



# Determining the requirements for software components

Author: Richard Veryard  
Version: January 20<sup>th</sup> 1999

[richard@veryard.com](mailto:richard@veryard.com)  
<http://www.veryard.com>

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

[info@scipio.org](mailto:info@scipio.org)  
<http://www.scipio.org>



# Preface

## Purpose of document

- To describe the SCIPIO approach to determining the requirements for software components.
- To critique traditional OOAD approaches to requirements engineering, especially those based on Use Cases.

## Questions for reader

The text is interspersed with open questions, which the reader is invited to consider.

- |   |
|---|
|  How will software engineering practices need to change, if they are to accommodate Component-Based Development? |
|  How soon do you expect these changes to be widespread within the software industry?                           |

## Acknowledgements

Thanks to David Iggulden and Paul Winstone for useful discussions.

# Introduction

## Summary

The popular engineering approach to determining the requirements for software components is in three stages. This approach has been carried forward from structured methods to Object Oriented Analysis and Design (OOAD).

- First you identify a group of users who need a software solution for an identified business problem.
- Then you define the requirements on the software system. In OOAD this is usually specified as a set of use cases. These requirements may be based on a model of the business process, and are negotiated with the users.
- Then you design the software system as a set of interacting components.

Of course, if you are trying to build generic components for multiple use, there may not be a specific business process to analyse, or even a specific software system to design. Furthermore, there may not be any specific users to negotiate requirements with. Undaunted by this, software engineers typically adopt the same approach but at a different level of abstraction. A **domain** is defined, which is a generic business process or generic area of automation.

- First you identify a group of domain experts, who are supposed to stand proxy for a class of potential users.
- Then you define the requirements on the domain, in collaboration with the domain experts.
- Then you design a generic kit of interacting components, which will be usable for any system or business process that satisfies the generic domain description.
- Then you assemble systems from these components that satisfy the specific needs of particular users within the target area.

This approach implies a division of labour: some engineers specialize in the creation of small lumps of functionality (called software components); while other engineers specialize in assembling these components to produce large lumps of functionality (known as software applications or systems).

But this approach assumes that it is meaningful to think about software requirements in terms of a fixed lump of functionality, known as the software system or application, delivered to a fixed community of users. It assumes that one person or team (possibly known as the system architect) has design control over this lump.

The limitations of this approach emerge when we are faced with large open distributed dynamic networks of software. It is both a business imperative and a technological imperative for business organizations to connect their business processes into these networks. These networks lack central design authority or architectural control, and evolve organically. Overall functionality and structure may change unpredictably from one day to the next.

Connecting to these networks raises a number of difficult management dilemmas, including control, security and stability.

These networks are “Out of Control”. Traditional engineering approaches are inadequate for operating effectively in this environment. Biological and ecological metaphors seem to have more relevance than engineering metaphors. [Reference: Kevin Kelly]

The SCIPIO approach to creating software components is radically different to the traditional engineering approach, and is based on biological and ecological metaphors.

- First we identify an **ecosystem**, which may contain both human users and existing artefacts.
- Then we identify **services** that would be meaningful and viable in this ecosystem.
- Then we procure **devices** that enable the release and delivery of these services into the ecosystem.

## Context

In this section, we aim to understand the context in more detail. We define a series of what we call **ecosystems**. We use this word, rather than **markets** or **industries** or **networks**, because it is more general. An ecosystem may contain human agents or organizations, or it may contain intelligent software agents, or it may be a hybrid.

The natural world can be regarded as a single global ecosystem, because there are some connections between all the parts. (Birds may visit even a remote island, carrying new species of plant or insect.) However, for many purposes it is simpler to regard a semi-isolated part as a separate ecosystem.

Instead of the biological/ecological metaphor of **ecosystem**, some readers may prefer to think in terms of a **political economy**.

## Separation

To analyse the context for software component development, we separate the whole into separate ecosystems.

The first separation we draw is between the demand-use side and the supply side. This separation will be familiar to most readers.

Component-based development enables a further separation, between the external service (accessed through a component interface), and the internal component assets or devices that deliver the service. This second separation applies equally on the demand-use side and on the supply side, yielding four ecosystems altogether.

<p style="text-align: center;"><b>Service Use Ecosystem</b></p> <ul style="list-style-type: none"> <li>➤ using services</li> <li>➤ demanding services</li> <li>➤ architecting / configuring use of services</li> <li>➤ subscribing to service publications</li> </ul>	<p style="text-align: center;"><b>Service Supply Ecosystem</b></p> <ul style="list-style-type: none"> <li>➤ providing / delivering services through stable interfaces</li> <li>➤ architecting services</li> <li>➤ publishing available services</li> </ul>
<p style="text-align: center;"><i>Ecological principles: pleasure, critical mass</i></p>	<p style="text-align: center;"><i>Ecological principle: commodity/availability</i></p>
<p style="text-align: center;"><b>Device Use Ecosystem</b></p> <ul style="list-style-type: none"> <li>➤ configuring devices</li> <li>➤ installing / connecting / calling devices</li> <li>➤ predicting device behaviour</li> <li>➤ predicting system behaviour</li> </ul>	<p style="text-align: center;"><b>Device Supply Ecosystem</b></p> <ul style="list-style-type: none"> <li>➤ architecting devices</li> <li>➤ providing devices to deliver services (build, buy, assemble, reuse)</li> <li>➤ managing devices as assets</li> </ul>
<p style="text-align: center;"><i>Ecological principles: quality, flexibility</i></p>	<p style="text-align: center;"><i>Ecological principle: conservation of energy / economies of scale / reuse</i></p>

### Ecological principle

In each of the four ecosystems, we can identify one or more ecological principles or economic imperatives.

#### Service use ecosystem

An important element of strategic thinking around a business process is to decide: which bits are to be routine and mechanical, consuming as little management time and attention as possible; and which bits are to be strategically interesting, on which management time and attention is to be focused. Thus some business services need to be as boring as possible, while others need to be as exciting as possible. And it's important to get the balance right - too much excitement is painful or stressful, while too little excitement is death. This balance is a critical survival factor for the ecosystem as a whole; we call this the **pleasure principle**.

Furthermore, some services give value to their users by being unique, while others give value to their users by being common. The best-known example of the latter is communication services: how much value you get from your email or fax service depends on how many of your friends and associates are also using email or fax. Thus your decision to use a given service sometimes depends on your estimate of the number of other users. We call this the **critical mass principle**.

## Service supply ecosystem

In this ecosystem, services are competing for survival. Between two services, the more available service will usually win over the less available. Hence there is a strong technological and commercial pressure for services to increase their availability; we call this the **availability principle**. (It is sometimes known as the **commodity principle**.)

Some aspects of availability are as follows (depending on the nature of the service):

- Global 24-hour access. Instant response.
- Any hardware and software platform. Available in Arabic, Chinese, English, Hindi, Russian and Spanish.
- Easy to use. Low entry cost. Good support. Minimum learning curve.
- High reliability. Safe and secure. Low risk.

## Device supply ecosystem

In this ecosystem, competitive survival depends on delivering the greatest quantity of service with the smallest amount of work. This is often called **reuse**; software reuse should be focused on achieving **economies of scale** in software, based on effective asset management and knowledge management. We call this the **energy conservation principle**.

## Device use ecosystem

In this ecosystem, competitive survival depends on getting the expected services (and their associated benefits) from a given configuration of devices. This in turn relies on an ability to predict and control the behaviour of components-in-use, including the emergent properties of large distributed systems. We call this the **quality principle**.

Also relevant in this ecosystem is the ability to easily substitute devices and reconfigure systems. We call this the **flexibility principle**.

**A good component is one that satisfies the ecological principles of all four ecosystems.** Such a component is viable and meaningful, and is likely to survive and develop.

## Triggers

What then triggers the creation of a new component?

The initial stimulus may come from any of the four ecosystems.

**Demand-driven.** An agent may identify some service that he/she/it requires. This generates an unsatisfied service demand, which may be communicated to the service supply ecosystem in hopes of a response. Alternatively, the service demand may be translated into a device demand, and communicated from the device use ecosystem to the device supply ecosystem.

**Supply-driven.** A supplier may identify an opportunity to extend an existing service to increase its availability. Or to package a bundle of existing services and/or devices to provide a new service. An engineer may identify an opportunity to wrap or modify an existing device to provide new services. Prototype devices may be reengineered to increase availability of services.

Regardless of the initial stimulus, a successful component needs to find sufficient acceptability on the demand-side, and sufficient economies of scale on the supply-side. The requirements process needs to connect with both imperatives, but in no particular sequence.

## Types of supplier

In the past, the supply-side could be divided into two modes of software supply.

**Off-the-shelf.** Some suppliers designed and marketed products for sale into an identified market (or ecosystem). These were standard products, with little or no variation, and were typically expensive to modify or integrate. If many users bought the same product, then the development and marketing costs could be shared between them, reducing the individual cost to each user.

**Bespoke.** Some suppliers designed and delivered one-off products to a particular customer's specification. These were typically much more expensive per user than standard off-the-shelf products, and took longer to deliver, but were (at least in theory) much closer to the customer's requirements.

Component-based development enables a third mode of software supply.

**Mass customization.** Suppliers who are able to respond to the needs of a single customer, while achieving economies of scale across multiple customers.

There are many ecosystems containing only off-the-shelf and bespoke suppliers, in which neither mode of supply can eliminate the other. However, as soon as mass customization becomes effective in a given supply ecosystem, then off-the-shelf and bespoke supply are ecologically doomed, and will eventually be eliminated from that ecosystem. These suppliers may survive in the short term by switching to other (perhaps smaller niche) ecosystems, but for how long?



Do you think there are any software supply ecosystems in which a supplier can be safe from competition from software mass customization? How would you be sure?

## Supply challenge

**Bespoke.** A software house that specializes in bespoke software development may detect increasing difficulties competing on price. If your competitors are achieving better economies of scale, without compromising quality and flexibility, then they will be able to undercut your prices.

Most software houses still bid for bespoke work on the basis of a simple formula: estimated cost plus contingency plus profit. There may be some opportunities for you to reduce costs or contingency, by improving your software process. But if your competitors are doing this too, this won't be enough.

There are two possible strategies for survival. You can either move up the “food chain”, concentrating on supply and packaging of services (while subcontracting the software engineering side to cheap suppliers in Bangalore). Or you can embrace the ecological imperative: conservation of energy.

Instead of bidding for bespoke work on a cost-plus basis, you must try to determine what the customer is willing to pay. If this isn’t enough to cover your costs, then you need to find a way of satisfying the customer that leaves you with some residual value. If you have developed some software components that you can sell to other customers as well, this might well make up the difference. (There are other forms of residual value, but this is the most likely one for a software house to exploit.)

**Off-the-shelf.** Meanwhile, a software product vendor with a standard fixed range of products may detect increasing difficulties maintaining market share, or entering new markets. If your competitors can offer more flexible products, with greater availability and lower total cost of ownership, then they can erode your customer base.

The challenge for such suppliers is to leverage the economies of scale, to get wider flexibility and availability from an equivalent device base, and to get much greater internal levels of reuse. This is basically an architectural issue: how to improve the internal configuration and layering of the product. (Some suppliers will choose to keep the benefits of this improved architecture to themselves, while others will choose to open up the architecture to customers and third parties.)

## Perspectives on open distributed systems

### Five viewpoints

Initial work on architecture for open distributed systems was carried out by the ANSA project under the UK Alvey Programme. This identified five different formal descriptions of an open distributed system. (The ANSA architecture uses the term **projection** for these five descriptions.)

This work developed into an international standard Reference Model for Open Distributed Processing [ISO 10746] - variously known as RM-ODP or ODP-RM.

The RM-ODP allows for any coherent set of formal descriptions, which it calls **viewpoints**. It is usually assumed that there will be five viewpoints, corresponding to the five projections of the ANSA architecture.

Enterprise Viewpoint	Describes the <b>purpose</b> of the system.	This viewpoint includes the intended roles and responsibilities of human and software agents within the system.
Information Viewpoint	Describes the <b>meaning</b> of the system.	This viewpoint includes the semantics of the conversations (or messages) passed between the agents.
Computational Viewpoint	Describes the <b>causal behaviour</b> of the system.	This viewpoint includes the operational rules governing and triggering the behaviour of the system components.



Engineering Viewpoint	Describes the <b>design mechanisms</b> of the system.	This viewpoint includes the devices that implement the behaviour.
Technology Viewpoint	Describes the <b>physical infrastructure</b> of the system.	This viewpoint describes the physical assets that support the engineering devices.

## Component viability and compatibility

For a component to be viable within a given ecosystem, it needs to be viable within all five viewpoints.

- It needs to serve a useful **purpose**, relative to the **intentions** of some **community of agents**.
- It needs to be **semantically meaningful**. Its interface has to have an information model that is recognisable by other agents within the service use ecosystem.
- Its **behaviour** needs to fit with the expectations of other agents. It must conform to standard or local interfaces and protocols.
- Its internal **design** must be compatible with the quality and performance demands of the device use ecosystem.
- Its technical requirements must be compatible with the available **infrastructure** and **resources** within the device use ecosystem.

## Component economies of scale

Let us now return to the supplier challenge of **economies of scale**. We can now articulate a supply strategy for achieving economies of scale in terms of the five viewpoints.

A supplier of software components achieves effective reuse by combining **customization** (i.e. maximum variation) in one viewpoint with **standardization** (i.e. minimum variation) in another viewpoint.

For example, a supplier may define a standard information model, and then seek to deliver this on as many platforms as possible.

Or a supplier may make a strategic commitment to a single platform (e.g. Enterprise Java Beans), and then attempt to deliver a highly flexible information model from a common code-base.

# CBD Methods

## Critique of traditional OOAD methods

Let us use this analysis to identify the limitations of traditional OOAD methods.

### Focus on user requirements

Traditional engineering methods, including OOAD, start from the assumption that somebody **requires** something to use, and the engineering task is to construct a **solution** that satisfies that user requirement.

This assumption fits most software development projects, but not all. Some of the time the project objective is to build something that nobody knew they needed. The only known requirement may be the supplier's need to remain in business, but this is not usually regarded as a user requirement, and is usually ignored by requirements engineering methods.

But if CBD projects are supposed to be focused on reuse, this conflicts with the supposed need to be focused on user requirements. This is because **reuse has no meaning within the service use ecosystem**.

### Focus on software system behaviour

Furthermore, most traditional engineering methods formulate the requirements in terms of the behaviour of some system, usually restricted to a computer information system or software system. OOAD methods usually specify these requirements as a set of **use cases**.

Where a software development project can exercise design control over an entire system, then this approach still seems to be valid. However, this approach fails to support the design of components within open distributed systems, because such systems generally lack central design control.

**Q** Is it possible to define a use case for a single component? Do you think it is meaningful or useful to do so? Or would you regard this as a trivialization or perversion of the use case concept?

Furthermore, there are problems with flexibility. The demand for flexibility (which we have positioned within the device use ecosystem) needs to be balanced against the demand for particular services (which we have positioned within the service use ecosystem).

**Q** Is it possible to formulate requirements for flexibility in terms of use cases?

## Methodological implications of our analysis

Any CBD method that is truly focused on software reuse and economies of scale should contain the following elements.

- A systematic way of understanding the design and operational constraints of a given ecosystem.
- A systematic way of defining component interfaces that will be useful, meaningful, relevant, and compatible within the target ecosystem.
- A systematic way of predicting the amount of use that a given component is likely to achieve in a given ecosystem.
- A systematic way of balancing standardization with customization.

Such a method will almost certainly need to articulate several different viewpoints on a distributed system, as required by RM-ODP.

## SCIPIO Approach

### Viewpoints

We define five viewpoints. These are based loosely on the ANSA/RM-ODP set; we have adopted some modifications to make them more generalized.

<b>Viewpoint</b>	<b>Focus</b>	<b>Key Concepts</b>
Enterprise	Business relationships	Stakeholder Intention Role Responsibility
Exchange	Conversations  Workflows and information flows	Joint action Collaboration Transaction Flow
Behaviour	Activities	Service Interface Rule Operation Use cases
Design	Components	Component Component kit Connector
Physical	Infrastructure	Platform Mechanism Protocol

Not all five viewpoints may be applicable or relevant in every situation or project.

## Steps

The SCIPIO Method analyses the requirements for software components in the following way.

Scope Ecosystem	Define the target ecosystem for components.	We use informal techniques to produce an initial scope definition, which may be a combination of text and rich pictures.  Later refinements are more formal, and are based on the models of the ecosystem.
Model Ecosystem	Develop formal descriptions of the ecosystem.	We may produce models in all five viewpoints, or some of them, depending on the situation.
Identify Service Opportunities	Identify unfilled or weakly filled niches in the service use ecosystem.	A service opportunity may be based on any of the following: <ul style="list-style-type: none"> <li>➤ An unsatisfied intention of some agent</li> <li>➤ An unsatisfactory existing service, lacking quality or availability</li> <li>➤ An opportunity to subdivide existing roles.</li> </ul>
Create Service Opportunities	Specify interface of proposed service.	Must include quality of service characteristics as well as 'functional' behaviour.
Confirm Service Meaningful within Ecosystem	Estimate the likely adoption of the proposed component in the target ecosystem.	We check the specification against our analysis of the ecosystem, in all five viewpoints.  In some cases, the use of a component depends on achieving a critical mass within the ecosystem within a defined period.
Confirm Service Economies	Verify that the service can be delivered cost-effectively and profitably.	This may be based on an analysis of the opportunities to reuse existing assets, or to otherwise leverage economies of scale.
Extend Ecosystem	Identify additional ecosystems for potential use.	If the proposed component is not cost-effective or profitable within the original target ecosystem, or it is unlikely to achieve a critical mass of usage, we may need to find additional ecosystems to support the development of this component.

## Scenarios

This method allows for several scenarios of user/requirements, including the traditional ones:

- Provision of whole systems or individual components for a specified user or user community, funded by (or on behalf of) that user/community.
- Speculative development of whole systems or individual components, for publication and sale within a defined market.
- Development of intelligent software agents for release into a global network (such as the Internet), with various payment and funding models.
- Collaborative speculative development, in which a relatively small number of users participate in a development, but a significant portion of the funding comes from speculation against future reuse by a much wider community.

## References

**The Biology of Machines.** Kevin Kelly, Out of Control: The New Biology of Machines. UK edition, Fourth Estate 1994.

**RM-ODP** - the reference model for Open Distributed Processing.

The official website for RM-ODP is <http://www.iso.ch:8000/RM-ODP/>

There are some key papers downloadable from the ANSA website <http://www.ansa.co.uk> but the website itself is so badly signposted that you would be unlikely to find what you wanted.

See instead [http://www.dstc.edu.au/AU/research\\_news/odp/ref\\_model/](http://www.dstc.edu.au/AU/research_news/odp/ref_model/)

**SCIPIO.** For more information on SCIPIO, including a detailed task structure, please see the SCIPIO website at <http://www.scipio.org/>