

5 TECHNIQUES AT A GLANCE

5.1 INTRODUCTION

This chapter presents a set of core modeling concepts for applying CBD for effective e-business systems. The purpose of this chapter is not describe a complete definitive methodology, but to establish “just enough” semantics and notation using practical techniques that lend themselves to a component oriented approach. Hints and tips have been included to guide the reader. The following techniques are presented in catalog form for ease of use:

- Business modeling.
- Business type modeling.
- Use case modeling.
- Collaboration modeling.
- Architecture modeling.
- Interface specification modeling.

The techniques have evolved from numerous sources, including Sterling Software’s Advisor method (Dodd, 1999), which was itself inspired by the ideas of Catalysis™ (D’Souza and Wills, 1998), as well as the author’s experiences in developing and applying the SELECT Perspective™ (Allen and Frost, 1998) approach. We’ll close the chapter by providing some pointers on how the techniques are used in relation to one another.

5.1.1 The Rise of UML

The Unified Modeling Language (OMG, 2000) has become the generally accepted standard for object oriented analysis and design modeling notation and semantics. Our notations use UML as a base, only extending notation where it is not explicitly provided.

It is important to understand that we do not cover specifically OO techniques in this book. There are seemingly countless books on applying UML on OO analysis and design projects. If you wish to design your components using OO implementations specifically for e-business systems the choice is more limited though books are now available (Conallen, 2000).

5.2 BUSINESS MODELING

Pressure for fast business results has caused many current Internet systems to have difficulties with implicit, confused or quickly changing business objectives. This lends a new urgency to business modeling that help with understanding rapidly changing business contexts and to clarify and plan new e-business processes at four levels:

- Strategy: Identifying objectives, strategies, and goals. The balanced score card approach (Kaplan and Norton) is used to ensure that the measures reflect the customer, financial, internal and learning aspects of performance.
- Activity: Understanding process flow and collaboration using, for example, process flow diagrams and process collaboration diagrams.
- Organization: Clarifying organizational structure using, for example, organization flow diagrams.
- Information: Understanding business information and rules using, for example, business concepts.

5.2.1 Business Modeling Concepts

There are many different modeling techniques that can be used for business modeling and the reader should feel free to use their favorites. However, cosmetics are irrelevant here: it is correct application and understanding of concepts that is critical for effective e-business using business components as we emphasize throughout this book.

We restrict ourselves here to the core concepts that we defined back in 3.3 under three headings: business concepts, business propositions and business processes. We summarize those concepts here for convenience.

A business concept represents key information categories.

A business proposition can be a rule that defines or constrains aspects of the business, a goal that directs or a problem that inhibits an aspect of the business.

A business process is a group of related business capabilities that add value to a customer. The capabilities are realized by families of tasks, that may collaborate in different event driven groups to fulfill the business process.

There are three types of business process:

- Customer processes provide value add and provide deliverables to an external client.
- Sustaining processes add value to the external customer, however they do not exchange information or material directly with the customer.
- Enabling processes have internal customers. They provide the services necessary

to support the core processes and manage the business.

A business process may also be distinguished an atomic process to assist with achieving a good level of granularity and prevent over-analyzing to unhelpful levels of detail. An atomic process is task that is the responsibility of one role acting in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state. Note that although one role is responsible for the atomic process, that role may collaborate with several other roles in order to complete the atomic process.

A role is a set of responsibilities hat may correspond with a job title (purchasing agent) or organizational unit name (purchasing), but may also be abstract (purchaser).

5.2.2 Business Modeling Notation

Business Concept Diagram:

Key business concepts are sketched out on a business concept diagram as illustrated in figure 5.1. The diagram is very much a thinking tool used to help understand the business domain. Business concepts are declared as boxes; lines between boxes represent associations between business concepts.

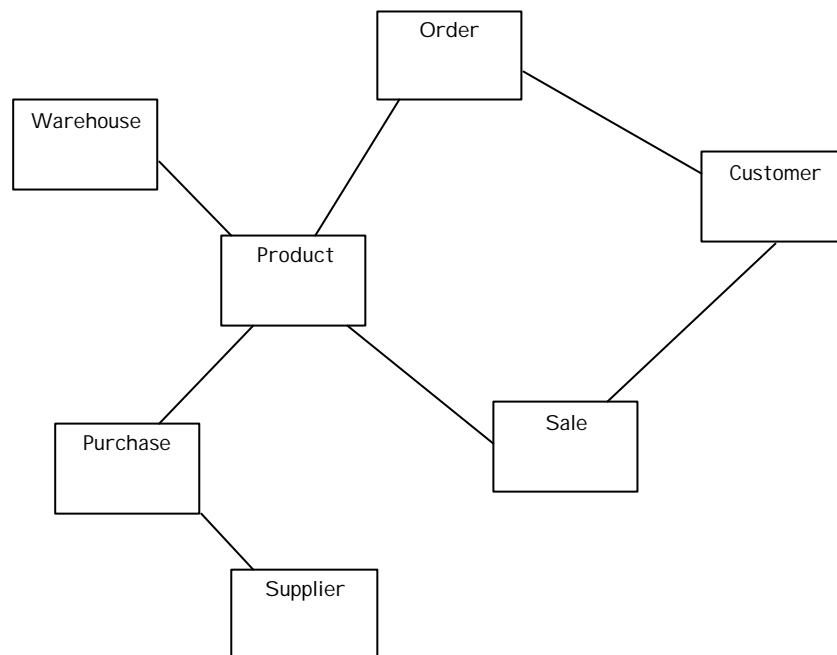


Figure 5.1 Business Concept Diagram

Organization Flow Diagram:

An example organization flow diagram is shown in figure 5.2. Roles are shown as boxes and arrowed lines represent information dependencies. The roles shown on an organization flow diagram are usually physical things: organization units, external agents (including customers and suppliers), or computer systems. Both “as is” and “to be” diagrams are possible.

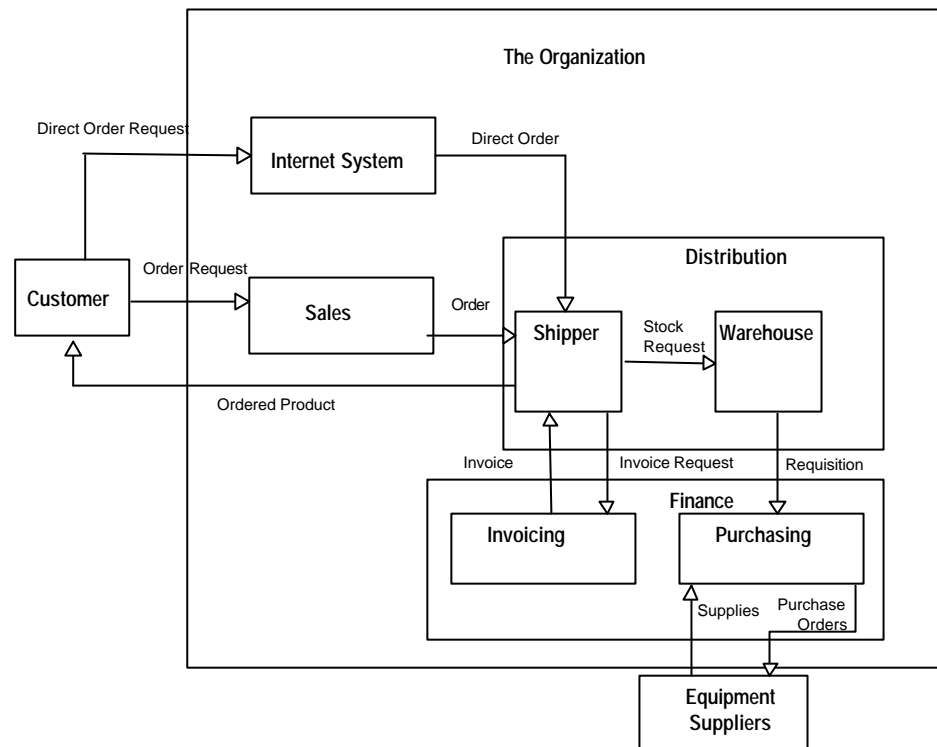


Figure 5.2 Organization Flow Diagram

An organization flow diagram illustrates all external and internal organization interactions that significantly influence performance. This diagram provides “the big picture” and helps determine the business context, as well as helping uncover possible end to end processes and suggesting channels to market and third party involvement.

Process Flow Diagram:

The process flow diagram notation is indicated in figure 5.3. Sometimes the diagram is known as a swim-lane diagram as roles are diagrammed as parallel bands. Additionally decision points, shown as diamonds, may be included. Also note that a process may span more than swim-lane where the responsibility is held by more

than one role; such a process is known as a committee process.

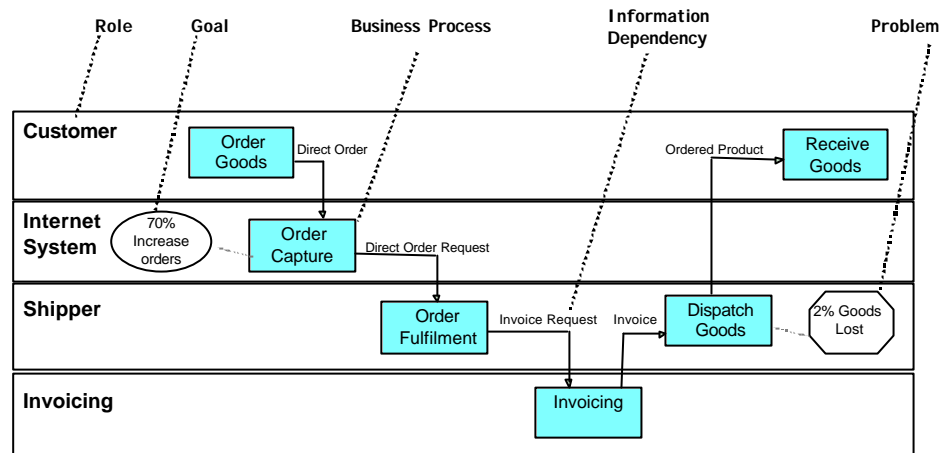


Figure 5.3 Process Flow Diagram

In one sense the process flow diagram is a lower level version of the organization flow diagram, with roles shown as swim-lanes instead of boxes. However, process flow diagrams are essentially prototypical and best employed to model scenarios, whereas the organization flow diagram provides an overview of the business.

Rule and Goal Hierarchy Diagrams:

Hierarchy diagrams such as rule hierarchies and goal hierarchies can be created, as shown in figure 5.4. More detailed notations are also available for modeling business rules (Veryard, 1999).

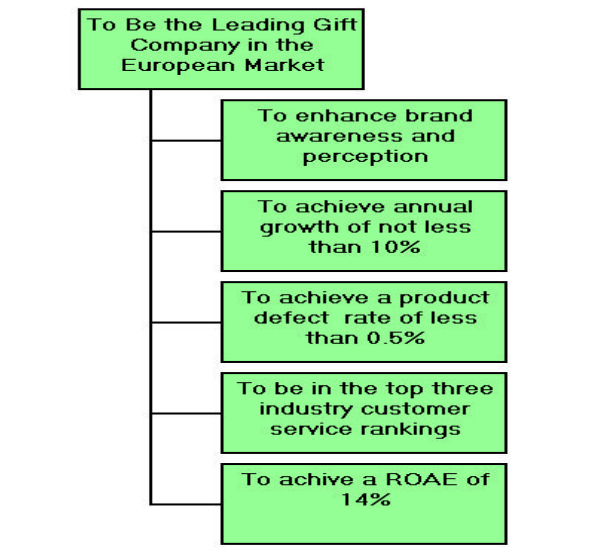


Figure 5.4 Goals hierarchy diagram

5.2.3 Business Modeling Tips and Hints

Here's a quick guide to business modeling, based around the activities of –business process planning itemized in 2.2.2:

- **Envision:** sketch out “to be” business concept and organization flow diagrams for the overall strategy and use these to scope the e-business improvement plan. Set out the overall business goals and problems.

- **Reflection:** if appropriate¹, build an “as is” organization flow diagram for the chosen scope. On subsequent iterations build process flow diagrams. Refine business goals and problems.
- **Conception:** refine the “to be” business concept and organization flow diagrams for the chosen scope. On subsequent iterations build process flow diagrams. Refine business goals and problems.
- **Organizing:** use the “to be” models to scope areas for improvement, identifying e-business solution needs for software projects (typically in terms of sets of customer processes) and scoping criteria for architecture planning (typically in terms of enabling or sustaining processes). Use the “as is” models to help define migration paths.
- **Iterate!**

In process flow analysis it is important not to over-analyze (or under-analyze) the threads and achieve a useful level of atomic granularity. “Time-slice” groups of activities by events that denote essential constraints imposed by the business, not by technology. That way, you stand a greater chance of finding atomic processes.

Business processes that are strategically important need to be understood and commonality identified across those business processes. Use sustaining processes and enabling processes to help identify areas of commonality.

Process flow diagrams are helpful for gaining business insight. However, e-business processes seldom support information flow in regulated chains. The effective e-

¹ If starting up a new .com business this does not apply. In other cases, it is often appropriate to build at least an “as is” organization flow diagram if only to have an understanding of your starting point for improvement. However, it is may sometimes be better to leave this until the “to be” model has been built in order to avoid prejudicing your thinking about where the business is going with preconceptions based on where it currently is.

business provides business capabilities that are configured in adaptable fashion to meet customer needs. Therefore do not worry too much about the detail and rigor of the flow – workflow technology can be used to help with those issues. Concentrate rather on achieving a good set of atomic processes and the roles involved. Add warning about detail, usefulness of atomic process.

Use process flow diagrams selectively to model the interesting, problematic or strategically important scenarios. Do not try and use these diagrams to “define the world” in huge detail – such a path leads to guaranteed failure..

Committee processes are normally an indication of complex behavior involving collaboration between roles. As such they provide an ideal input to collaboration modeling as described in 5.7.

Business rules may appear in various guises and at various places through the different models, for example as business goals on a process flow diagram or textual constraints attached to a business concept or business process. Where there are lots of potential rules consider modeling rules separately, for example using techniques such as rule hierarchies (Veryard, 1999).

5.3 BUSINESS TYPE MODELING

Business type modeling is a technique for modeling business concepts and their inter-relationships. It helps establish the information needs of a domain independently of any implementation. In turn this helps provide a sharper focus to the project. The business type model is used to drive the project architecture. Business type modeling is primarily a software requirement capture activity.

5.3.1 Business Type Modeling Concepts

A type is a pure specification construct that does not define how its instances are implemented. Another way of saying this is to say that a type is a classifier² whose instances have identity. Types may be related through association, inheritance or dependency.

There are three main categories of type, with notations as indicated below:

- **Specification Type:** defines data structure (attributes) but not operations
- **Interface:** The specification of a set of software services provided (and required) by any component supporting this interface; (one of our component forms; see chapter 3).
- **Component Specification:** The specification of a unit of software that describes the behavior of a set of objects, and defines a unit of implementation; (another one of our component forms; see chapter 3).

A business type is a specification type whose instances must be tracked by the business.
--

5.3.2 Business Type Modeling Notation

Notations for the three main categories of type are indicated in figure 5.5.

² A classifier is a general construct for defining data structure and behavior.

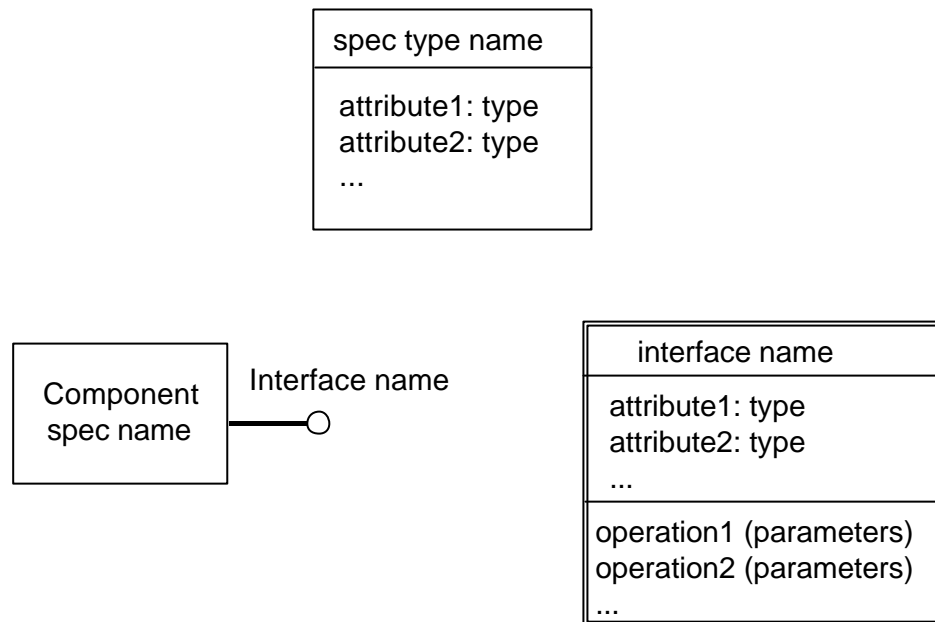
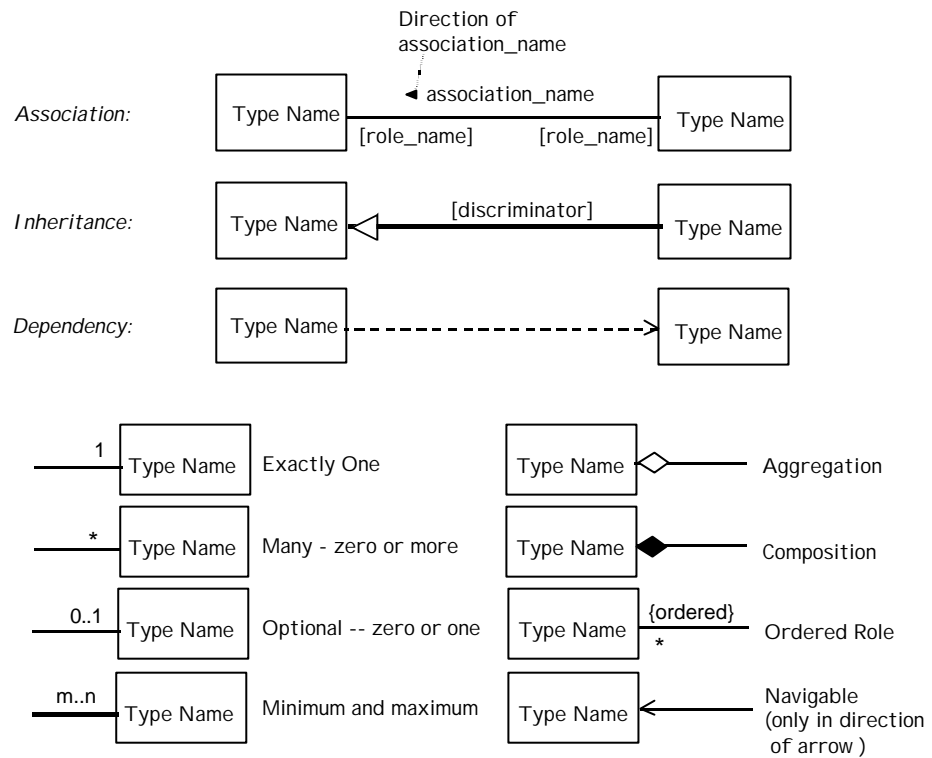
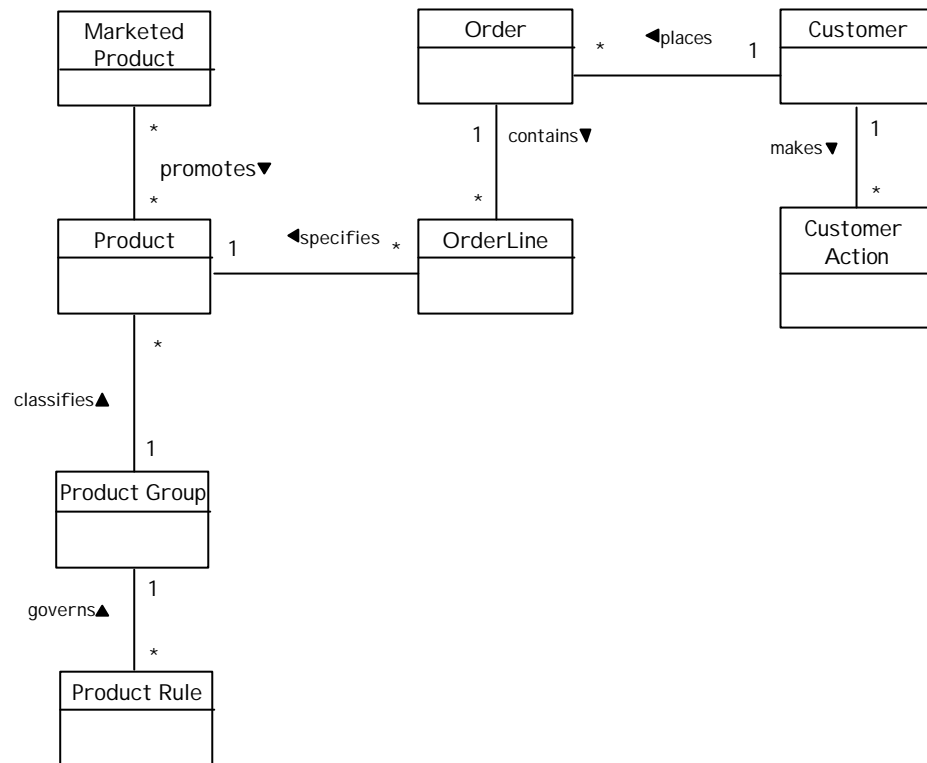


Figure 5.5 Type notations

A summary of main type modeling notations is shown in figure 5.6; the reader is referred to UML (UML, 1999) for full definitions.

**Figure 5.6** Type diagram notations**Business Type Diagram**

A business type diagram models business types and the associations between them. In figure 5.7 attributes are suppressed.

**Figure 5.7** Business Type Diagram

5.3.3 Business Type Modeling Tips and Hints

Here's a quick guide to business type modeling:

Identify and map key business concepts: think about the main subjects of

interest in the business domain.

- **Discover and scope candidate business types:** search use cases and requirements documents, consider information needs Add key attributes and responsibilities³ but do not define details yet.
- **Abstract further types:** look for roles and apply type model patterns (Fowler, 1997).
- **Refine in the light of the component architecture:** consider changing the type model⁴ to reflect common business capabilities or help with closure and reuse.
- **Define type details:** gradually add and define attributes, relationships, operations and invariants as they are discovered, only providing detailed definitions once the type model has stabilized.
- **Iterate!**

Generally discovery of types occurs earlier than invention or abstraction of types; “Key abstractions reflect the vocabulary of the problem domain and may either be discovered from the problem domain, or invented as part of the design.” (Booch 1994). This is sometimes referred to as the “discovery before abstraction” principle.

Knowledge of system requirements and use case descriptions is used to refine the "concept map", in particular by adding attributes, developing types and associations, and removing redundancy. Requirement Prototypes are used to help verify the use

³ A responsibility is a candidate business service. Responsibilities have key business relevance.

⁴ Changing the type model may involve combining or splitting, removing or introducing model items. Model items are types, responsibilities or attributes.

cases.

In applying use cases to help with the type model do not make the mistake of trying to factor out low-level operations such as Get and Set routines. These operations are available “by default” when you declare a type within an interface type model (see 5.7). Instead focus on responsibilities, on key business capabilities, that relate to the business process in the BPM. The responsibilities will later be assigned to specific interfaces as services and factored into constituent operations as appropriate.

Types must be clearly and concisely defined. It is often useful to use the form: “a <generalization> that <qualifies> noun”; for example, Customer: “A person who buys goods from our company”.

Roles are types that participate in collaborations and form coherent sets of responsibilities. An important feature of roles is that they separate behavior from its physical packaging and therefore help to identify reusable interfaces.

Roles may be modeled by specialization or generalization as illustrated in figures 5.8 and 5.9.

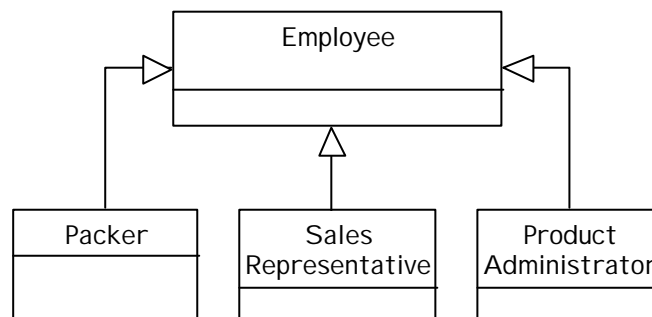
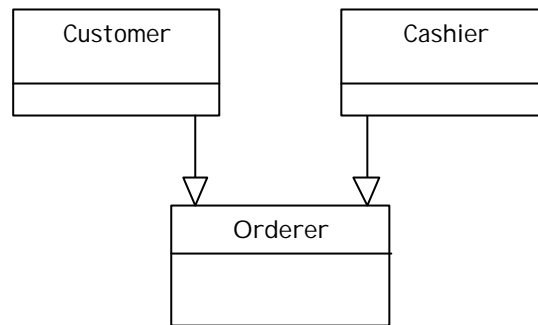
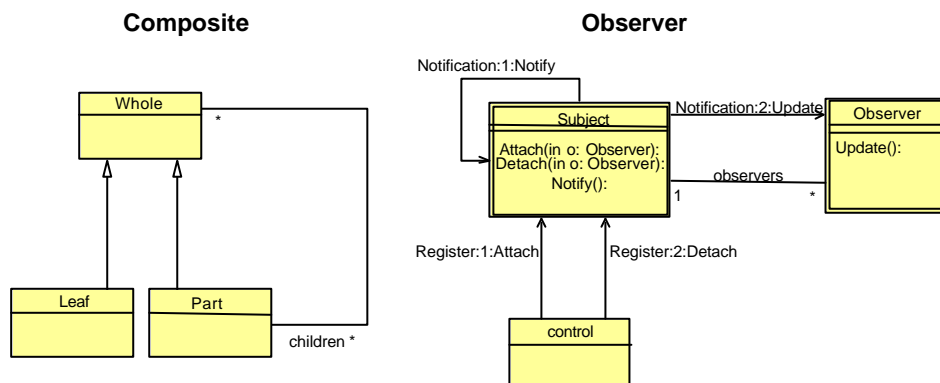


Figure 5.8 Role specialization

**Figure 5.9** Role generalization

Patterns are reusable analysis or design knowledge. A pattern describes a problem and its recommended solution. A good pattern is concise: small but packing a heavy punch. Two example patterns (Gamma, 1995) are shown in figure 5.10.

Analyst's Health Warning: Patterns are not complete solutions; you must understand the problem you're trying to solve first, then use the right pattern for the job!

**Figure 5.10** Examples of patterns using type diagrams

In one sense the business type model is similar to a business data model. Data abstraction, a well established technique from the world of data modeling, is an important part of business type modeling. However there are some subtle differences. For example, a business type model that differs from an entity relationship model in that repeating attributes, repeating structures of attributes, and conditional groups of attributes are not allocated to additional types. Each type in the model should be some type of thing from the business perspective. It should not be generalized or specialized or partitioned or normalized to suit an envisaged database design.

The business type model is not a logical database design.

5.4 USE CASE MODELING

Use case modeling is applied to help verify business requirements, involve business people and understand software scope. The technique must focus externally, on what the software is *for*, before getting into internal software mechanics. This is critical in CBD because use cases are used as one of the starting points for identification of business services and thence interfaces (separate from internal implementations). Use cases also provide a natural mechanism for identification of software increments in project planning and test cases in the testing phase. Use case modeling is primarily a software requirement capture activity.

5.4.1 Use Case Modeling Concepts

The concept of use case modeling is in essence very simple. Definitions follow:

Use Case: “A behaviorally related sequence of interactions performed by an actor in a dialogue with the system to provide some measurable value to the actor” (Jacobson, 1994). A use case represents a collection of scenarios. A use case is formed from the subset of actor - system interactions from an atomic process.

Actor: A role (or set of roles) that is played in relation to a software system; it could be a person, a group of persons, an organization unit, another software system or a piece of equipment. An actor can be external or internal to the business; for example

Customer or Credit Controller.

However, there are many pitfalls that await the unwary. We'll return to those shortly in the Tips and Hints section.

5.4.2 Use Case Modeling Notation

Use Case Diagrams

Use case diagrams are used to depict the main actors and use cases within the software scope as shown in figure 5.11. Lines between actors and use cases are associations which are assumed to be bi-directional unless specifically arrowed (as in the case of a system report, for example).

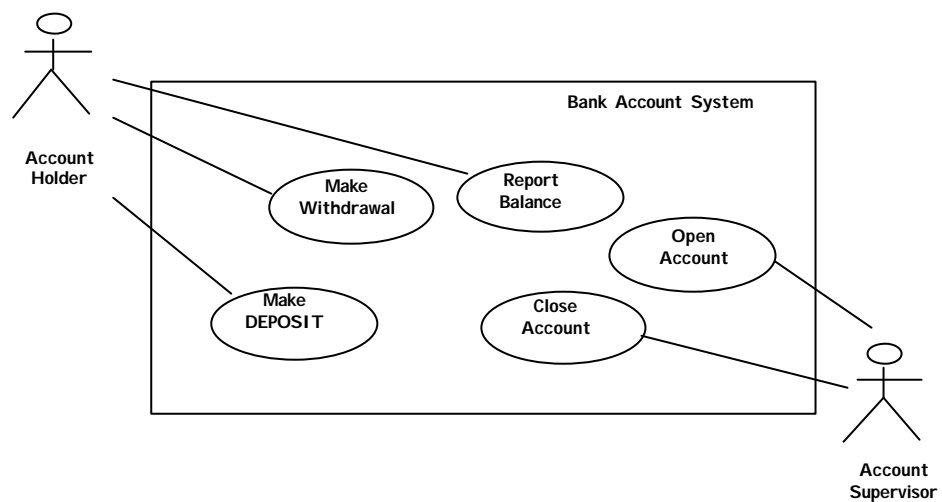


Figure 5.11 Use case diagram

Includes and extends relationships, which are stereotyped usage dependencies⁵, may also be shown on the use case diagram in figure 5.12. The includes relationship is used to abstract out commonly occurring use cases, that often correspond to business services. The extends relationship is used to separate out exception processing in order to avoid cluttering the main use case with unnecessary detail.

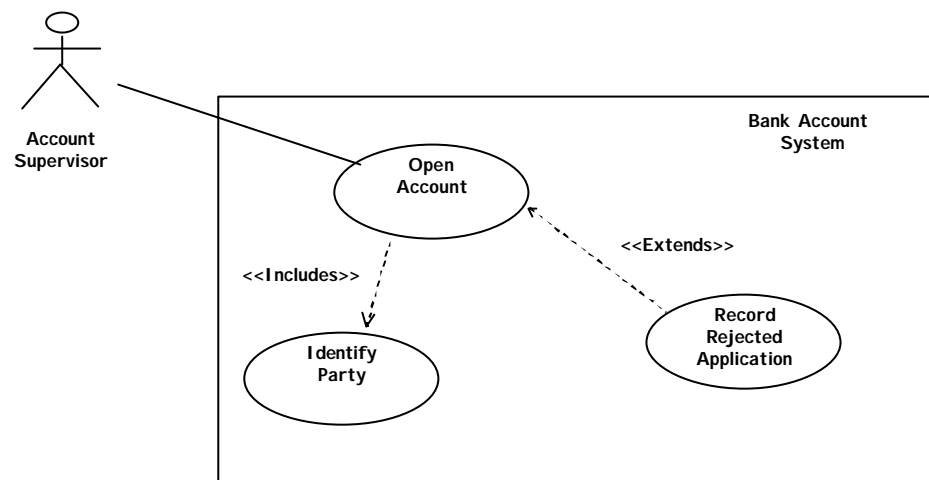


Figure 5.11 Use case diagram showing “includes” and “extends” relationships

Use Case Descriptions

A use case description effectively talks us through the normal sequence of actor-system interactions. It is vitally important to describe the use case in business language, from the initiating actor’s point of view.

Use case descriptions should be formatted. A simple example is as shown in figure 5.12. The intent of the use case is a concise description of its business purpose. The description itself should be partitioned into steps. Variations from the normal

⁵ . A separate generalization relationship is also available in UML.

sequence are usefully appended at the foot of the description.

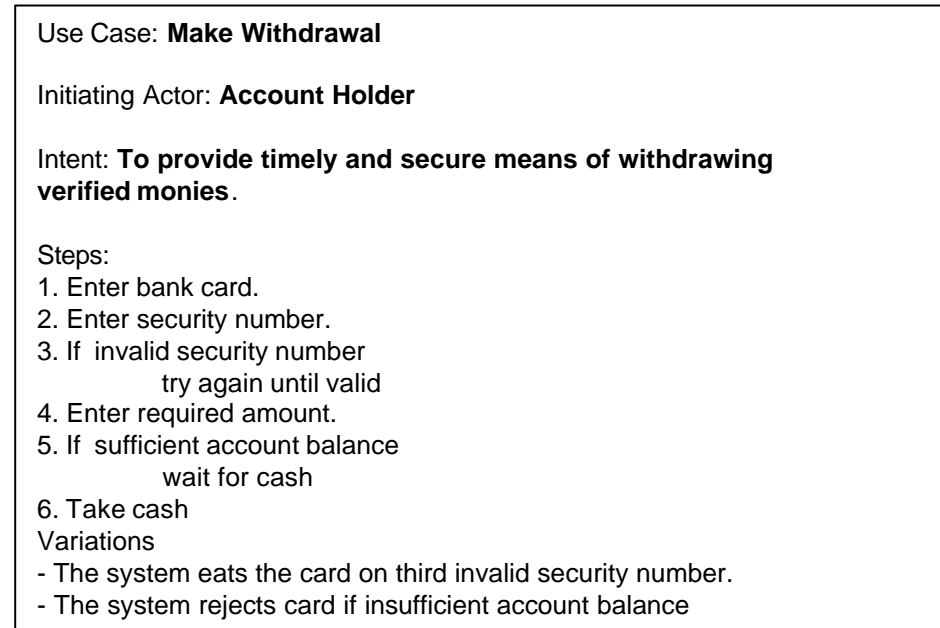


Figure 5.12 Use case description

Other information may also be included in the format including:

- **Preconditions and postconditions:** the states of the system before and after successful execution of the use case..
- **Non Functional Requirements:** implementation constraints, such as platform, as well as quality attributes such as response time and mean time to repair.
- **Business Rules:** business conditions governing the the user interface. Other business rules are attributed to appropriate model items; for example the rule “bay slots must not exceed 3 hours” is attributed as an invariant to the appropriate class (Bay Slot) in the class model.

- **Example screen layouts:** Illustrations of screens associated with the use case, including sample user data where available.

5.4.3 Use Case Modeling Tips and Hints

Here's a quick guide to use case modeling:

- **Identify business need:** ideally, start with atomic processes and roles from your BPM Focus on operational customer facing processes, not information management processes.
- **Discover use cases and actors:** map atomic processes and roles to use cases and actors respectively; usually one for each atomic process, but can be more. Draw use case diagrams.
- **Storyboard the use cases:** use JAD sessions to storyboard the use cases with your business people. Break the use cases into simple steps.
- **Factor out “includes” and “extends” use cases :** look for commonality of use cases or parts of use cases and model using the “includes” dependency. Similarly look for error paths and exception conditions and model using the “extends” dependency.
- **Describe each use case:** use JAD sessions to detail the use cases with your business people, refining to include business rules and nonfunctional requirements. Use prototyping techniques to help understand the use cases and include sample screens with the documentation.
- **Iterate!**

A common problem with use cases is lack of clear definition. If you asked 10 analysts to define the term “class” then you'd probably get very similar definitions. If you asked the same 10 to define “use case” chances are the replies would vary significantly. So clear definition is a basic worry. Use the basic definition above but

also apply guidelines on abstraction (coming up next).

Aim for useful levels of abstraction. A very high level abstraction is a major function like Sales. A very low level would be specific routines like Find Customer Address. We can apply some guidelines to help. A use case must:

- be a *self contained* unit with no intervening time delays imposed by the business.
- be initiated by a *single actor in a single place*, although it might result in output flows which are sent to other passive actors.
- Have a measurable business related *intent*.
- Leave the software in a *stable state*, it cannot be left half done.

Apply the 80:20 principle (Koch, 1997). It is important stick to the normal paths through a use case that constitute 80% of the functionality; an example might be “Schedule instructor to teach course”. Focussing unnecessarily on low-level detail may result in over-long and complicated use case descriptions; an example might be “Requested date is out of range”.

At the same time there is another side to the 80:20 principle. Often 80% of the value of a solution may result from its ability to innovate in the more unusual cases. It’s important therefore to also think of key business problems that could occur and how your solution will respond; an example might be “Find alternative venue (in the event that the usual one is unavailable”.

Use cases can lead to a functional decomposition analysis if not applied carefully. A top-down analysis results in function components, which are little more than main routines, controlling dumb data components – in other words components that are not responsible. Avoiding this trap means using use cases in their true spirit - focussing externally, on what the system is *for*, before getting into internal system mechanics. This is actually very critical with CBD as the goal is specification of

interfaces (separate from internal implementations).

Use cases are ideally suited to a particular subset of functional requirements: operational or event driven functionality. It's important not to forget other types of functional requirements that are less suited to use cases: for example information, maintenance, reporting and business policy requirements. Nonfunctional requirements, like security and performance, also need to be clearly stated. Unfortunately obsession with use cases often causes these other facets to be neglected.

Sometimes the obsession can lead projects totally astray as in the case of a major telecommunications company who had reached 400 use cases, and still rising, on one of their projects. On closer investigation it was revealed that this was a management information system involving flexible inquiries over a range of varied telephone network information. Thinking of the possible inquiries for such a system was like an open ended shopping list! And yet the team had not even started building a class diagram, which would have provided useful insights into the required information structure.

In medicine, we often read of latest miracle cures, usually in the form of a pill - for example a pill to cure obesity. The truth is that correct weight is a function of many things. Obviously pills can help, but we also need to consider metabolism, diet, exercise, environment, stress levels and so on. The same is true of good software practices - balance is needed. The point is that use cases have acquired something of silver-bullet status. Used selectively and correctly, like any tool, they will produce results but they are not a miracle cure for the problems of software development.

5.5 COMPONENT ARCHITECTURE MODELING

Component architecture modeling explores and defines the scope of components and their interfaces, the dependencies between components and interfaces, and also

dependencies concerning non-component *software units*⁶. It is important to appreciate that the models and diagrams shown in this section are only part of the component architecture; elements such as patterns, design guidelines, nonfunctional requirements and architecture policies (such as mechanisms for implementing cross-component associations in a way that maintains the integrity of the associations) are also included.

The *scope* of the component architecture may be a set of detailed requirements realting to a single project (project architecture) or set of high-level requirements for the enterprise as a whole, or more commonly, a significant sub-set of the enterprise (enterprise architecture).

5.5.1 Component Architecture Modeling Concepts

A component architecture diagram is a plan of software units⁷ and their dependencies.

Architecture modeling can be applied at different levels of refinement corresponding to the forms of component defined in chapter 3.

Specification Architecture:

⁶ Software units include software packages, legacy systems, special subroutines, and software assemblies which are not themselves formal components.

⁷ A software unit may be a component or non-component software such as a legacy system or software package.

- Interface
- Component Specification

Implementation Architecture:

- Component
- Component Implementation
- Component Module

Deployment Architecture:

- Installed Module
- Installed Component
- Component Object

In this book we make the pragmatic assumption that for the most part it is components, interfaces and their dependencies that are modeled. Having defined components and interfaces we need to look more closely at dependencies.

A dependency between software units⁸ means that a change to one software unit (the independent software unit) will affect the other software unit (the dependent software unit). It is worth noting that in all forms of architecture model, the dependency is actually ultimately dependent upon the component specification form, even though it may be diagrammed against the implementation. A dependency may be refined to state the specific interface upon which the component depends.

Where the dependent component (the consumer) is a component specification, then the dependency expresses a specification rule, which all implementations of that component specification must adhere to. Where the dependent component (the consumer) is a component implementation, then the dependency expresses an implementation design choice.

There are different types of dependency:

Usage: A model item requires the presence of another model item to function successfully.

Invocation: A model item calls an operation of another model item.

Instantiation: A model item instantiates another model item.

In deployment architecture there are two further concepts to note:

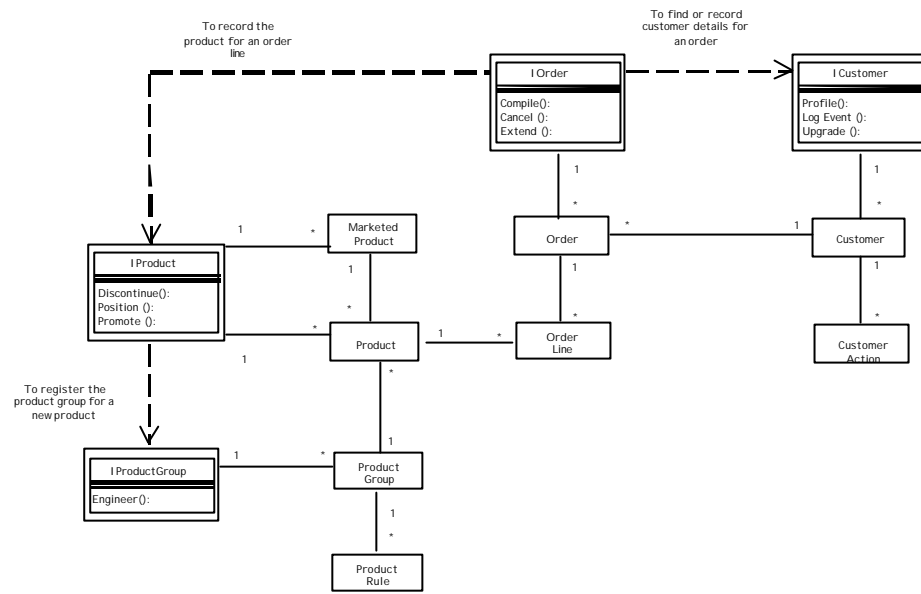
Nodes. Nodes are the processors of the technical infrastructure; for example, PCs, servers or mainframes.

Connections. Connections are communication paths between nodes. These can be labeled with the nature of the communication path. For example, the protocol used for communication.

5.5.2 Component Architecture Modeling Notation

Interface Responsibility Diagram.

Interface Responsibility Diagrams are primarily a thinking tool for identifying interfaces and planning specification architecture. They are simply type diagrams that include interfaces and the associations between interfaces and business types, and (optionally) dependencies between interfaces as shown in figure 5.13. Responsibilities of interfaces are noted along with the reasons for the dependencies.

**Figure 5.13** Interface responsibility diagram**Interface Dependency Diagrams.**

Where the interface responsibility diagram is large or complex, dependencies between interfaces are usefully shown separately on an interface dependency diagram as shown in figure 5.14.

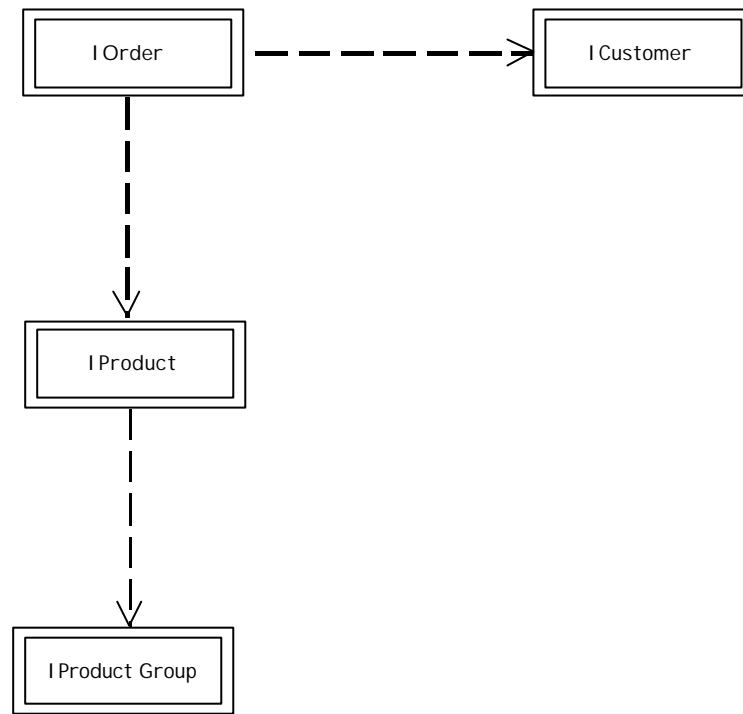
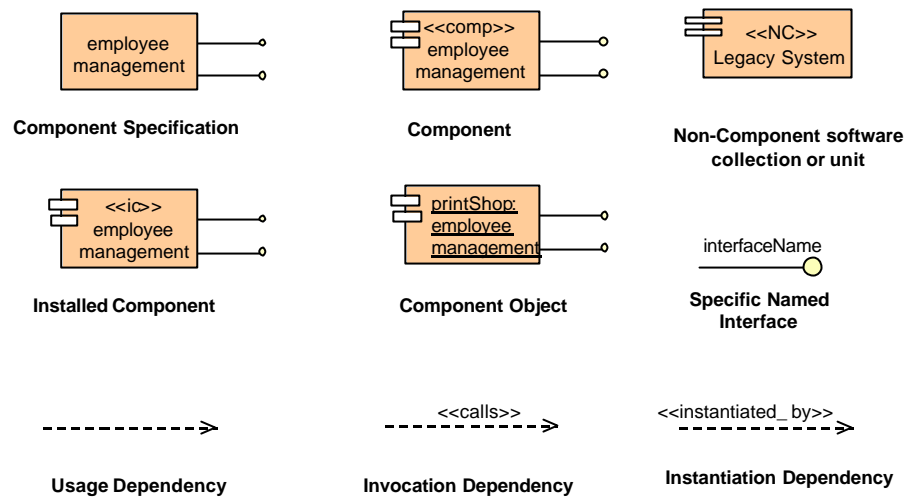


Figure 5.14 Interface dependency diagram

Architecture Diagrams.

Architecture diagrams are used in different ways at specification, implementation and deployment levels. Hybrid diagrams showing elements from different architectures are also possible. All types of architecture diagram are used to explore dependencies. The overall set of notations is shown in figure 5.15. Note the use of stereotypes to describe the component form and dependency relationship. It is possible to stereotype any component form; not all these are shown. By convention, the stereotypes of component specification and usage dependency are suppressed.

**Figure 5.15** Component architecture notations**Component Specification Architecture Diagram**

A component specification architecture diagram is used to model component specifications, interfaces and their inter-dependencies as shown in figure 5.16. As a first step each interface identified in the interface responsibility diagram has been allocated to a single component specification. Each specification dependency captures a specification rule requiring *all* implementations of the dependent component to use the independent component. It acts as a constraint on implementation and deployment architectures.

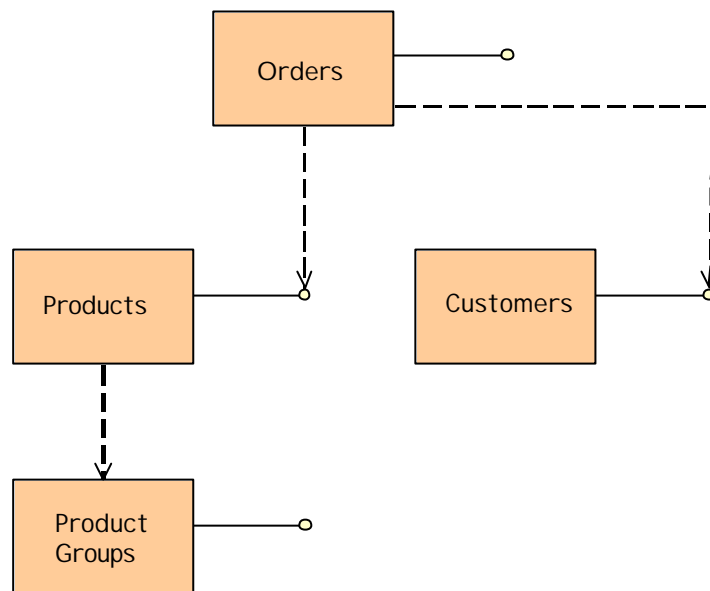


Figure 5.16 Component specification architecture diagram

Implementation architecture diagrams

Implementation Architecture Diagrams depict components, component specifications and interfaces, plus their dependencies; component implementation and component module forms may also be shown though this is quite rare in practice.

In the example shown in figure 5.17, we have decided that the Orders component uses a specific interface of the Products component. But in the case of the Products component, we decide it uses the Product Groups component without being specific about which interfaces, or even which implementation is to be consumed.

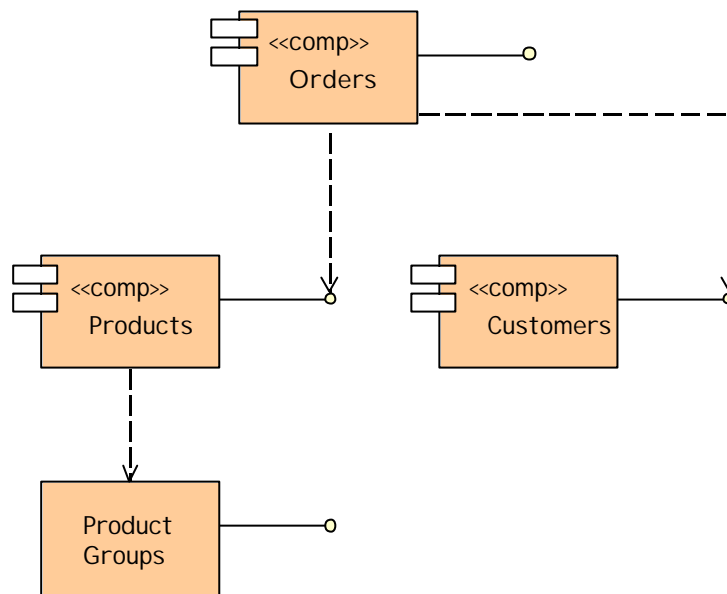


Figure 5.17 Component implementation architecture diagram

Deployment Architecture Diagrams

Deployment architecture diagrams are used to show allocation of software units to nodes, the connections between nodes, and the dependencies between software units, as illustrated in figure 5.18. Note that, as well as run-time component objects, installed components, installed modules, non-component modules, the databases or files managed by a particular node may also be shown. This architecture is only used selectively, to explore distribution issues.

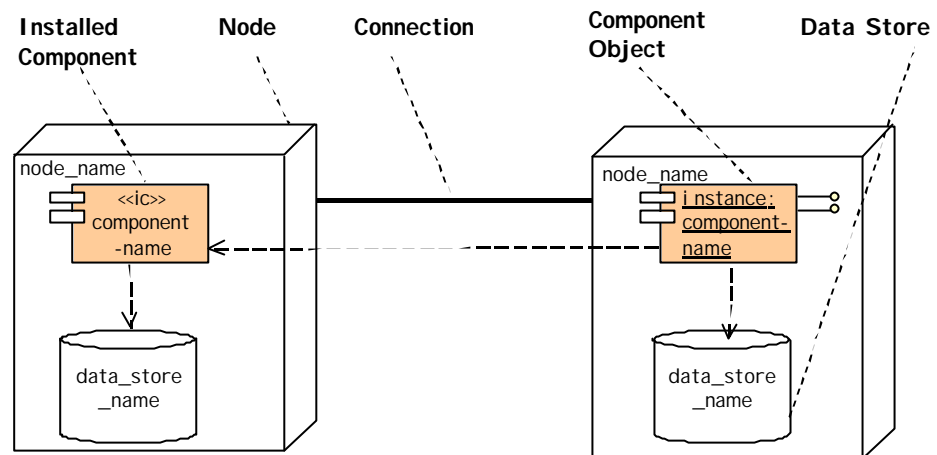


Figure 5.18 Component deployment architecture diagram

5.5.3 Component Architecture Modeling Hints and Tips

Here's a quick guide to component architecture modeling⁹:

- Identify and scope interfaces using the business type model, the use cases and the collaborations; see interface identification guidelines, below. Interface dependency modeling helps understand usage dependencies between interfaces,

⁹ The assumption here is that we'll use a single diagram to show both component specification and implementation dependencies.

providing a basis for a first-cut component architecture, by initially¹⁰ assuming one business component will be implemented for each interface.

- Adjust the initial set of interfaces in the light of the enterprise component architecture for consistency. Seek to reuse existing interfaces where appropriate.
- Check out “large” interfaces and review for splitting. Conversely check out “small” interfaces and review for combining. (See size guidelines below).
- Appraise partitioning of interfaces using partitioning guidelines below.
- Examine possible component implementations and implementation dependencies to help ensure our choice of interfaces is pragmatic and achievable. Review the dependencies to determine whether they should be specification level dependencies. Note that interface dependencies will conversely constrain the implementation.
- Review grouping of interfaces to component specifications.
- Check out cross-component associations making sure it’s possible to maintain the integrity of the associations.¹¹
- Additionally, if the implementation is highly distributed it may also be useful to model run-time dependencies between installed components or component objects. This again helps to ensure that the component specification architecture is achievable.
- Iterate

¹⁰ Later, interfaces may be grouped into component specifications.

¹¹ Some mechanisms for implementing cross-component associations, such as relationship management components or the call-back mechanisms may involve changes to the initial architecture, such adding new components or dependencies (ref Sterling document).

Interface identification guidelines:

- Types that have been assigned cohesive groups of responsibilities and play specific roles are often good candidate interfaces.
- Use cases and collaboration models are used alongside the business type model to expose roles and help identify interface responsibilities. They are also used to help identify associated existing systems and interfaces.
- Responsibilities that relate specifically to the needs of an actor initiating a use case or collaboration, often map to interfaces that offer local business capabilities using local business rules.
- Responsibilities that are common to several use cases or collaborations (for example, “includes” use cases) often map to interfaces that offer common business capabilities using generic business rules.

Interface “size” guidelines:

Aim for around 5-15 types per set of interfaces allocated to the same component specification as a rough rule of thumb

Interface partitioning guidelines:

The main theme is to partition to minimize undesirable dependencies and to

Appraise associations between types managed by different interfaces:

- Minimize associations between types managed by different interfaces. More than two indicates strong coupling and suggests it may be beneficial to merge the two interfaces or allocate the two interfaces to the same component specification.
- Group together types that have multiple associations and assign to the same

interface.

- Ensure that mandatory associations only apply “downward” from user to used component specifications; that is associations between types assigned to different interfaces that have been allocated to different component specifications.

Appraise usage dependencies between types managed by different interfaces:

- Adjust usage dependencies according to the position of the component in the architectural layers or “main sequence” (see chapter 3)
- Avoid circular dependencies
- Aim for cohesive interfaces, more by types commonly reused together, rather than simply functionally cohesive.
- Aim for “closure”: try to group together types that are changed together. Minimize the ripple effects of change; see guideline on change below..

Keep pattern aware, to take advantage of the best work of others. There are several important industry initiatives in this area, including OMG work through the business object task force (BOTF) as well as useful work on architectural patterns (Buschmann) and business related patterns (Fowler, 1997), not only design patterns which are now well-covered in the literature (Gamma et al 1995).

Two architectural patterns commonly crop up. A hierarchical architecture pattern is reflected in the use “application interfaces” which assumes responsibility for flow of control within collaborations. A network architecture pattern is reflected in a set of interfaces each of which has delegated responsibility for carrying out part of the collaborations. In real life an architecture may exhibit dual characteristics. However the question that often comes up is “which pattern is better?”

An object oriented purist answer is that the network pattern is better as responsibility is distributed evenly across interfaces; each interface collaborates just with those

interfaces that can help with a specific behavior. Application interfaces it is argued are structured programming (where control is centralized in control modules) in object oriented clothes. This leads to a structured functional architecture in which most interfaces become no more than dumb data transformers under a function blob.

A non-purist answer is that the hierarchical pattern is better as it is more flexible in catering for change, particularly changes to the ordering of tasks. Network patterns it is argued lead to convoluted and messy designs which are difficult to unpick and prone to hacking in the face of change.

In truth both patterns have their advantages and disadvantages that need to be weighed up according to individual circumstances. The most important criterion is to place responsibility for an action at the point of natural ownership for that action. Where several interfaces are involved in a use case and results need to be compared from several sources, there is often no natural point of ownership. Generally, a hierarchy pattern works best for such situations. However, this is not always the case. For example, suppose we want to know whether a product qualifies for a discount according to the rules of its product group. One approach might be to centralize invocation of all required interfaces from a product application interface, which also makes the decision on whether a discount applies. A second approach would be to delegate invocation of the product group interface localizing responsibility for making the decision on whether a discount is due to the product interface. In this example the application interface treats the other interfaces as pure data handlers; the network pattern is preferred because responsibility for making the decision is at the point of natural ownership for that action: that is, product is naturally responsible for working out whether it qualifies for a discount. It is after all the subject of that decision.

There are different ways of dealing with cyclic dependencies. Merging the components doesn't mean that the interfaces have to be merged. That depends on whether it is easier to redefine them as one interface or define the connection between the two interfaces. In figure 5.19, (ref to Sterling Architecture course material) the original specification is that I Orders Manager will keep track of which employee took each order. Orders Component therefore depends on I Employees Manager to get the details of the employee. Meanwhile Employees Component depends on I Orders Manager to get details what commission is due to an

employee.

In the first refined version, Orders component keeps track of which employee took each order through invoking I Employees Manager and supplies I Employees Manager with information that an employee took an order.

The advantage of this resolution is that Employees Component could now be implemented independently. The disadvantage is that there will be some duplication of data across components. There will also be the need to consider how to handle any referential integrity issues.

In the second example, both orders and employees are kept independent, and the connection between them is managed by a user of both.

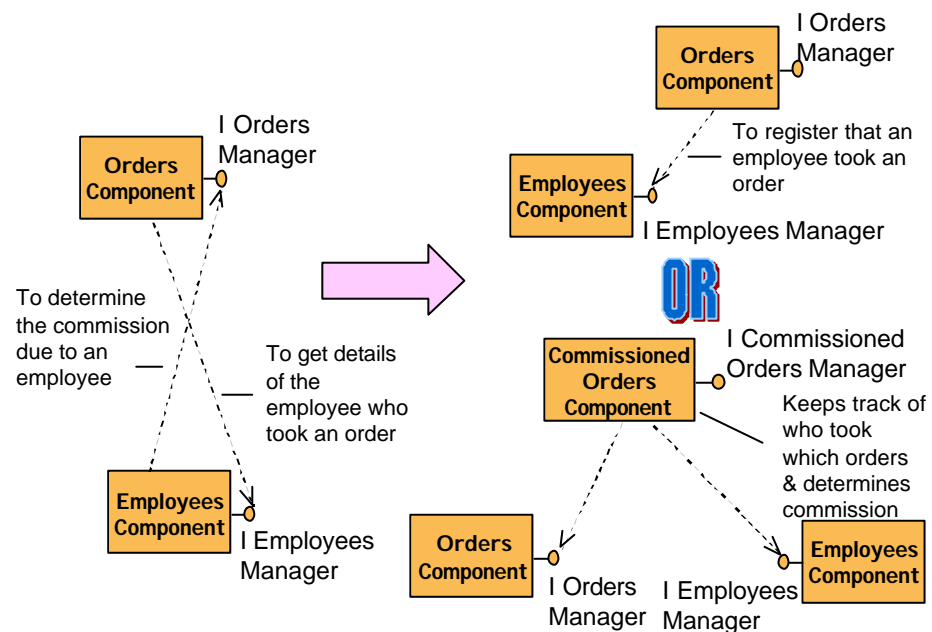


Figure 5.19 Resolving a cyclic dependency

A situation that commonly occurs is wrapping or adapting a legacy system or software package as a short-term makeshift component that can be substituted

software package, as a short-term makeshift component, that can be substituted with something better in future. Ideally, the makeshift software would offer the desired future interfaces, in which case we can define the dependency as a specification rule, and substitute the makeshift software with the real thing when it's available. This is the replaceability principle in practice. But if we're calling a "make-do" interface that we intend to improve in future, then that dependency should remain an implementation dependency.

Having constructed an initial component architecture, think of the things that could typically change as the organization migrates through the Internet spectrum. Play the changes through the architecture, testing that the interfaces "hold water". Look for the ripple effects of change. Try grouping together components that change together, or splitting a component that handles diverse types of change. Aim to localize the effects of change and for increasing reusability "downward" through the layers of the architecture.

5.6 COLLABORATION MODELING

Collaboration modeling¹² helps identify and understand behavior and the roles that types play in that behavior, through decreasing levels of abstraction. An important aspect of collaboration modeling is to assign responsibilities to interfaces, pinpointing the interfaces at the lower levels. A good distribution of responsibilities across interfaces is a keynote of CBD.

Collaboration modeling can be used at any level of abstraction from business process modeling to implementation. For example CRC cards (Wilkinson, 1995) are a form of collaboration modeling that is particularly useful in identification of business types and in allocating responsibilities to those types. Another example of the technique is in understanding allocation of behavior to interfaces as part of the

¹² It is important to understand that we are using collaboration modeling in the sense described by Catalysis (D'Souza and Wills), as an overall set of techniques. This can include the more restricted sense used in the UML (OMG, 1999), where the collaboration diagram is used to show the messaging between objects that is required to realize required behavior; we use the term "interaction diagram" for this specific case; see below.

specification activity, where a special type of collaboration diagram known as an interaction diagram shows collaboration between all interfaces participating in an action.

The basic idea is two fold:

- To factor behavior into appropriate interfaces.
- To group together collaborating types into components.

5.6.1 Collaboration Modeling Concepts

Collaboration: An abstraction of a set of related actions between typed objects playing defined roles.

Action: A discrete piece of behavior

Joint Action: An action that has not been localized to an interface (use cases are examples of joint actions).

Localized Action: An action that is a feature of an interface. A synonym of “interface operation”. The term tends to be used in contexts where the action is the result of the refinement of a joint action, which has now been "localized"— that is, assigned to a type.

Role: A type that represents a set of responsibilities within the context of a collaboration or use case. Roles are often refined to interfaces.

Refinement: The activity of analyzing a model (an abstraction) and creating a more detailed model (a refinement). The refinement is said to conform to the abstraction. Refinement is the opposite of abstraction.

Abstraction: The activity of synthesizing a detailed model (a refinement) into a less detailed model (an abstraction) by omitting details not relevant to the details of the abstraction. Abstraction is the opposite of refinement.

5.6.2 Collaboration Modeling Notation

Explain difference with UML.

Context diagrams

Context diagrams are collaboration diagrams that are used to focus on the scope of a joint action. As such they may be used at varying levels of detail from business process to discrete pieces of behavior. The example illustrated in figure 5.20 models the context of the joint action Take Order; note that Take Order is actually a use case because it represents a set of interactions between an actor (the role Orderer) and other roles which are likely to be automated.

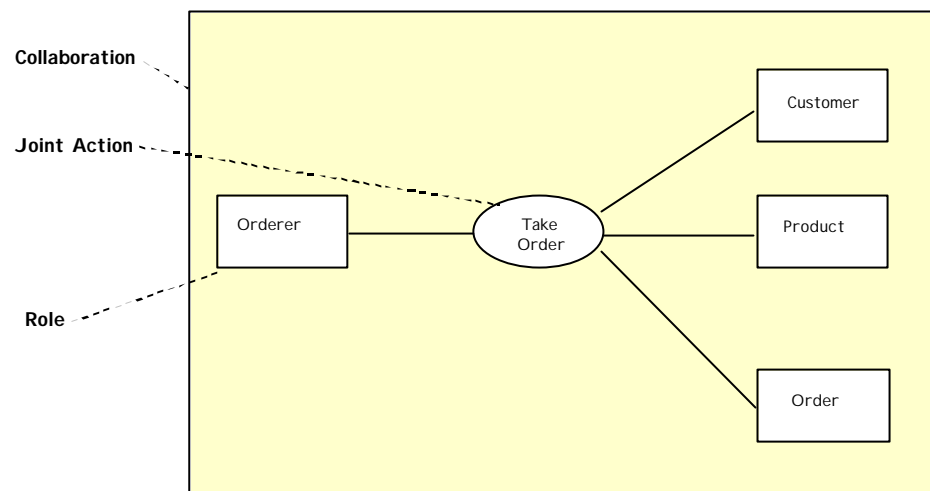


Figure 5.20 Context diagram

The collaboration is refined into a set of further joint actions and roles as shown in figure 5.21. Figure 5.21 is said to conform to figure 5.20.

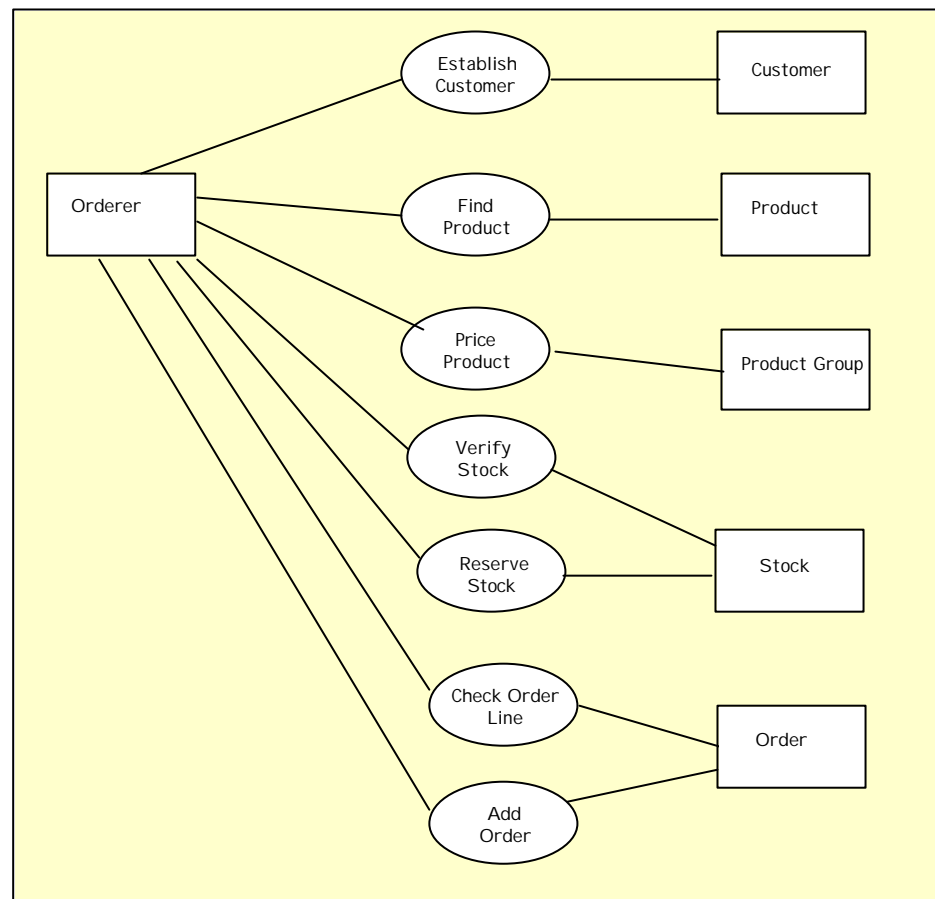
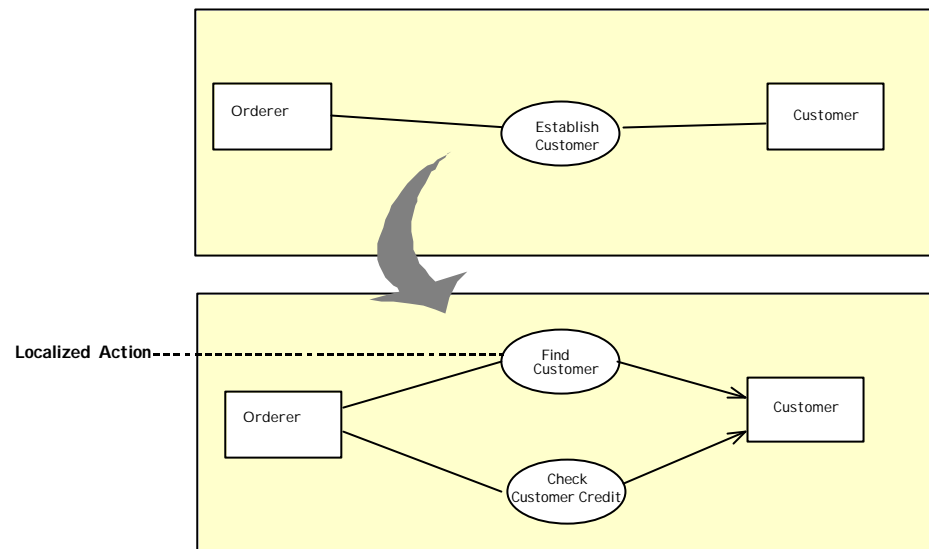
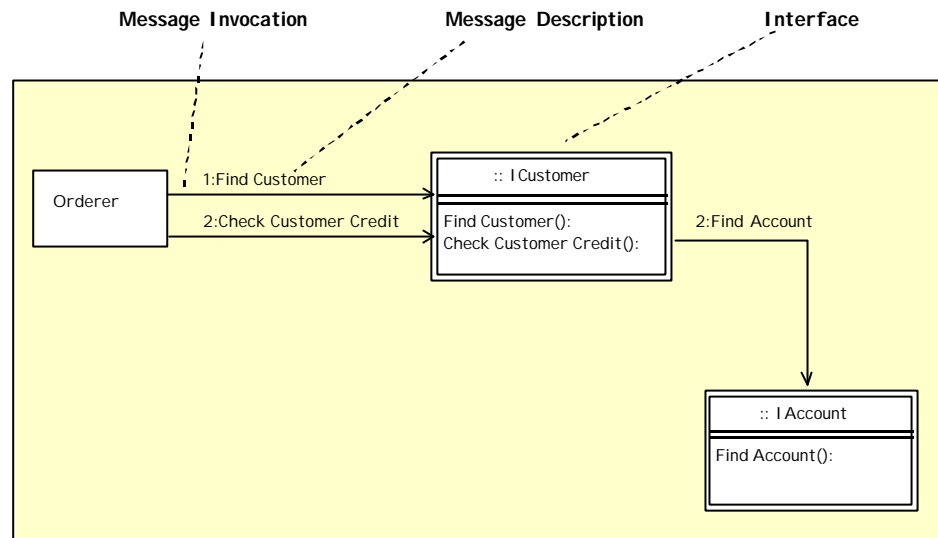


Figure 5.21 Refined context diagram

Once a reasonable level of detail has been achieved, actions may be localized to their responsible roles. Establish Customer is refined as shown in figure 5.22 with actions now localized to Customer.

**Figure 5.22** Localizing actions**Interaction diagrams**

Interaction diagrams help to assign responsibilities to interfaces. An interaction diagram is shown in figure 5.23. Note that the localized actions Find Customer and Check Customer Credit are now assigned as operations on the interface *Icustomer*, refined from the type *Customer*. Note it can be useful to model both actions and interfaces on the same diagram (a hybrid between context and interaction diagrams) where only a sub-set of actions have been localized to interfaces.

**Figure 5.23** Interaction diagram

The message notation is, of necessity, more complex if you wish to specify the sequence, choice and repetition involved in a complete description of a collaboration. Message description can be structured as described in (UML, 1999) as follows:

- a) A sequence number using Dewey decimal notation. The triggering message has no number. The numbers are separated by dots to represent nested procedure calling sequence. For example message 2.1 is part of the procedure invoked by message 2 and follows message 2 within that procedure. Note that “*” indicates a potentially iterative message. Letters can be appended to indicate concurrency of messages; for example 2.1a and 2.1b. Also a question mark, optionally followed by a Boolean expression can be used to model conditionality.
- b) Return value name(s); followed by assignment sign (`:=`). This is optional.
- c) The name of the message which may be an event or an operation.

d) Arguments; contained in brackets. This is optional.

Incidentally, if this level of detail is really needed then it is likely that a sequence diagram is a much more appropriate tool for tackling the problem, as described below.

Sequence diagrams

Sequence diagrams help to explore the detailed mechanics of interaction between typed objects and are commonly used in object oriented design. However they may also be used to assist in architecture and specification modeling. Time runs downward through the diagram. A structured language description is usefully appended to the left margin of the diagram to describe sequence, choice and repetition. This helps to avoid cluttering the diagram itself. The notation is shown in figure 5.24.

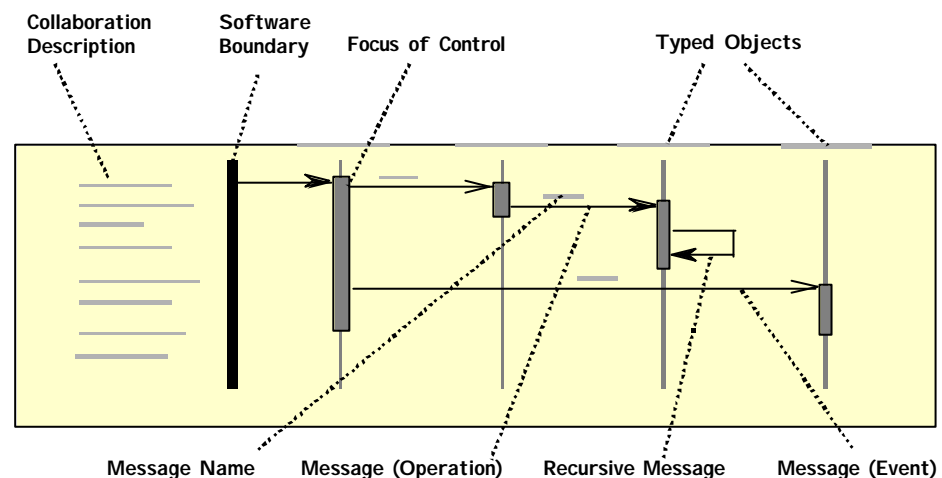
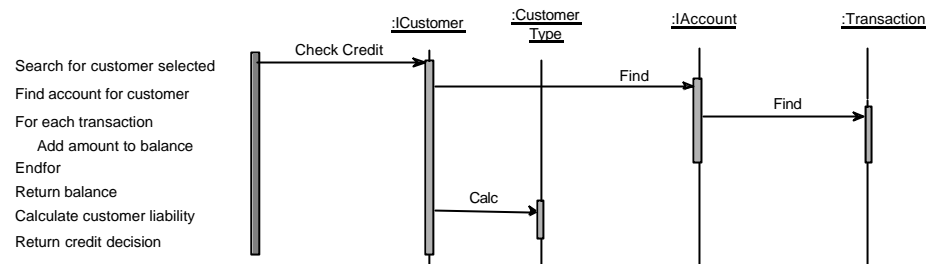


Figure 5.24 Sequence diagram notation

In the sequence diagram shown in figure 5.25 the interaction described previously has been refined to show the message invocations occurring “behind” the interfaces so to speak.

**Figure 5.25** Sequence diagram

5.6.3 Collaboration Modeling Hints and Tips

Here's a quick guide to collaboration modeling¹³:

- Identify joint actions at an appropriate level of abstraction (see next guideline).
- Use context diagrams to focus on the scopes of joint actions.
- Refine into further joint actions and roles.
- Once a reasonable level of detail has been achieved, localize the joint actions their responsible types (or "roles").
- Look for existing interfaces that can supply the required behavior.
- Use interaction diagrams to help to assign responsibilities to interfaces; localized actions are now assigned as operations on the interfaces.
- Apply viewpoint analysis (see below) to the interfaces, depending on level of

¹³ actually if you look at 5.7.2 you'll see that we're generalizing the steps we followed in that section

desired reusability.

- Use sequence diagrams¹⁴ to help to explore the detailed mechanics of interaction between interfaces.
- Iterate!

Level of abstraction:

Collaboration modeling is a very widely applicable technique and can be used within different technique families as shown in the table 5.1. Note that collaboration modeling can be applied as part of business modeling, as a precursor or alternative to use case modeling. On the other hand it can applied at a low-level in the analysis of a single use case.

One of the most illuminating uses of collaboration modeling however is at architectural level in the analysis of context and in the design of interface interactions.

Table 5.1 Collaboration Abstraction Levels

Level of Abstraction	Action	Technique Family
High	Business Process	Business Process Modeling
Mid	Use Case	Requirements Modeling

¹⁴ if appropriate also use these diagrams to model collaboration between typed objects as part of an internal object oriented design for the component.

	Use Case Step	
Mid	All levels	Architecture (Context Model)
Mid-Low	Operation	Interface Specification (Interaction Diagram)
Low	Operation	Implementation (Sequence Diagram)

Viewpoint Analysis:

Context modeling is also very usefully applied in helping to evaluate the reusability of an interface and to engineer generic interfaces. The idea is to examine the interface from as many viewpoints as reasonable, given the amount of reusability that is being sought; hence the term “viewpoint analysis”.

If an interface plays a specific role in relation to that other types, it's always worth asking the question, “Might this interface also be used in other contexts?” For example, we might have identified an interface to provide tickets for theatre-goers using the Internet. Could the interface be used similar contexts, for example sports events, TV or cinema, with the minimum of extension or additions?

A danger of collaboration model is to use it in the spirit of function decomposition. Remember to look for existing interfaces that can serve the collaboration and to seek collaborations that are reusable in different contexts.

Use cases are a special case of the joint action concept. Providing the granularity of use case is appropriate (see guideline in 5.4.3), the set of use cases can provide a good starting point for collaboration modeling.

Interaction diagrams are most appropriate for modeling specific threads or scenarios. It's generally best keep the diagram simple and avoid cluttering with the

scenarios. It's generally best keep the diagram simple and avoid cluttering with the extra notation that needed for modeling complete collaborations. If timing, sequencing and conditionality are important then use sequence diagrams, which are often more useful for modeling complete collaborations.

5.7 INTERFACE SPECIFICATION MODELING

Interface specification modeling is used to rigorously define interfaces. Interface type models are used for this purpose. An interface type model declares information that the interface must remember. In a simple case this may take the form of a simple list of attributes. More commonly, the interface type model consists of a collection of types, attributes and associations, attributes depicted as a type diagram together with a catalog of operations and invariants.

The relevance of interface specification modeling is twofold: the interface type model provides consumers of the interface with an “information and services” catalog. Equally it acts as a compliance document for suppliers seeking to provide implementations for the interface.

5.7.1 Interface Specification Modeling Concepts

An interface specification must describe the operations offered by the interface, the invariants that constrain the interface and the information that the interface must remember.

A type model is used to specify the information that the interface must remember.

State charts (UML,) may be used to specify state related behavior of an interface.

Precise operation specification is achieved by writing a set of pre- and post-conditions for each operation.

Precondition: A precondition is an assertion that must be true prior to execution of

an operation. A pre-condition can be empty, which means it is always true.

Postcondition: A postcondition is an assertion that must be true following an execution, given that the precondition is true. A post-condition always has a corresponding pre-condition.

Pre-conditions and post-conditions are conditional expressions whose value is either true or false. As such, they may contain any number of sub-conditions connected by Boolean operators. An operation may have one or more pre- and post-condition pairs. If at execution no pre-condition is true, the effect of the operation is undefined.

It is up to the initiator to ensure that at least one pre-condition holds true on invoking an action. The action only guarantees effects if a pre-condition holds. This is the detail of the "contract" between the consumer and service provider.

Invariant: An invariant is a condition concerning the elements of a type model, which must always hold true while no action is in progress. Examples:

- A Customer can hold either a Gold Account or a Silver Account but not both.
- If Account Balance < Overdraft Limit, then no withdrawals allowed.
- The sum of (order item price x order item quantity) holds true for the order's total value across all operations
- Order item quantity range 0 to 20
- All order items for an order < 10,000

Some invariants may be specified graphically, in type interface model (e.g. multiplicity).

5.7.2 Interface Specification Modeling Notation

The notation is illustrated in figure 5.26.

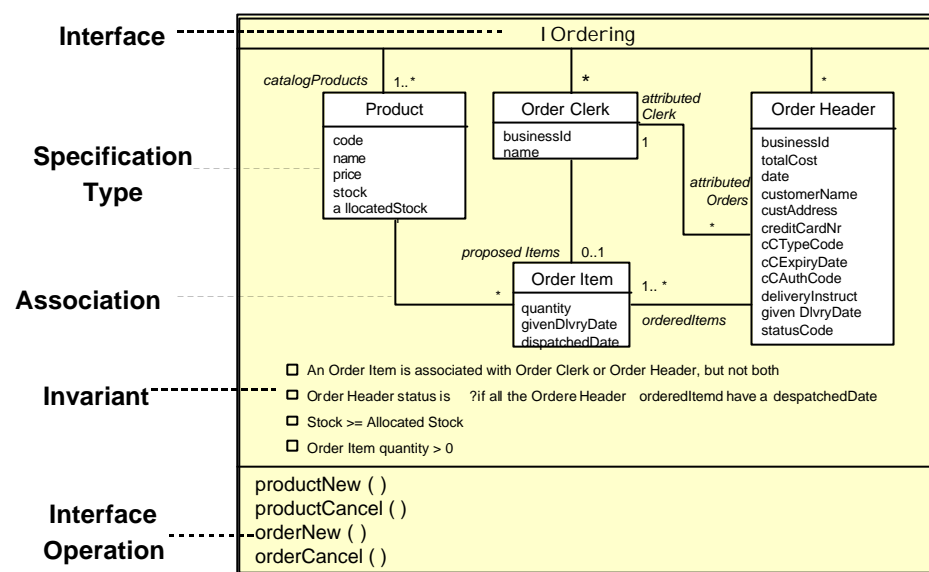


Figure 5.26 Interface type diagram notation

Each operation is specified using pre and postconditions as illustrated in figure 5.27. Note that a formal notation such as OCL can be used to define pre- and post-conditions.

```
productNew(in n:string, in busID:number, return r:Result)

pre  NO Product q EXISTS IN products WITH q.businessID = busID
post Product p is CREATED IN products WITH
      p.instanceID = some unique value
      p.businessID = busID
      p.name = n

pre  Product p EXISTS IN products WITH p.businessID=busID
post r.errorcode=12
```

Figure 5.27 Preconditions and postconditions

5.7.3 Interface Specification Modeling Hints and Tips

Here's a quick guide to interface specification modeling:

- Use the interface responsibility model (see section X) to help identify types that must be remembered by each interface.
- Declare these types should be declared on the interface type model.
- Declare interface responsibilities in the form of a first cut list of operations with objectives.
- Appraise interface for overall quality (see below)
- Evaluate interfaces for reusability (see below)
- Assess and record the mechanism to be used for implementing cross-component associations in the interface definition.
- Specify invariants.
- Use state charts to model state related behavior, if appropriate.

- Specify responsibilities as services using pre and postconditions.
- Specify suggested implementation mechanism (outsource, internal design etc.)
- Consider different presentation styles before publishing the interface.

Here are some guidelines for checking out interface quality:

- Good interfaces tend to do one thing well
- Not necessarily something simple
- Single, coherent set of responsibilities
- Not a collection of dissimilar behaviors

Interfaces often correspond to roles

The reusability of the interface should be evaluated, for example using viewpoint analysis (see above 5.7.3), before specifying operations and invariants. Here are some questions to ask in checking out reusability of interfaces:

- Could another team use this interface the way we specified it?
- Would significant changes be required?
- What is likely to change in the business?
- How could this interface be changed to accommodate likely changes?

Would significant rework be required?

Do not try to specify inquiry or CRUD operations in the interface type model. These operations are implicitly available on all types declared in the interface type model. Inquiries involving multiple types are implicitly supported by navigating

associations declared in the interface type model. This greatly assists in streamlining the specification.

Be willing to customize your interface specifications for use by different audiences:

- customers browsing for suitable components
- consumer developers
- supplier developers

In specifying pre and postconditions it is advisable to focus on "normal, successful" processing first, so as not to be overwhelmed with too much detail. Often these will correspond to the basic courses of use cases. As with the use cases, we can work on various alternative outcomes once we have understood the normal cases.

Invariants should be used to simplify the pre and postconditions. It is important to check that each variable to which the pre and postcondition pairs refers is either a parameter of the operation, or an element of the interface type model.

5.8 SUMMARY

Let's summarize by considering how the different techniques work together.

5.8.1 A Techniques Roadmap

The overall relationship between the different techniques is depicted in figure 5.28. The reader should note that the diagram is illustrate and not meant to imply a rigid sequence.

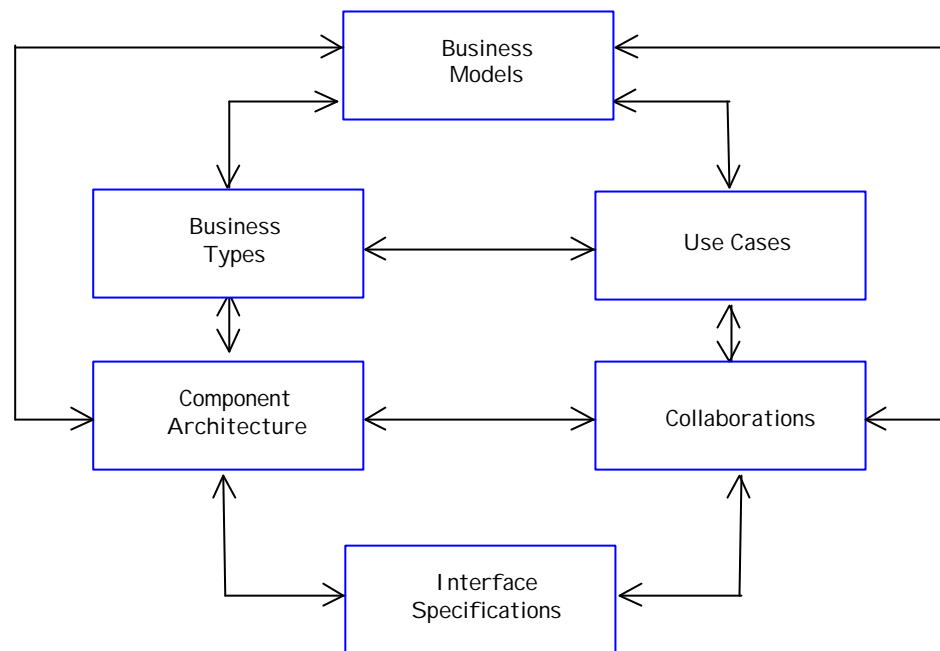


Figure 5.28 Techniques Route-Map

The business models provide a context for modeling business types and use cases by setting business goals and objectives in order to help provide the business case. Business process flow diagrams help to identify use cases. Business concept models provide an initial pathway toward developing business types. The business case helps scope the component architecture and set a context for collaboration modeling. Conversely information unearthed as a result of any of the four software related techniques may result in a revision to the business models.

Modeling of use cases and business types is two-way. Both help in understanding software requirements. The analyst should be confident that the use cases reflect functional requirements and that the business types can support the use cases. At the same time the business type model should reflect requirements not covered by use cases, such as business policy, business rules and information requirements.

Business type modeling drives both collaboration modeling and architecture

modeling (and vice-versa). Interface types are initially declared by consideration of the business type model. An interface type should manage cohesive sets of business types. Dependencies between the interface types and associations on the business type model must be catered for by the component architecture. Business types are declared on collaboration models, which help to further refine understanding of the business types.

Use case modeling drives collaboration modeling (and vice-versa). Use cases are factored into lower level actions and the required types, including interfaces, declared on collaboration models. (Note: In fact it is possible to proceed direct to the collaborations without prior use case modeling). Thinking through collaborations in this way may also cause use cases to be adjusted.

Collaboration modeling and use case modeling drive architecture modeling (and vice-versa). The architecture provides interfaces that are declared on the collaboration models. Conversely the collaborations (and use cases) are “played through” and test out the architecture, causing adjustment of interface responsibilities, questioning of dependencies and new interfaces and dependencies to be identified.

Collaboration modeling drives interface type modeling (and vice-versa). Operations required to support the collaborations and attributes and types that need to be remembered are declared on the interface type model.

Architecture modeling drives interface type modeling (and vice-versa). The interface type models must support dependencies identified in the architecture. Conversely, thinking through interfaces in this way may also cause the architecture to be adjusted.

5.9 REFERENCES

Allen, P., Abuse of the Use Case Miracle Cure, Cutter Email Advisor, Cutter Consortium, Jan 1999

Consortium, Jan 1999

Allen, P. and Frost, S., *Component-Based Development for Enterprise Systems: Applying The SELECT Perspective*, Cambridge University Press - SIGS Publications, 1998

Booch, G., *Object Oriented Analysis and Design with Applications*, 2nd Ed., Benjamin Cummins, 1993.

Conallen, J., *Building Web Applications with UML*, Addison Wesley Longman, 2000

Dodd, J., et al., *Advisor 2.4*, Sterling Software, 1999

Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.

Gottesdeiner, E., *Turning Rules into Requirements*, Application Development Trends, Vol 6 No 7, July 1999.

Harmon, P., *Building Systems that Integrate COM, CORBA, and JAVA*, *Component Development Strategies*, Cutter Information Corp, October, 1999

Koch, R., *The 80:20 principle*, Nicholas Brealey, 1997

Martin, J., *Information Engineering Vols 1-3*, Prentice Hall, Englewood Cliffs, 1989.

McMenamin, S and Palmer, J., *Essential Systems Analysis*, Prentice Hall (Yourdon Press), 1984.

Page-Jones, M. *The Practical Guide to Structured Systems Design*, (2nd ed), Prentice Hall (Yourdon Press), 1988.

Penker, M. and Eriksson, H., *Business Modeling with UML: Business Patterns at*

Work, Wiley, 2000

Rummler, G.A., and Brache, A.P., *Improving Performance*, Jossey-Bass, 1995.

Veryard, R., Notes on Business Rules,
<http://www.users.globalnet.co.uk/~rxv/kmoi/rulemodelling.htm>, 1999.

Wilkinson, N.M., *Using CRC Cards: An Informal Approach to Object Oriented Development*, SIGS Books, 1995.

Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1979.
