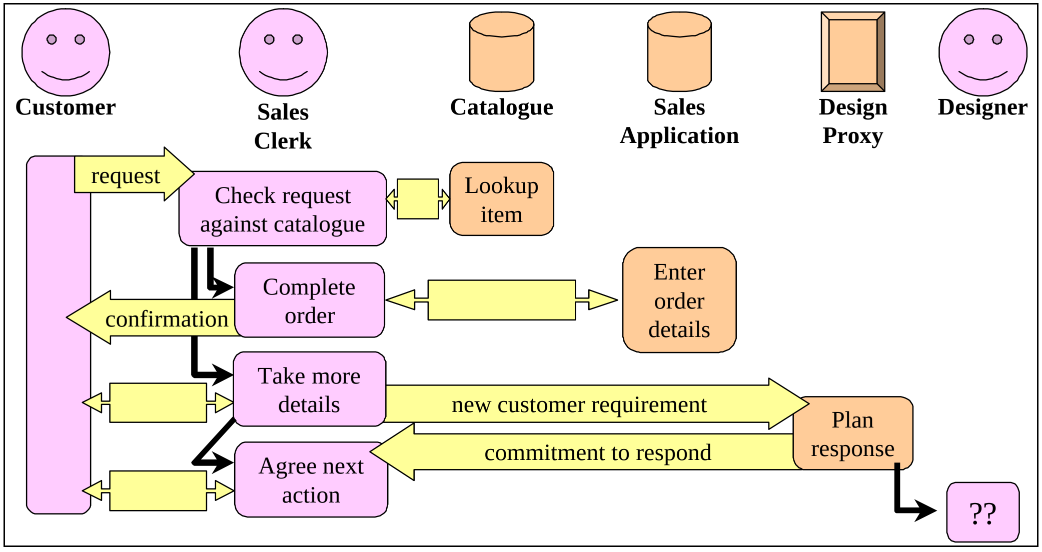
Q How does interaction continue?

Q Does designer get feedback from customer?

Q Who specifies delivery?

Q Who specifies delivery date?

The workflow includes human roles and software components.



The new operations are added, and the whole thing is strung together using workflow management software or appropriate middleware.

In our solution, the Design Proxy handles the workflow.

The new workflow is negotiated between the participants.

So far, we have assumed that the product design process will be able to make use of the new information collected by the sales clerk, and will be able to provide a prompt response.

In this example, we assume that a representative of the design organization has the authority to agree this, and commit to the new exchange.

Thanks to component-based development, we can install the new connection between the sales process and the design process without necessarily concerning ourselves with the internals of the design process.

Modelling Behaviour and Services

We start to consider the agents as objects.

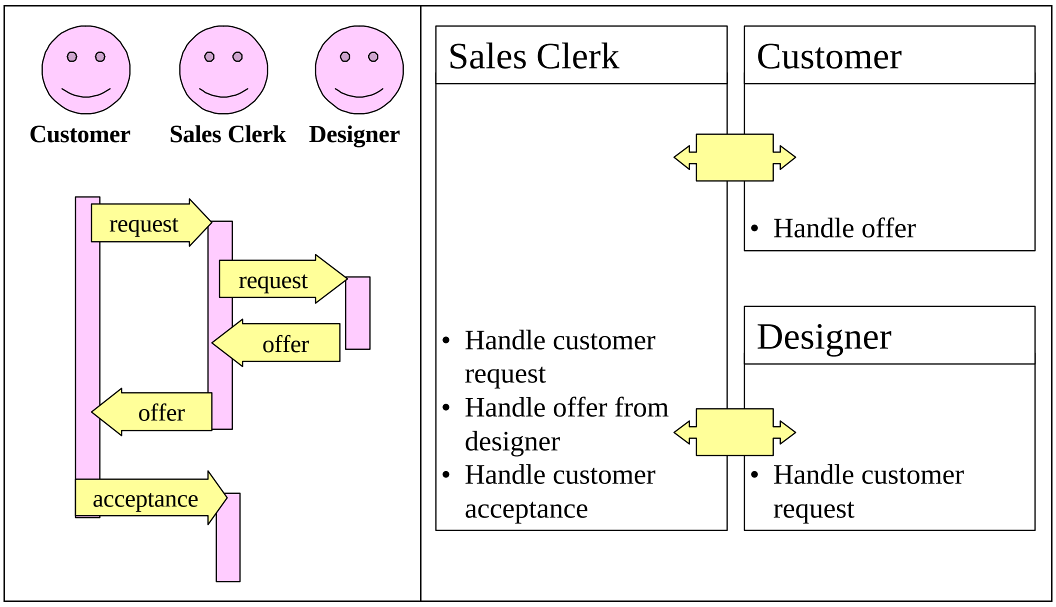
Each human role, each software artefact, each lump of activity, can be described as an **object**. We use object modelling techniques to describe the **behaviour** of an object as a set of **operations** performed by a **class**. We now view the whole system as a collection of behaviours.

To give meaning to the operations of an object, we need to understand the **vocabulary** in which the operations are expressed. Each object knows about some other objects, and may make reference to them. This vocabulary is modelled as a class diagram, for each object.

An **interface** represents a service provided by some objects to some other objects. The interface of an object consists of a set of operations, together with that part of the vocabulary that is referenced in these operations. For many purposes, we wish to specify the external behaviour of an object, without specifying its internal model.

In many object modelling methods, the behaviour of a whole system or subsystem is described as a set of **use cases**. This defines the interface between the system and a human role external to the system. (This human role is usually known as The User.)

We define the behaviour of each object in response to each message.



At this point in the case study, we describe the external behaviour of the sales clerk, as supported by one or more software systems. At this point, we do not distinguish the behaviour of the clerk from the behaviour of the software supporting the sales clerk - this distinction comes later.

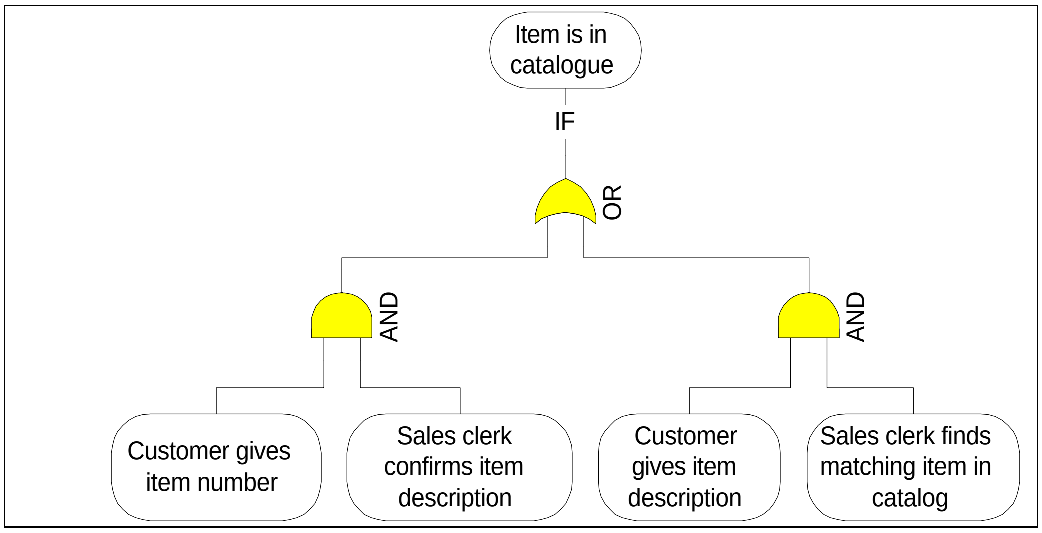
Q Define what the sales clerk does:

- when receiving a request from the customer;
- when receiving an offer from the designer;
- when receiving an acceptance from the customer.

Q Define what the designer does.

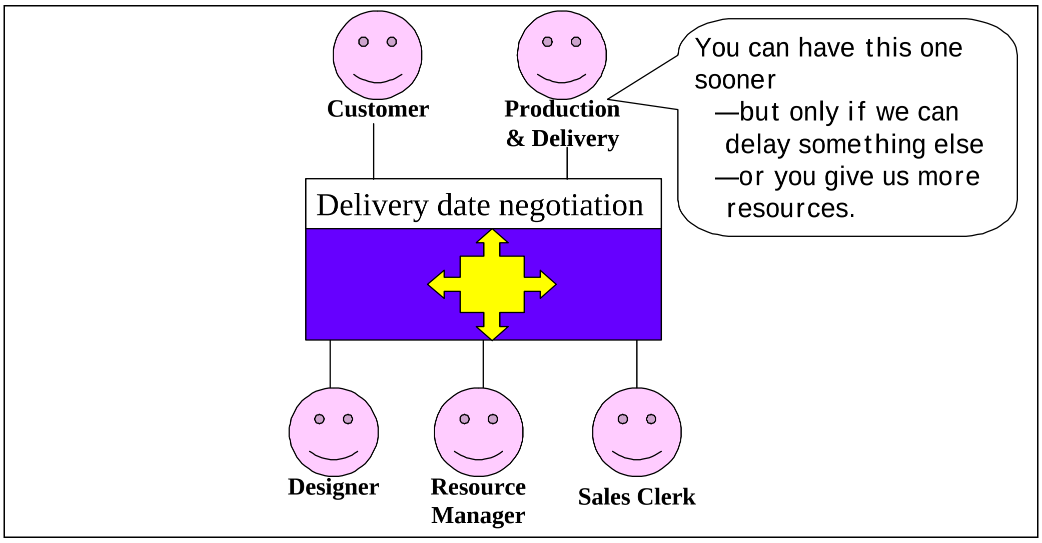
Q Define what the customer does.

We define the rules for each behaviour.


Item is in catalogue
IF
OR
AND
AND
Customer gives item number
Sales clerk confirms item description
Customer gives item description
Sales clerk finds matching item in catalog

To define each operation declaratively, we define its preconditions and postconditions. These can be expressed as formulas in predicate logic, but we prefer to represent them as hierarchical rule diagrams.

Even small details are analysed in the same way.

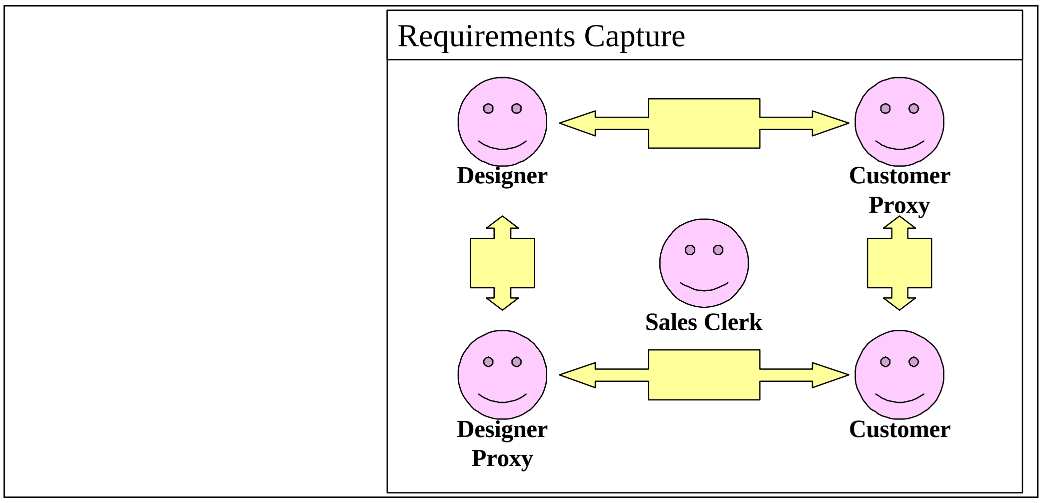

Customer
Production & Delivery
You can have this one sooner —but only if we can delay something else —or you give us more resources.
Delivery date negotiation
Designer
Resource Manager
Sales Clerk

Note that the Resource Manager will only be included in this collaboration under certain circumstances, viz. if the possible outcomes (postconditions) of the exchange may include alterations to resource allocations.

Q What are the options for designing the exchange of messages?

Q What are the options for designing software components?

Exchanges may be implemented through proxies.



One way of looking at the role of the sales clerk is that he is merely a two-way proxy. The designer talks to the sales clerk as a proxy for the customer, and the customer talks to the sales clerk as a proxy for the designer.

In some telephone sales situations, a computer may stand proxy for a human. For example, in some cities it is possible to order cinema tickets over the phone without talking to a real person at all.

Proxies may be human or mechanical. Indeed, in the Behaviour View we do not need to specify whether behaviours shall be human or mechanical - this is determined in the Design View.

An earlier version of this material provoked a comment about the way that I've named the components - particularly the Designer Proxy. Software engineers have traditionally been taught to name modules or components in terms of their functions. Thus, instead of 'Designer Proxy', they would probably have called it 'Requirements Capture'.

If you were looking at the component purely from the perspective of the sales clerk, this would seem the natural choice. But I prefer to name components in terms of the exchanges they implement. This focuses attention on the other end of the exchange: in this case, the product designer.

However, for those components where I haven't analysed the exchange, I've used traditional functional names.

A proxy has special advantages as an exchange partner

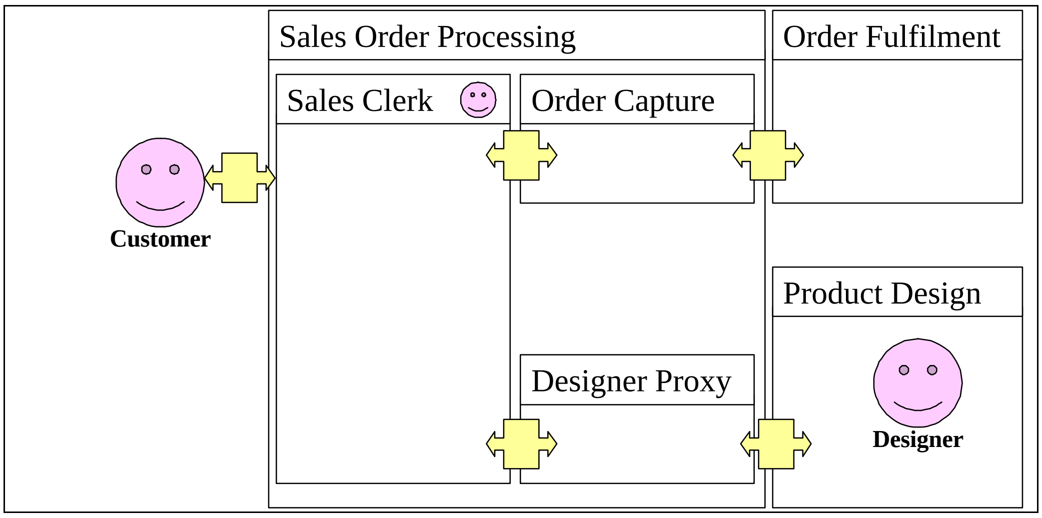
Location	Accessible from anywhere you need it.
Timing	Always available when you need it.
Ease of use	Can use your vocabulary.
Reliability	Always provides a good response.

The proxy stands for someone or something else. It should provide better availability - in terms of location, timing, ease of use, and reliability. It may also improve maintainability: since only the proxy itself needs to know the exact relationship with the people or things it is standing for, only the proxy needs to be changed if this relationship changes.

So what does this mean when we come to designing the behaviour of the proxy?

Q What behaviours does a proxy need to have in order to provide these advantages?

The sales clerk interacts with Product Design via a software component: the Designer Proxy.



This view shows how the required **behaviour** of sales order processing is divided between one human role (in this case, although in other cases there may be multiple human roles) and two software components. When the analysis is complete, each box will contain a detailed specification of the required behaviour of the given role or component.

What is achieved by having the sales clerk talk to a designer proxy (implemented in software) rather than directly to the designer?

Firstly, the designer proxy will have a much higher level of availability than the real designer. The sales clerk can nearly always access the designer proxy, whereas the real designer may be in meetings.

But the introduction of a software component between the sales clerk and the designer also gives us an opportunity to get more flexibility into the process, since we no longer need to fix the identity or location of the designer in advance. The designer proxy can stand for many designers, perhaps not yet identified. Instead of being given access to a single product designer, the sales clerk is potentially given access to an entire network/system/process.

Q If there is a large network of designers available, on what basis does a particular task get routed to a particular designer?

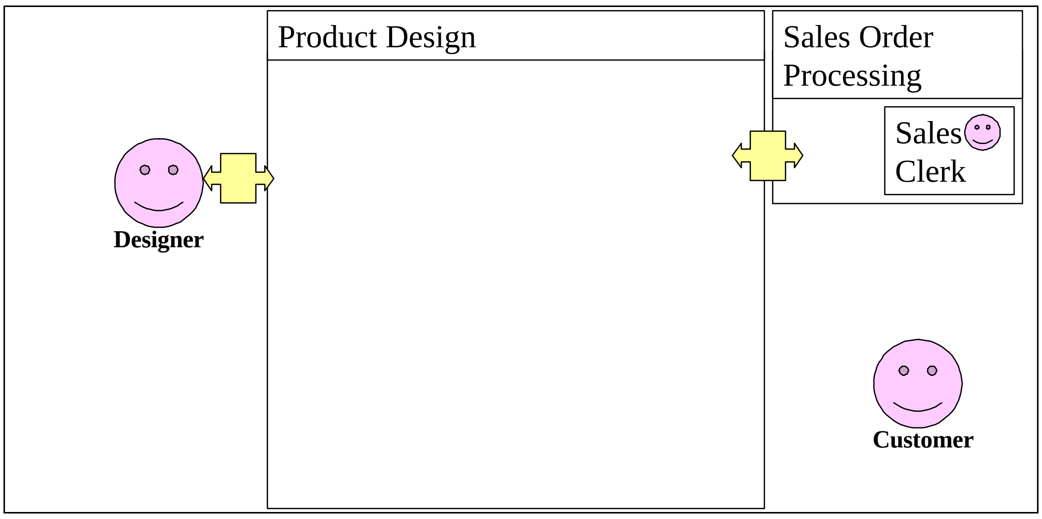
Except in a very small organization, the sales clerk will not know all the designers personally, and is very unlikely to know which of them are busy at a given time. In any case, it may not be appropriate for the sales clerk to select a designer, based on whatever knowledge or opinions the sales clerk might have of a given designer's capabilities. With proxies, our solution is both scaleable, and allows for different criteria of designer selection and work allocation to be implemented.

Perhaps there is a group of designers in Hong Kong, working a different shift pattern to the European designers. (Many distributed workflow systems take full advantage of the shifting time-zones around the world.) Perhaps the company has recently merged with a company in Germany, whose designers specialize in leather garments. (The system could route tasks according to specialist requirements of knowledge and skill.) Perhaps the company has a joint venture with a chemical company, developing new man-made fabrics.

Q How does the behaviour of the design proxy relate (if at all) to the behaviour of the designer?

The designer proxy needs to anticipate the behaviour of the designer. Ideally, the designer proxy needs to ask all the questions that the designer himself will ask. One way of achieving this is to build an intelligent proxy that will learn from experience - adding questions according to the history of previous exchanges. However, this could result in the designer proxy asking considerably more questions than a real designer would ask.

Product Design also needs proxies.



Just as the sales clerk talks to a design proxy, the designer may need to be able to talk to a sales proxy, or even a customer proxy, which would be another software component giving a similar degree of flexibility and transparency in reverse.

Q What proxies are needed by Product Design? Sketch a plausible model showing this.

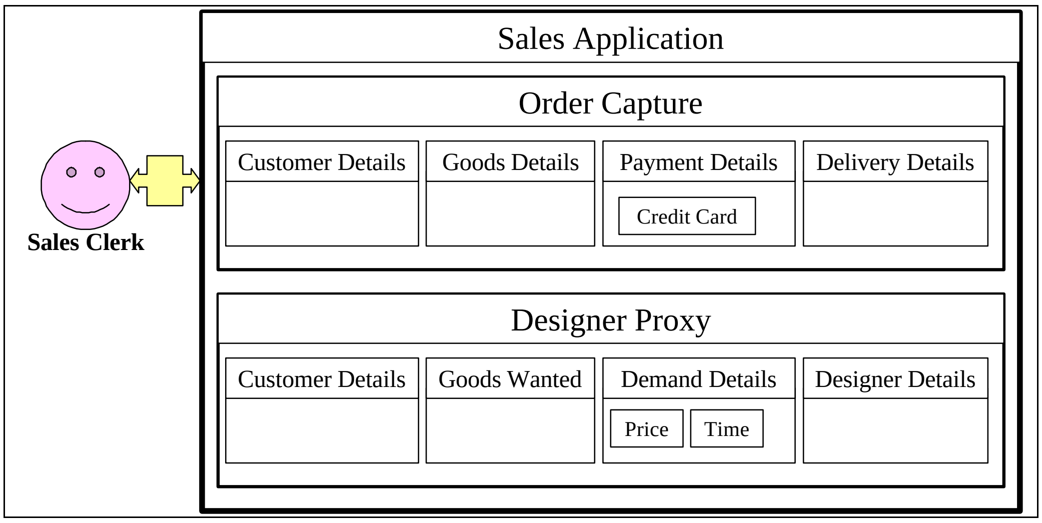
Q Does the Designer need to know whether he is talking to the Sales Clerk or directly to the Customer? Would a combined proxy provide greater flexibility?

Specifying Components and Interfaces

When we have a complete understanding of the behaviours and services that are required, we can partition them into **components**, with clearly specified **interfaces**. A component can only be used through an interface. Several behaviours and services may be bundled into a single component. In some technical environments, a component may offer its services through several interfaces.

It is at this stage that we consider the opportunities to reuse and enhance legacy systems. We use similar techniques to describe the behaviour of chunks of legacy system, and this defines what is required to convert these chunks into proper components - for example, stripping and wrapping techniques.

We design the sales application as a hierarchy of services, delivered by components.



This view shows the **design** of the sales application as a hierarchy of components delivering services to one another. This diagram shows the hierarchy in the form of nested boxes - tree diagrams provide an alternative (equivalent) representation.

The software design is decomposed to the level of individual screens/windows, and to individual database accesses.

Ideally, many of these low-level components will already exist, either in the form of existing legacy functionality that can be wrapped to provide the required interface, or in the form of generic components from an existing component library that can be plugged in to provide the required service.

We adjust the design to make best use of the components we already have.

Some credit card functionality already exists. We can subdivide the Credit Card component into the existing functionality and the new functionality. There are then three design options:

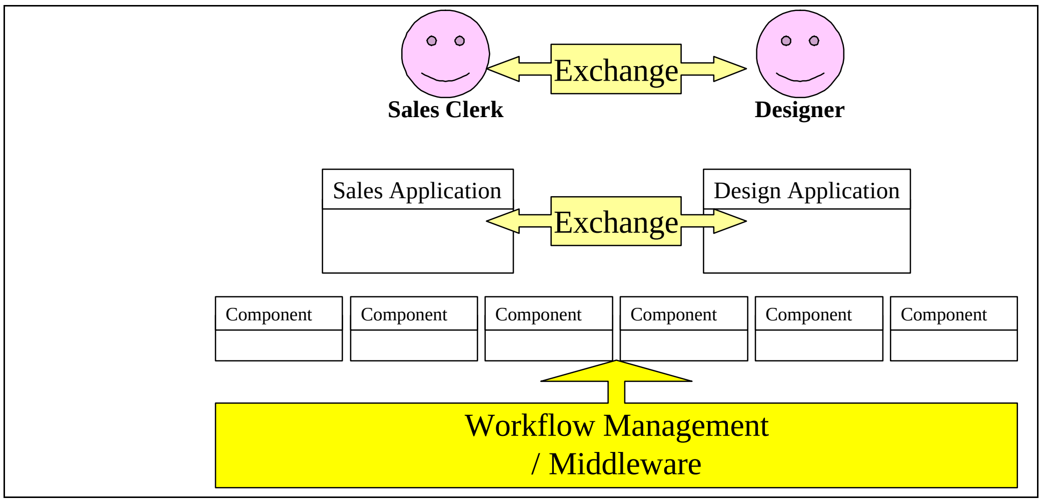
1. Preserve the interface of the existing component. Adjust the interfaces of the components that interact with it.

2. Modify the existing component to extend its interface. (This would normally need to be done in a way that preserved the existing interface for existing uses.)

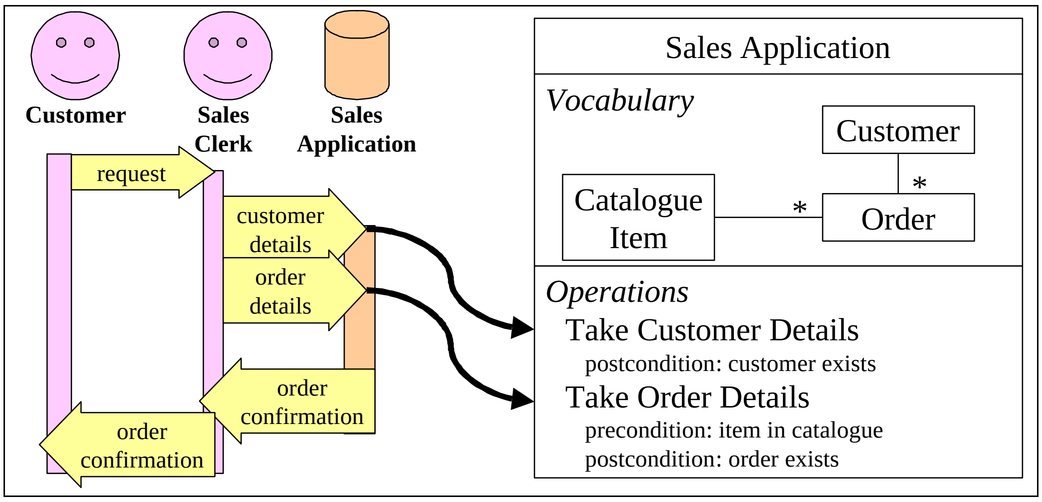
3. Add a bridging component that allows the existing component to be used without modification, and without changing the design.

The workflow is implemented as a string of components.

We can use workflow management software and/or middleware to administer the actual exchange of messages between the relevant people/departments.

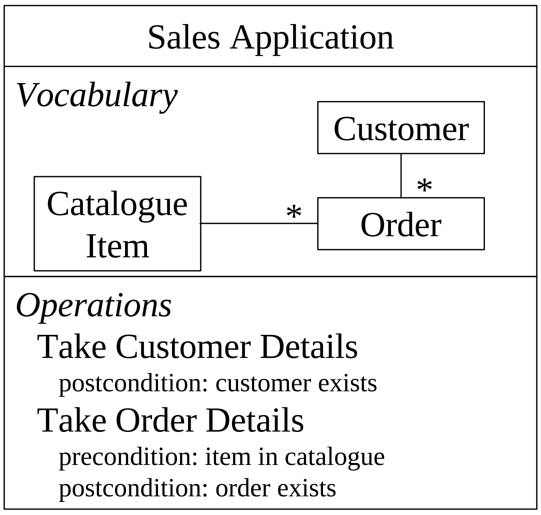


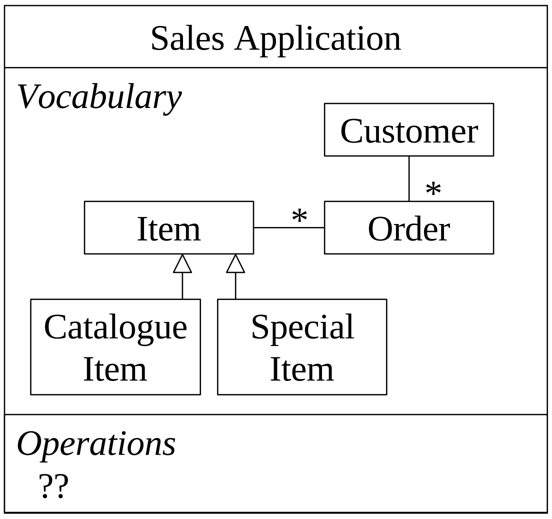
Most of the order capture functionality already exists - and we want to reuse it.



Most of the order capture functionality already exists in the legacy system. We don't want to rebuild this if we don't have to. Instead, we can wrap the legacy code so that it performs the required operations in the manner of a component.

Our new system complicates the data structure.





Q What are the operations of the new component?

There are many existing applications that reference CATALOGUE ITEM. Some of these should continue to reference CATALOGUE ITEM, while others will now need to reference the more general ITEM.

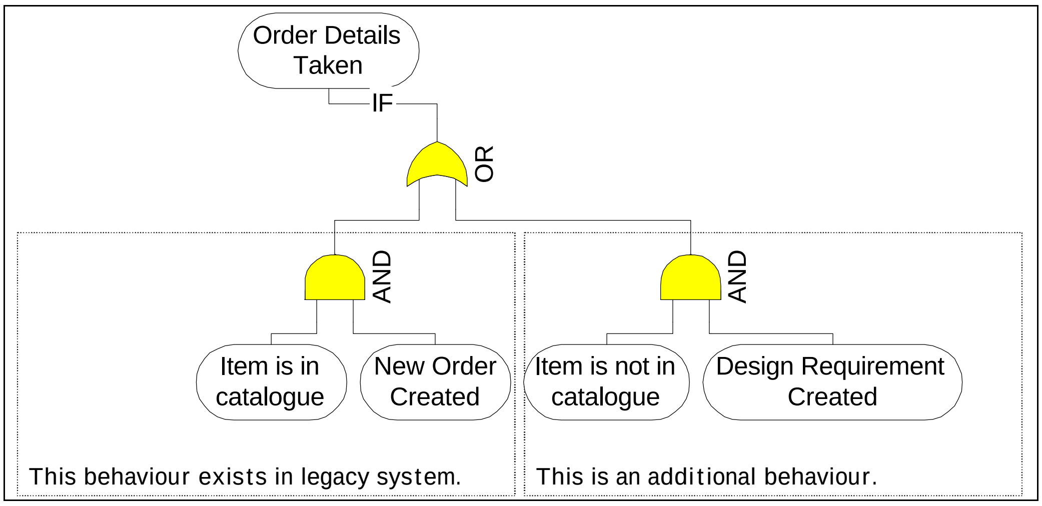
Q Which applications would you expect to require CATALOGUE ITEM only?

Q Which applications would you expect to require all occurrences of ITEM?

We may need to implement this by creating a component with multiple interfaces. We can then gradually reconnect existing applications to this new component. In the short term, some manual adjustment (e.g. to management information reports) may be required.

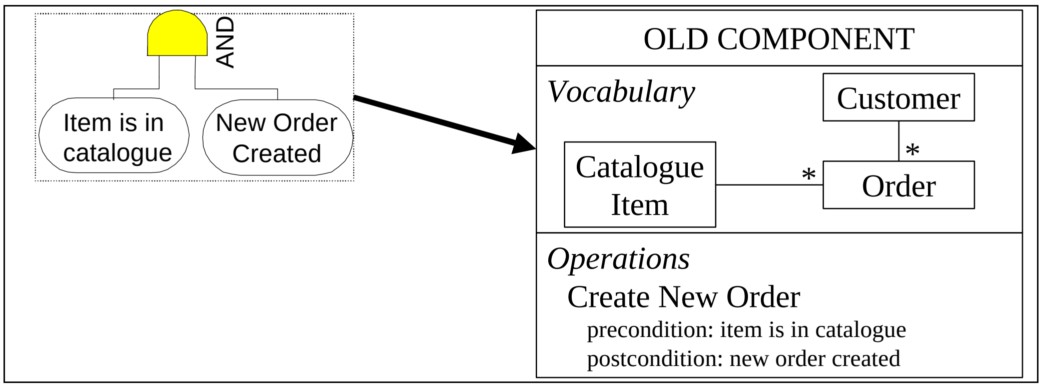
Q Which applications would you expect to be most critical for reconnecting to the correct data structure? Which applications (if any) would you expect to be least critical?

Our new system complicates the rules ("postconditions") for taking an order.

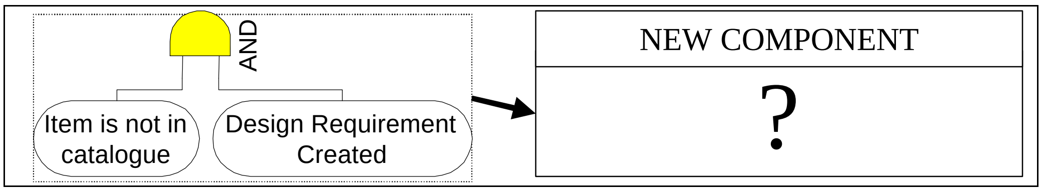

Order Details Taken
IF
OR
AND
AND
Item is in catalogue
New Order Created
Item is not in catalogue
Design Requirement Created
This behaviour exists in legacy system.
This is an additional behaviour.

The preferred option for implementing this additional behaviour is to wrap the existing behaviour as a component, and create the additional behaviour as another component.

Existing behaviours can be extracted from legacy systems.

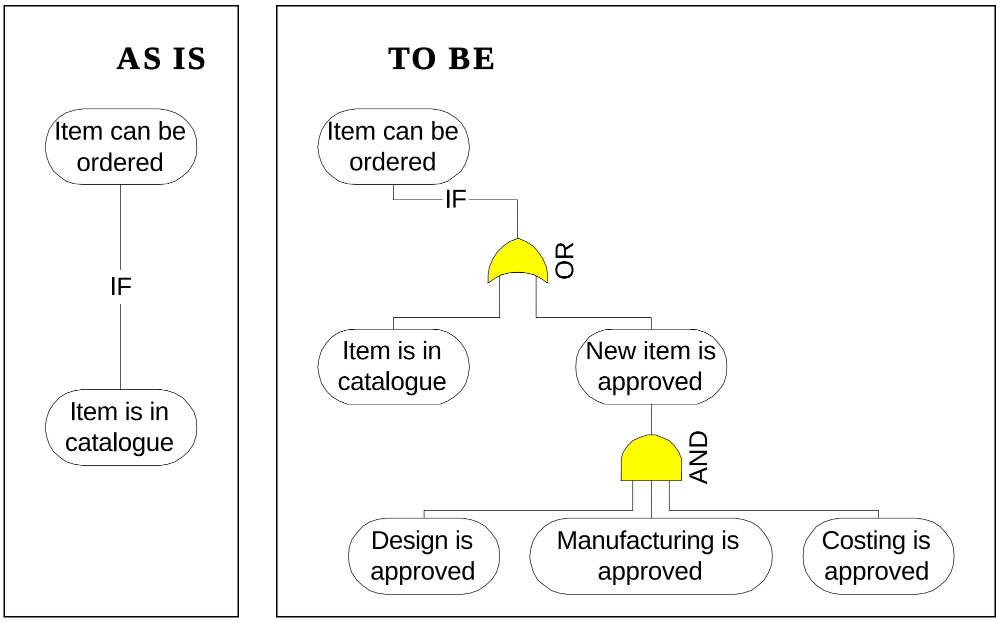


Additional behaviours can be specified as new components.



Q Specify the new behaviours that are required.

Our new system relaxes the rules ("preconditions") for selling an item.



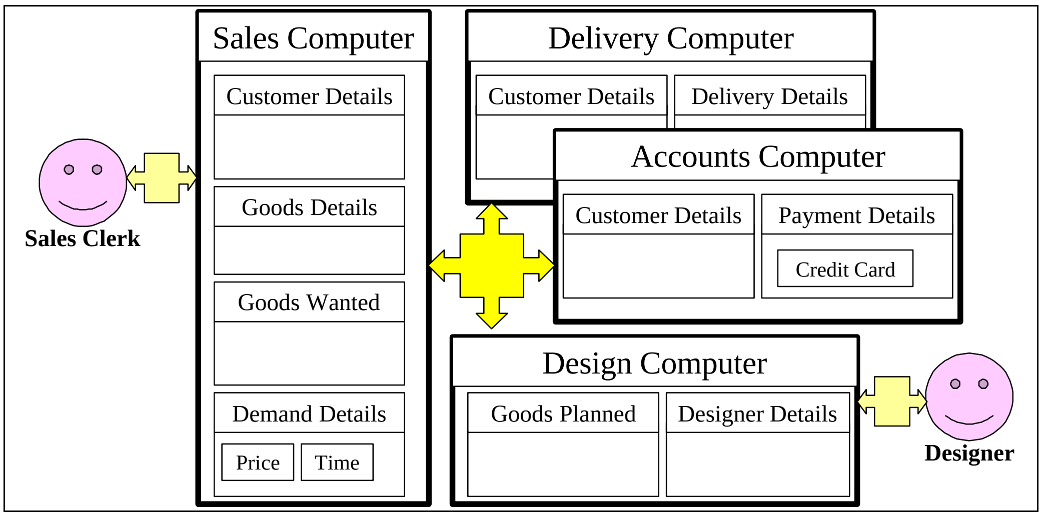
Q How do we implement this additional complexity? How do we enforce the preconditions for ordering an item?

Q How does the existence of non-catalogue items affect data structures? How does the existence of non-catalogue items affect other processes?

Q How do we minimize impact on other systems?

Specifying Physical Mechanisms

The components are distributed onto the appropriate physical platforms.



This view shows the **physical** configuration and distribution of the components onto elements of the technical infrastructure. The technical choices made here affect such quality characteristics of the final system as reliability, availability, robustness and performance - these are often known as 'non-functional' requirements. These characteristics connect back (or at least they ought to) to the distribution of responsibilities and risks identified in the enterprise view.

This physical view establishes some requirements for the technical platform(s), and allows the desired technical performance characteristics (of hardware, network and system software) to be calculated or estimated. But of course there are many other requirements on a technical architecture and on the procurement of technical platform components, and we shall not be discussing these further here.

We select the appropriate communication mechanisms.

There are many ways of passing messages from one human role to another. For example, they may be transmitted by email or phone, or embedded in a computer system (e.g. workflow management software).

For messages between computer components, we may use workflow management software or middleware for transmission.

We may also need to consider control mechanisms, to ensure that all messages arrive promptly and securely.

Stepwise Implementation

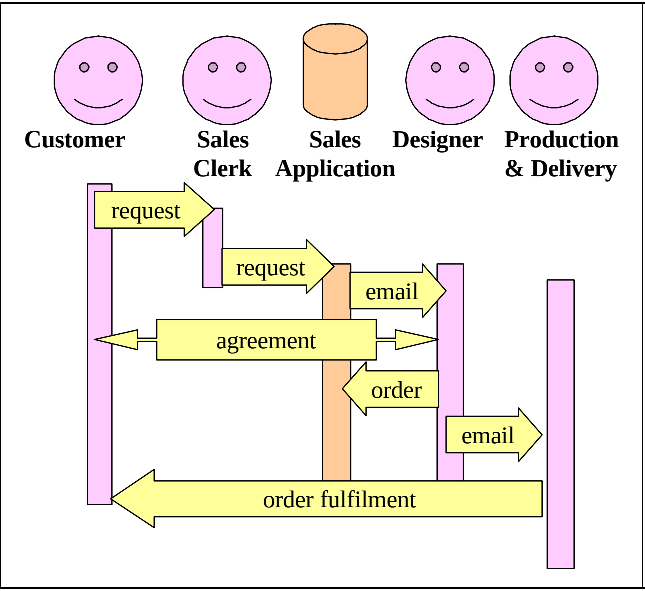
We specify each step of the implementation.

STEP 1	➢	Sales application sends email to design department.
	➢	Designer talks individually to customer.
	➢	Design department creates sales paperwork.
	➢	Production / delivery handles special order manually.
STEP 2	➢	Sales and design as step 1.
	➢	Production / delivery handles special order automatically.
STEP 3	➢	Finish automating exchange between sales and design departments.
	➢	Implement design proxy.

Component-based development allows us to implement the solution in cleanly separate stages.

The separation of these steps is primarily focused on reducing disruption to the user departments. When we're thinking about business process improvement, convenience to the IT professionals should come a long way second.

The initial solution is highly restricted.



- Designer exercises control of workflow
- Designer talks directly to customer
- Designer operates sales application
- Only appropriate for small volumes.

The first step is quick and cheap, and may be regarded as a business prototype, to establish a proof-of-concept. This will help management to decide whether to continue with the change, and will help users to understand the details of the change.

Final Points and Review

SCIPIO aims and principles

Q In the light of the case study, what is now your view of the aims and principles of SCIPIO?

Component-Based Development

Q To what extent is this story dependent on new technological possibilities, such as Component-Based Development?

In what way is CBD a necessary part of this story? After all, many of the supposed advantages of CBD, such as reuse, have been promised by earlier approaches, including modular programming and design. Is CBD really any different from these earlier approaches?

In my opinion, the key difference lies not in the way the software systems are developed - although there are certainly some differences there - but in the way the new software is installed and deployed. In our example, it is possible to define a new component connecting sales with product design, without knowing the details of the sales process or the product design process. This has several potentially valuable implications:

- The same component can be deployed across a diverse organization, without enforcing a single global standard set of operating procedures upon the design departments or sales teams.
- A design department can continue to use the same component while it is undergoing other operational changes and improvements.

Modelling Notations and Use of diagrams

The case study shows how we can analyse a business process opportunity through to the specification of software components, using the same modelling notations throughout.

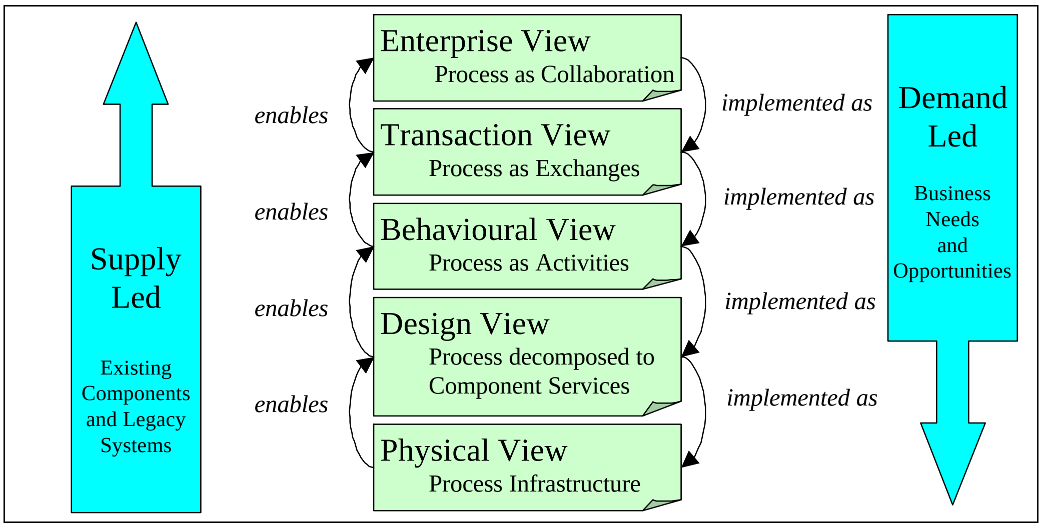
The modelling notations used in this example are an extension of UML. The main difference from UML is the concept of **exchange**, which we have found indispensable for modelling both business conversations and aggregations of message traffic between software components. We use the exchange diagram in a way that is complementary to the interaction diagrams and class diagrams of UML.

Q Does each diagram make sense on its own? Do the diagrams support each other? What difficulties do you foresee in practice?

Sequence of tasks

We have shown a progression of models through this case study. We started with an outline view of a business process, which led to models of business relationships. We then explored transactions and exchanges in more detail, before focusing on the behaviours required from the system - both from human roles and from software components. In this story, the constraints and opportunities of the legacy systems and physical infrastructure were left until towards the end.

It may be very tempting to regard the sequence of diagrams presented in this case study as a traditional waterfall. In practice, we have to work in parallel with all five views. In particular, many elements of the design may already be given from legacy systems or from available component sources, and may dominate the solution. Only in an idealistic 'greenfield' project can we pretend to follow a top-down approach.



Furthermore, although many of the requirements traditionally known as 'non-functional' are addressed in the physical view, it is usually unwise to leave these requirements to late stages of a development project.

Q Does the (implied) sequence make sense? What other sequences might have worked? What difficulties do you foresee in practice?