Developing Large-scale Systems with the Rational Unified Process

Maria Ericsson

Rational Software White Paper

Rational®
the e-development company™

Table of Contents

History

This paper is drawn on "Systems of Interconnected Systems", published in ROAD, May-June 1995, by Ivar Jacobson, Karin Palmkvist, and Susanne Dyrhage [1]. This paper benefits from input from several large-scale systems development projects, and is also intended to align with the Rational Unified Process® version 5.1 [2] and the Unified Modeling Language [3].

Systems of Interconnected Systems

There is a considerable increase in complexity when developing large-scale systems. Not only does it require that you are capable of comprehending a more complex set of artifacts, you are also introducing overhead since you need to manage a larger set of resources. This paper describes an architectural pattern that is used to help control the added complexity overhead. The architectural pattern, among other places discussed in [4], is referred to as a **system of interconnected systems**.

This construct is useful when building very large or complex systems such as command and control systems or highly integrated IT solutions. These type of "super systems" are in most cases divided into several separate parts, each developed independently as a separate system. A super system is implemented by a set of interconnected systems, communicating with each other to fulfill the duties of the super system. One of these systems represents the overall capabilities, and we call it the **superordinate system**. The other systems represent a part of the whole, and we call them **subordinate systems**. A superordinate system is clearly distinct from the subordinate systems that implement it. The relation between the different types of systems is made distinct: from the perspective of the superordinate systems, the subordinate systems are subsystems, see figure 1.

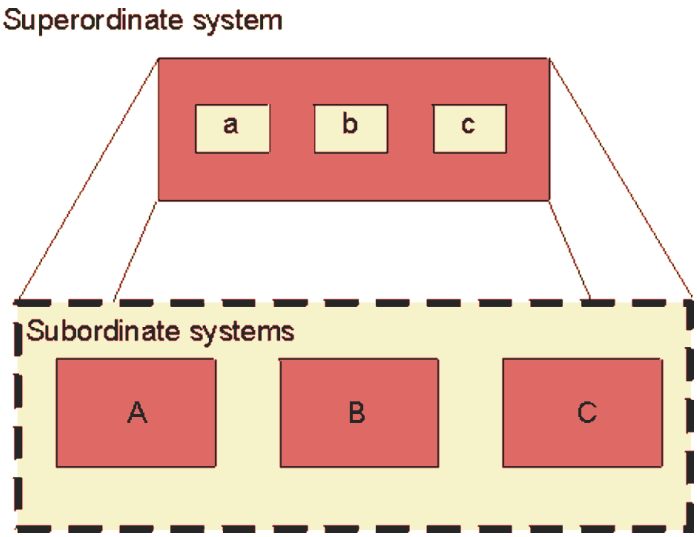

Figure 1. The specification of a superordinate system is implemented by a system of interconnected systems, in which the systems A, B, and C are implementations of the superordinate system's subsystems a, b, and c, respectively.

Separating the superordinate system from its subordinate systems has several advantages:

- The subordinate systems can be managed separately during all life-cycle activities, including sales and delivery.
- It makes it easy to use a subordinate system to implement other superordinate systems by plugging it into other systems of interconnected systems.

- You do not always know when you start building a system whether it is a system of interconnected systems or not. You can start working with a "simple" system view and fairly late in the lifecycle determine whether you need to apply the system of interconnected systems pattern or not.
- It allows you to make internal changes to the subordinate systems without developing a new version of the superordinate system. New versions of superordinate systems are required only due to major functional changes.

Each subordinate system has an associated set of artifacts, with a clear traceability between them. There is also traceability maintained from the artifact sets of the subordinate systems to the corresponding artifact sets of the superordinate system. Each subordinate system can be managed as a separate development project with its own lifecycle phases: inception, elaboration, construction, and transition.

If the "super system" you are building is very large, a subordinate system may need to be divided even further and thus be treated as a system of interconnected systems.

The Software Development Lifecycle

In the Rational Unified Process, the development lifecycle is presented and discussed from two perspectives: the management perspective and the development perspective, see figure 2.

From a management perspective, you go through four lifecycle phases to develop a system, or a new generation of a system. From the development perspective, you develop iteratively versions of the system that are incrementally more and more complete. The activities you perform during an iteration have in the Rational Unified Process been grouped into a set of core workflows. Each core workflow focuses on describing some aspect of the system, resulting in a model of the system, or a set of documents.

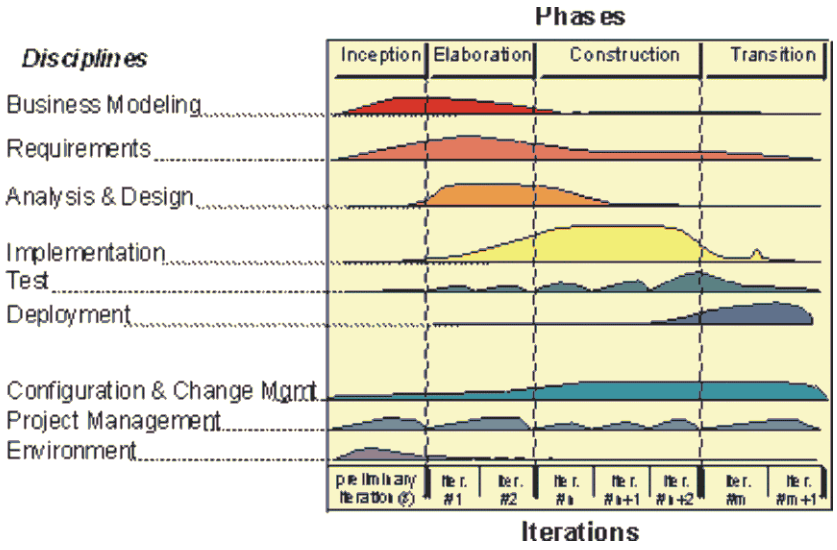

Figure 2. The iterative model

Applying this to systems of interconnected system, the superordinate system as well as each of its subordinate systems all go through its own lifecycle, and are often treated as separate projects.

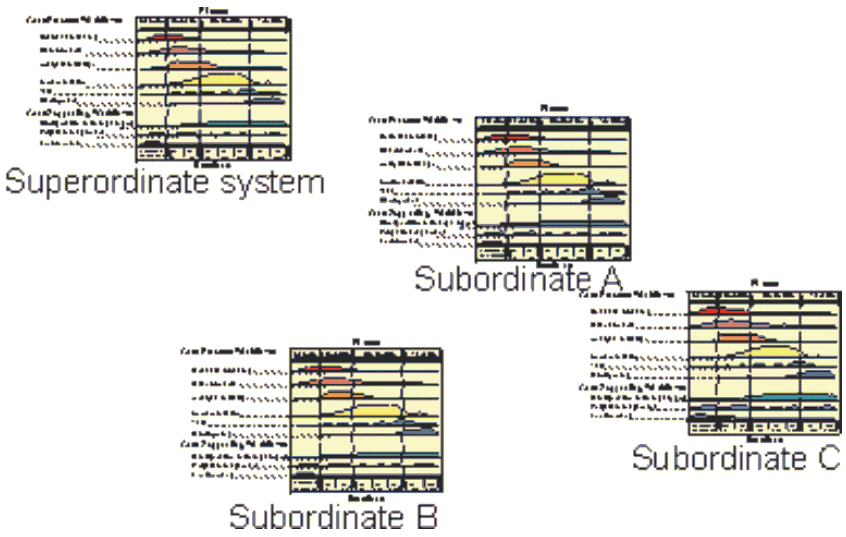

Figure 3. Each system, superordinate and subordinate, goes through its own lifecycle.

The lifecycles do, of course, have dependencies. To manage those dependencies correctly is one of the challenges of developing a system of interconnected systems. The dependencies are of the following kind:

- The lifecycles are dependent in time. The lifecycle of the superordinate system starts first. Once the superordinate system has gone through at least one iteration and the interfaces of the subordinate systems are relatively stable, the lifecycles of the subordinate systems can start. In fact, you may not even know which the subordinate systems are until you've gone through at least one iteration on the superordinate system.
- The lifecycle of the superordinate system may go into maintenance once the interfaces of the subordinate systems are stable. This means that no active development is done, only if issues occur that requires changes of the interfaces of the subordinate systems.
- The interfaces of the subordinate systems are owned by those who develop the superordinate system. For more on interfaces, see [3] and [5].
- The classes that implement the subordinate system's interfaces are owned by those who develop the subordinate systems.

Workflows and Artifacts of System Development

A natural assumption is that the superordinate system, as well as the subordinate systems, can be developed with the same set of artifacts and developed through the same workflows as are typical for non-composite, systems. Before we can proceed showing how this can be done, these artifacts and workflows have to be introduced. In the Rational Unified Process, we introduce five core process workflows, see figure 4. These are:

- Business engineering – the purpose is to assess the organization in which the system will be used, to better understand the needs and problems that is to be solved by the system. The result is a business use-case model and a business object model. This workflow can be considered optional. If the organization in which the system is to be employed is very simplistic it may not add value.
- Requirements – with the purpose to capture and evaluate the requirements, placing usability in focus. This results in a use-case model, with actors representing external units communicating with the system, and use cases representing transaction sequences, yielding measurable results of value to the actors.

- Analysis & Design – with the purpose to investigate the intended implementation environment and the effect it will have on the construction of the system. This results in an object model (the design model), including use-case realizations that show how the objects communicate to perform the flow of the use cases. This might include interface definitions for classes and subsystems, specifying their responsibilities in terms of provided operations. This object model is also adapted to the implementation environment in terms of implementation language, distribution etc. It is sometimes useful to consider the results of analysis a separate model, then called the analysis model.

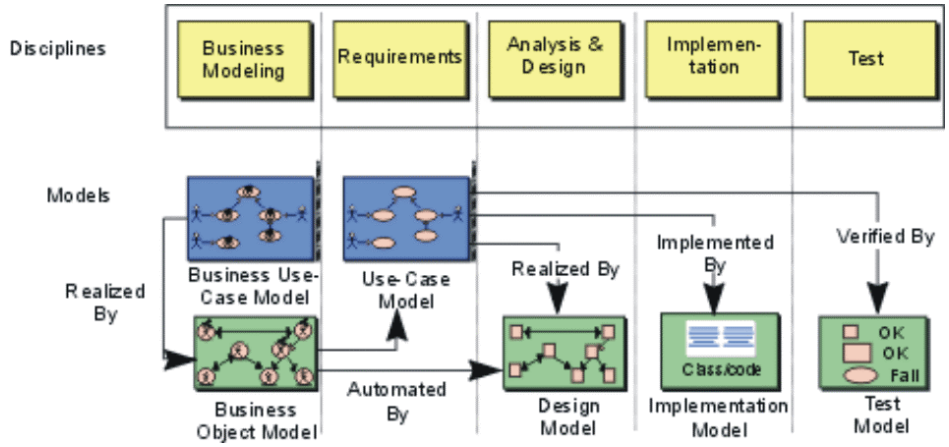

Figure 4. Each core process workflow is associated with a particular set of models.

- Implementation – with the purpose to implement the system in the prescribed implementation environment. This results in source code, executables, and files.
- Test – with the purpose to ensure that the system is the one intended, and that there are no errors in the implementation. This results in a certified system that is ready for delivery.

Development of a System of Interconnected Systems

What we really must do is define how a system's responsibilities can be distributed over several systems, each one taking care of a well-defined subset of these responsibilities. This means that the main goal is to define the interfaces among these subordinate systems. When we have accomplished this, the rest of the work can be done separately for each subordinate system, according to the "divide-and-conquer" principle. Therefore, this is all we must do for the system as a whole, apart from testing it once implementation is done.

Criteria for Decomposition

So how do you decide whether to decompose your system into a system of interconnected systems? There are a couple of characteristics that should be considered:

- For a system of considerable size and complexity, you can partition the problem into smaller pieces that are easier to comprehend one at a time.
- Are you dealing with physically separate systems? Often this is the case when working with legacy systems or legacy architecture.
- Decomposing helps define natural and narrow interfaces between parts of the system.
- You may decide to implement one part of the system using some major COTS (Commercial-Off-The-Shelf) product. Decomposing will help clarify how you intend to use the COTS product.

- Partitioning allows you to get the most of a distributed development organization, and clearly partition the work among several geographically dispersed teams.

Risks to be considered are:

- Overuse of decomposition can hide the overall problem for all the details of it.
- By using physically separate systems or physically separate teams you run the risk of killing any form of reuse, and end up with a rigid stove-pipe system.

Organization

To mitigate the risks mention above, it is critical to have a group of people assigned to overlook the whole development effort. This group is often called an architecture team and should focus on the following key concerns:

- That there is an overall architecture defined and that it is followed in the subordinate systems.
- That there is a reasonable focus on reuse and sharing of experience between the subordinate systems.
- That there is a clear understanding of what artifacts to produce and what the relations are between artifacts of the subordinate and superordinate systems.
- That an effective change management strategy is defined and followed by all teams.

The architecture team can, but doesn't always, own the development of the superordinate system. For a more thorough discussion on organization, see [6].

The Lifecycle of the Superordinate System

First, you may optionally choose to do **business engineering** to better understand the context of the system. This is value adding if:

- there is a need for the developers to better understand the organization,
- the organization itself is heterogeneous in how it does its business and terminology and processes need to be aligned, or
- the software engineering effort is done in conjunction with a business re-engineering effort.

See also [6].

This effort would result in a business use-case model and a business object model. Alternatively, you may choose to do limited business engineering, only looking at key concepts in the business domain and document those in a business object model. This is often referred to as domain modeling.

Once you have "primed the pump" with a set of business models you need to begin elicit **requirements** on the entire system. We have the same need for requirements modeling for a system of interconnected systems as for any other system. A use-case model is a very natural way to express the results, see [7]. The most straightforward way to look at this superordinate use-case model is to assume that it completely captures the system's behavioral requirements. This is however probably seldom the case. Since we need to implement the system with other systems, the overall system is probably quite complex. Therefore it is not a good idea to try to be exhaustive at this level. Thus, a superordinate use-case model usually gives a complete but simplified picture of the system's functional requirements. There is no need to be too detailed at this level, since the detailed modeling will be performed within each of the implementing subordinate systems. It is also often true that many requirements are not necessarily visible in a superordinate use case that cuts several subsystems. Such requirements can be said to be "local" to a subsystem.

The purpose of **analysis & design** is to achieve a robust architecture of the system, which is of course of vital importance for a system of interconnected systems. The developers of the superordinate system must achieve a robust structure of subordinate systems, while they need not bother at all with their inner structures. We will therefore model a division of the system into smaller parts using subsystems. In order to get the right set of subsystems, and to get a first idea of how to distribute the responsibilities of the superordinate system over these subsystems, we develop an analysis model. The analysis classes should represent the roles played by things in the system when the high-level use cases are performed. Therefore, the analysis model gives a simplified picture of the complete object structure, in analogy with the high-level use-case model.

Functionally related analysis classes are grouped together into subsystems. Thus we get a subsystem structure that is ideal in the sense that it is based upon only functional criteria, for example, we have not taken into account any distribution requirements. An often highly influential factor is the existence of legacy systems. Legacy systems may fulfill some or many of the responsibilities defined in the analysis model. The existence of such systems may even cause a re-partitioning of the responsibilities found in analysis so that you can achieve maximal reuse of existing capabilities.

The result of design may be a subsystem structure that is very different from the one we defined based upon functional criteria during analysis. Thus we end up with a structure of design subsystems, which will each be implemented by a subordinate system, see figure 5. In order to be able to continue the development work for each such system separately, interfaces are defined for each subsystem. In fact, defining interfaces is the most important activity performed at the superordinate level, since interfaces provide rules for development of the subordinate systems. No design classes are defined, the only thing you do is define the interfaces of the design subsystems.

No **implementation** is done as part of the lifecycle of the superordinate system, other than maybe some prototyping work to explore particular technical aspects of the system.

The final workflow is **test**, which in this case means integration test when the different subordinate systems are assembled, and also testing that every superordinate use case is performed according to its specification by the interconnected systems in cooperation.

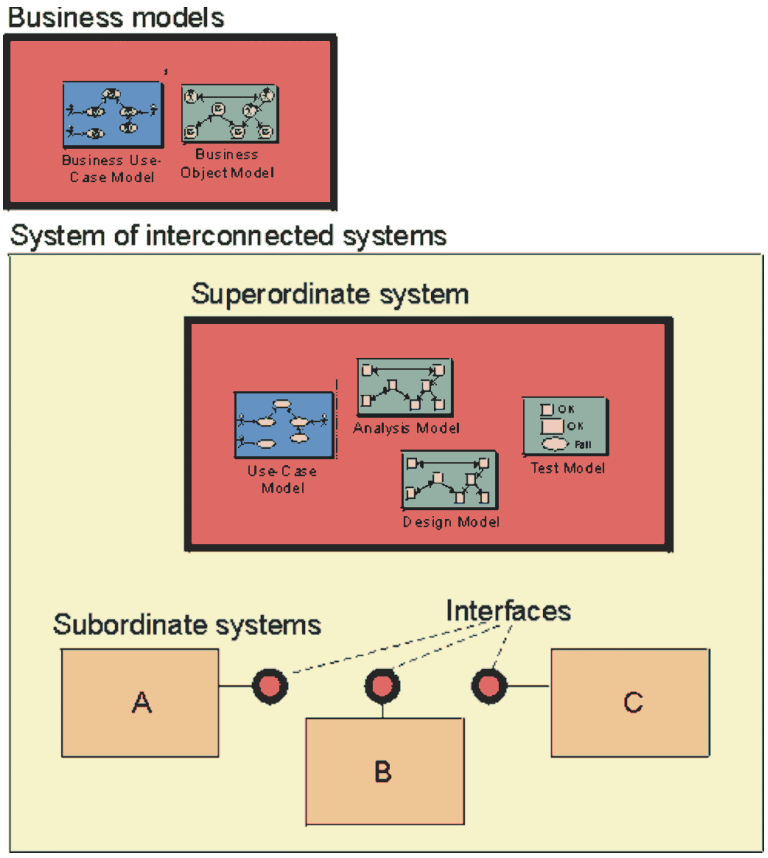

Figure 5. The superordinate system is described by a set of models, where the subsystems defined in the high-level design model will be implemented by subordinate systems. Interfaces to the subordinate systems are owned by the superordinate system.

To show how you would work with the superordinate system, here are several sample iteration plans; one for an iteration in the inception phase of the lifecycle of the superordinate system, one for an iteration in the elaboration phase. We use activity diagrams to describe the iteration plans. The action states in these diagrams correspond to workflow details as defined in the Rational Unified Process.

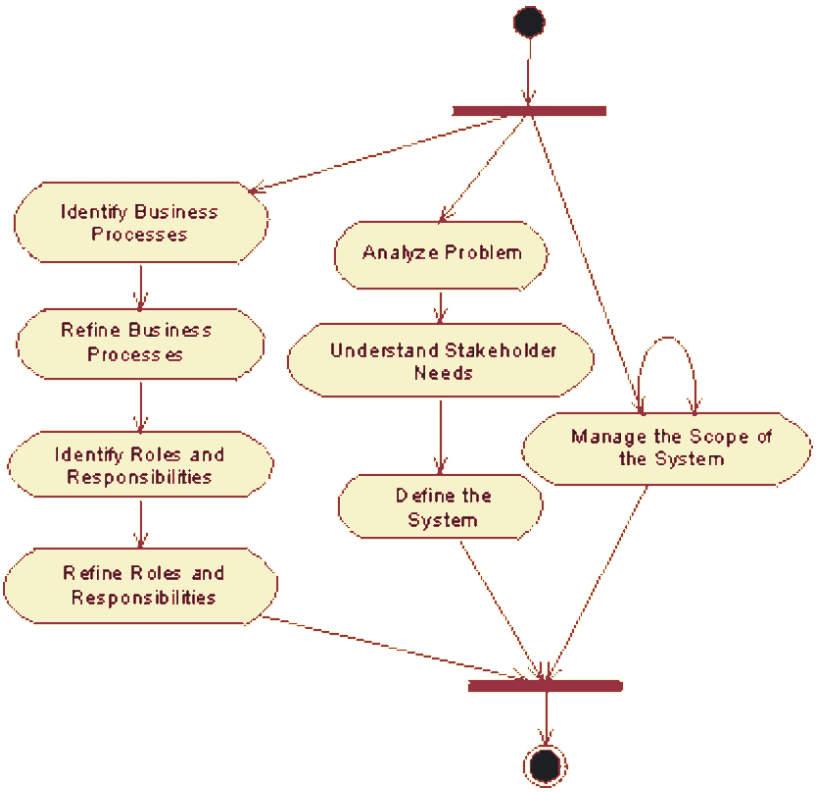

Figure 6. An activity diagram describing an example of an inception iteration plan for the superordinate system.

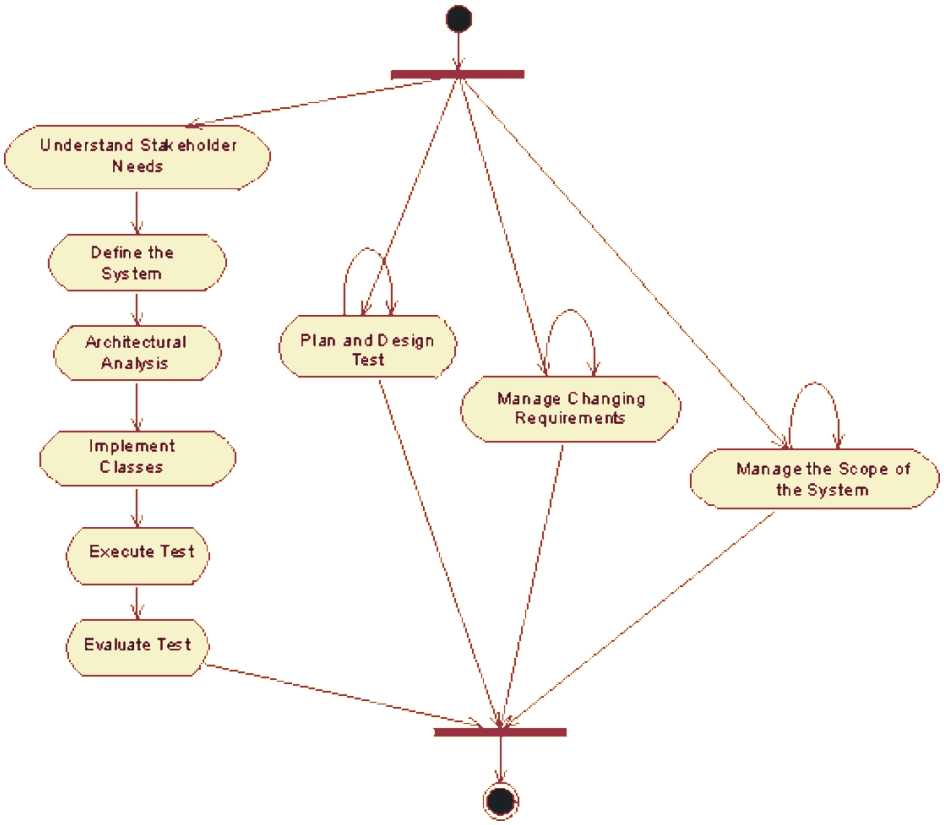

Figure 7. An activity diagram describing an example of an elaboration iteration plan for the superordinate system. The action state "implement classes" is here since you may do some limited implementation of prototypes to explore technical aspects of the system.

The Lifecycle of the Subordinate System

Each subordinate system is developed in the usual way, as a black box considering other systems with which it communicates as actors. You perform the usual set of activities and develop the usual set of models, as described above, for each such system. If the models at the superordinate level is defined in all details, you get complete recursion between the models at different levels, but as mentioned above, this is in practice seldom the case.

For the subordinate system, you will perform the **requirements** workflow. The interfaces and the use cases of the superordinate system will be your primary input to understand the boundaries of the subordinate system and what its actors are.

When performing **analysis & design** for the subordinate system, the interfaces defined in the superordinate system will be your "boundary conditions", together with the high-level use cases.

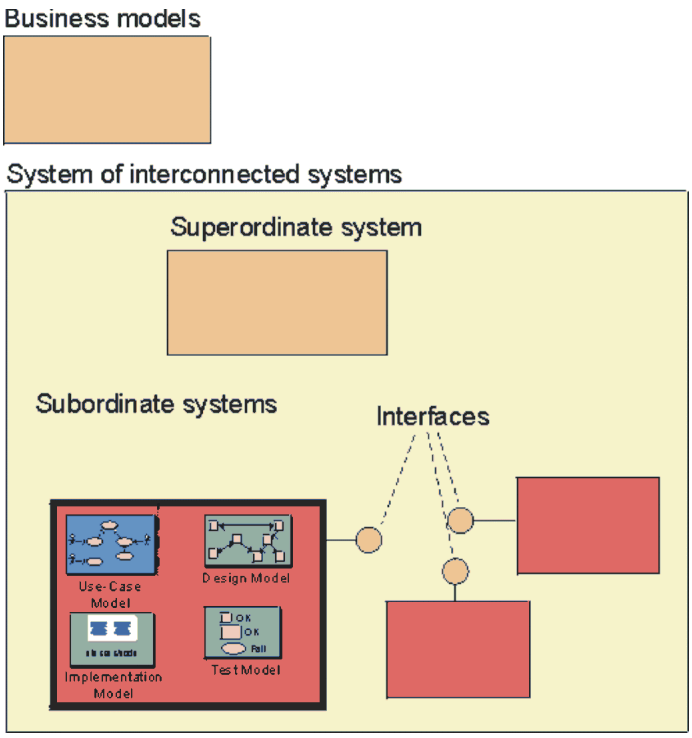

Figure 8. The subordinate systems are described by their own sets of models.

To show how you would work with the subordinate system, here are two sample iteration plans from its lifecycle.

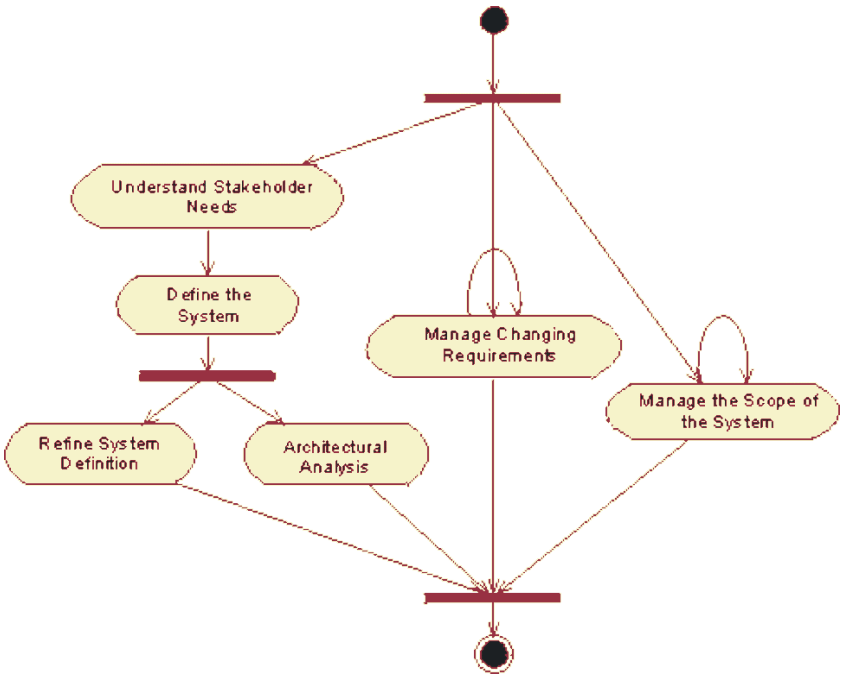

Figure 9. A sample inception iteration plan for the subordinate system. This is an incomplete iteration, since no executable is produced.

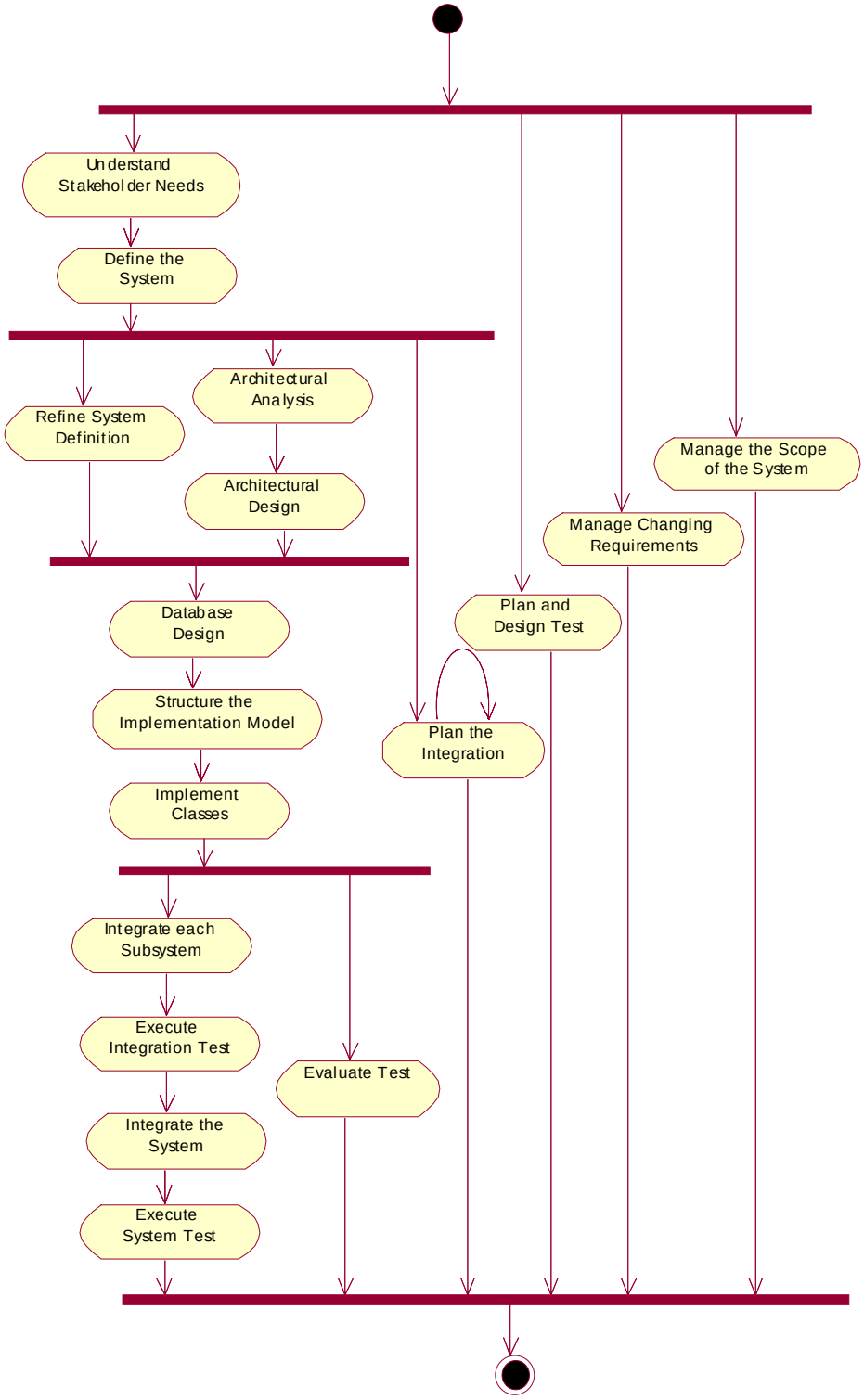

Figure 10. A sample elaboration iteration plan for the subordinate system. The focus in elaboration is on completing the refined system definition and the architecture.

Use Cases in Systems of Interconnected Systems

You should build one use-case model for each of your systems, both superordinate and subordinate, in your systems of interconnected systems. They are dependent in the following way (see also figure 11):

- A high-level use case in the superordinate system is split (not always, but often) onto the subsystems. Each 'split' becomes a use case in the model for its subordinate system, see figure 11.
- From the perspective of one subordinate system, the other subordinate systems are actors in its use-case model, see figure 12.

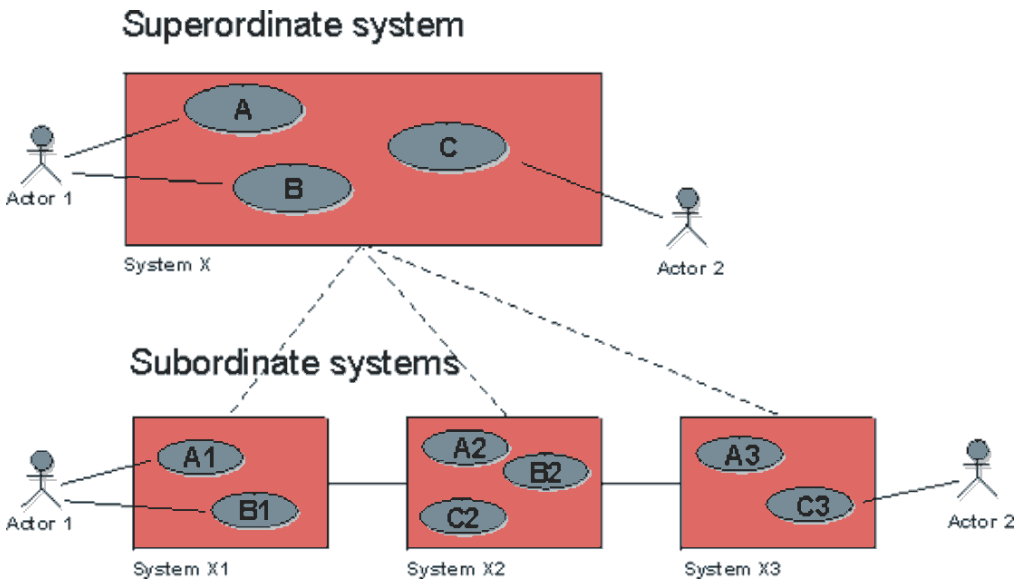

Figure 11. The relationship between high-level use case in the superordinate system, and detailed use cases in the subordinate systems.

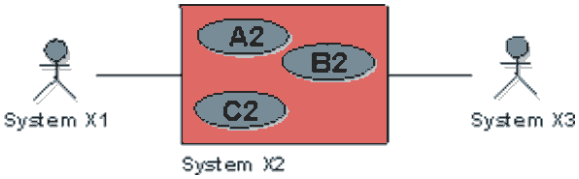

Figure 12. In the use-case model of the subordinate system X2, the other subordinate systems X1 and X3 are viewed as actors.

There are some special considerations for the use cases describing the superordinate system. Since you will in a sense re-describe all requirements for each subordinate system, there is no point in delving too much in detail with these use cases. In the normal case, it is usually sufficient to only write down the step-by-step outline to the flow of events of the high-level use cases and not detail it to narrative text.

In this use-case model you should not use any of the use-case relationships (generalization, extend, include). In general, it does not add value for the following reasons:

- You will not describe the high-level use cases in detail, so you do not need to be worried about text appearing in several places.

- You will anyway structure the information when you "split" high-level use cases onto subordinate systems. Mixing this with other structuring mechanisms may be confusing.

There is an important exception this, which is if you are aiming at finding reusable components in your system of interconnected systems. Structuring the superordinate use-case model to find generic use cases is a powerful method to find reusable components. See [6] for more details on this topic.

Design Models in Systems of Interconnected Systems

Each system in your system of interconnected systems, both superordinate and subordinate, should have its own design model. The design models are related in the following way:

- Subsystems in the design model for the superordinate system define the boundaries of the subordinate systems.
- The operations you define on subsystems of the superordinate system are input to defining interfaces to the subordinate systems.

The design model of the superordinate system is described in less detail than the subordinate design models. You would produce the following:

- Subsystems, described briefly.
- Use-case realizations, in terms of how the subsystems collaborate. The usual way to document these high-level use-case realizations is to draw sequence diagrams. It is by producing these diagrams that you define the "split" of a high-level use case onto subordinate systems, see figure 13.
- Operations of the subsystems.
- Interface definitions for the subsystems.

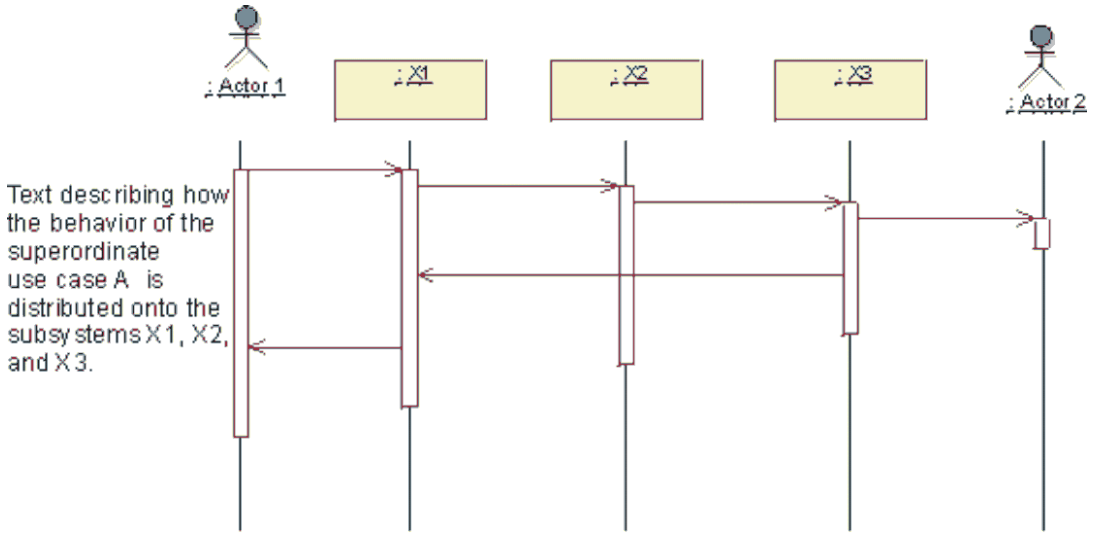

Figure 13. A sequence diagram for the realization of the superordinate use case A.

Information Sets in Systems of Interconnected Systems

An area where most organizations spend a lot of effort is to understand how to manage their artifacts and correctly understand their dependencies. We have in previous section discussed specifically the dependency between superordinate and subordinate use-case models and design models. There are also general dependency issues that need to be considered.

When a system goes through a pass of the lifecycle, you will produce artifacts that can be organized into information sets [8], see figure 14. These sets are organized based on what artifacts evolve "together".

- You can customize the exact contents of each set depending on what type of application you are building, but the sets remain the same.
- You need to understand the dependencies between the sets, so that you can maintain traceability between artifacts in an effective way.

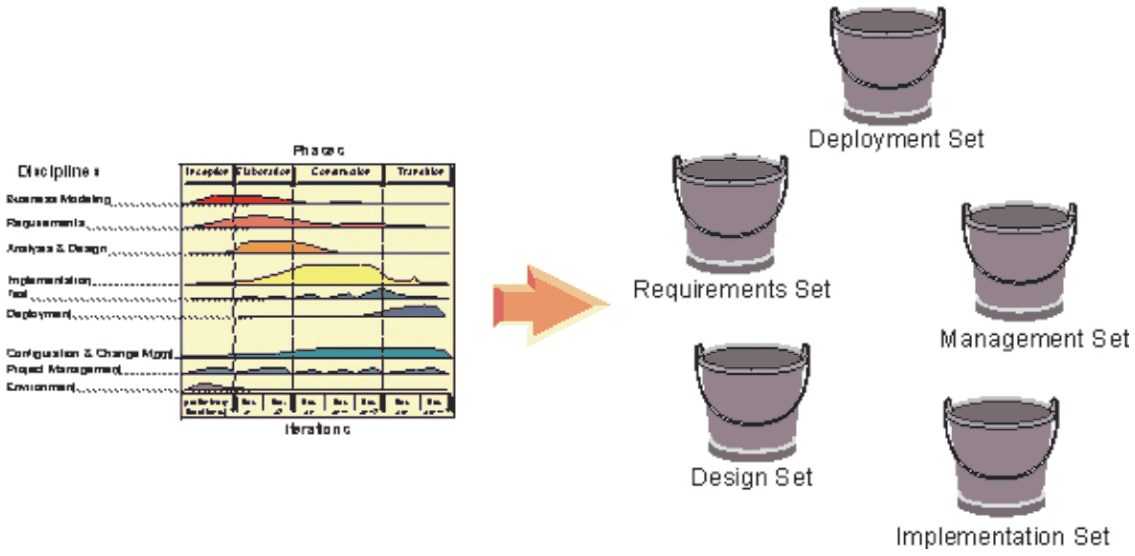

Figure 14. The lifecycle of a system will produce information sets.

In a system of interconnected system, the superordinate and each subordinate system will produce its own set of information sets, see figure 15.

- A subordinate information set has a dependency to its corresponding superordinate information set.
- The type of content may differ in corresponding information sets between subordinate systems, since the application types may differ.
- Corresponding subordinate information set should be independent, except they fulfill the same subsystem interfaces defined in the superordinate system.

The effort put into maintaining traceability between artifacts in the superordinate system and subordinate systems should be kept at a minimum. Maintaining traceability within the system should be prioritized.

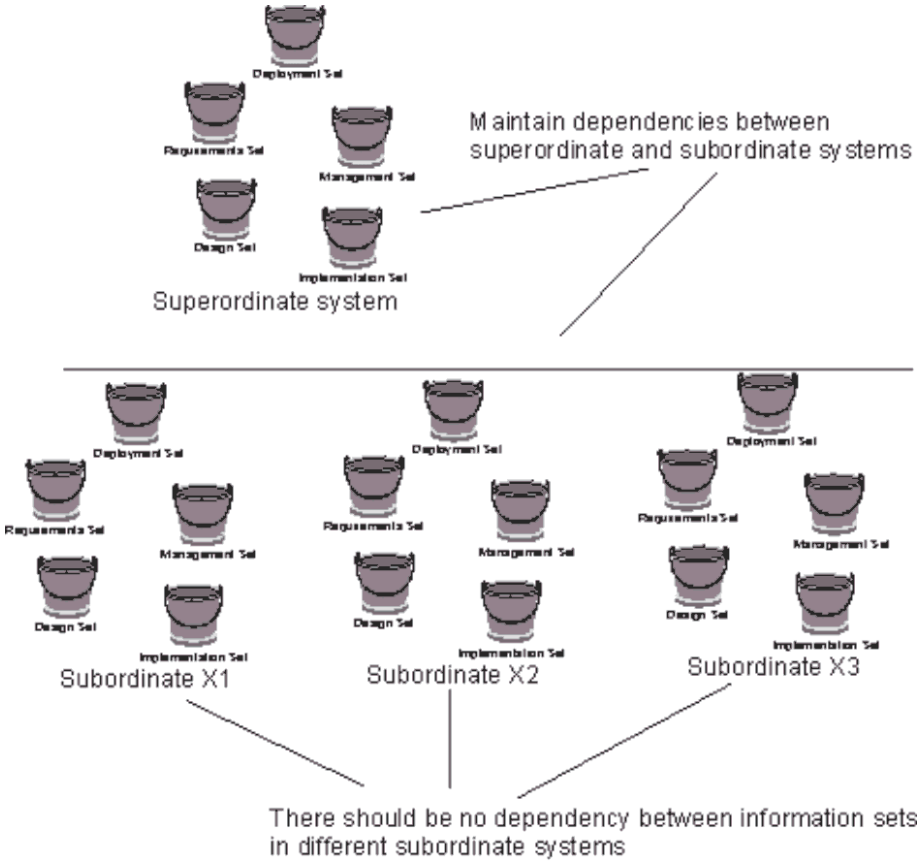

Figure 15. Each system in a system of interconnected systems will produce its own set of information sets.

Architecture in Systems of Interconnected Systems

Each system in your systems of interconnected systems, both superordinate and subordinate, should have its architecture defined.

For the superordinate system, an architecture document should discuss:

- Key use cases or scenarios for the superordinate system.
- Layering of the system of interconnected system.
- How to handle reuse between subordinate systems, and what to reuse.
- Key mechanisms and their implementation, the ones generic enough to be used by all subordinate systems. For example, all subordinate systems should use common mechanisms for communication, error reporting, and fault management, or the superordinate system will not behave as a homogenous system.

For the subordinate systems, an architecture document should clarify:

- The role of the subordinate system within the system of interconnected systems.
- Key use cases or scenarios for the subordinate system.
- How the subordinate system will use the layered structure defined for the system of interconnected system. Another way of saying this is that you need to define how the subordinate system will fulfill the role that has been defined for it in the layered architecture of the superordinate system.

- What generic key mechanisms will be used and how, and what application-specific key mechanisms are added.
- How reuse will be applied. Specifically, which subsystems are common across two or more subordinate systems, and which mechanisms are built to allow subordinate systems to communicate.

Relations Between Systems

You have seen that the usual system-development activities also can be applied to systems implemented by systems of interconnected systems. This is advantageous because it means that you do not need to handle such systems in a way significantly different from that used with other systems. You also get a nice separation of the superordinate system from its implementation in the form of other subordinate systems. Each system in a system of interconnected system has its own lifecycle. Since each system may have different characteristics, you may use variations of the development process to produce them. In the terms of the Rational Unified Process [2], you would have a different development case for each system.

A final note on the independence between systems involved in a system of interconnected systems:

First, take a look at the subordinate systems. Each such system implements one subsystem in the superordinate system's design model. The subsystems depend on each other's interfaces and not explicitly on each other, see figure 12. Thus, you can exchange one subsystem for a new version of it without affecting other subsystems, as long as the new subsystem conforms to the same interface. You get exactly the same relation between the subordinate systems. Each subordinate system views its surroundings as a set of interfaces. This means that you can exchange a system with another, as long as the new system plays the same roles towards other systems, i.e. as long as it can be represented with the same set of interfaces. Systems refer to each other's interfaces as specified by the corresponding relations between subsystems and interfaces in the superordinate model.

In the use-case model of a subordinate system, the interfaces of the other subordinate systems it interacts with is represented as actors. You can say that a subordinate system looks upon the interfaces of another system as offered by the corresponding actors, and therefore never has to refer directly to the other system, see figure 16. Note that interface B occurs in several places in figure 12, indicating that it is really the same interface referred to by subsystems in the superordinate system and by the corresponding subordinate systems.

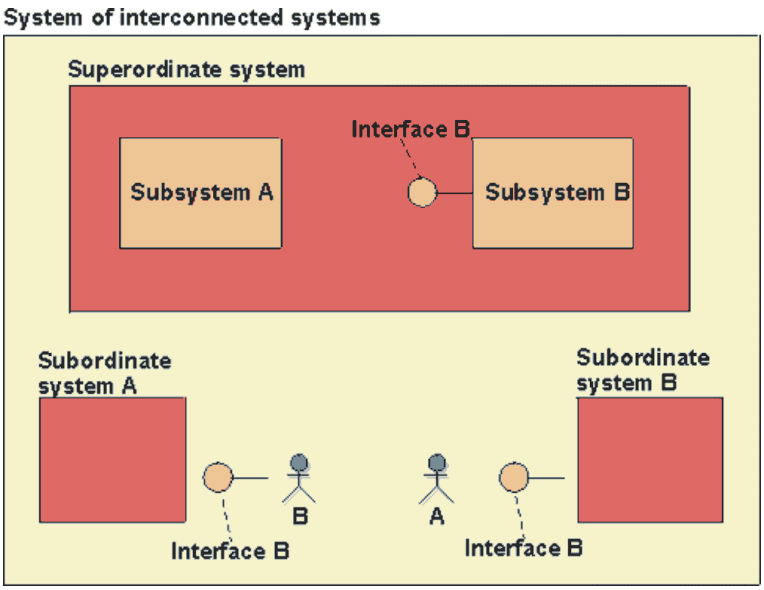

Figure 16. The subsystems of the superordinate system depend on each other only through their interfaces. The implementing subordinate systems therefore get the same kind of independence. In the model of the superordinate system, subsystem B provides interface B to other subsystems. The corresponding subordinate system B therefore needs to provide the same interface B to other subordinate systems.

How about the superordinate system, what is its relation to its subordinate systems? It is independent of its implementing systems in the following sense: Each such system is only an implementation of what we have specified in the models of the superordinate system, it is not part of its specification. For practical reasons you have to define traceability links between systems at different levels, in order to trace requirements, and the most "tidy" way to do this is to define such links only between the interfaces, see figure 11. In fact, one may even say that the subordinate systems are nothing more than implementations providing the interfaces defined in the superordinate models.

But, this is for systems that are anything more than simple examples not enough. An interface does not specify anything else than what goes on at a particular interaction point. A subordinate system may have 100's of interfaces and each interface 10's of operations. Relating an input at one interface to one or many outputs at another interface is impractical to do in an interface description. This is why you need use cases to explain the semantics of the subordinate system.

You can conclude that each system involved when a system is implemented by a system of interconnected systems are independent of the other systems, but they depend strongly upon each other's interfaces. This gives you a very good platform for parallel development of the subordinate systems.

Application Areas

The architecture and modeling techniques for systems of interconnected systems can be used for different types of systems, such as:

- distributed systems
- very large or complex systems
- systems combining several business areas
- systems reusing other systems
- distributed development of a system

The situation may also be the reverse: from a set of already existing systems, we define a system of interconnected systems by assembling the systems. In fact, in some cases this is how the large system evolves in earlier phases of its evolution. You realize you have systems that could be interconnected, and doing so creates a "large system" that adds more value than the two separate systems.

In fact, for any system where it is possible to view different parts of the system as systems of their own, it is advisable to define it as a system of interconnected systems. Even if it is a single system today, it may later prove necessary to split the system into several separate products, due to distributed development, reuse reasons or customers' needs to buy only parts of it, to mention some examples.

As a conclusion we will take a closer look at a couple of cases where the architecture for systems of interconnected systems can be used. We will, for each of the examples, show that the system in question has to be considered *both* as a single system *and* as a set of separate systems, indicating that it should be treated as a superordinate system implemented by a system of interconnected systems.

Large-scale systems

The telephone network is probably the world's largest system of interconnected systems. This is an excellent example where more than two system levels are needed to manage complexity. It is also an example of a case where the top-level superordinate system is owned by a standardization body, and different competing companies develop one or several subordinate systems that must conform to this standard. Here, we will discuss the mobile telephone network GSM (Global System of Mobile Telephony) to show the advantages from implementing a large-scale system as a system of interconnected systems.

The functionality of a very large system usually combines several business areas. For example, the GSM standard covers the entire system, from the calling subscriber to the called subscriber. In other words, it includes both the behaviors of the mobile telephones and the network nodes. Because different parts of the system are products of their own that are bought separately, even by different kinds of customers, they should be treated as systems of their own. For example, a company that develops complete GSM systems will sell the mobile telephones to subscribers, and network nodes to telephone operators. This is one reason for treating different parts of a GSM system as different subordinate systems. Another reason is that it would take too long to develop such a large and complex system as GSM as one single system; the different parts must be developed in parallel by several development teams.

On the other hand, because the GSM standard covers the entire system, there is reason to also consider the system as a whole, that is, the superordinate system. This will help developers understand the problem domain and how different parts are related to each other.

Distributed systems

For systems distributed over several computer systems, the architecture for systems of interconnected systems is very suitable. By definition, a distributed system always consists of at least two parts. Because well-defined interfaces are necessary in distributed systems, these systems are very well suited also to be *developed* in a distributed fashion, that is, by several autonomous development teams working in parallel. The subordinate systems of a distributed system can even be sold as products of their own. Thus, it is natural to regard a distributed system as a set of separate systems.

The requirements for a distributed system usually cover the functionality of the entire system, and sometimes the interfaces between the different parts are not pre-defined. Moreover, if the problem domain is new for the developers, they first have to consider the functionality of the entire system, regardless of how it will be distributed. These are two very important reasons to view it as a single system.

Reuse of Legacy Systems

Almost more often than not, large systems reuse legacy systems. The legacy can be described as a subordinate system. You would then "re-engineer" a use-case model and maybe an analysis model for the legacy system, to understand how it can work in the larger context of the superordinate system. These re-engineered models does not necessarily have to be complete, at a minimum they need to cover the functionality of the legacy that has a direct impact to the functionality of the rest of the system of interconnected systems, or that may require modification.

Use of Prefabricated Packages

A system can be an integration and customization of two or more prefabricated packages. A good example are the Enterprise Resource Planning (ERP) systems. Many ERP systems are a composition of subordinate systems such as MRP (Material Resource Planning), Inventory Management, Supply Chain Management, etc. Similar compositions are available in other areas such as human resource or payroll applications. They are like prefabricated systems, which you have to specialize and interconnect with other standard packages in order to get a complete system. To understand what the set of packages do to together you need the superordinate system. This case is what many customers in the financial community is faced with today.

Summary

This paper introduces an architectural pattern for systems of interconnected systems. This construct allows recursion not only within one model, it considers each subsystem a system in its own right and the recursition is between all the artifacts sets of each of the systems. The introduced architecture is used for systems that are implemented by several communicating systems. Each involved system is described by its own set of models, separate from other systems' models.

The advantages of using this technique are obvious: you can approach rather complex problems and understand them using a "divide and conquer" technique. However, the drawback is that you risk more overhead, desynchronized schedules. We have also seen examples that organizations find it very hard to employ an iterative lifecycle to the superordinate system, thereby running the risk that risks are being pushed towards the end of the lifecycle of the superordinate system. You also need to watch out that a reasonable and effective reuse strategy is followed to avoid developing a set of "stove-pipe" systems.

The examples given illustrate that the architecture for modeling systems of interconnected systems is useful in many different application areas. In fact, you may use the suggested architecture for any system where it is possible to view the different parts as systems of their own.

References

[1] Jacobson, I.; Palmkvist, K.; and Dyrhage, S., *Systems of Interconnected Systems*, ROAD, 2(1), 1995.

[2] – *Rational Unified Process* version 5.1.

[3] – Rumbaugh, J.; Booch, G.; Jacobson,I., *UML Reference Manual*, Addison Wesley Longman, 1999.

[4] – Herbert A. Simon, *The Sciences of the Artificial*, MIT Press,1981.

[5] – Jacobson, I.; Bylund, S.; Jonsson, P., *Using Contracts and Use Cases to Build Plugable Architectures*, Journal of Object-Oriented Programming, May/June, 1995.

[6] – Jacobson, J.; Griss, M.; Jonsson, P., *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley Longman, 1997.

[7] – Jacobson, I., *Use Cases in Large-Scale Systems*, ROAD, 1(6), 1995.

Corporate Headquarters
18880 Homestead Road
Cupertino, CA 95014
Toll-free: 800-728-1212
Tel: 408-863-9900
Fax: 408-863-4120
E-mail: info@rational.com
Web: www.rational.com

For International Offices: www.rational.com/worldwide

TP-#### /00. Subject to change without notice.