UNIT-I

INTRODUCTION

Unit I Syllabys:

Introduction : Introduction to OOAD; typical activities / workflows / disciplines in OOAD, Introduction to iterative development and the Unified Process, Introduction to UML; mapping disciplines to UML artifacts, Introduction to Design Patterns – goals of a good design, Introducing a case study & MVC architecture.

What Is Analysis and Design?

Analysis emphasizes an *investigation* of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used?
"Analysis" is a broad term, best qualified, as in *requirements analysis* (an investigation of the requirements) or *object analysis* (an investigation of the domain objects).

Design emphasizes a *conceptual solution* that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented. As with analysis, the term is best qualified, as in *object design* or *database design.*

Analysis and design have been summarized in the phase *do the right thing (analysis), and do the thing right (design).*

What Is Object-Oriented Analysis and Design?

During **object-oriented analysis,** there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book, Library,* and *Patron.*

During **object-oriented design,** there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChap-ter* method (see Figure 1.2). Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.
Figure 1.2 Object-orientation emphasizes representation of objects.

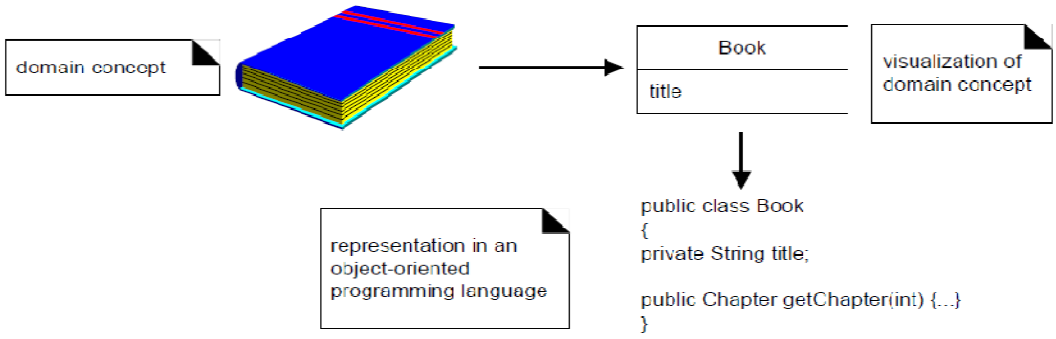


Figure 1.2 Object-orientation emphasizes representation of objects.

OOAD Concepts in UML

The **Unified Modeling Language(UML)** is **object-oriented modeling language** and used in designing object-oriented software applications. The applications based on the **object-oriented technologies** recommended by the **Object Management Group** (OMG)

We discuss the General concepts and methods of **object-oriented analysis and design** (**OOAD**)and how those were applied to UML itself.

Object-Oriented Design(OOD) is a software development approach to design and implement software system as a collection of interacting stateful objects with specified structure and behavior.

We will take a look at some of OOD concepts that seem relevant to the UML:

- class and object,
- message, operation, method,
- encapsulation,
- abstraction,
- inheritance,
- polymorphism.

Class and Object in UML

UML **class** is a classifier which describes a set of objects that share the same

- features,
- constraints,
- **semantics (meaning)**.

Class may be modeled as being **active**, meaning that an instance of the class has some autonomous behavior.

In all versions of UML from UML 1.x to UML 2.5, the essence of **object** is the same:

Object is an instance of a class.

An entity with a well defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. UML 2.5 describes object as

An object is an individual [thing] with a state and relationships to other objects. The state of an object identifies the values for that object of properties of the classifier of the object.

Message

Message concept is probably one of the most confusing in OOAD, especially for the software developers familiar with modern messaging systems and APIs such as the Java Message Service (JMS) or Microsoft Message Queuing (MSMQ), which allow separate, uncoupled applications or components to reliably communicate asynchronously. Message concept in OOAD and UML is quite a different thing.

Messages in Smalltalk-80 represent two-way communications between the objects of the system. Note, that "in Smalltalk everything is an object", including primitive values and classes. A message requests an operation from the receiver. Object selector and arguments transmit information to the receiver about what type of response to make. It is up to the

receiver to decide how to respond to the message. The receiver returns an object back that becomes the value of the message expression.

Let's consider simple Smalltalk example of a binary message to request arithmetic operation:

3 + 4

In this case, object '3' is the **receiver** of the **message** '+ 4'. The message contains **selector** '+' and **argument** '4'. Selector of the message is a name (or symbol) for the type of interaction required from the receiver. The receiver of the message ('3') returns back an object ('7') that becomes the value of the message expression.

If a message expression includes an assignment prefix, the object returned by the receiver will become the new object referred to by the variable. Even if no information needs to be communicated back to the sender, a receiver always returns a value for the message expression. Returning a value indicates that the response to the message is complete. For example,

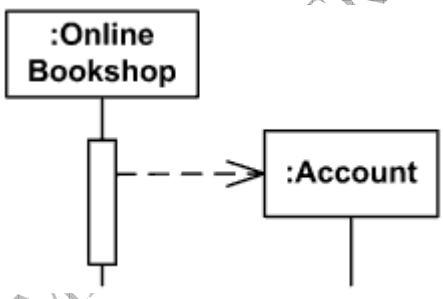
sum ← 3 + 4

makes 7 to be the new value of the variable sum.

Message in UML

Messages are intrinsic elements of UML interaction diagrams. A message defines a specific kind of communication between lifelines of an interaction. A communication can be, for example, invoking an operation, replying back, creating or destroying an instance, raising a signal. It also specifies the sender and the receiver of the message.

Create message is shown as a dashed line with open arrowhead, and pointing to the created lifeline's head.



Online Bookshop creates Account.

Note, that this weird convention to send a message to a nonexisting object to create itself is used both in UML 1.x and 2.x. As we saw above, in Smalltalk-80 new objects are created by sending messages to **classes**, with instance of the class created and returned back. So one way to interpret UML create message notation is probably as a shortcut for these actions.

Operation and Method in UML

Operation is defined in **UML 1.4.2** as a service that can be requested from an object to effect behavior. An operation has a **signature**, which may restrict the actual parameters that are possible.

Method is defined as the implementation of an operation. It specifies the algorithm or procedure associated with an operation.

Encapsulation

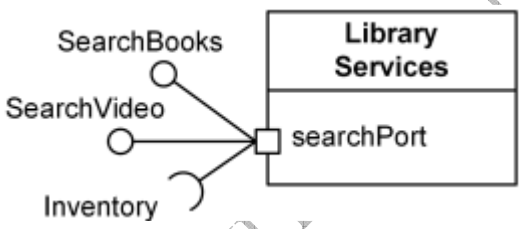
Encapsulation in UML

UML specifications provide no definition of **encapsulation** but use it loosely in several contexts.

For example, in UML 1.4 **object** is defined as an entity with a well defined boundary and identity that **encapsulates** state (attributes and relationships) and behavior (operations, methods, and state machines). Elements in peer packages are **encapsulated** and are not a priori **visible** to each other.

In UML 2.4 and 2.5 a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment, and also a Component is encapsulated and ... as a result, Components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together.

Encapsulated classifier in UML 2.4 and 2.5 is a structured classifier isolated from its environment (encapsulated ?) by using ports. Each port specifies a distinct interaction point between classifier and its environment.



Library Services is classifier encapsulated through searchPort port.

UML 2.4 specification also used term **completely encapsulated** without providing any explanation. It was removed in UML 2.5.

.

Abstraction in UML

Abstraction in UML corresponds to the concept of abstraction in OOD (as described above). UML provides different types (subclasses) of abstraction, including realizations (i.e. implementations).

Abstraction is a dependency relationship that relates two elements or sets of elements (called **client** and **supplier**) representing the **same concept** but at **different levels** of abstraction or from **different viewpoints**.

Realization is a specialized abstraction relationship between two sets of model elements, one representing a **specification** (the **supplier**) and the other represents an **implementation** of the latter (the **client**).

Inheritance

In OOAD and in UML 1.4 **inheritance** is defined as a mechanism by which more specific classes (called **subclasses** or **derived classes**) incorporate structure and behavior of the more general classes (called **superclasses** or **base classes**).

Generalization is defined as a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains some additional information. An instance of the more specific element may be used where the more general element is allowed.

Each generalizable element contains a list of features and other relationships that it adds to what it inherits from its ancestors.

For classifiers, these include features (attributes, operations, signal receptions, and methods) and participation in associations.]

If a generalizable element has more than one parent (**multiple inheritance**), then its full descriptor contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors.

Polymorphism

As it is usual with other OOAD concepts, **polymorphism** is also poorly defined. You can find all kinds of strange definitions of polymorphism, and there is no agreement which one is the best. To make things even worse, I will add my own definition of polymorphism:

Polymorphism is ability to apply different meaning (semantics, implementation) to the same symbol (message, operation) in different contexts.

When context is defined at compile time, it is called **static** or **compile-time polymorphism**. When context is defined during program execution, it is **dynamic** or **run-time polymorphism**.

It is believed that term "polymorphism" was introduced by Strachey in 1967 [CS 67] to describe operations and functions that could be applied to more than one type of arguments.

A typical example of static "ad hoc" polymorphism in procedural languages like ALGOL-60 or ALGOL-68 is:

```
sum := x + y;
```

In this example "+" is **polymorphic operation** which could be used with different types of operands - integer, real, string, complex, vector, etc. Specific static context - types of operands x and y - will determine at compile time which implementation of "+" is to be used.

This kind of static polymorphism is usually called **overloading** and means using the same operation symbol or function name on different types. Note, that overloading also allows different number of parameters and sometimes (ALGOL-68) even different priorities.

Another kind of static polymorphism is **parametric polymorphism** and is based on **templates**. In C++ example below, Vector template defines generic method 'elem' and operator '[ ]':

```
template<class T> class Vector {
  T* v;
  int size;
public:
  Vector();
  Vector(int);
  T& elem(int i) { return v[i]; }
  T& operator[] (int i);
  /* ... */
};
Vector<int> vi;
Vector<Shape*> vps;
```

In OOAD **polymorphism** means **dynamic polymorphism** and is commonly related to as **late binding** or **dynamic binding**. It could be defined as:

(Dynamic) Polymorphism is ability of objects of different classes to respond to the same message in a different way.

Virtual Functions

Procedural language Algol-60 allowed to pass a procedure as a parameter to another procedure. In Simula 67 classes may, like procedures, have formal parameters but they did not allow procedures as parameters to classes. Simula 67 authors found another way to have the same statements to have slightly different effect in different objects by declaring a procedure in a class C as "virtual", it could be redefined (overridden) in a subclass D. ... Thus, the same procedure call could activate different “versions” of the procedure, at least in objects of different subclasses.

Dynamic binding mechanism in C++, Java, C# allows to determine behavior (implementation) to be invoked in response to the message received by a specific object. To get this kind of polymorphic behavior in C++, the member functions must be **virtual** and objects must be manipulated through pointers or references.

Polymorphism in UML

There is no definition of polymorphism in UML specifications but there are some differences in how this term is used in different versions of UML.

In **UML 1.4.2** operation declares whether or not it may be realized by a different method in a subclass (which looks very similar to virtual functions in C++) by using **isPolymorphic** attribute. Methods realizing polymorphic operation have the same signature as the operation and have a body implementing the specification of the operation. Methods in descendants override and replace methods inherited from ancestors.

The **isPolymorphic** attribute is no longer present in **UML 2.4** and **UML 2.5**, without any explanation. Does it mean that all operations are now polymorphic (virtual), as if inspired by Java language? The **UML 2.4.1** specification had one obscure statement mentioning **polymorphism** in Chapter 11, Actions, and this statement is now removed from **UML 2.5**:

Operations are specified in the model and can be dynamically selected only through polymorphism.

I remember long time ago, Pascal language was promoted fiercely as a significant simplification compared to the huge, too formal, and complex Algol-68. The result was plain, restrained, and poorly defined language with ambiguous semantics. Hope, UML 2.5 simplification effort will not end up the same way, simple is not always lucid.

Software Development Life Cycle:

The UML is largely process-independent, means that it is not tied to any particular software development life cycle. However, to get the most benefit from UML, we should consider a process that is:

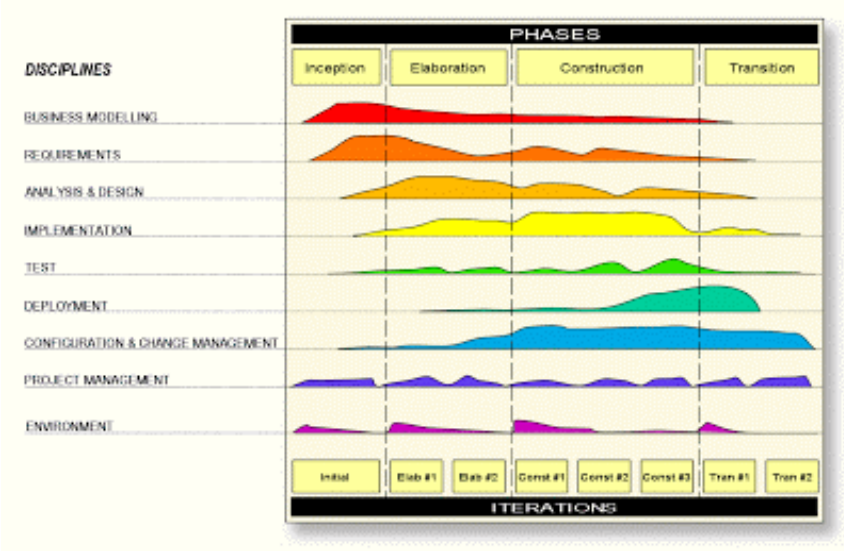
- Use case driven
- Architecture centric
- Iterative and Incremental

Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating system's architecture, for testing, and for communicating among the stakeholders of the project.

Architecture centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing and evolving the system under development.

An iterative process is one that involves managing a stream of executable releases. An iteration is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other.

This use case driven, architecture centric and iterative/incremental process can be broken down into phases. A phase is the span of time between two major milestones of the process. As the below figure shows, there are four phases in the software development life cycle: inception, elaboration, construction and transition. In the figure below, workflows are plotted against these phases, showing their varying degrees of focus over time.



Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are prioritized and baselined.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated and features that didn't make an earlier release are added.

An *iteration* is a distinct set of activities, with a baselined plan and evaluation criteria that result in a release, either internal or external. This means that the software development life cycle can be characterized as involving a continuous stream of executable releases of the system's architecture. It is this emphasis on architecture as an important artifact that drives the UML to focus on modeling the different views of a system's architecture.

Typical activities / workflows / disciplines in OOAD:

The UP(Unified Process) describes work activities, such as writing a use case, within **disciplines** (originally called **workflows).** Informally, a discipline is a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis. In the UP, an **artifact** is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on. There are several disciplines in the UP.

• **Business Modeling**—When developing a single application, this includes domain object modeling. When engaged in large-scale business analysis or business process reengineering, this includes dynamic modeling of the busi ness processes across the entire enterprise.

• **Requirements**—Requirements analysis for an application, such as writing use cases and identifying non-functional requirements.

• **Design**—All aspects of design, including the overall architecture, objects, databases, networking, and the like.

In the UP, **Implementation** means programming and building the system, not deployment. The **Environment** discipline refers to establishing the tools and customizing the process for the project—that is, setting up the tool and process environment.

list of UP disciplines is shown in the following Figure

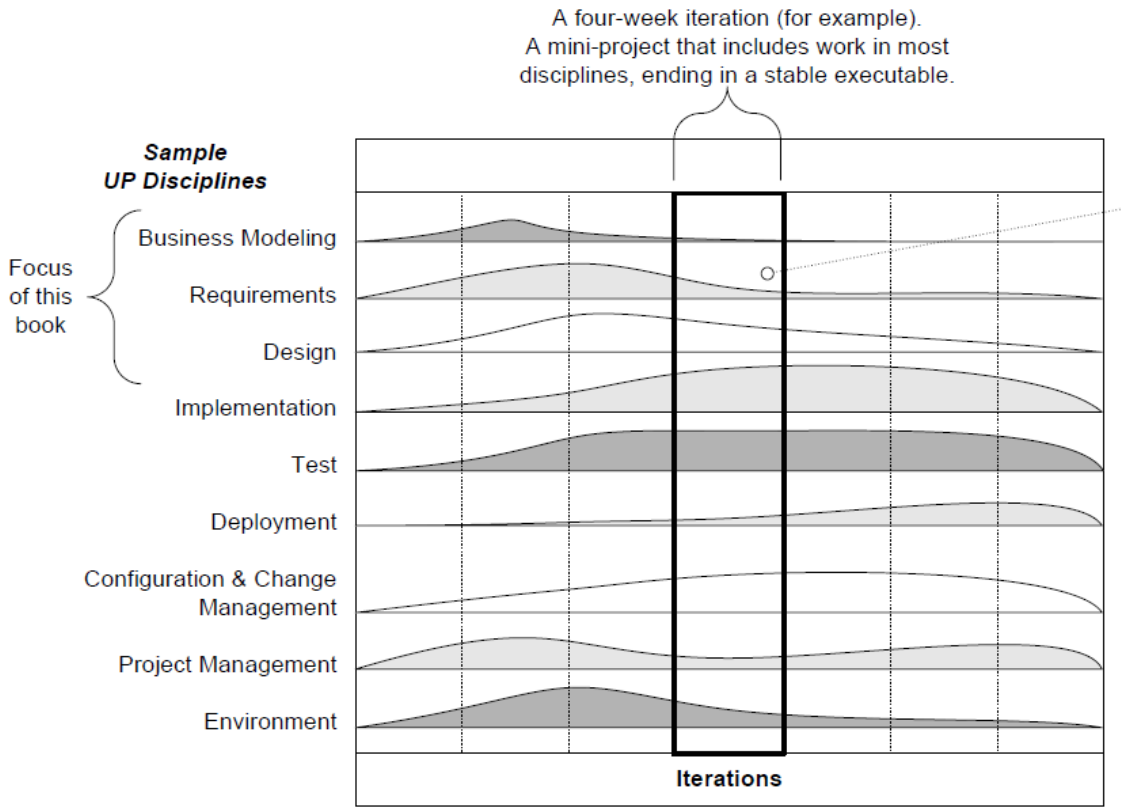


Figure 2.4 UP disciplines.[4]

UP Disciplines and Phases

During one iteration work goes on in most or all disciplines. However, the relative effort across these disciplines changes over time. Early iterations naturally tend to apply greater relative emphasis to requirements and design, and later ones less so, as the requirements and core design stabilize through a process of feedback and adaptation. Relating this to the UP phases (inception, elaboration, ...) the changing relative effort with respect to the phases; please note these are suggestive, not literal. In elaboration, for example, the iterations tend to have a relatively high level of requirements and design work, although definitely some implementation as well. During construction, the emphasis is heavier on implementation and lighter on requirements analysis.

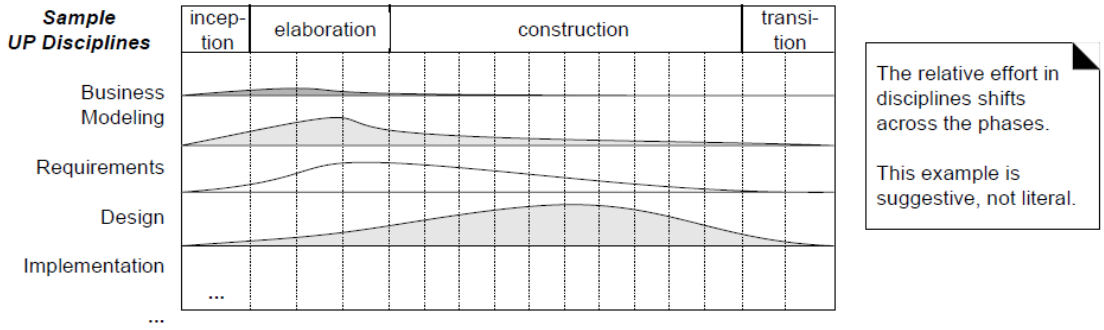


Figure 2.5 Disciplines and phases

Process Customization and the Development Case *Optional Artifacts*

a key insight into the UP is that all activities and artifacts (models, diagrams, documents, ...) are *optional.* on a UP project, a team should select a small subset of artifacts that address its particular problems and needs. In general, focus on a *small* set of artifacts that demonstrate high practical value.

The Development Case

The choice of UP artifacts for a project may be written up in a short document called the **Development Case** (an artifact in the Environment discipline). For example,the following table could be the Development Case describing the artifacts for the "NextGen Project" case study

Discipline	Artifact Iteration-*	Incep. I1	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

The Agile UP:

Processes can be heavy or light, and predictive vs. adaptive. A **heavy process** is one with the following qualities

• many artifacts created in a bureaucratic atmosphere
• rigidity and control
• elaborate, long-term, detailed planning
• predictive rather than adaptive

A predictive process is one that attempts to plan and predict the activities and resource (people) allocations in detail over a relatively long time span, such as the majority of a project. Predictive processes usually have a "waterfall" or sequential lifecycle—first, defining all the requirements; second, defining a detailed design; and third, implementing. In contrast, an **adaptive process** is one that accepts change as an inevitable driver and encourages flexible adaptation; they usually have an iterative lifecycle.

An **agile process** implies a light and adaptive process, quick in response to changing needs. it was meant to be adopted and applied in the spirit of an agile process—**agile UP.** Since the UP is iterative, requirements and designs are not completed before implementation. They adaptively emerge through a series of iterations, based on feedback. There isn't *a detailed* plan for the entire project. There is a high level plan (called the **Phase Plan)** that estimates the project end date and other major milestones, but it does not detail the fine-grained steps to those milestones. A detailed plan (called the **Iteration Plan)** only plans with greater detail one iteration in advance. Detailed planning is done adaptively from iteration to iteration.

Introduction to iterative development and the Unified Process

a **software development process** describes an approach to building, deploying, and possibly maintaining software. The **Unified Process** has emerged as a popular software development process for building object-oriented systems. In particular, the **Rational Unified Process or RUP.** a detailed refinement of the Unified Process, has been widely adopted. The Unified Process (UP) combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented description.

*Iterative Development***:**

The UP promotes several best practices, but one stands above the others: **iterative development.** In this approach, development is organized into a series of short, fixed-length (for example, four week) mini-projects called **iterations;** the outcome of each is a tested, integrated, and executable system. Each iteration includes its own requirements analysis, design, implementation, and testing activities. The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development** Early iterative process ideas were known as spiral

development and evolutionary Development.

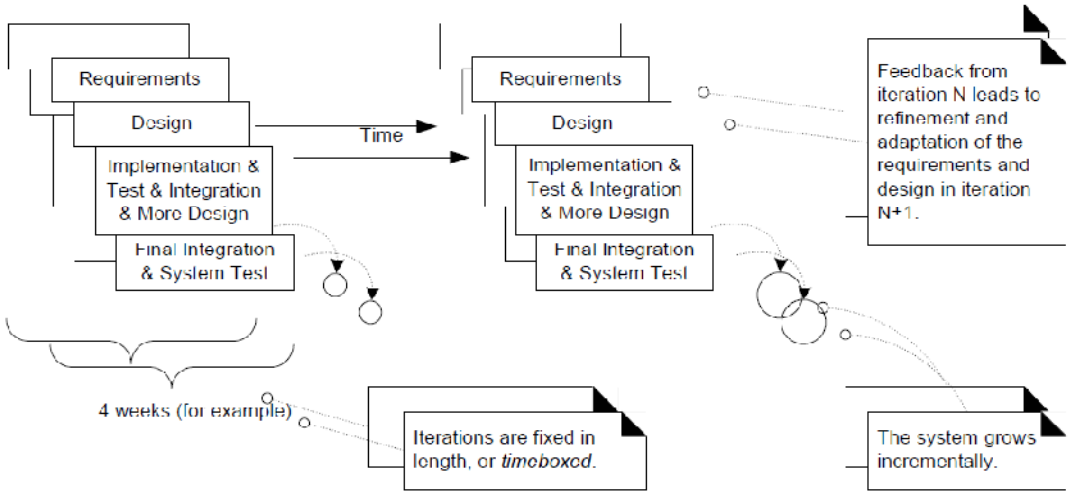


Figure 2.1 Iterative and incremental development.

The result of each iteration is an executable but incomplete system; it is not ready to deliver into production. The system may not be eligible for production deployment until after many iterations; for example, 10 or 15 iterations.

The output of an iteration is *not* an experimental or throw-away prototype, and iterative development is not prototyping. Rather, the output is a production-grade subset of the final system. Although, in general, each iteration tackles new requirements and incrementally extends the system, an iteration may occasionally revisit existing software and improve it; for example, one iteration may focus on improving the performance of a subsystem, rather than extending it with new features.

Feedback and Adaptation:

Each iteration involves choosing a small subset of the requirements, and quickly designing, implementing, and testing. In early iterations the choice of requirements and design may not be exactly what is ultimately desired. This early feedback is worth its weight in gold. the feedback from realistic building and testing something provides crucial practical insight and an opportunity to modify or adapt understanding of the requirements or design.

End-users have a chance to quickly see a partial system and say, "Yes, that's what I asked for, but now that I try it, what I really want is something slightly different. Feedback is a skillful way to make progress and discover what is of real value to the stakeholders. In addition to requirements clarification, activities such as load testing will prove if the partial design and implementation are on the right path, or if in the next iteration, a change in the core architecture is required. Better to resolve and *prove* the risky and critical design decisions early rather than late—and iterative development provides the mechanism for this. Consequently, work proceeds through a series of structured build-feedback-adapt cycles.

Benefits of Iterative Development:

Benefits of iterative development include:

• early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
• early visible progress
• early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
• managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
• The learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

Iteration Length and Timeboxing:

The UP (and experienced iterative developers) recommends an iteration length between two and six weeks. Small steps, rapid feedback, and adaptation are central ideas in iterative development; long iterations subvert the core motivation for iterative development and increase project risk. Much less than two weeks, and it is difficult to complete sufficient work to get meaningful throughput and feedback; much more than six or eight weeks, and the complexity becomes rather overwhelming, and feedback is delayed. A very long iteration misses the point of iterative development. Short is good.

A key idea is that iterations are **timeboxed,** or fixed in length. For example, if the next iteration is chosen to be four weeks long, then the partial system should be integrated, tested, and stabilized by the scheduled date—date slippage is discouraged.If it seems that it will be difficult to meet the deadline, the recommended
response is to remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.
The UP recommends that normally an iteration should be between two and six weeks in duration.

INTRODUCTION TO UML:

UML definition:
The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software intensive system.

The UML is a Language

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. **A modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of the system.** A modeling language like UML is thus a standard language for software blueprints.

The vocabulary and rules of a language such as the UML tell you how to create and read well-formed models. But they don't tell you what models you should create and when you should create them. That's the role of software development process. A well-defined process will guide you in deciding what artifacts to produce, what activities and what workers to use to create them and manage them.

UML is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then converting it out into code is close to zero. You think it, you code it. In fact, some things are best represented in code. Text is a wonderfully minimal and direct way to write expressions and algorithms.

The programmer is still doing some modeling. He or she may even sketch out a few ideas on a white board or paper. However, there are several problems with this. *First, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language. Second, there are some things about a software system you can't understand unless you build models.* For example, the meaning of a class hierarchy can be inferred, but not directly grasped, by looking at the code for all the classes in the hierarchy. *Third, if the developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever or, at best, only partially re-creatable from the implementation, once that developer moved on.*

Writing models in UML addresses the third issue: An explicit model facilitates communication. Some things are best modeled textually; others are best modeled graphically. Indeed, in all software systems there are structures that transcend what can be represented in a programming language. The UML is such a graphical language. This addresses the second problem described earlier.

The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, can interpret that model without any errors or confusion. This addresses the first issue described earlier.

UML is a Language for Specifying

Specifying means building models that are precise, unambiguous and complete. In particular, the UML addresses the specification of all the important analysis, design and implementation decisions that must be made in developing and deploying a software-intensive system.

UML is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in

the UML to a programming language such as Java, C++ or VB, or even to tables in a relational database or object-oriented database.

This mapping permits forward engineering: The generation of code from a UML model into a programming language. The reverse is also possible: you can reconstruct a model from an implementation back into the UML which is known as reverse engineering. Combining these two paths of forward code generation and reverse engineering yields round-trip engineering, meaning the ability to work with in either a graphical or a textual view, while the tools keep the two views consistent.

UML is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include:

- Requirements
- Architecture
- Design
- Source Code
- Project Plans
- Tests
- Prototypes
- Releases

The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modeling the activities of project planning and release management.

Where can UML be used? (Application domains of UML)

UML can be used in domains such as:

- Enterprise information systems
- Banking and Financial services
- Telecommunications
- Transportation
- Defense
- Retail
- Medical electronics
- Scientific
- Distributed web-based services

UML is not limited to modeling software. In fact, it is expressive enough to model non-software systems, such as workflow in the legal system, the structure and behavior of a patient healthcare system and the design of hardware.

Conceptual Model of the UML/ Mapping Disciplines into UML artifacts/ Artifacts of UML:

To understand the UML, we need to form a conceptual model of the language, and this requires learning three major elements: 1) UML's basic building blocks, 2) The rules that tell us how these building blocks may be put together and 3) Some common mechanisms that apply throughout the UML.

Building Blocks of UML

The vocabulary of the UML consists of three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together and diagrams group interesting collections of things.

Things in UML

There are four kinds of things in UML:

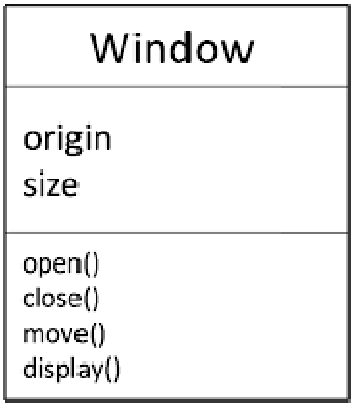
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

These things are the basic object-oriented building blocks of the UML. We use them to write well-formed models.

Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things:

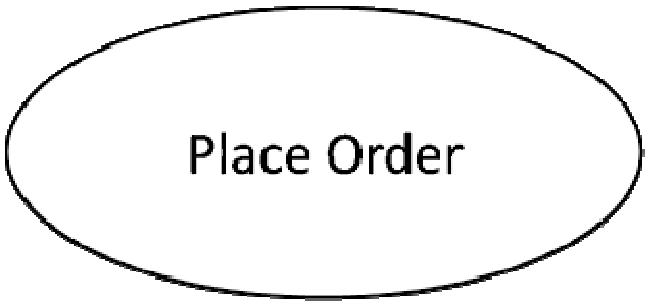
First, a class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically a class is rendered as a rectangle, usually including its name, attributes and operations as shown below:



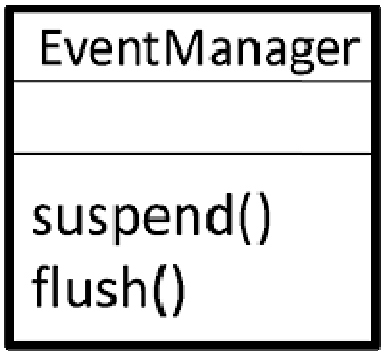
Second, an interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. Graphically, an interface is rendered as a circle together with its name. An interface is typically attached to the class or component that realizes that interface.

Third, a collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural as well as behavioral dimensions. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name as shown below:

Fourth, a use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name as shown below:

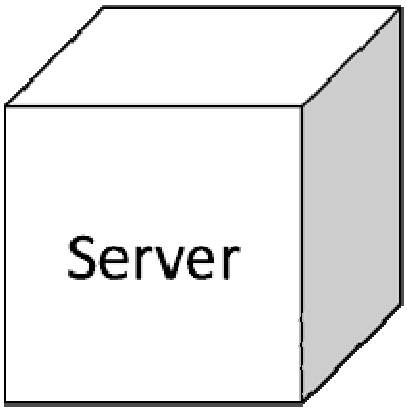


Fifth, an active class is a class whose object own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with thick lines, usually including its name, attributes and operations as shown below:



Sixth,a component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Generally, a component is rendered as a rectangle with tabs, usually including its name as shown below:

Seventh, a node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube, usually including only its name as shown below:



Behavioral ***Things***

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

First, an interaction is a behavior contains a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. An interaction involves messages, action sequences and links. Graphically, a message is rendered as a directed line, almost always including the name of its operation as shown below:

Second, a state machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. A state machine involves other elements like states, transitions, events and activities. Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates as shown below:

Grouping ***Things***

Grouping things are the organized parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

A package is general purpose mechanism for organizing elements into groups. Structural things, behavioral things and even other things may be placed in a package. Graphically, a

package is rendered as a tabbed folder, usually including only its name and sometimes, its contents as shown below:

Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments we may apply to describe, illuminate and remark about any element in a model. There is one primary kind of annotational thing, called a note. Graphically a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment as shown below:

Relationships in UML

There are four kinds of relationships in UML:

1. Dependency
2. Association
3. Generalization
4. Realization

These relationships are the basic relational building blocks of the UML. We use them to write well-formed models.

First, a dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing. Graphically, a dependency is rendered as a dashed line, possibly directed and occasionally including a label as shown below:

Second, an association is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments such as multiplicity and role names as shown below:

Third, a generalization is a specialization/generalization relationship in which objects of the specialized element (child) are substitutable for objects of the generalized element (parent). Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent as shown below:

Fourth, a realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship as shown below:

Diagrams in UML

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of things and relationships. We draw diagrams to visualize a system from different perspectives. So, a diagram is a projection into a system. In theory, a diagram may contain any combination of things and relationships. In practice, however a small number of common combinations arise, which are consistent with the five most useful views that make up the architecture of a software-intensive system. For this reason, UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

A class diagram shows a set of classes, interfaces, collaborations and their relationships. These diagrams are the most common diagrams found in modeling the object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams.

A use case diagram shows a set of use cases and actors and their relationships. Use case diagrams address the static use case view of a system. Theses diagrams are especially important in organizing and modeling the behaviors of a system.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages. A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

A statechart diagram shows a state machine, consisting of states, transitions, events and activities. Statechart diagrams address the dynamic view of a system.

An activity diagram is a special kind of statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system.

A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. They are related to component diagrams in that a node typically encloses one or more components.

Rules of UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any other language, UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for:

Names	What we can call things, relationships and diagrams
Scope	The context that gives specific meaning to a name
Visibility	How those names can be seen and used by others
Integrity	How things properly and consistently relate to one another
Execution	What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are:

Elided	Certain elements are hidden to simplify the view
Incomplete	Certain elements may be missing
Inconsistent	The integrity of the model is not guaranteed

These less-than-well-formed models are unavoidable as the details of a system unfold during the software development life cycle.

Common Mechanisms in UML

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language:

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specifications

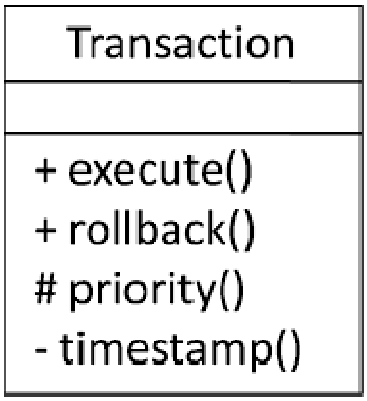
The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full

set of attributes, operations and behaviors that the class embodies. We use the UML's graphical notation to visualize the system. We use UML's specification to state the system's details.

Adornments

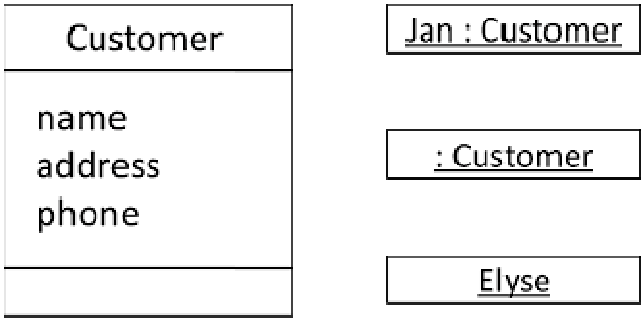
Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems.

A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. The below figure shows a class, adorned to indicate that it is an abstract class with two public, one protected and one private operation:



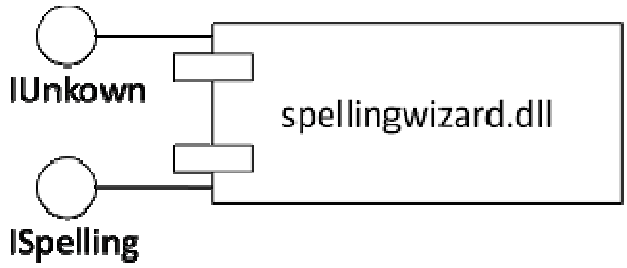
Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways. First, there is division of class and object. A class is an abstraction and an object is one concrete manifestation of that abstraction. In UML, we can model classes as well as objects, as shown below:



In the above figure, there is one class, named customer, together with three objects: Jan, :Customer and Elyse.

Second, there is separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract. In UML, you can model both interfaces and their implementations as shown below:



In the above figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and ISpelling.

Extensibility Mechanisms

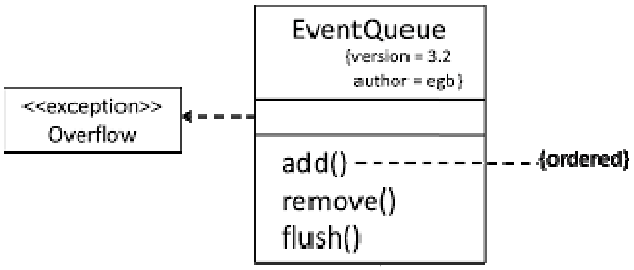
The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be enough to express all possible nuances of all models across all domains across all time. For this reason, UML is opened-ended, making it possible for us to extend the language in controlled ways. UML's extensibility mechanisms include:

- Stereotypes
- Tagged values
- Constraints

A *stereotype* extends the vocabulary of the UML, allowing us to create new kinds of building blocks that are derived from existing ones but that are specific to our problem. For example, if we are working in a programming language, such as Java or C++, we will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways.

A *tagged value* extends the properties of a UML building block, allowing us to create new information in that element's specification. For example, if we are working on a shrink-wrapped product that undergoes many releases over time, we often want to track the version and author of certain critical abstractions. Version and author are not primitive concepts in UML. They can be added to any building block, such as a class, by introducing new tagged values to that building block as shown in the figure below:

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. Below figure shows that you can add a constraint that explicitly marks these for the operation add.



Collectively, these three extensibility mechanisms allow you to shape and grow UML to our project's needs.

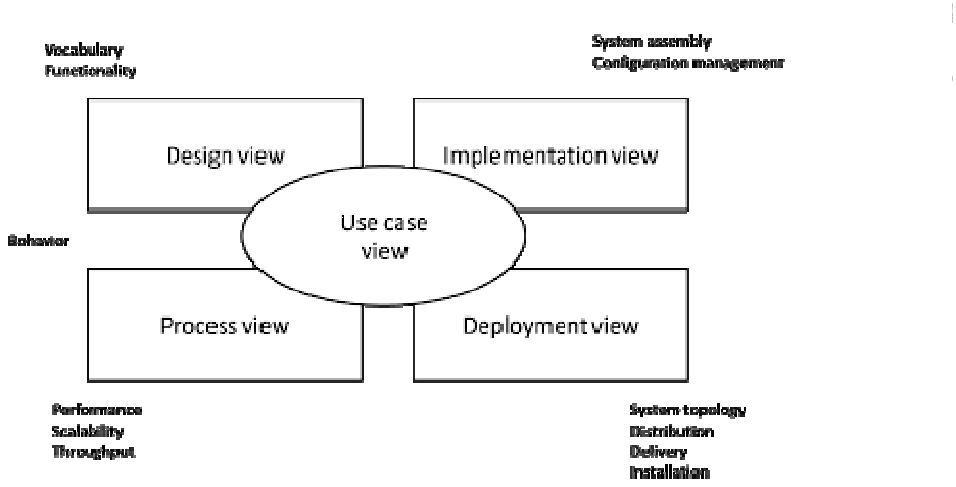
Architecture

Visualizing, specifying, constructing and documenting a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders like end users, analysts, developers, system integrators, testers, technical writers and project managers each looks at the system in different ways at different times over the project's life. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about:

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger sub systems
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations and their composition

As the below figure illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of the system.



The *use case view* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts and testers. With UML, the static aspects of this view are captured in use case diagrams and the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams and activity diagrams.

The *design view* of a system encompasses the classes, interfaces and collaborations that form the vocabulary of the problem and its solution. With UML, the static aspects of this view are captured in class diagrams and object diagrams while the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams and activity diagrams.

The *process view* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. With UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The *implementation view* of a system encompasses the components and files that are used to assemble and release the physical system. With UML, the static aspects of this view are captured in component diagrams and the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams and activity diagrams.

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. With UML, the static aspects of this view are captured in deployment diagrams and the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. The UML permits us to express every one of these five views and their interactions.

Introduction to Design Patterns

Def:

The ***design pattern*** describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
Each design pattern focuses on a particular object-oriented design problem or issue

In general, a pattern has four essential elements:

1. pattern name
2. The problem
3. Solution
4. consequences

The ***pattern name*** is used to describe a design problem, its solutions, and consequences in a word or two in such a way that
creates the design vocabulary,
designing of higher level abstractions,.
Useful for documentation
facilitates the communication of trades-offs to others.

2. The ***problem*** describes when to apply the pattern. It explains the problem and its context

3. The ***solution*** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, Instead, the pattern provides an abstract description of a design problem

4. The ***consequences*** are the results and trade-offs of applying the pattern. the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.
design patterns describe object-oriented designs. They are based on
object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada)

MVC(Model/View/Controller) architecture / DESIGN PATTERNS IN SMALL TALK MVC:

The Model/View/Controller (MVC) is used to build user interfaces in Smalltalk.

MVC consists of three kinds of objects.

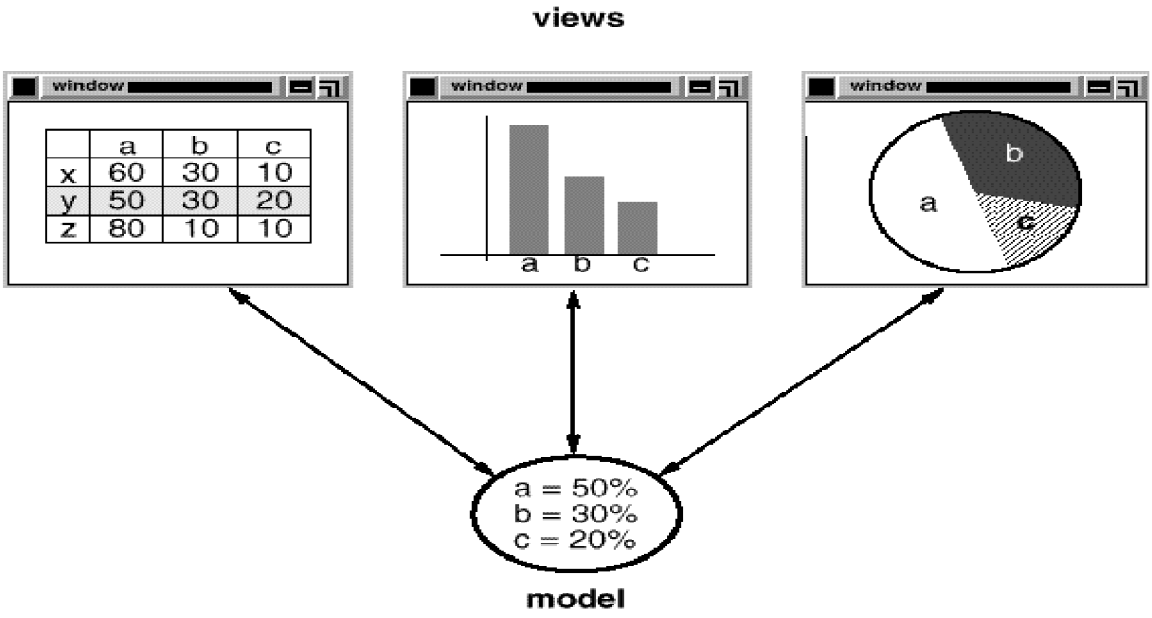
-Model
-View
-Controller

The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input.
Before MVC, user interface designs combine these objects together. MVC decouples them to increase flexibility and reuse.

MVC decouples views and models by establishing a ***subscribe/notify*** protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself.

This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The following diagram shows a model and three views. The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.



this example reflects a design that decouples views from models. Decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This is the responsibility of Observer design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. MVC supports nested views with the CompositeView class,

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller.

The View-Controller relationship is an example of the Strategy design pattern. MVC uses other design patterns, such as Factory Method to specify the default controller class for a view and Decorator to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

Describing Design Patterns:

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template makes the design patterns easier to learn, compare, and use.

Pattern Name and Classification

The pattern's name conveys the essence of the patter, briefly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the

Intent : A short statement that answers the following questions: What does the design pattern do?. What particular design issue or problem does it address?

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

Applicability What are the situations in which the design pattern can be applied?

Structure: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT)

Participants: The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations: How the participants collaborate to carry out their responsibilities.

Consequences: results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation: What hints, or techniques should you be aware of when implementing the pattern?

Sample Code: Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Related Patterns: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

The Catalog of Design Patterns:

Abstract Factory : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter : Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge : Decouple an abstraction from its implementation so that the two can vary independently.
Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command : Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Composite : Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator : Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Factory Method : Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Flyweight Use sharing to support large numbers of fine-grained objects efficiently.

Interpreter : Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator : Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator : Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer : Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Prototype : Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Proxy: Provide a surrogate or placeholder for another object to control access to it.

Singleton : Ensure a class only has one instance, and provide a global point of access to it.

State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy : Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor : Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Organizing the Catalog:

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them.

We classify design patterns by two criteria. The first criterion, called ***purpose***, reflects what a pattern does. Patterns can have either

1. creational,
2. structural, or
3. behavioural purpose..

	Purpose		
	Creational	Structural	Behavioral

Scope	**Class**	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Design Patterns: Elements of Reusable Object-Oriented Software

The ***second criterion, called scope***, specifies whether the pattern applies primarily to classes or to objects. ***Class patterns*** deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are ***static*** fixed at ***compile-time***. ***Object patterns*** deal with object relationships, which can be changed at run-time and are more dynamic.

There are other ways to organize the patterns. Some patterns are often used together. Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar.

Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns" sections. Following diagram depicts these relationships graphically.

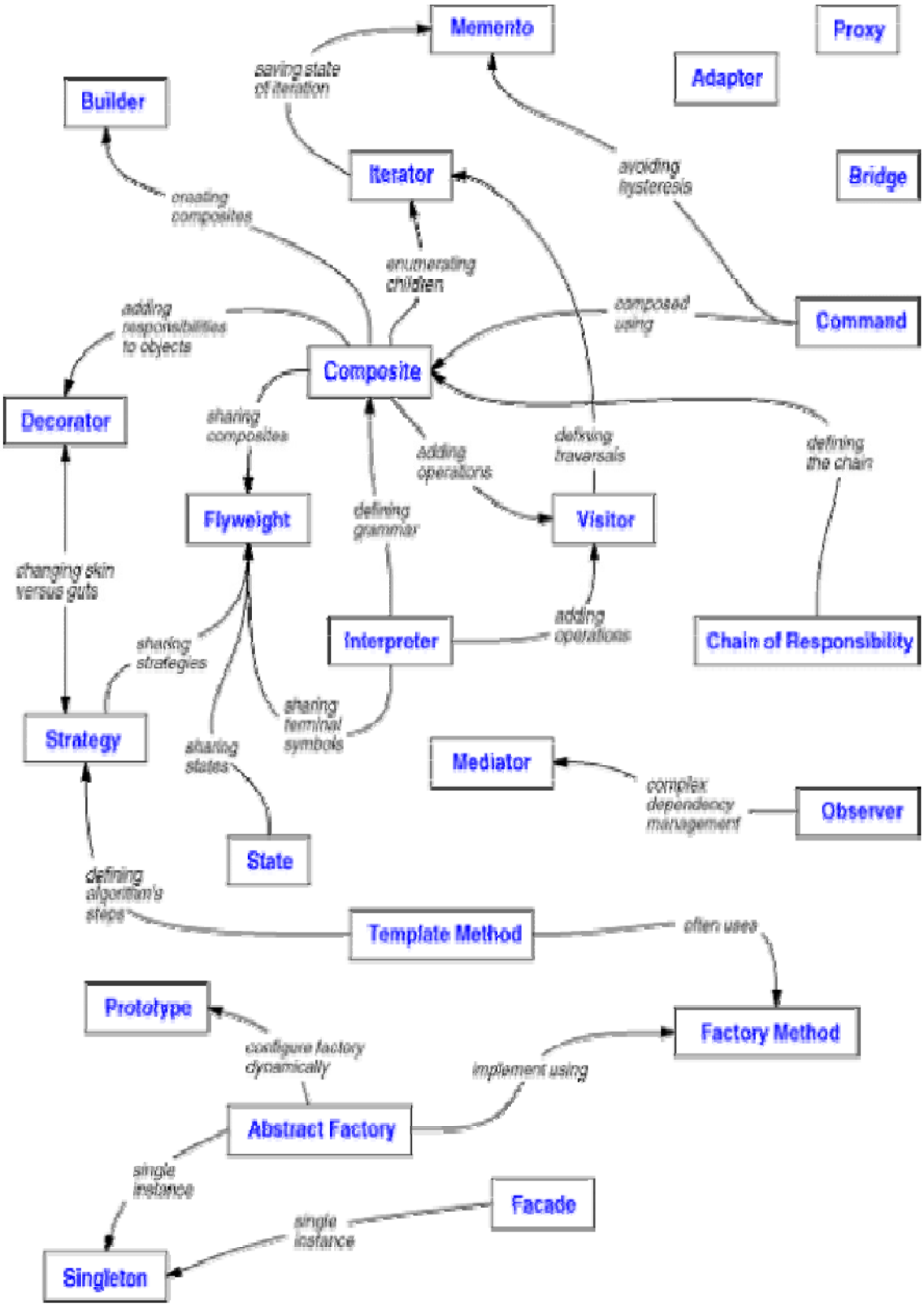


Figure 1.1: Design pattern relationships

How Design Patterns Solve Design Problems?:

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Finding Appropriate Objects:

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors should be considered : encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on.

Many objects in a design come from the analysis model. Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy pattern describes how to implement interchangeable families of algorithms. The State pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Determining Object Granularity:

Objects can vary in size and number. They can represent everything such as the hardware or all the way up to entire applications. How do we decide what should be an object?

Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities.

Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory and Builder yield objects whose only responsibilities are creating other objects. Visitor and Command yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces:

An operation the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object.

An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.
A type is a name used to denote a particular interface.

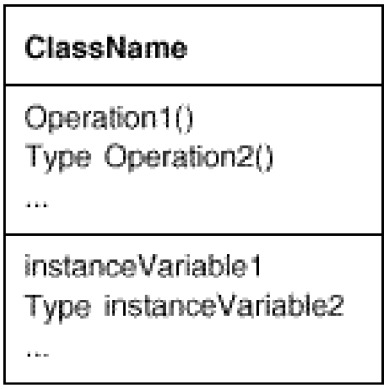
Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces. The run-time association of a request to an object and one of its operations is known as dynamic binding. Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. This substitutability is known as polymorphism, and it's a key concept in object-oriented systems.

Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what *not* to put in the interface. The Memento pattern is a good example. It describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later.

Design patterns also specify relationships between interfaces . or they place constraints on the interfaces of some classes. For example, both Decorator and Proxy require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor , the Visitor interface must reflect all classes of objects that visitors can visit.

Specifying Object Implementations:

An object's implementation is defined by its class. The class specifies the object's internal data and defines the operations the object can perform.

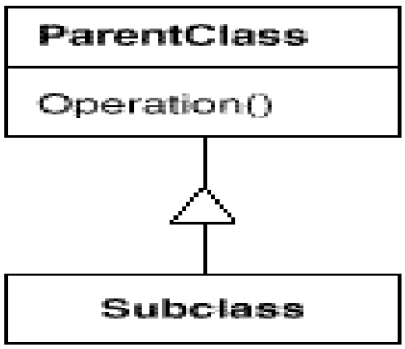


Objects are created by instantiating a class. The object is said to be an instance of the class. The process of instantiating a class allocates storage for the object's internal data (made up of instance variables) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.

A dashed arrowhead line indicates a class that instantiates objects of another class.

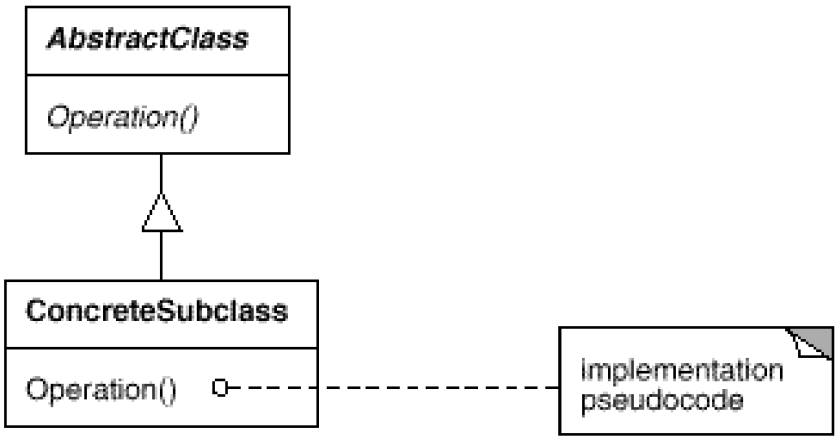
New classes can be defined in terms of existing classes using class inheritance. When a subclass inherits from a parent class, it includes the definitions of all the data and operations

that the parent class defines. Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes, and they'll be able to perform all operations defined by this subclass and its parents. We indicate the subclass relationship with a vertical line and a triangle:

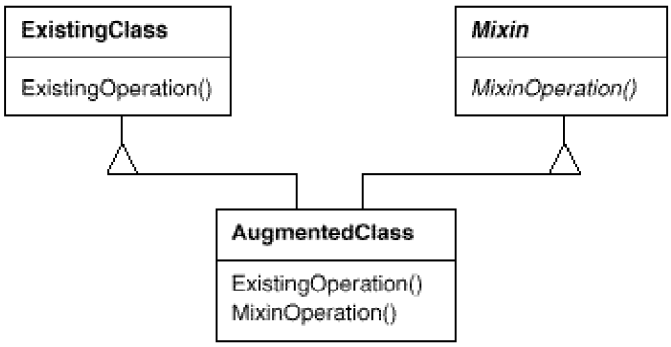


An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated. The operations that an abstract class declares but doesn't implement are called abstract operations. Classes that aren't abstract are called concrete classes. Subclasses can refine and redefine behaviors of their parent classes.

More specifically, a class may override an operation defined by its parent class. Overriding gives subclasses a chance to handle requests instead of their parent classes. Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.



A mixin class is a class that's intended to provide an optional interface or functionality to other classes. It's similar to an abstract class in that it's not intended to be instantiated. Mixin classes require multiple inheritance:



Class versus Interface Inheritance:

It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

Putting Reuse Mechanisms to Work:

Most people can understand concepts like objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible, reusable software, and design patterns can show you how.

1. Inheritance versus Composition:

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. Reuse by subclassing is often referred to as white-box reuse. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes."

Inheritance and composition each have their advantages and disadvantages. Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Implementation dependencies can cause problems when you're trying to reuse a subclass. This dependency limits flexibility and ultimately reusability.

Object composition is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff. Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type.

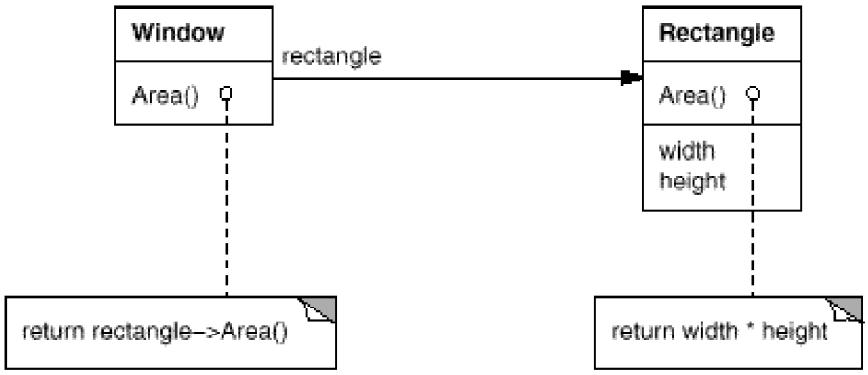
Object composition has another effect on system design. Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.

2 .***Delegation:***

Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request:

a receiving object delegates operations to its delegate. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the this member variable in C++ and self in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

For example, instead of making class Window a subclass of Rectangle (because windows happen to be rectangular), the Window class might reuse the behavior of Rectangle by keeping a Rectangle instance variable and *delegating* Rectangle-specific behavior to it. In other words, instead of a Window *being* a Rectangle, it would *have* a Rectangle. Window must now forward requests to its Rectangle instance explicitly, whereas before it would have inherited those operations.
The following diagram depicts the Window class delegating its Area operation to a Rectangle instance.



The main ***advantage of delegation*** is that it makes it easy to compose behaviours at run-time and to change the way they're composed. Our window can become circular at run-time simply by replacing its Rectangle instance with a Circle instance, assuming Rectangle and Circle have the same type.

Delegation has a ***disadvantage*** it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software. There are also run-time inefficiencies, but the human inefficiencies are more important in the long run.

Delegation is a good design choice only when it simplifies more than it complicates. Several design patterns use delegation. The State , Strategy , and Visitor patterns depend on it. In the State pattern, an object delegates requests to a State object that represents its current state. In the Strategy pattern, an object delegates a specific request to an object that represents a strategy for carrying out the request.

An object will only have one state, but it can have many strategies for different requests. The purpose of both patterns is to change the behavior of an object by changing the objects to which it delegates requests. In Visitor, the operation that gets performed on each element of an object structure is always delegated to the Visitor object.

Other patterns use delegation less heavily. Mediator introduces an object to mediate communication between other objects. Sometimes the Mediator object implements operations simply by forwarding them to the other objects; other times it passes along a reference to itself and thus uses true delegation. Chain of Responsibility handles requests by forwarding them from one object to another along a chain of objects. Sometimes this request carries with it a reference to the original object receiving the request, in which case the pattern is using delegation. Bridge decouples an abstraction from its implementation. If the abstraction and a particular implementation are closely matched, then the abstraction may simply delegate operations to that implementation. Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

Inheritance versus Parameterized Types:

Another (not strictly object-oriented) technique for reusing functionality is through parameterized types, also known as generics (Ada, Eiffel) and templates (C++). This technique lets you define a type without specifying all the other types it uses. The unspecified types are supplied as *parameters* at the point of use.

For example, a List class can be parameterized by the type of elements it contains. To declare a list of integers, you supply the type "integer" as a parameter to the List parameterized type. To declare a list of String objects, you supply the "String" type as a parameter.

Parameterized types give us a third way (in addition to class inheritance and object composition) to compose behavior in object-oriented systems. Many designs can be implemented using any of these three techniques.

There are important differences between these techniques.

Object composition lets you change the behavior being composed at run-time, but it also requires indirection and can be less efficient.

Inheritance lets you provide default implementations for operations and lets subclasses override them. Parameterized types let you change the types that a class can use. But neither inheritance nor parameterized types can change at run-time. Which approach is best depends on your design and implementation constraints.

Parameterized types aren't needed at all in a language like Smalltalk that doesn't have compile-time type checking.

Relating Run-Time and Compile-Time Structures:

The run-time and Compile-time structures are largely independent. The code structure is frozen at compile time. It consists of classes in fixed inheritance relationships. A program run-time structure consists of rapidly changing networks of communicating objects.

Consider the distinction between object aggregation and acquaintance and how differently they manifest themselves at compile- and run-times. Aggregation implies that one object owns or is responsible for another object. Generally we speak of an object *having* or being *part of* another object. Aggregation implies that an aggregate object and its owner have identical lifetimes.

Acquaintance implies that an object merely *knows of* another object. Sometimes acquaintance is called "association" or the "using" relationship. Acquainted objects may request operations of each other, but they aren't responsible for each other. Acquaintance is a weaker relationship than aggregation and suggests much looser coupling between objects.

In our diagrams, a plain arrowhead line denotes acquaintance. An arrowhead line
with a diamond at its base denotes aggregation

It's easy to confuse aggregation and acquaintance, because they are often implemented in the same way. The distinction may be hard to see in the compile-time structure, but it's significant. Aggregation relationships tend to be fewer and more permanent than acquaintance. Acquaintances, in contrast, are made and remade more frequently, sometimes existing only for the duration of an operation. Acquaintances are more dynamic as well, making them more difficult to discern in the source code.

With such disparity between a program's run-time and compile-time structures, it's clear that code won't reveal everything about how a system will work. The system's run-time structure must be imposed more by the designer than the language. The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is.

Many design patterns (in particular those that have object scope) capture the distinction between compile-time and run-time structures explicitly. Composite and Decorator are especially useful for building complex run-time structures. Observer involves run-time structures that are often hard to understand unless you know the pattern. Chain of Responsibility also results in communication patterns that inheritance doesn't reveal. In general, the run-time structures aren't clear from the code until you understand the patterns.

Designing for Change:

The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

To design the system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and eimplementation, client modification, and retesting.

Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.
Design patterns help you avoid this by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.
Here are some common causes of redesign along with the design pattern(s) that address them:

1. ***Creating an object by specifying a class explicitly***. Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.
Design patterns: Abstract Factory , Factory Method , Prototype
.
2. ***Dependence on specific operations***. When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time. Design patterns: Chain of Responsibility , Command .

3. ***Dependence on hardware and software platform***. External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms. It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies. Design patterns: Abstract Factory , Bridge .

4. ***Dependence on object representations or implementations***. Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.
Design patterns: Abstract Factory , Bridge , Memento , Proxy .

5. ***Algorithmic dependencies***. Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore algorithms that are likely to change should be isolated. Design patterns: Builder , Iterator , Strategy , Template Method , Visitor
.
6. ***Tight coupling.*** Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain. ***Loose coupling*** increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily. Design patterns use techniques such as ***abstract coupling and layering*** to promote loosely coupled systems. Design patterns: Abstract Factory , Bridge , Chain of Responsibility , Command , Facade , Mediator , Observer

7. **Extending functionality by subclassing**. Customizing an object by subclassing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And subclassing can lead to an explosion of classes,

because you might have to introduce many new subclasses for even a simple extension. Object composition in general and delegation in particular provide flexible alternative inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand.

Many design patterns produce designs in which you can introduce customized functionality just by defining one subclass and composing its instances with existing ones.

Design patterns: Bridge , Chain of Responsibility , Composite , Decorator , Observer , Strategy .

8. ***Inability to alter classes conveniently***. Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it. Or maybe any change would require modifying lots of existing subclasses. Design patterns offer ways to modify classes in such circumstances. Design patterns: Adapter , Decorator , Visitor .

These examples reflect the flexibility that design patterns can help you build into your software. How crucial such flexibility is depends on the kind of software you're building.

Let's look at the role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.

Application Programs

If you're building an application program such as a document editor or spreadsheet, then *internal* reuse, maintainability, and extension are high priorities. Internal reuse ensures that you don't design and implement any more than you have to. Design patterns that reduce dependencies can increase internal reuse. Looser coupling boosts the likelihood that one class of object can cooperate with several others. For example, when you eliminate dependencies on specific operations by isolating and encapsulating each operation, you make it easier to reuse an operation in different contexts. The same thing can happen when you remove algorithmic and representational dependencies too.

Design patterns also make an application more maintainable when they're used to limit platform dependencies and to layer a system. They enhance extensibility by showing you how to extend class hierarchies and how to exploit object composition. Reduced coupling also enhances extensibility. Extending a class in isolation is easier if the class doesn't depend on lots of other classes.

Toolkits:

Often an application will incorporate classes from one or more libraries of predefined classes called toolkits. A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is a set of collection classes for lists, associative tables, stacks, and the like. The C++ I/O stream library is another example. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They let you as an implementer avoid recoding common functionality. Toolkits emphasize *code reuse*. They are the object-oriented equivalent of subroutine libraries. Toolkit design is arguably harder than application design, because toolkits have

to work in many applications to be useful. Moreover, the toolkit writer isn't in a position to know what those applications will be or their special needs. That makes it all the more important to avoid assumptions and dependencies that can limit the toolkit's flexibility and consequently its applicability and effectiveness.

Frameworks

A framework is a set of cooperating classes that make up a reusable design for a specific class of software For example, a framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical . Another framework can help you build compilers for different programming languages and target machines . Yet another might help you build financial modeling applications . You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework. The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application.

The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize *design reuse* over code reuse, though a framework will usually include concrete subclasses you can put to work immediately. Reuse on this level leads to an inversion of control between the application and the software on which it's based. When you use a toolkit (or a conventional subroutine library for that matter), you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code *it* calls. You'll have to write operations with particular names and calling conventions, but that reduces the design decisions you have to make. Not only can you build applications faster as a result, but the applications have similar structures. They are easier to maintain, and they seem more consistent to their users. On the other hand, you lose some creative freedom, since many design decisions have been made for you. If applications are hard to design, and toolkits are harder, then frameworks are hardest of all. A framework designer gambles that one architecture will work for all applications in the domain. Any substantive change to the framework's design would reduce its benefits considerably, since the framework's main contribution to an application is the architecture it defines. Therefore it's imperative to design the framework to be as flexible and extensible as possible.

Furthermore, because applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it. That makes loose coupling all the more important; otherwise even a minor change to the framework will have major repercussions.
The design issues just discussed are most critical to framework design. A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't. Mature frameworks usually incorporate several design patterns. The patterns help make the framework's architecture suitable to many different applications without redesign. An added benefit comes when the framework is documented with the design patterns it uses . People who know the patterns gain insight into the framework faster. Even people who don't know the patterns can benefit from the structure they lend to the framework's documentation. Enhancing documentation is important for all types of software, but it's particularly important for frameworks. Frameworks

often pose a steep learning curve that must be overcome before they're useful. While design patterns might not flatten the learning curve entirely, they can make it less steep by making key elements of the framework's design more explicit.

Difference between frameworks and design patterns:

Even though because patterns and frameworks have some similarities, they are different in three major ways:

1. *Design patterns are more abstract than frameworks.* Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, the design patterns in this book have to be implemented each time they're used. Design patterns also explain the intent, trade-offs, and consequences of a design.

2***. Design patterns are smaller architectural elements than frameworks.*** A typical ramework contains several design patterns, but the reverse is never true.

3. ***Design patterns are less specialized than frameworks***. Frameworks always have a particular application domain. A graphical editor framework might be used in a factory simulation, but it won't be mistaken for a simulation framework. In contrast, the design patterns in this catalog can be used in nearly any kind of application. While more specialized design patterns than ours are certainly possible (say, design patterns for distributed systems or concurrent programming), even these wouldn't dictate an application architecture like a framework would.

Frameworks are becoming increasingly common and important. They are the way that object-oriented systems achieve the most reuse. Larger object-oriented applications will end up consisting of layers of frameworks that cooperate with each other. Most of the design and code in the application will come from or be influenced by the frameworks it uses.

How to Select a Design Pattern:

With more than 20 design patterns in the catalog , it might be hard to find the one that addresses a particular design problem
Here are several different approaches to finding the design pattern that's right for your problem:

1. ***Consider how design patterns solve design problems***.
In the above topic we discussed, how the design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems.

2. ***Scan Intent sections.*** Read through each pattern's intent to find one or more that sound relevant to your problem. You can use the classification scheme to narrow your search.

3. ***Study how patterns interrelate***. Studying these design pattern relationships can help direct you to the right pattern or group of patterns.

4. **Study patterns of like purpose.** The catalog consists of creational patterns, structural patterns, and behavioral patterns. Compare and contrast them give you insight into the similarities and differences between patterns of like purpose.

5. **Examine a cause of redesign.** Look at the causes of redesign starting on page 37 to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

6. **Consider what should be variable in your design**. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The following table lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

Purpose Design Pattern Aspect(s) That Can Vary

Creational
Abstract Factory : families of product objects
Builder : how a composite object gets created
Factory Method : subclass of object that is instantiated
Prototype : class of object that is instantiated
Singleton: the sole instance of a class

Structural:
Adapter : interface to an object
Bridge : implementation of an object
Composite : structure and composition of an object
Decorator : responsibilities of an object without subclassing
Facade : interface to a subsystem
Flyweight : storage costs of objects
Proxy: how an object is accessed; its location

Behavioral:
Chain of Responsibility: object that can fulfill a request
Command : when and how a request is fulfilled
Interpreter : grammar and interpretation of a language
Iterator : how an aggregate's elements are accessed, traversed
Mediator : how and which objects interact with each other
Memento (316) what private information is stored outside an object, and when
Observer: number of objects that depend on another
Object: how the dependent objects stay up to date
State: states of an object
Strategy : an algorithm
Template Method : steps of an algorithm
Visitor operations that can be applied to object(s) without changing their class(es)

How to Use a Design Pattern:

Once you've picked a design pattern, how do you use it? Here's a step-by-step approach to applying a design pattern effectively:

1. **Read the pattern once through for an overview**. Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.

2. **Go back and study the Structure, Participants, and Collaborations sections.** Make sure you understand the classes and objects in the pattern and how they relate to one another.

3. ***Look at the Sample Code section to see a concrete example of the pattern in code.*** Studying the code helps you learn how to implement the pattern.

4. **Choose names for pattern participants that are meaningful in the application context.** The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application.

5. ***Define the classes.*** Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

6. ***Define application-specific names for operations in the pattern.*** Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.

7. **Implement the operations to carry out the responsibilities and collaborations in the pattern**. The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

Goals of good design:

There are many aspects to consider in the design of a piece of software. The importance of each consideration should reflect the goals and expectations that the software is being created to meet. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.

- **Modularity** - the resulting software comprises well defined, independent components which leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.
- **Reliability** (Software durability) - The software is able to perform a required function under stated conditions for a specified period of time.
- **Reusability** - The ability to use some or all of the aspects of the preexisting software in other projects with little to no modification.
- **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security** - The software is able to withstand and resist hostile acts and influences.
- **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.[5]
- **Performance** - The software performs its tasks within a time-frame that is acceptable for the user, and does not require too much memory.
- **Portability** - The software should be usable across a number of different conditions and environments.
- **Scalability** - The software adapts well to increasing data or number of users.

CASE STUDY:

The NextGen POS System

The case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are very interesting requirement and design problems to solve. In addition, it is a realistic problem; organizations really do write POS systems using object technologies.

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled). A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth. Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is

initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation. Architectural Layers and Case Study Emphasis A typical object-oriented information system is designed in terms of several architectural layers or subsystems (see Figure 3.1). The following is not a complete list, but provides an example:

• **User Interface**—graphical interface; windows.

• **Application Logic and Domain Objects—**software objects representing domain concepts (for example, a software class named *Sale)* that fulfil application requirements.

• **Technical Services**—general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems. OOA/D is generally most relevant for modeling the application logic and technical service layers.

The NextGen case study primarily emphasizes the problem domain objects, allocating responsibilities to them to fulfill the requirements of the application.

Object-oriented design is also applied to create a technical service subsystem for interfacing with a database.

In this design approach, the UI layer has very little responsibility; it is said to be *thin.* Windows do *not* contain code that performs application logic or processing. Rather, task requests are forwarded on to other layers.

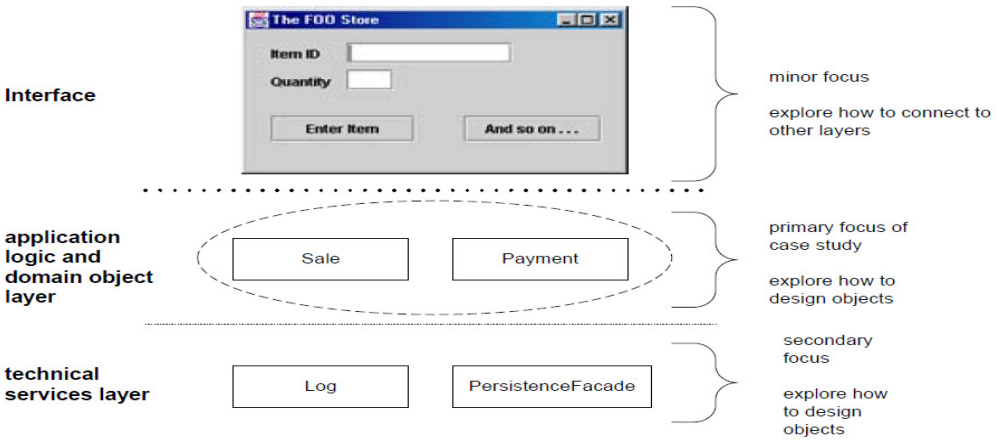


Figure 3.1 Sample layers and objects in an object-oriented system, and the case study focus.

UNIT-II
INCEPTION

Unit II Syllabus:

Inception: Artifacts in inception, Understanding requirements – the FURPS model, Understanding Use case model – introduction, use case types and formats, Writing use cases – goals and scope of a use case, elements / sections of a use case, Use case diagrams, Use cases in the UP context and UP artifacts, Identifying additional requirements, Writing requirements for the case study in the use case model.

Introduction:

This chapter defines the inception phase of a project. Most projects require a short initial step in which the following kinds of questions are explored:

- What is the vision and business case for this project?
- Feasible?
- Buy and/or build?
- Rough estimate of cost: Is it $10K-100K or in the millions?
- Should we proceed or stop?

However, the purpose of the inception step is not to define all the requirements, or generate a believable estimate or project plan. the idea is to do just enough investigation to form a rational, justifiable opinion of the overall purpose and feasibility of the potential new system, and decide if it is worthwhile to invest in deeper exploration.
Thus, the inception phase should be relatively short for most projects, such as one or a few weeks long. Indeed, on many projects, if it is more than a week long.

Artifacts in Inception Phase:

Following table lists common inception (or early elaboration) artifacts and indicates the issues they address. These are only partially completed in this phase, will be refined in later iterations, And since it is inception, the investigation and artifact content should be light. For example, the Use-Case Model may list the *names* of most of the expected use cases and actors, but perhaps only describe 10% of the use cases in detail—done in the service of developing a rough high-level vision of the system scope, purpose, and risks. Note that some programming work may occur in inception in order to create "proof of concept" prototypes.

Artifact[1]	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements, and related non-functional requirements.
Supplementary Specification	Describes other requirements.
Glossary	Key domain terminology.
Risk List & Risk Management Plan	Describes the business, technical, resource, schedule risks, and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.

Artifact[1]	Comment
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

Table 4.1 Sample inception artifacts.

These artifacts are only partially completed in this phase. They will be itera-tively refined in subsequent iterations.

Understanding requirements – the FURPS+ model:

Requirements are capabilities and conditions to which the system must agree with. A prime challenge of requirements work is to find, communicate, and remember (record) what is really needed, in a form that clearly speaks to the client and development team members.
The UP promotes a set of best practices, one of which is *manage requirements.*
a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system is RUP. In UP requirements are gathered via techniques such as use case writing and requirements workshops.

As indicated in the following fig, challenged projects revealed that 37% of factors related to problems with requirements, making requirements issues the largest single contributor to problems . Consequently, masterful requirements management is important. In the waterfall Model the requirements are polished , stabilized, and freezed before design or implementation. The iterative approach is used the change and feedback as core drivers in discovering requirements.

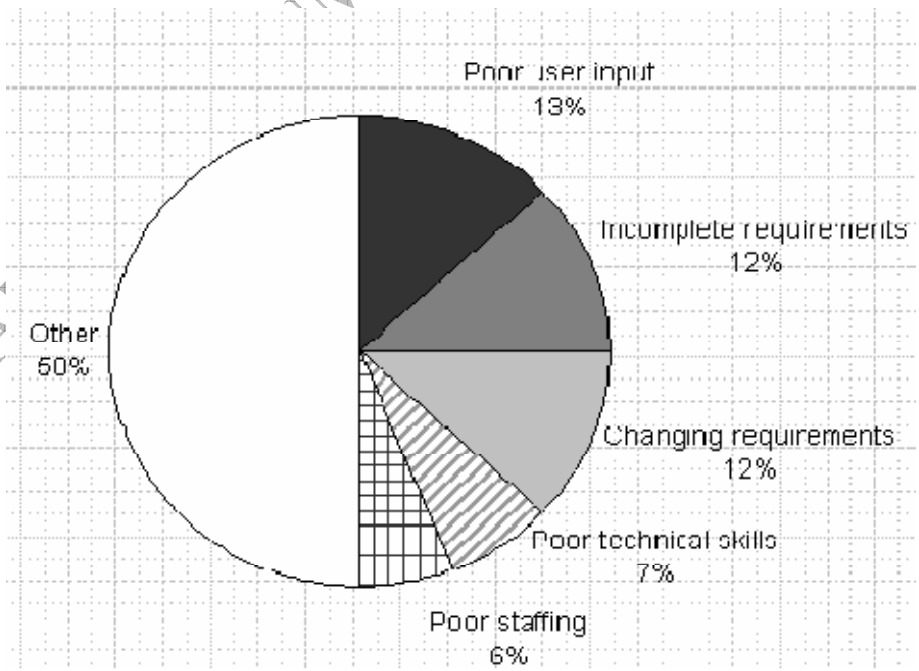


Figure: Factors on challenged software projects.

FURPS+ MODEL:

In the UP, requirements are categorized according to the FURPS+ model , a useful mnemonic with the following meaning:

• **Functional**—features, capabilities, security.
• **Usability**—human factors, help, documentation.
• **Reliability**—frequency of failure, recoverability, predictability.
• **Performance**—response times, throughput, accuracy, availability, resource usage.
• **Supportability**—adaptability, maintainability, internationalization, con figurability.
The "+" in FURPS+ indicates ancillary and sub-factors, such as:
• **Implementation**—resource limitations, languages and tools, hardware, ...
• **Interface**—constraints imposed by interfacing with external systems.
• **Operations**—system management in its operational setting.
• **Packaging**
• **Legal**—licensing and so forth.
It is helpful to use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk missing some important facet of the system.

Some of these requirements are collectively called the **quality attributes, quality requirements.** These include:

usability,
reliability,
performance,
and supportability.

In common usage, requirements are categorized as **functional** (behavioral) or **non-functional** (everything else).Functional requirements are explored and recorded in the Use-Case Model. Other requirements can be recorded in the use cases they relate to, or in the Supplementary Specifications artifact. The Vision artifact summarizes high-level requirements that are elaborated in these other documents. The Glossary records and clarifies terms used in the requirements. The Glossary in the UP also encompasses the concept of the **data dictionary,** which records requirements related to data, such as validation rules, acceptable values, and so forth. The quality requirements have a strong influence on the architecture of a system. For example, a high-performance,high-reliability requirement will influence the choice of software and hardware components, and their configuration. The need for easy adaptability due to frequent changes in the functional requirements would likewise fundamentally shape the design of the software.

Understanding Use case model:

Writing use cases(stories of using a system)—is an excellent technique to understand and describe requirements. The UP defines the **Use-Case Model** within the Requirements discipline. it is a model of the system's functionality and environment.

Customers and end users have goals (also known as *needs* in the UP) and want computer systems to help meet them, There are several ways to capture these goals and system requirements. Use cases are a mechanism to help keep it simple and understandable.
for example *brief format* use case:

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system

updates inventory. The customer receives a receipt from the system and then leaves with the items. stakeholders.

Simplicity and Utility are important factors in writing usecases.

Use Cases and Functional Requirements:

Use cases are requirements; primarily they are functional requirements that indicate what the system will do. In terms of the FURPS+ requirements types, they emphasize the "F" (functional or behavioral), but can also be used for other types, especially when those other types strongly relate to a use case.
In the UP—and most modern methods—use cases are the central mechanism that is recommended for their discovery and definition. Use cases define a promise or contract of how a system will behave. To be clear: Use cases *are* requirements (although not all requirements).
write use cases for the functional requirements. Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing. However, the UML defines a use case diagram to illustrate the names of use cases and actors, and their relationships.

Use Case Types:

There are two types of usecases:

1. Black box usecases
2. White box usecases

Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having *responsibilities,* which is a common unifying metaphorical theme in object-oriented thinking—software elements have responsibilities and collaborate with other elements that have responsibilities. By defining system responsibilities with black-box use cases, it is possible to specify *what* the system must do (the functional requirements) without deciding *how* it will do it (the design). Indeed, the definition of "analysis" versus "design" is sometimes summarized as "what" versus "how." This is an important theme in good software development: During requirements analysis avoid making "how" decisions, and specify the external behavior for the system, as a black box. Later, during design, create a solution that meets the specification.

Black-box style	Not
The system records the sale.	The system writes the sale to a data-base. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Usecase Formats:

Use cases are written in different formats, depending on need. In addition to the black-box versus white-box *visibility* type, use cases are written in varying degrees *of formality:*

• **brief**—terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.
• **casual**—informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.
• **fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees. The following example is a fully dressed case for our NextGen case study.

Example:
Use Case UC1: Process Sale
Primary Actor: Cashier
Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer short ages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants proof of pur chase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agen cies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.
Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10.Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

*a. At any time, System fails:
To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.
2a. System detects anomalies preventing recovery:
1. System signals error to the Cashier, records the error, and enters a clean state.
2. Cashier starts a new sale.
3a. Invalid identifier:
1. System signals error and rejects entry. 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
1. Cashier can enter item category identifier and the quantity.
3-6a: Customer asks Cashier to remove an item from the purchase:
1. Cashier enters item identifier for removal from sale.
2. System displays updated running total.
3-6b. Customer tells Cashier to cancel sale:
1. Cashier cancels sale on System.
3-6c. Cashier suspends the sale:
1. System records sale so that it is available for retrieval on any POS terminal.

4a.The system generated item price is not wanted (e.g., Customer complained about something and is offered a lower price):
1. Cashier enters override price.
2. System presents new price.
5a. System detects failure to communicate with external tax calculation system service:
1. System restarts the service on the POS node, and continues. 1a. System detects that the service does not restart.
1. System signals error.
2. Cashier may manually calculate and enter the tax, or cancel the sale.
5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
1. Cashier signals discount request.
2. Cashier enters Customer identification.
3. System presents discount total, based on discount rules.
5c. Customer says they have credit in their account, to apply to the sale:
1. Cashier signals credit request.
2. Cashier enters Customer identification.
3. Systems applies credit up to price=0, and reduces remaining credit.
6a. Customer says they intended to pay by cash but don't have enough cash:
1a. Customer uses an alternate payment method.
1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System.
7a. Paying by cash:
1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.
7b. Paying by credit: 1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

2a. System detects failure to collaborate with external system:
1. System signals error to Cashier.
2. Cashier asks Customer for alternate payment.
3. System receives payment approval and signals approval to Cashier.
3a. System receives payment denial:
1. System signals denial to Cashier.
2. Cashier asks Customer for alternate payment.
4. System records the credit payment, which includes the payment approval.
5. System presents credit payment signature input mechanism.
6. Cashier asks Customer for a credit payment signature. Customer enters signature.
7c. Paying by check...
7d. Paying by debit...
7e. Customer presents coupons:
1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
1a. Coupon entered is not for any purchased item:
1. System signals error to Cashier.
9a.There are product rebates:
1. System presents the rebate forms and rebate receipts for each item with a rebate.
9b. Customer requests gift receipt (no prices visible): 1. Cashier requests gift receipt and System presents it.

Special Requirements:
- Touch screen Ul on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.

Technology and Data Variations List:
3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.
7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:
- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

a fully-dressed use case can record many requirement details. This example will serve well as a model for many usecase problems.

Writing use cases:

Here we explain the simple ideas of elementary business processes and goals as a framework for identifying the use cases for an application.

Which of these is a valid use case?

. Negotiate a Supplier Contract
. Handle Returns
. Log In

An argument can be made that all of these are use cases *at different levels,* depending on the system boundary, actors, and goals.

Guideline: The EBP Use Case

For requirements analysis for a computer application, focus on use cases at the level **of elementary business processes** (EBPs).

EBP is a term from the business process engineering field,[4] defined as: A task performed by one person 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.
e.g., Approve Credit or Price Order

A common use case mistake is defining many use cases at too low a level; that is, as a single step, subfunction, or subtask within an EBP.

USECASES AND GOALS:

Actors have goals (or needs) and use applications to help satisfy them. Consequently, an EBP-level use case is called a **user** goal-level user case, to emphasize that it serves (or should serve) to fulfill a goal of a user of the system, or the primary actor. And it leads to a recommended procedure:

1. Find the user goals.
2. Define a use case for each.

Rather than asking "What are the use cases?", one starts by asking: "What are your goals?". Thus,

Here is a key idea regarding investigating user goals vs. investigating use cases:

We could ask either:
. "What do you do?" (roughly a use case-oriented question) or,
. "What are your goals?"
Answers to the first question are more likely to reflect current solutions and procedures, and the complications associated with them.
Answers to the second question, especially combined with an investigation to move higher up the goal hierarchy ("what is the goal of that goal?") open up the vision for new and improved solutions, focus on adding business value, and get to the heart of what the stakeholders want from the system under discussion.

investigating user goals for NextGen POS system during a requirements workshop as follows:

System analyst: "What are some of your goals in the context of using a POS system?"
Cashier: "One, to quickly log in. Also, to capture sales."
System analyst: "What do you think is the higher level goal motivating logging in?"

Cashier: "I'm trying to identify myself to the system, so it can validate that I'm allowed to use the system for sales capture and other tasks." **System analyst:** "Higher than that?"
Cashier: "To prevent theft, data corruption, and display of private company information."

"Prevent theft, ..." is higher than a user goal; it may be called an enterprise goal, and is not an EBP.
. Lowering the goal level to "identify myself and be validated" appears closer to the user goal level.

Although "identify myself and be validated" (or "log in") has been eliminated as a user goal, it is a goal at a lower level, called a **subfunction goal**.subgoals that support a user goal. Use cases should only occasionally be written for these subfunction goals.

The number and granularity of use cases influences the time and difficulty to understand, maintain, and manage the requirements.

Goals are usually composite, from the level of an enterprise to intermediate goals while using applications to supporting subfunction goals within applications .

Elements of usecases:

An **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A scenario is a specific sequence of actions and interactions between actors and the system under discussion; it is also called a **use case instance.** It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit card transaction denial.

a **use case** is a collection of related success and failure scenarios that describe actors using a system to support a goal. For example, here is a *casual format* use case that includes some alternate scenarios:

For example, here is a *casual format* use case that includes some alternate scenarios:
Handle Returns
Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...
Alternate Scenarios:
If the credit authorization is reject, inform the customer and ask for an alternate payment method.
If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted). If the system detects failure to communicate with the external tax calculator system, ...
An alternate, but similar definition of a use case is provided by the RUP:
A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor .

The phrasing *"an observable result of value"* is subtle but important, because it stresses the attitude that the system behavior should emphasize providing value to the user.

Sections of usecases:

Preface Elements:
Many optional preface elements are possible. Only place elements at the start which are important to read before the main success scenario. Move extraneous "header" material to the end of the use case.

> **Primary Actor:** The principal actor that calls upon system services to fulfill a goal.

Preconditions and Success Guarantees (Postconditions):
Preconditions state what *must always* be true before beginning a scenario in the use case. Preconditions are *not* tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case that has successfully completed, such as logging in, or the more general "cashier is identified and authenticated." Note that there are conditions that must be true, but are not of practical value to write, such as "the system has power." Preconditions communicate noteworthy ssumptions that the use case writer thinks readers should be alerted to. **Success guarantees** (or **postconditions)** state what must be true on successful completion of the use case.either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders.

> **Preconditions:** Cashier is identified and authenticated.
> **Success Guarantee (Postconditions):** Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario and Steps (or Basic Flow):
This has also been called the "happy path" scenario, or the more prosaic "Basic Flow." It describes the typical success path that satisfies the interests of the stakeholders. Note that it often does *not* include any conditions or branching. Although not wrong or illegal, it is arguably more comprehensible and extend-ible to be very consistent and defer all conditional handling to the Extensions section.

> *Suggestion*
>
> Defer all conditional and branching statements to the Extensions section.

The scenario records the steps, of which there are three kinds:
1. An interaction between actors.[3]
2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

Step one of a use case does not always fall into this classification, but indicates the trigger event that starts the scenario.

It is a common idiom to always capitalize the actors' names for ease of identification. Observe also the idiom that is used to indicate repetition.

Main Success Scenario:
1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...

Cashier repeats steps 3-4 until indicates done.

5. ...

Extensions (or Alternate Flows):

Extensions are very important. They indicate all the other scenarios or branches, both success and failure. Observe in the fully dressed example that the Extensions section was considerably longer and more complex than the Main Success Scenario section; this is common and to be expected. They are also known as "Alternative Flows." In thorough use case writing, the combination of the happy path and extension scenarios should satisfy "nearly" all the interests of the stakeholders. This point is qualified, because some interests may best be captured as non-functional requirements expressed in the Supplementary Specification rather than the use cases. Extension scenarios are branches from the main success scenario, and so can be notated with respect to it. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:

3a. Invalid identifier:
   1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers): 1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: Write the condition as something that can be *detected* by the system or an actor. To contrast:

5a. System detects failure to communicate with external tax calculation system service:

5a. External tax calculation system not working:

The former style is preferred because this is something the system can detect; the latter is an inference.

Extension handling can be *summarized* in one step, or include a sequence, as in this example, which also illustrates notation to indicate that a condition can arise within a range of steps:

3-6a: Customer asks Cashier to remove an item from the purchase:
   1. Cashier enters the item identifier for removal from the sale.
   2. System displays updated running total.

At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system).

Special Requirements:
If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:
- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 2 and 6.

Technology and Data Variations List:
Often there are technical variations in *how* something must be done, but not what, and it is noteworthy to record this in the use case. A common example is a technical constraint imposed by a stakeholder regarding input or output technologies.

For example, a stakeholder might say, "The POS system must support credit account input using a card reader and the keyboard." Note that these are examples of early design decisions or constraints; in general, it is skillful to avoid premature design decisions, but sometimes they are obvious or unavoidable, especially concerning input/output technologies. It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology. This list is the place to record such variations. It is also useful to record variations in the data that may be captured at a particular step.

Technology and Data Variations List:
3a. Item identifier entered by laser scanner or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.
7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Steps in writing usecases:

Use cases are defined to satisfy the user goals of the primary actors. Hence, the basic procedure for writing usecases is:

1.Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2. Identify the primary actors.those that have user goals fulfilled through using services of the system.
3. For each, identify their user goals. Raise them to the highest user goal level that satisfies the EBP guideline.
4. Define use cases that satisfy user goals; name them according to their goal. Usually, user goal-level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

1: Choosing the System Boundary

For example consider the POS system under design;

everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on.
Once the external actors are identified, the boundary becomes clearer. For example, is the complete responsibility for payment authorization within the system boundary? No, there is an external payment authorization service actor.

Steps 2 and 3: Finding Primary Actors and Goals

in a requirements workshop,we use brainstorming methods to identify actors and golas.
: Emphasize brainstorming the primary actors first, as this sets up the framework for further investigation.

Questions to Find Actors and Goals:

In addition to obvious primary actors and user goals, the following questions help identify others that may be missed:
Who starts and stops the system?
Who does user and security management?
Is there a monitoring process that restarts the system if it fails?
How are software updates handled?
Push or pull update?
Who does system administration?
Is "time" an actor because the system does something in response to a time event?
Who evaluates system activity or performance?
Who evaluates logs? Are they remotely retrieved?

Primary and Supporting Actors:

Recall that primary actors have user goals fulfilled through using services of the system. They call upon the system to help them. This is in contrast to *supportingactors,* which provide services to the system under design. For now, the focus is on finding the primary actors, not the upporting ones.

The Actor-Goal List

Record the primary actors and their user goals in an actor-goal list. In terms of UP artifacts it should be a section in the Vision artefact.

For example:

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out …	System Administra-tor	add users modify users delete users manage security manage system tables …
Manager	start up shut down …	Sales Activ-ity System	analyze sales and per-formance data
…	…	…	…

Primary Actor and User Goals Depend on System Boundary

Why is the cashier, and not the customer, the primary actor in the use case *Process Sale?* Why doesn't the customer appear in the actor-goal list?The answer depends on the system boundary of the system under design, as illustrated in the following fig.

If viewing the enterprise or checkout service as an aggregate system, the customer *is* a primary actor, with the goal of getting goods or services and leaving. However, from the viewpoint of just the POS system the cashier is the primary actor.

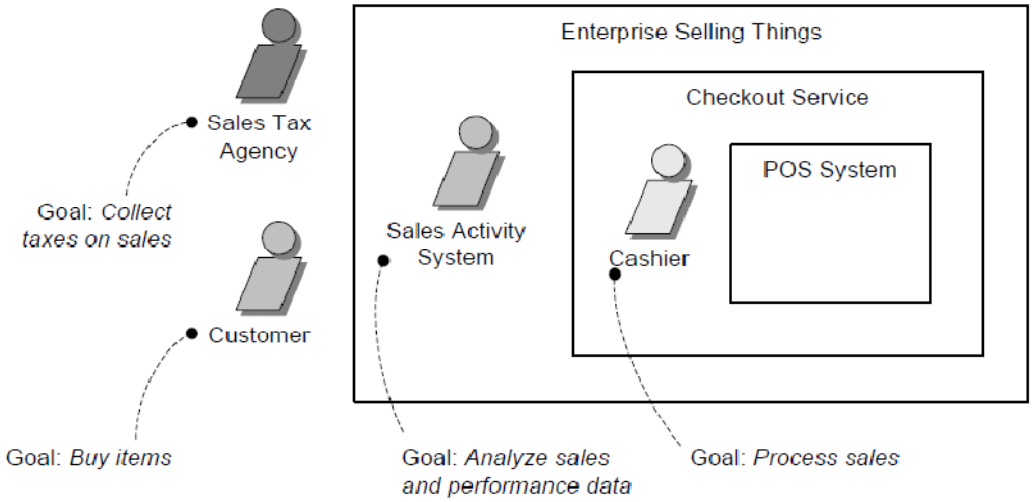


Figure 6.1 Primary actors and goals at different system boundaries.

**Actors and Goals depend on Event Analysis

Another approach finding actors, goals, and use cases is to identify external events. What are they, where from, and why? Often, a group of events belong to the same EBP-level goal or use case. For example:

External Event	From Actor	Goal
enter sale line item	Cashier	process a sale
enter payment	Cashier or Customer	process a sale
…		

4: Define Use Cases

In general, define one EBP-level use case for each user goal. Name the use case similar to the user goal.for example, Goal: process a sale; Use Case: *Process Sale.*

Also, name use cases starting with a verb.

A common exception to one use case per goal is to collapse CRUD (create, retrieve, update, delete) separate goals into one CRUD use case. For example, the goals "edit user," "delete user," and so forth are all satisfied by

the *Manage Users* use case. "Define use cases" has several levels of effort, ranging from a few minutes to simply record names, up to weeks to write fully dressed versions.

USECASE WRITING STYLE:

Usecases are written essential UI-Free style.

Essential style writing style **essential** when it avoids UI details and focuses on the real user intent. In an essential writing style, the narrative is expressed at the level of the user's *intentions* and system's *responsibilities* rather than their concrete actions. They remain free of technology and mechanism details, especially those related to the UI.

In contrast, there is a **concrete use case** style. In this style, user interface decisions are embedded in the use case text. The text may even show window screen shots, discuss window navigation, GUI widget manipulation and so forth.
For example:

1. Adminstrator enters ID and password in dialog box (see Picture 3).
2. System authenticates Adminstrator.
3. System displays the "edit users" window (see Picture 4).
4. . . .

These concrete use cases may be useful as an aid to concrete or detailed GUI design work during a later step, but they are not suitable during the early requirements analysis work. During early requirements work, "keep the user interface out. focus on intent."

TYPES OF ACTORS:

An actor is anything with behavior, including the system under discussion. itself when it calls upon the services of other systems. Primary and supporting actors will appear in the action steps of the use case text. Actors are not only roles played by people, but organizations, software, and machines. There are three kinds of external actors.
. Primary actor.has user goals fulfilled through using services.
For example, the cashier.
. Supporting actor.provides a service (for example, information) to the System Under discussion.The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
. Offstage actor.has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.

Use Case Diagrams:

The UML provides use case diagram notation to illustrate the names of use cases can actors, and the relationships between them .

a simple use case diagram provides a brief visual context diagram for the system, illustrating the external actors and how they use the system.

A use case diagram is an excellent picture of the system context; it makes a good **context diagram,** that is, showing the boundary of a system, what lies outside of it, and how it gets used. It serves as a communication tool that summarizes the behavior of a system and its actors.

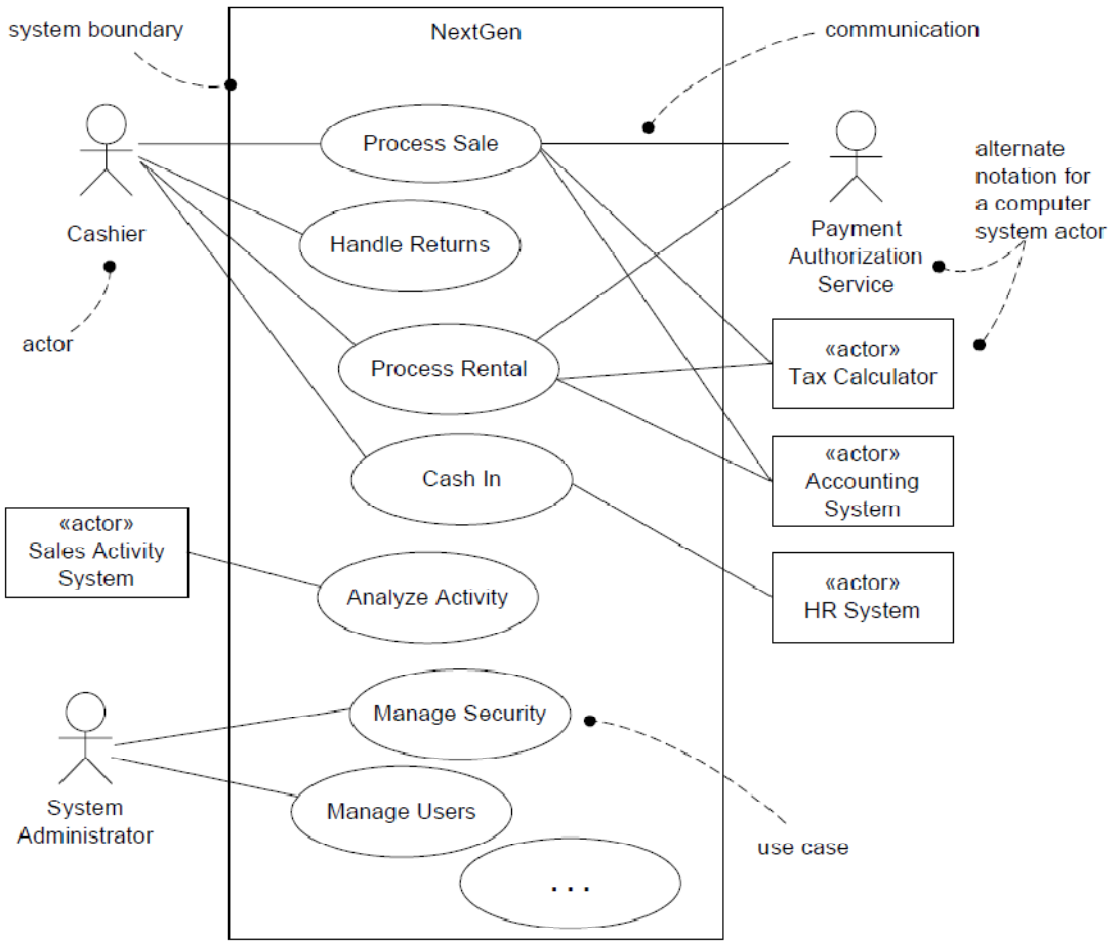


Figure 6.2 Partial use case context diagram.

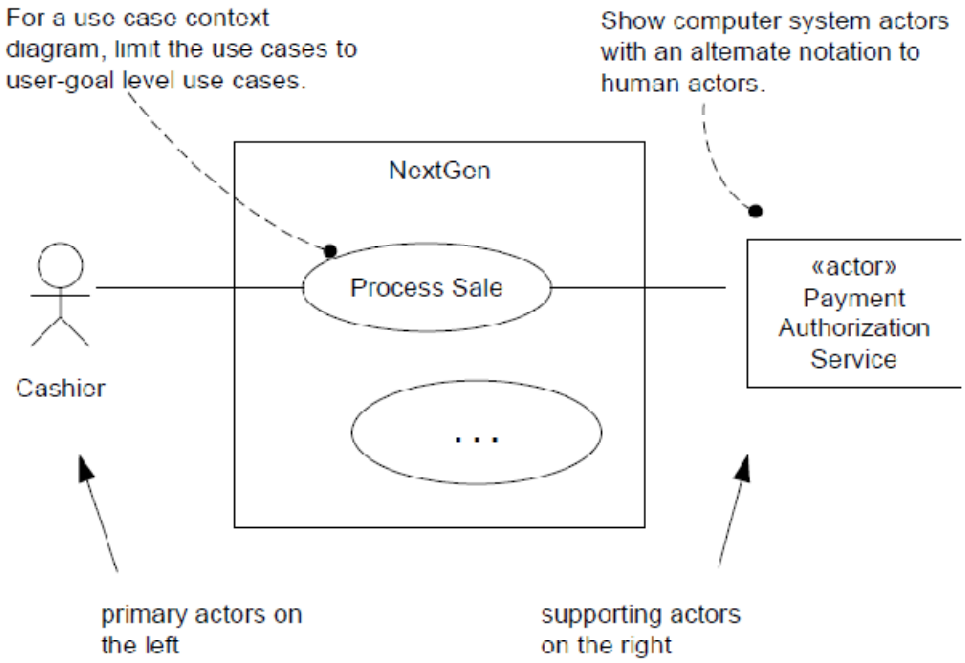
Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text.

Suggestion

Draw a simple use case diagram in conjunction with an actor-goal list.

Diagramming Suggestions:

Consider the following digaram. Notice the actor box with the symbol «actor». This symbol is called a UML **stereotype;** it is a mechanism to categorize an element in some way. A stereotype name is surrounded by guillemets symbols.special single-character brackets (not "«" and "»") most widely known by their use in French typography to indicate a quote.



An alternate notation for external computer system actors is illustrated in the following fig:

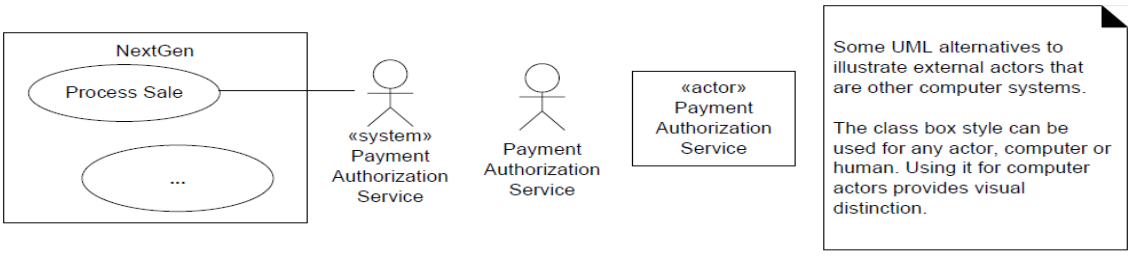


Figure 6.4 Alternate actor notation.

A Caution on Over-Diagramming:

The important use case work is to write text, not diagram or focus on use case relationships. If an organization is spending many hours (or worse, days) working on a use case diagram and discussing use case relationships, rather than focussing on writing text, relative effort has been misplaced.

Feature Lists/Function Lists:

Requirements in Context and Low-Level Feature Lists:

One idea behind use cases is to replace detailed, low-level feature lists (which were common in traditional requirements methods) with use cases (with some exceptions). These lists tended to look as follows, usually grouped into functional areas:

ID	Feature
FEAT1 .9	The system shall accept entry of item identifiers.

ID	Feature
…	…
FEAT2.4	The system shall log credit payments to the accounts receivable system.

Such detailed lists of low-level features are somewhat usable. However, the complete list is not a half-page; more likely it is dozens or a hundred pages. This leads to some drawbacks, which use cases help address. These include: . Long, detailed function lists do not relate the requirements in a cohesive context;
In contrast, use cases place the requirements in the context of the stories and goals of using the system.
. If both use case and detailed feature lists are used, there is duplication.

High-Level System Feature Lists:

It is common and useful to summarize system functionality with a terse, high-level feature list called system features in a Vision document. In contrast to 100 pages of low-level, detailed features, a system features list tends to include only a few dozen items. The list provides a very succinct summary of system functionality, independent of the use case view. For example:

Summary of System Features:

. sales capture
. payment authorization (credit, debit, check)
. system administration for users, security, code and constants tables, and so on
. automatic offline sales processing when external components fail
. real-time transactions, based on industry standards, with third-party systems, including inventory,
accounting, human resources, tax calculators, and payment authorization services
. definition and execution of customized "pluggable" business rules at fixed, common points in the
processing scenarios

When Are Detailed Feature Lists Appropriate?

Sometimes use cases do not really fit; some applications call out for a feature-driven viewpoint. For example, application servers, database products, and other middleware or back-end systems need to be primarily considered and evolved in terms *of features* ("We need XML support in the next release"). Use cases are not a natural fit for these applications.

Use cases in the UP context and UP artifacts:

Use cases are vital and central to the UP, which encourages **use-case driven development.** This implies:
. Requirements are primarily recorded in use cases (the Use-Case Model); other requirements techniques (such as functions lists) are secondary, if used at all.
. Use cases are an important part of iterative planning. The work of an iteration is.in part.defined by choosing some use case scenarios, or entire use cases. And use cases are a key input to estimation.
. **Use-case realizations** drive the design. That is, the team designs collaborating objects and subsystems in order to perform or realize the use cases.
. Use cases often influence the organization of user manuals.

The UP distinguishes between system and business use cases. **System usecases** are what have been examined in this chapter, such as *Process Sale.* They are created in the Requirements discipline, and are part of the Use-Case Model.

Business use cases are less commonly written. If done, they are created in the Business Modeling discipline as part of a large-scale business process reengi-neering effort, or to help understand the context of a new system in the business. They describe a sequence of actions of a business as a whole to fulfill a goal of a **business actor** (an actor in the business environment, such as a customer or supplier). For example, in a restaurant, one business use case is *Serve a Meal.*

Use Cases and Requirements Specification Across the Iterations:

It explains The timing and level of effort of requirements specification across the iterations. Following table shows the UP strategy of how requirements are developed.

The technical team starts building the production core of the system when only perhaps 10% of the requirements are detailed, and in fact, there is a deliberate delay in continuing with concerted requirements work until near the end of the first elaboration iteration.

This is the key difference in iterative development to a waterfall process: Production-quality development of the core of a system starts quickly, long before all the requirements are known.

Discipline	Artifact	Comments and Level of Requirements Effort				
		Incep 1 week	Elab 1 4 weeks	Elab 2 4 weeks	Elab 3 3 weeks	Elab 4 3 weeks
Requirements	Use-Case Model	2-day requirements workshop. Most use cases identified by name, and summarized in a short paragraph. Only *10%* written in detail.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 30% of the use cases in detail.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 50% of the use cases in detail.	Repeat, complete 707 of all use cases in detail.	Repeal with the goal of 80-90% of the use cases clarified and written in detail. Only a small portion of these have been built in elaboration; the remainder are done in construction.
Design	Design Model	none	Design for a small set of high-risk architecturally significant requirements.	repeat	repeat	Repeat. The high risk and architecturally significant aspects should now be stabilized.
Implementation	Implementation Model (code, etc.)	none	Implement these.	Repeat. 5% of the final system is built.	Repeat. 10% of the final system is built.	Repeat. 15% of the final system is built.
Project Management	SW Development Plan	Very vague estimate of total effort.	Estimate starts to take shape.	a little better...	a little bettor...	Overall project duration, major milestones, effort, and cost estimates can now be rationally committed to.

Table 6.1 Sample requirements effort across the early iterations; this is not a recipe.

Observe that near the end of the first iteration of elaboration, there is a second requirements workshop, during which perhaps 30% of the use cases are written in detail. This staggered requirements analysis benefits from the feedback of having built a little of the core software. The feedback includes user evaluation, testing, and improved "knowing what we don't know." That is, the act of building software rapidly surfaces assumptions and questions that need clarification.

Timing of UP Artifact Creation:

Following table illustrates some UP artifacts, and an example of their start and refinement schedule. The Use-Case Model is started in inception, with perhaps only 10% of the use cases written in any detail. The majority are incrementally written over the iterations of the elaboration phase, so that by the end of elaboration, a large body of detailed use cases and other requirements (in the Supplementary Specification) are written, providing a realistic basis for estimation through to the end of the project.

Discipline	Artifact Iteration->	Incep. I1	Elab. El. .En	Const. CL..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	*Use-Case Model*	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 6.2 Sample UP artifacts and timing. s - start; r - refine

Use Cases Within Inception:

Not all use cases are written in their fully dressed format during the inception phase. Rather, suppose there is a two-day requirements workshop during the early NextGen investigation. The earlier part of the day is spent identifying goals and stakeholders, and speculating what is in and out of scope of the
project. An actor-goal-use case table is written and displayed with the computer projector. A use case context diagram is started. After a few hours, perhaps 20 user goals (and thus, user goal level use cases) are identified, including *Process Sale, Handle Returns,* and so on. Most of the interesting, complex, or risky use cases are written in brief format; each averaging around two minutes to write.

The team starts to form a high-level picture of the system's functionality. After this, 10% to 20% of the use cases that represent core complex functions, or which are especially risky in some dimension, are rewritten in a fully dressed format; the team investigates a little deeper to better comprehend the magnitude, complexities, and hidden demons of the project, through a small sample of interesting use cases. Perhaps this means two use cases: *Process Sale* and *Handle Returns*.
A requirements management tool that integrates with a word processor is used for the writing, and the work is displayed via a projector while the team collaborates on the analysis and writing. The *Stakeholders and Interests* lists are written for these use cases, to discover more subtle (and perhaps costly) functional and key non-function requirements.or system qualities.such as for reliability or throughput.The analysis goal is not to exhaustively complete the use cases, but spend a few hours to obtain some insight. The project sponsor needs to decide if the project is worth significant investigation (that is, the elaboration phase). The inception work is not meant to do that investigation, but to obtain low-fidelity (and admittedly error-prone) insights regarding scope, risk, effort, technical feasibility, and business case, in order to decide to move forward, where to start if they do, or if to stop. Perhaps the NextGen project inception step lasts five days. The combination of the two day requirements workshop and its brief use case analysis, and other investigation during the week, lead to the decision to continue on to an elaboration step for the system.

Use Cases Within Elaboration:

This is a phase of multiple timeboxed iterations (for example, four iterations) in which risky, high-value, or architecturally significant parts of the system are incrementally built, and the "majority" of requirements identified and clarified. The feedback from the concrete steps of programming influences and informs
the team's understanding of the requirements, which are iteratively and adap-tively refined. Perhaps there is a two-day requirements workshop in each iteration.four workshops. However, not all use cases are investigated in each workshop. They are prioritized; early workshops focus on a subset of the most important use cases.
Each subsequent short workshop is a time to adapt and refine the vision of the core requirements, which will be unstable in early iterations, and stabilizing in later ones. Thus, there is an iterative interplay between requirements discovery, and building parts of the software.

During each requirements workshop, the user goals and use case list are refined. More of the use cases are written, and rewritten, in their fully dressed format. By the end of elaboration, "80-90%" of the use cases are written in detail. For the POS system with 20 user goal level use cases, 15 or more of the most complex and risky should be investigated, written, and rewritten in a fully dressed format. Note that elaboration involves programming parts of the system. At the end of this step, the NextGen team should not only have a better definition of the use cases, but some quality executable software.

Use Cases Within Construction:

The construction step is composed of timeboxed iterations (for example, 20 iterations of two weeks each) that focus on completing the system, once the risky and core unstable issues have settled down in elaboration. There will still be some minor use case writing and perhaps requirements workshops, but much less so than in elaboration. By this step, the majority of core functional and non-functional requirements should have iteratively and adaptively stabilized. That does not mean to imply requirements are frozen or investigation finished, but the degree of change is much lower.

UP Artifacts and Process Context:

As illustrated in Figure 6.5, use cases influence many UP artifacts.

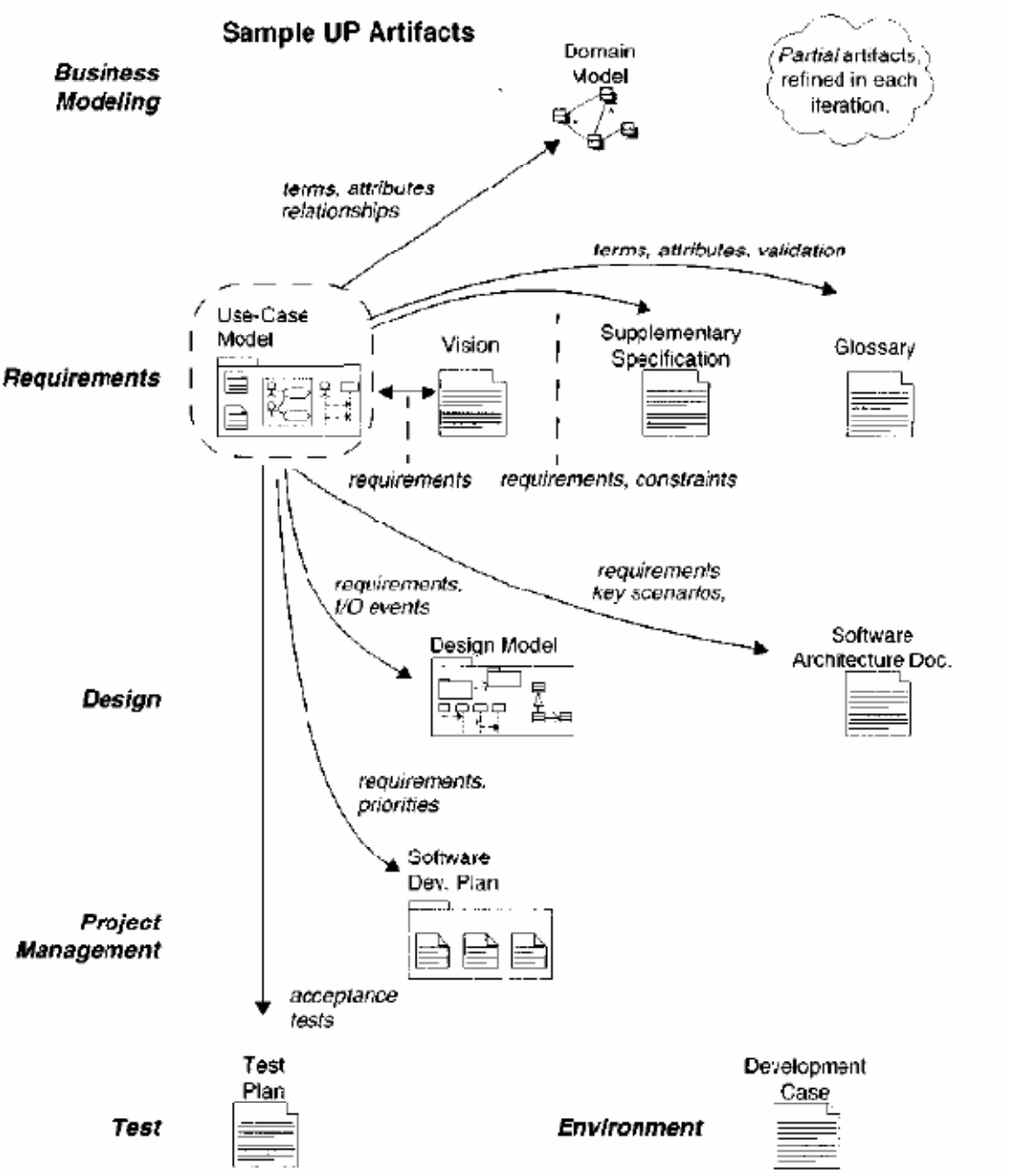


Figure 6.5 Sample UP artifact influence.

Identifying additional requirements:

It is not sufficient to write use cases. There are other kinds of requirements that need to be identified, such as documentation, packaging, supportability, licensing, and so forth. These are captured in the **Supplementary Specification.**

The **Glossary** captures terms and definitions; it can also play the role of a data dictionary.

The Vision summarizes the "vision" of the project. It serves to tersely communicate the big ideas regarding why the project was proposed, what the problems are, who the stakeholders are, what they need, and what the proposed solution looks like.

Supplementary Specification:

The Supplementary Specification captures other requirements, information, and constraints not easily captured in the use cases or Glossary, including system-wide "URPS+" quality attributes or requirements. Note that requirements specific to a use case can (and probably should) be first written with the use case, in a *Special Requirements* section, but some prefer to also consolidate all of them in the Supplementary Specification.. Elements of the Supplementary Specification could include:
. FURPS+ requirements.functionality, usability, reliability, performance, and supportability
. reports
. hardware and software constraints (operating and networking systems, ...)
. development constraints (for example, process or development tools)
. other design and implementation constraints
. internationalization concerns (units, languages, ...)
. documentation (user, installation, administration) and help
. licensing and other legal concerns
. packaging
. standards (technical, safety, quality)
. physical environment concerns (for example, heat or vibration)
. operational concerns (for example, how do errors get handled, or how often to do backups?)
. domain or business rules
. information in domains of interest (for example, what is the entire cycle of credit payment handling?)
Constraints are not behaviors, but some other kind of restriction on the design or project. They are also requirements, but are commonly called "constraints" to emphasize their *restrictive* influence.
For example:
Must use Oracle (we have a licensing arrangement with them).
Must run on Linux (it will lower cost).

Quality Attributes
Some requirements are called **quality attributes** (or "-ilities") of a system. These include usability, reliability, and so forth. Note that these refer to the qualities of the system, not that these attributes are necessarily of high quality (the word is overloaded in English). For

example, the quality of support-ability might deliberately be chosen to be low if the product is not intended to serve a long-term purpose. They are of two types:
1. Observable at execution (functionality, usability, reliability, performance, ...)
2. Not observable at execution (supportability, testability, ...)
Functionality is specified in the use cases, as are other quality attributes related to specific use cases (for example, the performance qualities in the *Process Sale* use case).
Other system-wide FURPS+ quality attributes are described in the Supplementary Specification.
Although functionality is a valid quality attribute, in common usage, the term "quality attribute" is most often meant to imply "qualities of the system other than functionality." Herein, the term is likewise used. This is not exactly the same as non-functional requirements, which is a broader term including *everything*
but functioality (for example, packaging and licensing). When we put on our "architect hat," the system-wide quality attributes (and thus the Supplementary Specification where one records them) are especially
interesting because.as will be introduced in Chapter 32.architectural analysis and design are largely concerned with the identification and resolution of the quality attributes in the context of the functional requirements. For example, suppose one of the quality attributes is that the NextGen system must be quite fault-tolerant when remote services fail. From an architectural viewpoint, that will have an overarching influence on large-scale design decisions. Quality attributes have interdependencies and involve trade-offs. As a simple example in the POS, "very reliable (fault-tolerant)" and "easy to test" are in some opposition, because there are many subtle ways a distributed system can fail.

Domain (Business) Rules:

Domain rules dictate how a domain or business may operate. They are not requirements of any one application, although an application's requirements are often by domain rules. Company policies, physical laws, and government laws are common domain rules.
They are commonly called **business rules,** which is the most common type, but that term is limited, as some software applications are for non-business problems, such as weather simulation or military logistics. A weather simulation has "domain rules" that influence the application requirements, related to physical laws and relationships. It is often useful to identify and record those domain rules that influence the requirements, usually realized in the use cases, because they can clarify incomplete or ambiguous use case content. For example, in the NextGen POS, if someone asks if the *Process Sale* use case should be written with an alternative to allow credit payments without signature capture, there is a business rule (RULE1) that clarifies whether this will not be allowed by any credit authorization company.
Note: Rules are not application requirements. Do not record system features as rules. They describe the constraints and behaviors of how the domain works, not the application.

Information in Domains of Interest:

It is often valuable for a subject matter expert to write (or provide URLs to) some explanation of domains related to the new software system (sales and accounting, the geophysics of underground oil/water/gas flows, ...), to provide context and deeper insight for the development team. It may contain pointers to important literature or experts, formulas, laws, or other references. For example, the arcana of UPC and EAN coding schemes, and bar code symbology, must be understood to some degree by the NextGen team.

NextGen Example: (Partial) Supplementary Specification

Supplementary Specification

Revision History

Version	Date	Description	Author
Inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Introduction

This document is the repository of all NextGen POS requirements not captured in the use cases.

Functionality

(Functionality common across many use cases)

Logging and Error Handling Log all errors to persistent storage. ***Pluggable Business Rules***

At various scenario points of several use cases (to be defined) support the ability to customize the functionality of the system with a set of arbitrary rules that execute at that point or event.

Security

All usage requires user authentication.

Usability

Human Factors

The customer will be able to see a large-monitor display of the POS.Therefore:

. Text should be easily visible from 1 meter.

. Avoid colors associated with common forms of color blindness.

Speed, ease, and error-free processing are paramount in sales processing, as the buyer wishes to leave quickly, or they perceive the purchasing experience (and seller) as less positive.

The cashier is often looking at the customer or items, not the computer display. Therefore, signals and warnings should be conveyed with sound rather than only via graphics.

Reliability

Recoverability

If there is failure to use external services (payment authorizer, accounting system, ...) try to solve with a local solution (e.g., store and forward) in order to still complete a sale. Much more analysis is needed here...

Performance

As mentioned under human factors, buyers want to complete sales processing very quickly. One potential bottleneck is external payment authorization. Our goal is to achieve authorization in less than 1 minute, 90% of the time.

Supportability

Adaptability

Different customers of the NextGen POS have unique business rule and processing needs while processing
a sale. Therefore, at several defined points in the scenario (for example, when a new sale is initiated,
when a new line item is added) pluggable business rule will be enabled.

Configurability

Different customers desire varying network configurations for their POS systems, such as thick versus
thin clients, two-tier versus N-tier physical layers, and so forth. In addition, they desire the ability to modify
these configurations, to reflect their changing business and performance needs. Therefore, the system
will be somewhat configurable to reflect these needs. Much more analysis is needed in this area to discover
the areas and degree of flexibility, and the effort to achieve it.

Implementation Constraints

NextGen leadership insists on a Java technologies solution, predicting this will improve long-term porting
and supportability, in addition to ease of development.

Purchased Components

. Tax calculator. Must support pluggable calculators for different countries.

Free Open Source Components

In general, we recommend maximizing the use of free Java technology open source components on this
project.
Although it is premature to definitively design and choose components, we suggest the following as likely
candidates:
. JLog logging framework
. .

Interfaces

Noteworthy Hardware and Interfaces

. Touch screen monitor (this is perceived by operating systems as a regular monitor, and the touch gestures as mouse events)
. Barcode laser scanner (these normally attach to a special keyboard, and the scanned input is perceived in software as keystrokes)
. Receipt printer
. Credit/debit card reader
. Signature reader (but not in release 1)

Software Interfaces

For most external collaborating systems (tax calculator, accounting, inventory, ...) we need to be able to plug in varying systems and thus varying interfaces.

Domain (Business) Rules

ID Rule Changeability Source
RULE1 Signature required for credit payments. Buyer "signature" will continue to be required, but within 2 years most of our customers want signature capture on a digital capture device, and within 5 years we expect there to be demand for support of the new unique digital code "signature" now supported by USA law. The policy of virtually all credit authorization companies.
RULE2 Tax rules. Sales require added taxes. See government statutes for current details.

High. Tax laws change annually, at all government levels. law
RULE3 Credit payment reversals may only be paid as a credit to the buyer's credit account, not as cash. Low credit authorization company policy RULE4 Purchaser discount rules. Examples: Employee. 20% off. Preferred Customer. 1 0% off. Senior. 15% off. High. Each retailer uses different rules. Retailer policy.

Vision:

Are We Solving the Same Problem? The Right Problem?.

The Problem Statement
During early requirements work in the inception phase, collaborate to define a terse problem statement; it will reduce the likelihood that stakeholders are trying to solve slightly different problems, and is usually quickly created. Occasion-ally, the effort reveals fundamental differences of opinion in what the parties are trying to achieve. Rather than plain prose, a table format offered in the RUP templates for problem.
statements is: The problem of . affects .the impact of which is . a successful solution would be .

The Key High-Level Goals and Problems of the Stakeholders:
This table summarizes the goals and problems at a higher level than task level use cases, and reveals important nonfunctional and quality goals that may belong to one use case or span many, such as:
. We need fault-tolerant sales processing. . We need the ability to customize the business rules.

What Are the Root Problems and Goals?
It is common for stakeholders to express their goals in terms of envisioned solutions, such as: "We need a full-time programmer to customize the business rules as we change them." The solutions are sometimes perceptive, because they understand their problem domain and options well. But sometimes stakeholder jump to solutions that are not the most appropriate or do not address the root
underlying major problems.

Thus, the system analyst needs to investigate the problem and goal chain.as discussed in the previous chapter on use cases and goals.in order' to learn the underlying problems, and their relative importance and impact, in order to prioritize and solve the most egregious concerns with a skillful solution.

Group Idea Facilitation Methods
Although outside the scope of this discussion, it is especially during activities such as high-level problem definition and goal identification that creative, investigative group work occurs. Here are some useful group facilitation techniques to discover root problems and goals, and support idea generation and pri-oritization: mind mapping, fishbone diagrams, pareto diagrams, brainstorming, multi-voting, dot voting, nominal group process, brainwriting, and affinity grouping. Check them out on the web. I prefer to apply several of these during the same workshop, to discover common problems and requirements from different angles.

System Features.Functional Requirements:

Use cases are not necessarily the only way one needs to express functional requirements for the following reasons: . They are detailed. Stakeholders often want a short summary that identifies
the most noteworthy functions. . What about simply listing the use case names *(Process Sale, Handle Returns, ...)* to summarize the functionality? First, the list may still be too long. Also, the names can hide interesting functionality stakeholders really want to know about; that is, the level of granularity can obscure noteworthy
functions. For example, suppose that the description of automated payment authorization functionality is embedded in the *Process Sale* use case. A reader of a list of use case names cannot tell if the system will do payment authorization. Furthermore, one may wish to group a set of use cases into one feature (for brevity), such as *System administration for users, security, code and constants tables, and so forth.* . Some noteworthy functionality is naturally expressed as short statements that do not conveniently map to use case names or Elementary Business Process-level goals. It may span or be orthogonal to the use cases. For example, during the first NextGen requirements workshop, someone might say "The system should be able to do transactions with existing third-party accounting, inventory, and tax calculation systems." This statement of func tionality does not represent one particular use case, but is a comfortable and succinct way to express, record, and communicate features.) As a stronger variation of the last point, some applications call out primarily for a description of functionality as features; use cases are not a natural fit. This is common, for example, with middleware
products such as application servers.use cases are not really motivated. Suppose the team is considering their next release. During a requirements discussion, people (such as marketing) will say, "The next version needs EJB 2.0 entity bean support." The requirements are primarily conceived in terms of a list of features, not use cases.

Therefore, an alternative, a complementary way to express system functions is with **features,** or more specifically in this context, **system features,** which are high-level, terse statements summarizing system functions. More formally, in the **UP, a system feature** is "an externally observable service provided by the system which directly fulfills a stakeholder need" .Features are things a system can *do.* They should pass this linguistic test: *The system shall do* <feature X>.

For example:

The system shall do payment authorization.

Recall that the Vision may be used as a formal or informal contract between development and business. System features are a mechanism to summarize in this contract what the system will *do.* This is complementary to the use cases, as the features are terse. Features are to be contrasted with various kinds of non-functional requirements and constraints, such as: *"The system must run on Linux, must have 24/7 availability, and must have a touch-screen interface."* Note that these fail the linguistic test.

At times, the admonition "an externally observable service..." is difficult to decide upon. For example, should the following be a system feature: *The system shall do transactions with third-party accounting, inventory, human resource, and tax calculation systems.* It is a kind of behavior, and probably noteworthy to the stakeholders, but the collaboration itself may not be externally visible, depending on your time frame, and how close and where you look. Include it.fine-grained classification questions are seldom worth the worry.

Finally, note that most system features will find detailed expression in use case text.

Notation and Organization:

First and foremost, short high-level descriptions are important. One should be able to read the system features list quickly.

It is not necessary to include the canonical "The system shall do..." or a variant phrase, although it is common.

Here is a features example at a high level, for a large multi-system project of which the POS is just one element:

The major features include:

. POS services

. Inventory management

. Web-based shopping

It is common to organize a two-level hierarchy of system features. But in the Vision document more than two levels leads to excessive detail; the point of system features in the Vision is to summarize the functionality, not decompose it into a long list of fine-grained elements. A reasonable example in terms of detail:

The major features include: .

. POS services:

) sales capture

) payment authorization

) . . .

.Inventory management:

) automatic reordering

) . . .

Sometimes, these second level features are essentially equivalent to use case names (or user-level goals), but that is not required; features are an alternative way to summarize functionality. Nevertheless, most system features will find detailed expression in the use cases. How many system features should the Vision contain?

Suggestion

A Vision with less than 50 features is desirable. If more, consider grouping and abstracting the features.

Other Requirements in the Vision:

In the Vision, system features briefly summarize functional requirements expressed in detail in the use cases. Likewise, the Vision *can* summarize other requirements (for example, reliability and usability) that are detailed in the *Special Requirements* sections of use cases, and in the Supplementary Specification (SS). However, there is some risk of unhelpful duplication. For example, the RUP product provides templates for the Vision and SS that contain identical or similar sections for other requirements such as usability, reliability, performance, and so forth. Such duplication is inevitably awkward to maintain. Fur-thermore, the level of detail for similar sections (for example, performance) in the Vision and the SS needs to be quite similar to be meaningful; that is, "essential" and "detailed" other requirement descriptions tend to be much the same,

Suggestion

For other requirements, avoid their duplication or near-duplication in both the Vision and Supplementary Specification (SS).and in use cases. Rather, record them only in the SS or uses cases (if use case specific). In the Vision, direct the reader to these for the other requirements.

This is a minor documentation nuance on the standard RUP templates that may reduce complications. If one prefers the standard template approach, that is also fine. *Vision, Features, or Use Cases.Which First?*
It is not useful to be rigid about the order of some artifacts. While collaborating to create different requirements artifacts, a synergy emerges in which working on one influences and helps clarify another. Nevertheless, a suggested sequence is:
1. Write a brief first draft of the Vision.
2. Identify user goals and the supporting use cases.
3. Write some use cases and start the Supplementary Specification.
4. Refine the Vision, summarizing information from these.

NextGen Example: (Partial) Vision

Vision

Revision History

Version Date Description Author
inception draft Jan 10, 2031 First draft. To be refined primarily during elaboration. Craig Larman

The analysis in this example is illustrative, but fictitious.

Introduction

We envision a next generation fault-tolerant point-of-sale (POS) application, NextGen POS, with the
flexibility to support varying customer business rules, multiple terminal and user interface mechanisms, and integration with multiple third-party supporting systems.

Positioning

Business Opportunity

Existing POS products are not adaptable to the customer's business, in terms of varying business
rules and varying network designs (for example, thin client or not; 2, 3, or 4 tier architectures). In
addition, they do not scale well as terminals and business increase. And, none can work in either
on-line or off-line mode, dynamically adapting depending on failures. None easily integrate with
many third-party systems. None allow for new terminal technologies such as mobile PDAs. There is
marketplace dissatisfaction with this inflexible state of affairs, and demand for a POS that rectifies
this.

Problem Statement

Traditional POS systems are inflexible, fault intolerant, and difficult to integrate with third-party
systems. This leads to problems in timely sales processing, instituting improved processes that
don't match the software, and accurate and timely accounting and inventory data to support measurement and planning, among other concerns. This affects cashiers, store managers, system
administrators, and corporate management.

Product Position Statement

.Terse *summary of who the system is for, its outstanding features, and what differentiates it from the competition.*

Alternatives and Competition...

Understand who the players are, and their problems.

Stakeholder Descriptions

Market Demographics...

Stakeholder (Non-User) Summary... User Summary...

Consolidate input from the Actor and Goals List, and the Stakeholder Interests section of the use cases.

A one day requirements workshop with subject matter experts and other stakeholders, and surveys at several retail outlets led to identification of the following key goals and problems: High-Level Goal Priority Problems and Concerns Current Solutions Fast, robust, integrated sales processing high Reduced speed as load increases.Loss of sales processing capability if components fail. Lack of up-to-date and accurate information from accounting and other systems due to non-integration with existing accounting, inventory, and HR systems. Leads to difficulties in measuring and planning. Inability to customize business rules to unique business requirements. Difficulty in adding new terminal or user interface types (for example, mobile PDAs). Existing POS products provide basic sales processing, but do not address these problems.

. . . .

This may be the Actor-Goal List created during use-case modeling, or a more terse summary.

User-Level Goals

The users (and external systems) need a system to fulfill these goals:

. *Cashier:* process sales, handle returns, cash in, cash out

. *System administrator:* manage users, manage security, manage system tables

. *Manager:* start up, shut down

. *Sales activity system:* analyze sales data

. .

User Environment...

Product Overview

Product Perspective

The NextGen POS will usually reside in stores; if mobile terminals are used, they will be in close proximity to the store network, either inside or close outside. It will provide services to users, and collaborate with other systems, as indicated in Figure Vision-1.

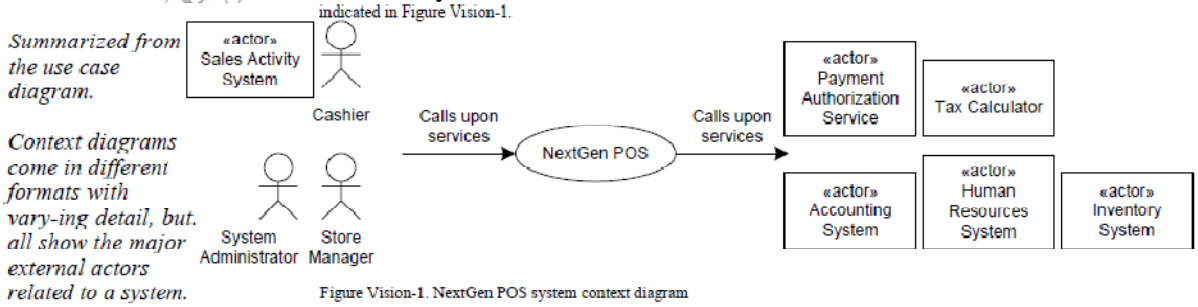


Figure Vision-1. NextGen POS system context diagram

Summary of Benefits

Similar to the Actor-Goal list, this table relates goals, benefits, and solutions, but at a higher level not solely related to use cases. It summarizes the value and differentiating

qualities of the product.As discussed below, system features are a terse format to summarize functionality. Supporting Feature Stakeholder Benefit Functionally, the system will provide all the common services a sales organization requires, including sales capture, payment authorization, return handling, and so forth. Automated, fast point-of-sale services. Automatic detection of failures, switching to local offline processing for unavailable services. Continued sales processing when external components fail. Pluggable business rules at various scenario points during sales processing. Flexible business logic configuration. Real-time transactions with third-party systems, using industry standard protocols. Timely, accurate sales, accounting, and inventory information, to support measuring and planning.
. .

Assumptions and Dependencies...
Cost and Pricing... Licensing and Installation...

Summary of System Features

. sales capture
. payment authorization (credit, debit, check)
. system administration for users, security, code and constants tables, and so forth.
. automatic offline sales processing when external components fail.
. real-time transactions, based on industry standards, with third-party systems, including inventory,
accounting, human resources, tax calculators, and payment authorization services . definition and execution of customized "pluggable" business rules at fixed, common points in the processing scenarios
.

Other Requirements and Constraints

Including design constraints, usability, reliability, performance, supportability, design constraints, documentation, packaging, and so forth: See the Supplementary Specification and use cases.

Glossary (Data Dictionary):

In its simplest form, the **Glossary** is a list of noteworthy terms and their definitions. It is surprisingly common that a term, often technical or particular to the domain, will be used in slightly different ways by different stakeholders; this needs to be resolved to reduce problems in communication and ambiguous requirements.

Suggestion

Start the Glossary early. I'm reminded of an experience working with simulation experts, in which the seemingly innocuous, but important, word "cell" was discovered to have slippery and varying meanings among the group members.

The goal is not to record all possible terms, but those that are unclear, ambiguous, or which require some kind of noteworthy elaboration, such as format information or validation rules.

Glossary as Data Dictionary:

In the UP, the Glossary also plays the role of a **data dictionary,** a document that records data about the data.that is, **metadata.** During inception the glossary should be a simple document of terms and descriptions. During elaboration, it may expand into a data dictionary.

Term attributes could include:
. aliases
. description

. format (type, length, unit)
. relationships to other elements
. range of values
. validation rules
Note that the range of values and validation rules in the Glossary constitute requirements with implications on the behavior of the system.

Units:

As Martin Fowler underscores in *Analysis Patterns*, units (currency, measures, ...) must be considered, especially in this age of internationalized software applications.

For example, in the NextGen system, which will hopefully be sold to many customers in different countries, *price* cannot be just a raw number. It must be in a *Money* or *Currency* unit that captures the notion of varying currencies.

Composite Terms:

The Glossary is not only for atomic terms such as "product price." It can and should include composite elements such as "sale" (which includes other elements, such as date and location), and nicknames used to describe a collection of data transmitted between actors in the use cases. For example, in the *Process Sale* use case, consider the following statement: System sends payment authorization request to an external Payment Authorization Service, and requests payment approval. "Payment authorization request" is a nickname for an aggregate of data, which needs to be explained in the Glossary.

NextGen Example: A (Partial) Glossary

Glossary

Revision History

Version	Date	Description	Author
Inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Definitions

Term	Definition and Information	Aliases
item	A product or service for sale	
payment authorization	Validation by an external payment authorization service that they will make or guarantee the payment to the seller.	
payment authorization request	A composite of elements electronically sent to an authorization service, usually as a char array. Elements include: store ID, customer account number, amount, and timestamp.	
UPC	12 digit code that identifies a product. Usually symbolized with a bar code placed on products. See http://www.uc-council.org for details.	Universal Product Code
...	...	

Other Requirement Artifacts Within the UP

Following table summarizes a sample of artifacts and their timing. All requirements artifacts are started in inception, and primarily worked on through elaboration.

Discipline	Artifact Iteration->	Incep. Il	Elab. El. .En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 7.1 Sample UP artifacts and timing. s - start; r - refine

UP Artifacts and Process Context:

Artifact influence emphasizing the Vision, Supplementary Specification, and Glossary are shown in the following Figure:

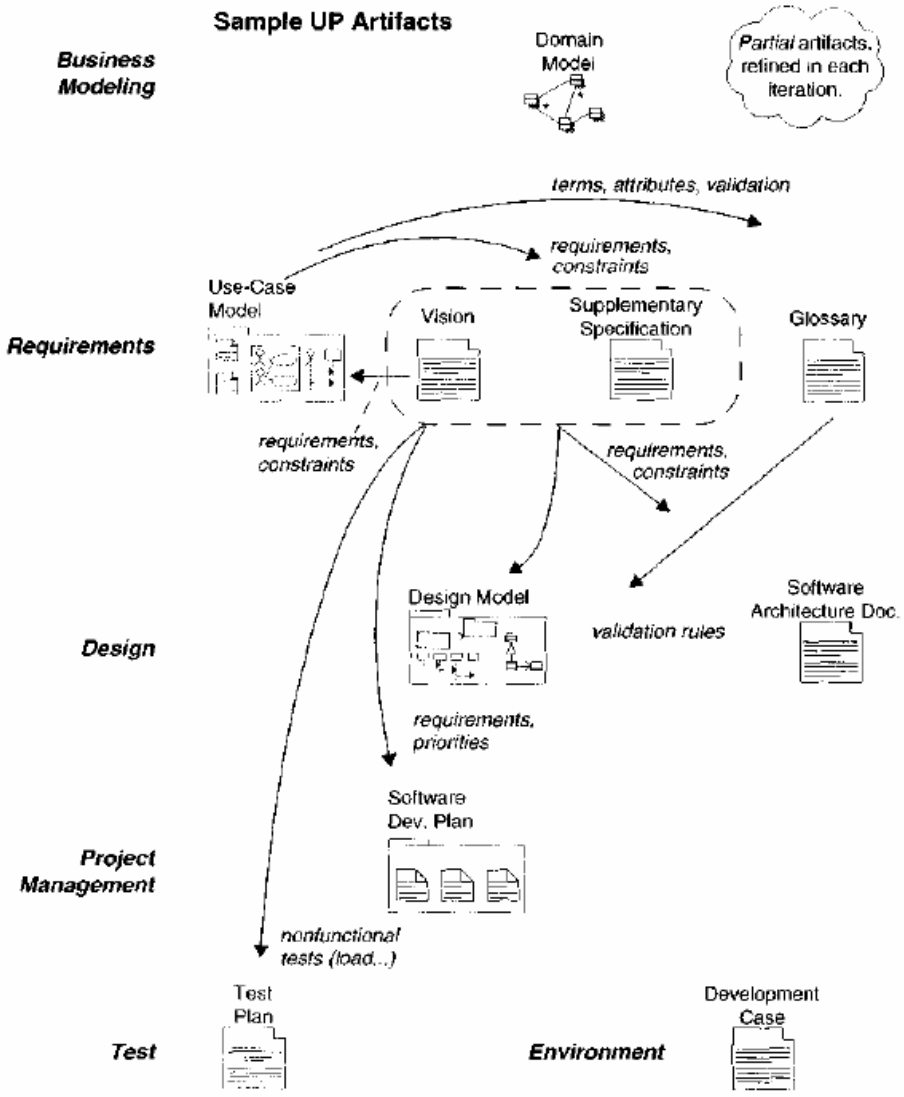


Figure 7.1 Sample UP artifact influence.

Unit-III

Elaboration

Unit III Syllabus:

Elaboration: System sequence diagrams for use case model, Domain model : identifying concepts, adding associations, adding attributes, Interaction Diagrams, Introduction to GRASP design Patterns ,Design Model: Use case realizations with GRASP patterns, Design Class diagrams in each MVC layer Mapping Design to Code, Design class diagrams for case study and skeleton code.

Introduction

Elaboration is the initial series of iterations during which:

1. The majority of requirements are discovered and stabilized.
2. The major risks are mitigated or retired.
3. The core architectural elements are implemented and proven.

What Happened in Inception?

The inception step of the NextGen POS project may be only one week. The artifacts created should be brief and incomplete, the phase quick, and the investigation light.

Some likely activities and artifacts in inception include:

- a short requirements workshop
- most actors, goals, and use cases named
- most use cases written in brief format; 10-20% of the use cases are written in fully dressed detail to improve understanding of the scope and complexity
- most influential and risky quality requirements identified
- version one of the Vision and Supplementary Specification written
- risk list
- technical feasibility of special requirements
- user interface-oriented prototypes to clarify the vision of functional requirements
- recommendations on what components to buy/build/reuse, to be refined in elaboration
- high-level *candidate* architecture and components proposed
- plan for the first iteration
- candidate tools list

It is not the requirements phase of the project, but a short step to determine basic feasibility, risk, and scope, and decide if the project is worth more serious investigation, which occurs in elaboration

Elaboration:

Build the core architecture, resolve the high-risk elements, define most requirements, and estimate the overall schedule and resources.

Some key ideas t in elaboration include:

- do short timeboxed risk-driven iterations
- start programming early
- .adaptively design, implement, and test the core and risky parts of the architecture test early, often, realistically
- adapt based on feedback from tests, users, developers
- write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

Architecturally Significant activities in Elaboration:

Early iterations build and prove the core architecture. That is, identifying the separate processes, layers, packages, and subsystems, and their high-level responsibilities and interfaces.
Partially implement these in order to connect them and clarify the interfaces. Modules may contain mostly "stubbed" code.

. Refining the inter-module local and remote interfaces (this includes the finest details of the parameters and return values).
. Integrating existing components.
. Implementing simplified end-to-end scenarios that force design, implementation, and test across many major components.
Elaboration phase testing is important, to obtain feedback, adapt, and prove that the core is robust. Early testing for the NextGen project will include:
. Usability testing of the user interface for *Process Sale.*
. Testing of recovery when remote services, such as the credit authorizer, fail. . Testing of high load to remote services, such as load on the remote tax calculator.

Planning the Next Iteration:
Planning and project management include some key ideas to organize requirements and iterations by risk, coverage, and criticality.
. **Risk** includes both technical complexity and other factors, such as uncertainty of effort or usability.
. **Coverage** implies that all major parts of the system are at least touched on in early iterations.perhaps a "wide and shallow" implementation across many components.
. **Criticality** refers to functions of high business value.
These criteria are used to rank work across iterations. Use cases or use case scenarios are ranked for implementation.early iterations implement high ranking scenarios.

The ranking is done before Iteration 1, but then again before Iteration 2, and so forth, as new requirements and new insights influence the order. That is, the plan is adaptive, rather than speculatively frozen at the beginning of the project.
.
For example:

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging ...	Scores high on all ranking criteria. Pervasive. Hard to add late. ...
Medium	Maintain Users ...	Affects security subdomain. ...
Low	...	...

Based on this ranking, we see that some key architecturally significant scenarios of the *Process Sale* use case should be tackled in early iterations.

The chosen requirements for the next iteration are briefly listed in an **Iteration Plan.** This is not a plan of all the iterations, only a plan of the next.
The overall requirements ranking is recorded in the **Software Development Plan.**

Iteration 1 Requirements and Emphasis:

Iteration 1 of the elaboration phase emphasizes a range of fundamental and common OOA/D skills used in building object systems, such as assigning responsibilities to objects, database design, usability engineering, and UI design are needed to build software.

The requirements for the first iteration of the NextGen POS application:

- .Implement a basic, key scenario of the *Process Sale* use case: entering items and receiving a cash payment.
- Implement a *Start Up* use case as necessary to support the initialization needs of the iteration.
- There is no collaboration with external services, such as a tax calculator or product database.
- No complex pricing rules are applied.
- The design and implementation of the supporting UI would also be done.

Incremental Development for the Same Use Case Across Iterations:

Note that not all requirements in the *Process Sale* use case are being handled in iteration 1. It is common to work on varying scenarios or features of the same use case over several iterations and gradually extend the system to ultimately handle all the functionality required (see Figure 8.1). On the other hand, short, simple use cases may be completed within one iteration.

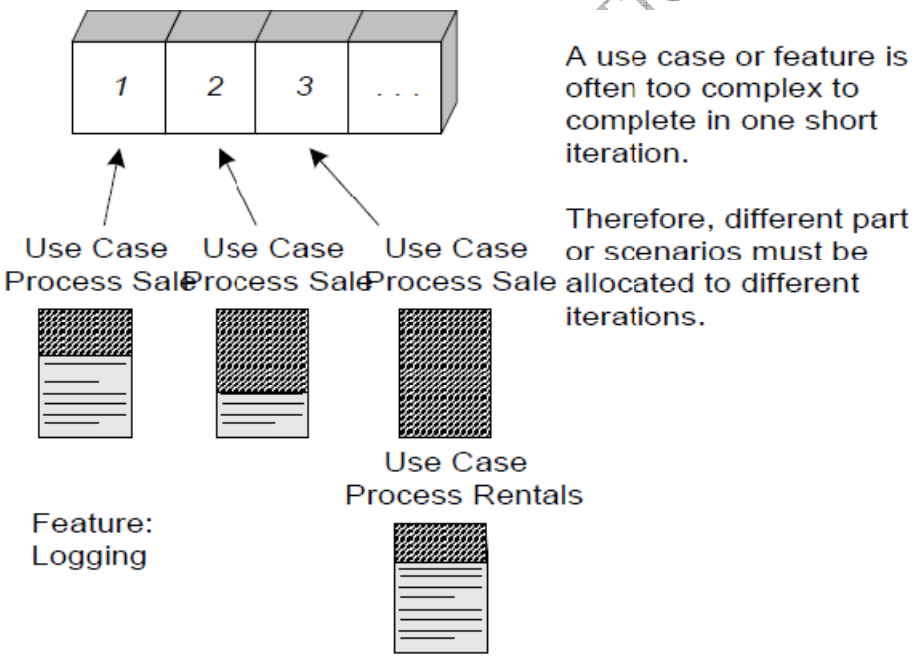


Figure 8.1 Use case implementation may be spread across iterations.

What Artifacts May Start in Elaboration?

Table 8.1 lists *sample* artifacts that may be started in elaboration, and indicates the issues they address.

Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.

Artifact	Comment
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Test Model	A description of what will be tested, and how.
Implementation Model	This is the actual implementation — the source code, executables, database, and so on.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

Table 8.1 Sample elaboration artifacts, excluding those started in inception.

System sequence diagrams(SSD) for use case model:

Objectives:

Identify system events.
Create system sequence diagrams for use cases.

Introduction:

A system sequence diagram is a fast and easily created artifact that illustrates ***input and output events*** related to the systems under discussion. The UML contains notation in the form of sequence diagrams to illustrate events from external actors to a system.

System Behavior:

Before proceeding to a logical design of how a software application will work, it is useful to investigate and define its behavior as a **"black box"**.

System behaviour is a description *of what* a system does, without explaining how it does it. One part of that description is a system sequence diagram. Other parts include the use cases, and system contracts.

System Sequence Diagrams:

Use cases describe how external actors interact with the software system. During this interaction an actor generates events to a system, usually requesting some operation in response.

For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale. That request event initiates an operation upon the system.

The UML includes **sequence diagrams** as a notation that can illustrate actor interactions and the operations initiated by them.
A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events. All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.

> An SSD should be done for the main success scenario of the use case, and frequent or complex alternative scenarios.

The UML does not define something called a "system" sequence diagram, but simply a sequence diagram.

Example of an SSD:
An SSD shows, for a particular course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system events that the actors generate (see Figure 9.1). Time proceeds downward, and the ordering of events should follow their order in the use case. System events may include parameters.

This example is for the main success scenario of the *Process Sale* use case. It indicates that the cashier generates *makeNewSale, enteritem, endSale,* and *makePayment* system events.

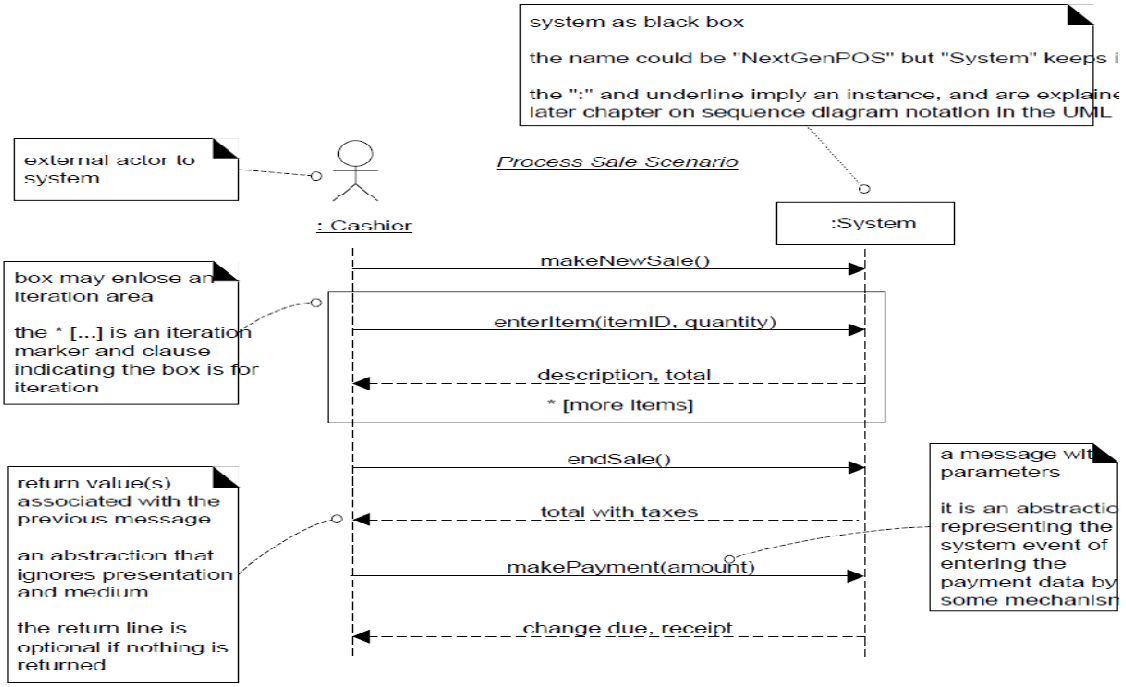


Figure 9.1 SSD for a *Process Sale* scenario.

Inter-System SSDs:

SSDs can also be used to illustrate collaborations between systems, such as between the NextGen POS and the external credit payment authorizer.

SSDs and Use Cases:
An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case (see Figure 9.2).

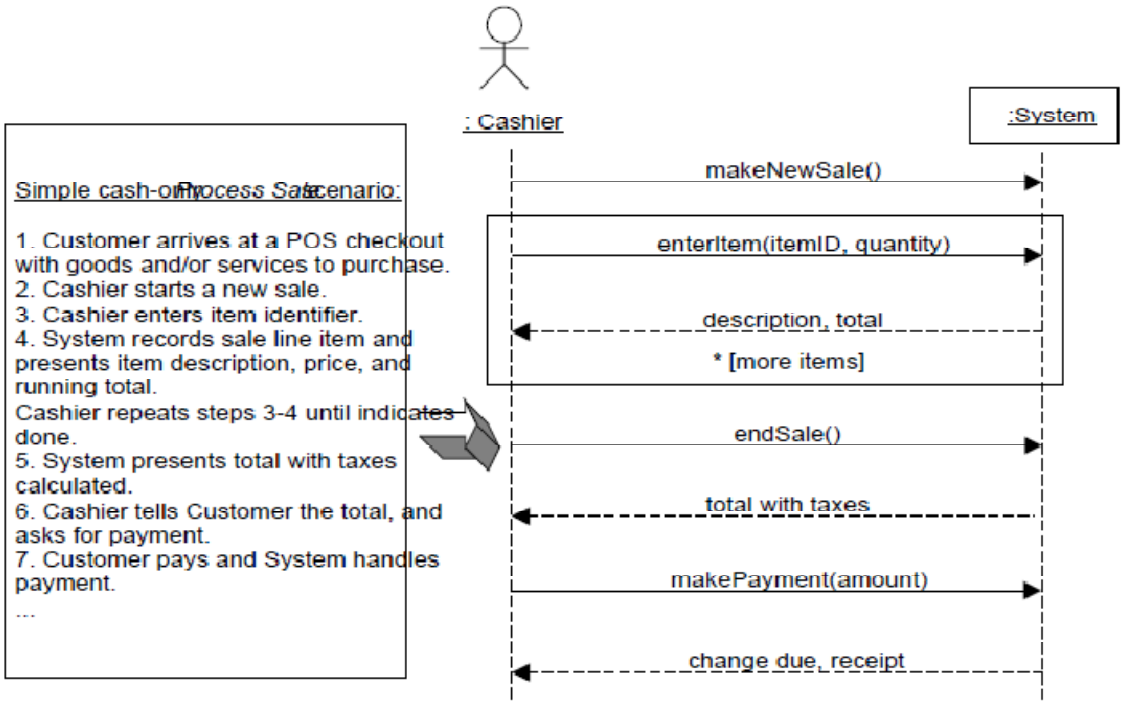


Figure 9.2 SSDs are derived from use cases.

System Events and the System Boundary:
To identify system events, it is necessary to be clear on the choice of system boundary. For software development, the system boundary is usually chosen to be the software (and possibly hardware) system itself; in this context, a system event is an external event that directly stimulates the software (see Figure 9.3).

Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash-only scenario, does not directly interact with the POS system—only the cashier does.
Therefore, the customer is not a generator of system events; only the cashier is.

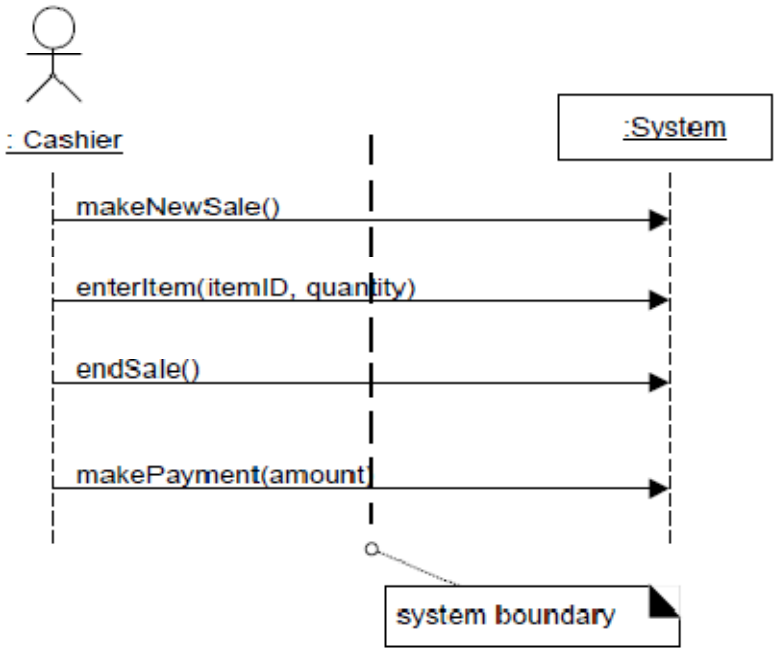


Figure 9.3 Defining the system boundary.

SSDs and the Glossary:

The terms shown in SSDs (operations, parameters, return data) are brief. These may need proper explanation. For this purpose the Glossary could be used.

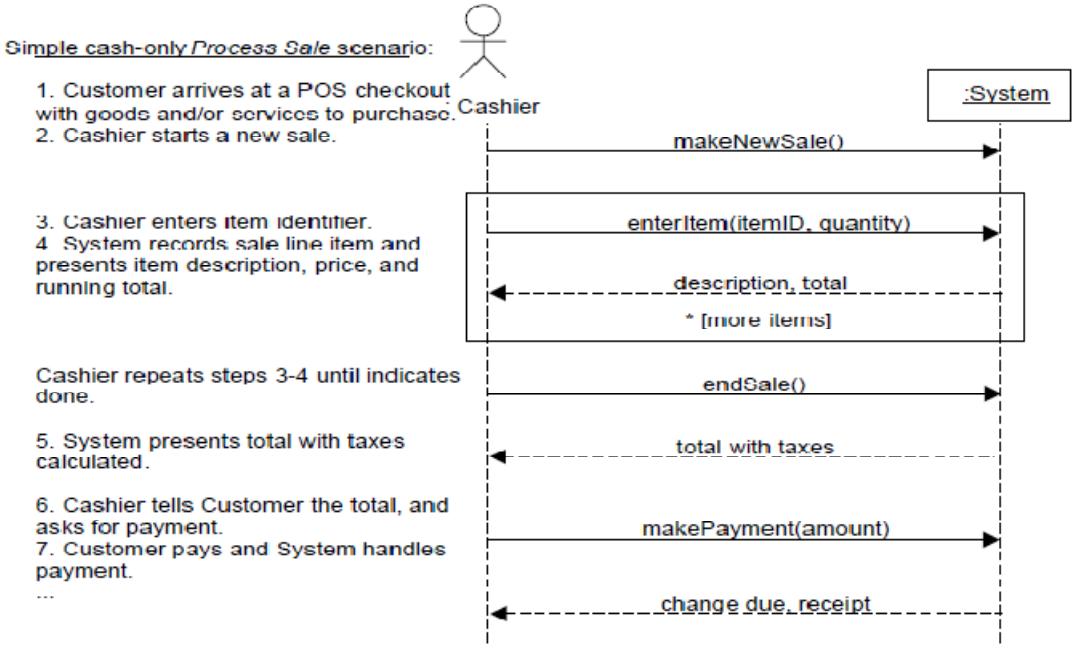


Figure 9.5 SSD with use case text.

SSDs Within the UP:

SSDs are part of the Use-Case Model—a visualization of the interactions implied in the use cases. SSDs were not explicitly mentioned in the original UP description, although the UP creators are aware of and understand the usefulness of such diagrams. SSDs are an example of the many possible skillful analysis and design artifacts or activities that the UP or RUP documents do not mention.

Phases:

Inception—SSDs are not usually motivated in inception.

Elaboration—Most SSDs are created during elaboration, when it is useful to identify the details of the system events to clarify what major operations the system must be designed to handle.

Note that it is not necessary to create SSDs for all scenarios of all use cases rather, create them only for some chosen scenarios of the current iteration. Finally, it should only take a few minutes or an half hour to create the SSDs

Discipline	Artifact Iteration→	Incep. I1	Elab. E1. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	*Use-Case Model (SSDs)*	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	R
Project Management	SW Development Plan	s	r	r	R
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 9.1 Sample UP artifacts and timing, s - start; r - refine

2 UP Artifacts

Sample relationships of SSDs to other artifacts are shown in Figure 9.6.

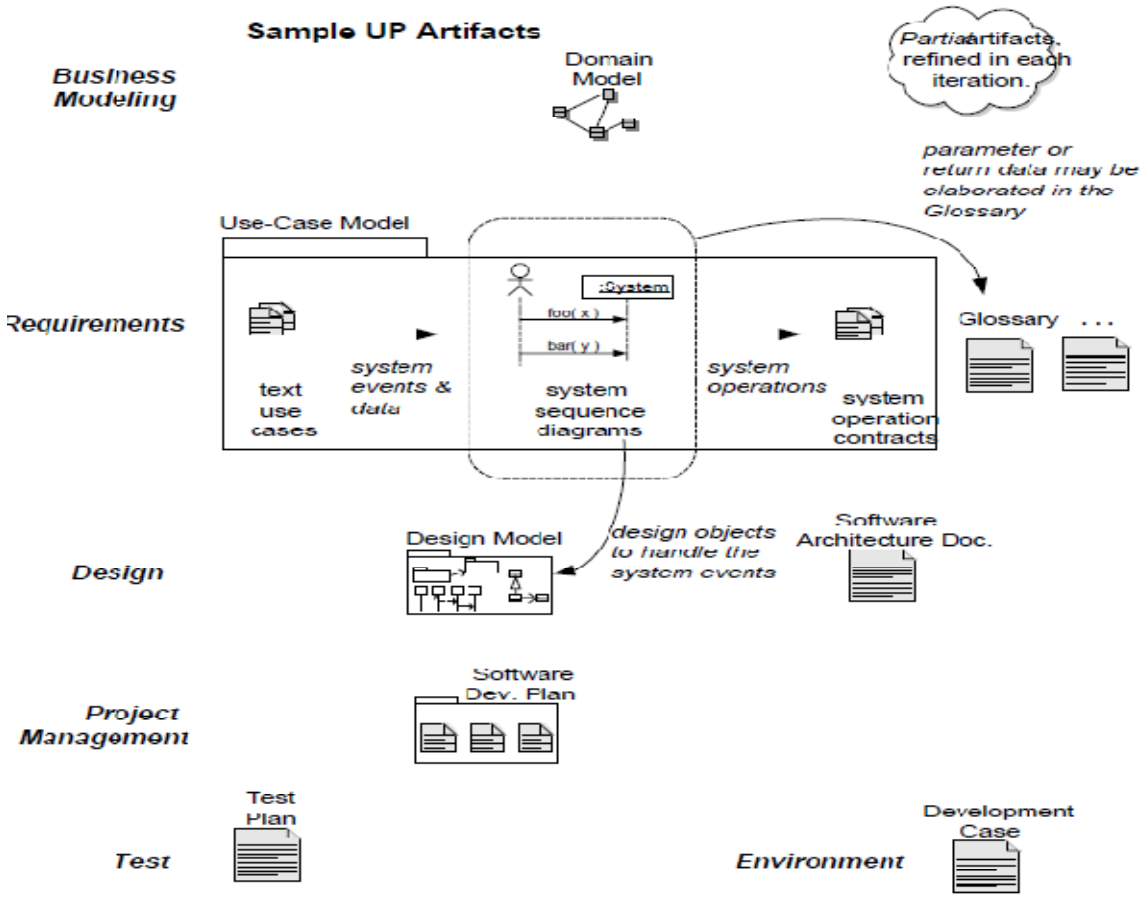


Figure 9.6 Sample UP artifact influence.

DOMAIN MODEL:

Objectives:

- ✓ Identify conceptual classes related to the current iteration requirements.
- ✓ Create an initial domain model.
- ✓ Distinguish between correct and incorrect attributes.
- ✓ Add *specification* conceptual classes, when appropriate.
- ✓ Compare and contrast conceptual and implementation views.

Introduction:
A domain model is widely used as a source for designing software objects, and will be a required input to several subsequent artifacts.
A domain model illustrates conceptual classes in a problem domain; it is the most important artifact to create during object-oriented analysis.

1. Use cases are an important requirements analysis artifact, but are not object-oriented. They emphasize a process view of the domain models.
Identifying a rich set of objects or conceptual classes is at the heart of object-oriented analysis, and well worth during the design and implementation work.
The identification of conceptual classes is part of an investigation of the problem domain. The UML contains notation in the form of class diagrams to illustrate domain models.

Key Idea
A domain model is a representation of real-world conceptual classes, not of software components. It is *not* a set of diagrams describing software classes, or software objects with responsibilities.

Domain Models:
The object-oriented step in analysis or investigation is the decomposition of a domain of interest into individual conceptual classes or objects. A **domain model** is a *visual* representation of conceptual classes or real-world objects in a domain of interest.
They have also been called **conceptual models domain object models, and analysis object models.**
The UP defines a Domain Model as one of the artifacts that may be created in the ***Business Modeling discipline***.
Using UML notation, a domain model is illustrated with a set of **class diagrams** in which ***no operations*** are defined.

For example, Figure 10.1 shows a partial domain model, showing purely conceptual views of domains. Domain models are not data models.

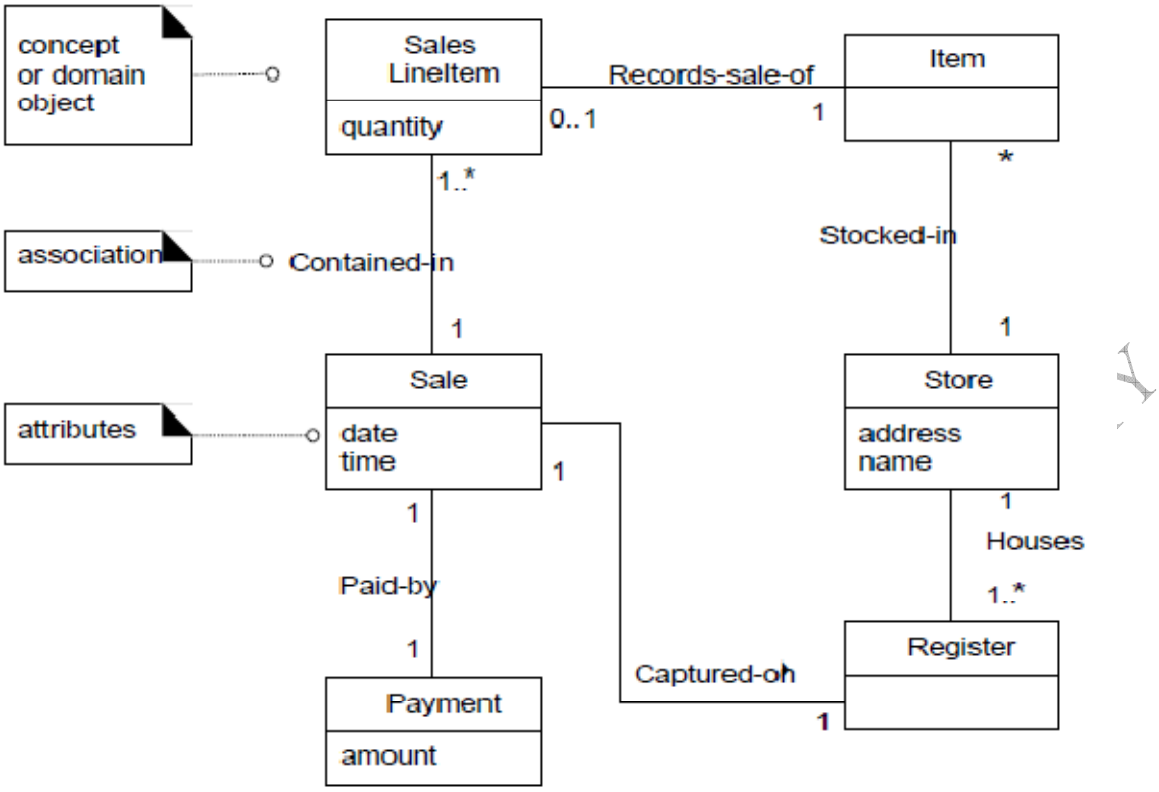


Figure 10.1 Partial domain model—a visual dictionary.

Domain Model—A Visual Dictionary of Abstractions:

Figure 10.1 visualizes conceptual classes in the domain. It also depicts an *abstraction* of the conceptual classes. The model displays a partial view, or abstraction, and ignores uninteresting (to the modelers) details.

Thus, the domain model may be considered a *visual dictionary* of abstractions, domain vocabulary, and information content of the domain.

Domain Models Are not Models of Software Components:

A domain model, as shown in Figure 10.2, is a visualization of things in the realworld domain of interest, *not* of software components such as a Java or C++ class or software objects with responsibilities.

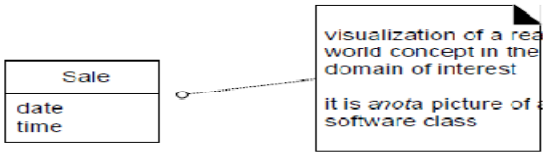


Figure 10.2 A domain model shows real-world conceptual classes, not software classes.

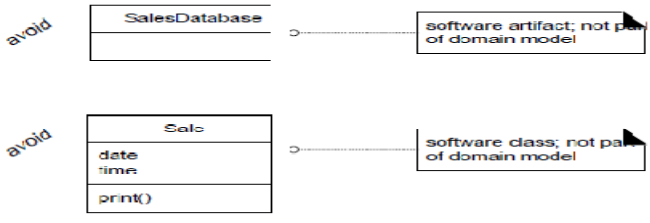


Figure 10.3 A domain model does not show software artifacts or classes.

The domain model consisting of:

- Domain objects or conceptual classes
- Associations between conceptual classes
- attributes of conceptual classes

Conceptual Classes:
The domain model illustrates conceptual classes or vocabulary in the domain.
A conceptual class may be considered in terms of its ***symbol, intension, and extension.***
• **Symbol**—words or images representing a conceptual class.
• **Intension**—the definition of a conceptual class.
• **Extension**—the set of examples to which the conceptual class applies.

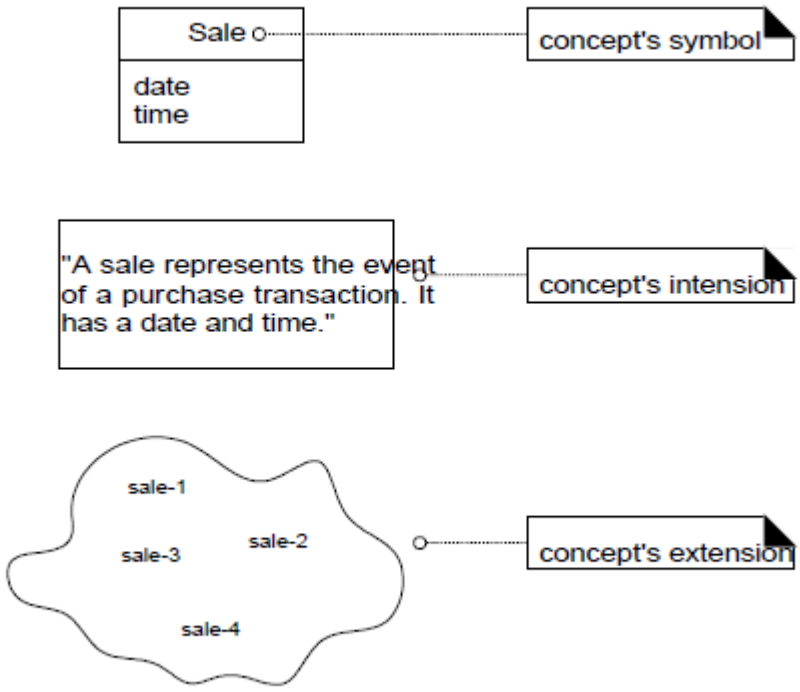


Figure 10.4 A conceptual class has a symbol, intension, and extension.

.

Conceptual Class Identification:
Our goal is to create a domain model of interesting or meaningful conceptual classes in the domain of interest In iterative development, one incrementally builds a domain model over several iterations in the elaboration phase.
In each, the domain model is limited to the prior and current scenarios under consideration, rather than to capture all possible conceptual classes and relationships.
For example, this iteration is limited to a simplified cash-only *Process Sale* scenario;
The central task is therefore to identify conceptual classes related to the scenarios under design.
The following is a useful guideline in identifying conceptual classes:

- Do not think that a domain model is better if it has fewer or more conceptual classes.

- Do not exclude a conceptual class simply because the requirements do not indicate
- any obvious need to remember information about It.
- It is valid to have attributeless conceptual classes

Strategies to Identify Conceptual Classes:

Two techniques are presented in the following sections:

1. Use a conceptual class category list.

2. Identify noun phrases.

Use a Conceptual Class Category List:

Start the creation of a domain model by making a list of candidate conceptual classes. Table 10.1 contains many common categories that are usually worth considering.

Examples are drawn from the store and airline reservation domains.

Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places	*Store* *Airport*
transactions	*Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people	*Cashier* *Pilot*
containers of other things	*Store, Bin* *Airplane*
things in a container	*Item* *Passenger*
other computer or electro-mechanical systems external to the system	*CreditPaymentAuthorizationSystem* *AirTrafficControl*
abstract noun concepts	*Hunger* *Acrophobia*
organizations	*SalesDepartment* *ObjectAirline*
events	*Sale, Payment, Meeting* *Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct* *BookingASeat*
rules and policies	*RefundPolicy* *CancellationPolicy*
catalogs	*ProductCatalog* *PartsCatalog*
records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract* *MaintenanceLog*
financial instruments and services	*LineOfCredit* *Stock*
manuals, documents, reference papers, books	*DailyPriceChangeList* *RepairManual*

Table 10.1 Conceptual Class Category List.

Finding Conceptual Classes with Noun Phrase Identification:

identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

The fully dressed use cases are an excellent description to draw from for this analysis. For example, the current scenario of the *Process Sale* use case can be used.

Main Success Scenario (or Basic Flow):

1. Customer arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. Cashier starts a new **sale.**
3. Cashier enters **item identifier.**
4. System records **sale line item** and presents **item description, price,** and running **total.** Price calculated from a set of price rules.
Cashier repeats steps 2-3 until indicates done.
5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment.**
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).
9. System presents **receipt.**
10.Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

7a. Paying by cash:
1. Cashier enters the cash **amount tendered.**
2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

A weakness of this approach is different noun phrases may represent the same conceptual class or attribute, among other ambiguities. it is recommended in combination with the *Conceptual Class Category List* technique.

Candidate Conceptual Classes for the Sales Domain:

From the Conceptual Class Category List and noun phrase analysis, a list is generated of candidate conceptual classes for the domain for the simplified scenario *of Process Sale.*

Register
Item
Store
Sale
Payment
ProductCatalog
ProductSpecification
SalesLineItem
Cashier
Customer
Manager

Domain Modeling Guidelines:

Apply the following steps to create a domain model:

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory (discussed in a subsequent chapter).
4. Add the attributes necessary to fulfill the information requirements (discussed in a subsequent chapter).

A UML class is a special case of the very general UML model element **classifier**—something with structural features and/or behavior, including classes, actors, interfaces, and use cases.

To keep things clear, this book will use class-related terms as follows, which is consistent with the UML and the UP:

- Conceptual class — real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
- Software class — a class representing a specification or implementation perspective of a software component, regardless of the process or method.
- Design class — a member of the UP Design Model. It is a synonym for software class, but for some reason I wish to emphasize that it is a class in the Design Model. The UP allows a design class to be either a specification or implementation perspective, as desired by the modeler.
- Implementation class — a class implemented in an object-oriented language such as Java.
- Class — as in the UML, the general term representing either a real-world thing (a conceptual class) or software thing (a software class).

Example: The NextGen POS Domain Model

The list of conceptual classes generated for the NextGen POS domain may be represented graphically (see Figure 10.11) to show the start of the Domain Model.

Figure 10.11 Initial Domain Model.

Consideration of attributes and associations for the Domain Model will be deferred to subsequent chapters.

Domain Models Within the UP:

As suggested in the example of Table 10.2, a Domain Model is usually both started and completed in elaboration.

DOMAIN MODELS WITHIN THE UP

Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	*Domain Model*		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 10.2 Sample UP artifacts and timing, s - start; r - refine

Inception:

Domain models are not strongly motivated in inception, since inception's purpose is not to do a serious investigation, but rather to decide if the project is worth deeper investigation in an elaboration phase.

Elaboration:

The Domain Model is primarily created during elaboration iterations, when the need is highest to understand the noteworthy concepts and map some to software classes during design work.

domain. ... This is often referred to as a domain model. [RUP]

UP Artifacts:

Artifact influence emphasizing the Domain Model is shown in Figure 10.12.

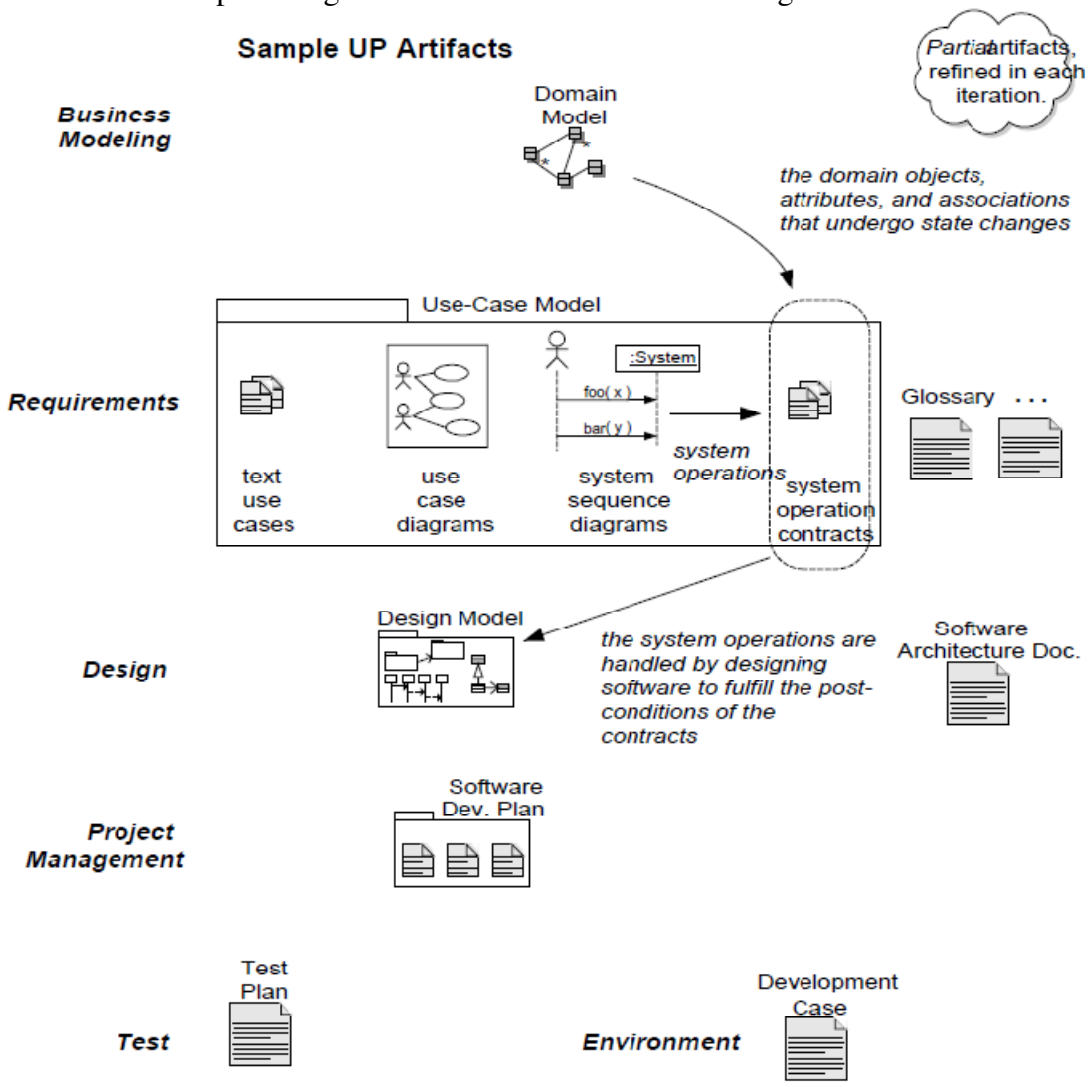


Figure 10.12 Sample UP artifact influence.

2) Adding associations:

Objectives:
• Identify associations for a domain model.
• Distinguish between need-to-know and comprehension-only associations.

Introduction:
It is useful to identify those associations of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.

Associations:
An **association** is a relationship between types that indicates some meaningful and interesting connection (see Figure 11.1).
In the UML associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances."

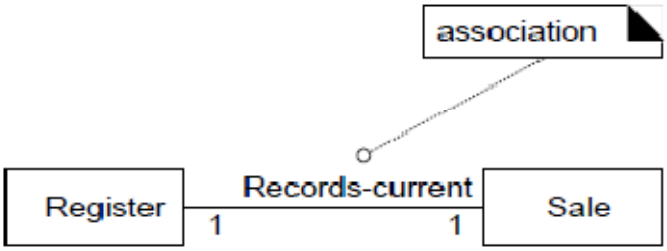


Figure 11.1 Associations.

Criteria for Useful Associations
Associations imply knowledge of a relationship that needs to be preserved for some duration—it could be milliseconds or years, depending on context.
For example, do we need to remember what *SalenLineItem* instances are associated with a *Sale* instance to reconstruct a sale, print a receipt, or calculate a sale total.

Consider including the following associations in a domain model:
- Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- Associations derived from the Common Associations List.

The UML Association Notation:
An association is represented as a line between classes with an association name. The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible. This traversal is purely abstract; it is *not a* statement about connections between software entities.

The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes. An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If not present, it is conventional to read the association from left to right or top to bottom, although the UML does not make this a rule (see Figure 11.2).

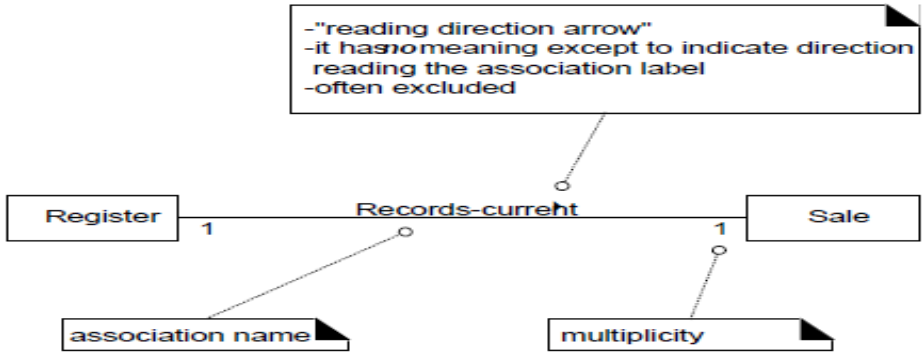


Figure 11.2 The UML notation for associations.

Finding Associations—Common Associations List:

Start the addition of associations by using the list in Table 11.1. It contains common categories that are usually worth considering. Examples are drawn from the store and airline reservation domains.

Category	Examples
A is a physical part of B	*Drawer — Register (or more specifically, a POST)* *Wing — Airplane*
A is a logical part of B	*SalesLineItem — Sale* *FlightLeg—FlightRoute*
A is physically contained in/on B A is logically contained in B	*Register — Store, Item — Shelf* *Passenger — Airplane* *ItemDescription — Catalog* *Flight— FlightSchedule*
A is a description for B	*ItemDescription — Item* *FlightDescription — Flight*
A is a line item of a transaction or report B A is known/logged/recorded/reported/captured in B	*SalesLineItem — Sale* *Maintenance Job — Maintenance-Log* *Sale — Register* *Reservation — FlightManifest*
A is a member of B	*Cashier — Store* *Pilot — Airline*
A is an organizational subunit of B A uses or manages B	*Department — Store* *Maintenance — Airline* *Cashier — Register* *Pilot — Airplane*
A communicates with B	*Customer — Cashier* *Reservation Agent — Passenger*
A is related to a transaction B A is a transaction related to another transaction B	*Customer — Payment* *Passenger — Ticket* *Payment — Sale* *Reservation — Cancellation*
A is next to B	*SalesLineItem — SalesLineItem* *City— City*

Category	Examples
A is owned by B	*Register — Store* *Plane — Airline*
A is an event related to B	*Sale — Customer*; *Sale — Store* *Departure — Flight*

Table 11.1 Common Associations List.

High-Priority Associations:

Here are some high-priority association categories that are invariably useful to include in a domain model:

• A is a *physical or logical part* of B.
• A is *physically or logically contained* in/on B.
• A is *recorded in* B.

Association Guidelines:

• Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
• It is more important to identify *conceptual classes* than to identify associations.
• Too many associations tend to confuse a domain model rather than illuminate it. Their discovery can be time-consuming, with marginal benefit.
• Avoid showing redundant or derivable associations.

Roles:

Each end of an association is called a **role.** Roles may optionally have:

• name
• multiplicity expression
• navigability

Multiplicity:

Multiplicity defines how many instances of a class *A* can be associated with one instance of a class *B* (see Figure 11.3).

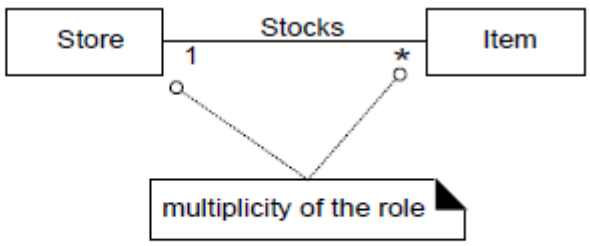


Figure 11.3 Multiplicity on an association.

For example, a single instance of a *Store* can be associated with "many" (zero or more, indicated by the *) *Item* instances.

Some examples of multiplicity expressions are shown in Figure 11.4.

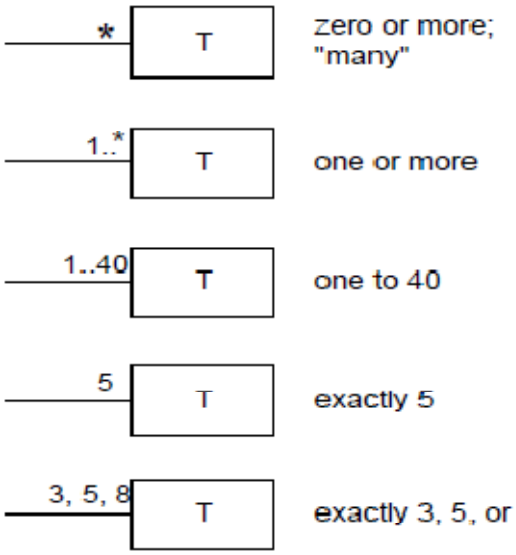


Figure 11.4 Multiplicity values.

Naming Associations:

Name an association based on a *TypeName-VerbPhrase-TypeName* format where the verb phrase creates a sequence that is readable and meaningful in the model context.

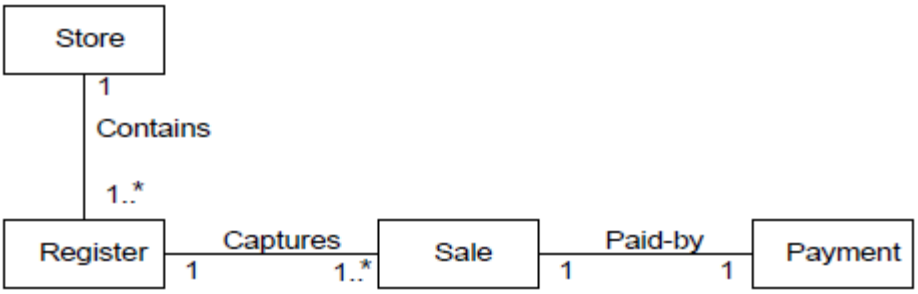
Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

Two common and equally legal formats for a compound association name are:

• *Paid-by*

• *PaidBy*

In Figure 11.6, the default direction to read an association name is left to right or top to bottom. This is not a UML default, but a common convention.



Multiple Associations Between Two Types:

Two types may have multiple associations between them. an example from the domain of the airline is the relationships between a *Flight*and *an Airport* (see Figure 11.7); the flying-to and flying from associations are distinctly different relationships, which should be shown separately.

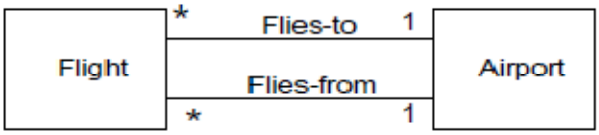


Figure 11.7 Multiple associations.

Associations and Implementation:

When creating a domain model, we may define associations that are not necessary during implementation. Conversely, we may discover associations that need to be implemented but were missed during domain modeling. In these cases, the domain model can be updated to reflect these discoveries.

Suggestion

Should prior investigative models such as a domain model be updated with insights (such as new associations) revealed during implementation work? Do not bother unless there is some future practical use for the model. If it is just (as is sometimes the case) a temporary artifact used to provide inspiration for a later step, and will not be meaningfully used later on, why update it? Avoid making or updating any documentation or model unless there is a concrete justification for future use.

NextGen POS Domain Model Associations:

We can now add associations to our POS domain model. We should add those associations which the requirements (for example, use cases) suggest or imply a need to remember, or which otherwise are strongly suggested in our perception of the problem domain.

The following sample of associations is justified in terms of a need-to-know. It is based on the use cases currently under consideration.

Register Records Sale
Sale Paid-by Payment
ProductCatalog Records ProductSpecification
To know the current sale, generate a total, print a receipt.
To know if the sale has been paid, relate the amount tendered to the sale total, and print a receipt.
To retrieve an *ProductSpecification,* given an itemID.

- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- Avoid showing redundant or derivable associations.

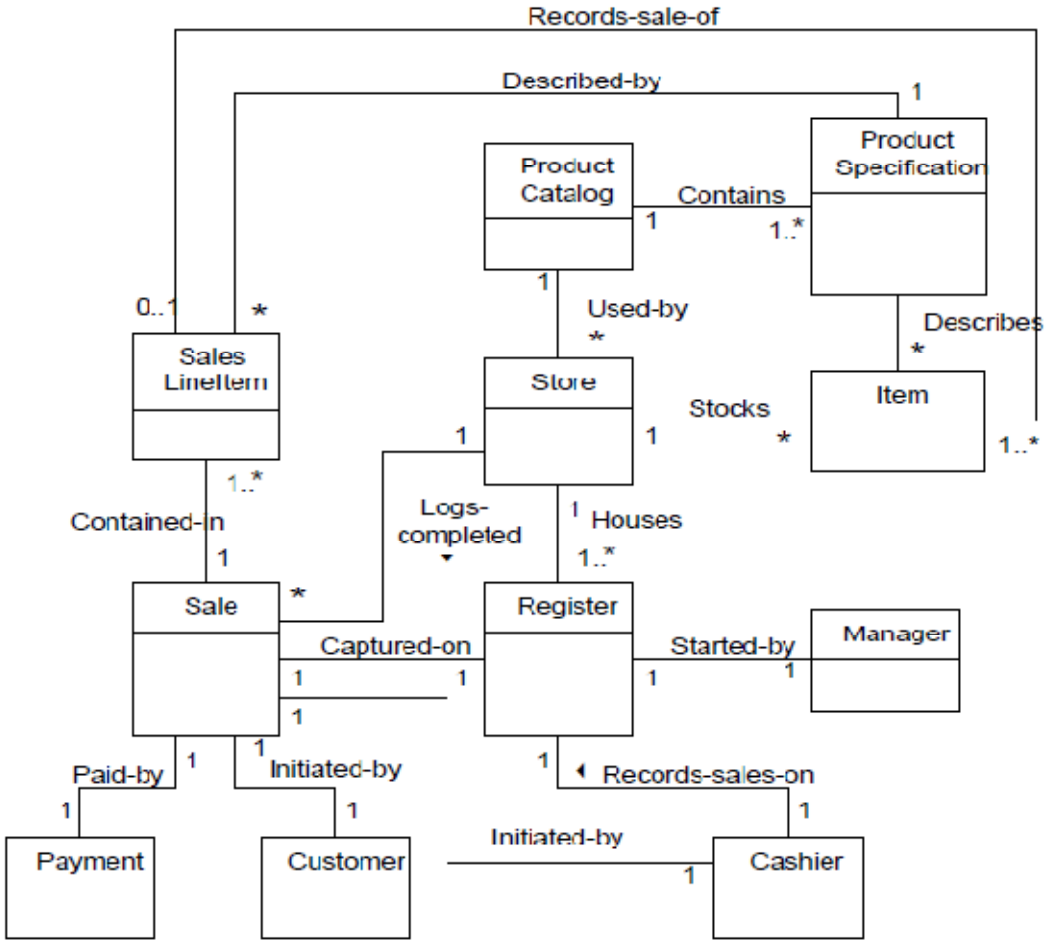


Figure 11.8 A partial domain model.

Need-to-Know vs. Comprehension Associations :

Emphasize need-to-know associations, but add choice comprehension-only associations to enrich critical understanding of the domain.

3) Adding Attributes:

Objectives:
- Identify attributes in a domain model.
- Distinguish between correct and incorrect attributes.

Introduction:
It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.

Attributes:
An attribute is a logical data value of an object.

Include the following attributes in a domain model: Those for which the requirements (for example, use cases) suggest or imply a need to remember information.

For example, the *Sale* conceptual class needs a *date* and *time* attribute.

UML Attribute Notation:

Attributes are shown in the second compartment of the class box (see Figure 12.1). Their type may optionally be shown.

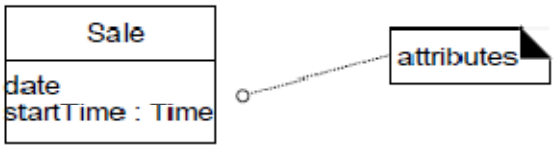


Figure 12.1 Class and attributes.

Valid Attribute Types:

There are some things that should not be represented as attributes, but rather as associations. This section explores valid attributes.

Keep Attributes Simple

Intuitively, most simple attribute types are what are often thought of as primitive data types, such as numbers. The type of an attribute should not normally be a complex domain concept, such as a *Sale or Airport.* For example, the following *currentRegister* attribute in the *Cashier* class in Figure 12.2 is undesirable because its type is meant to be a *Register,* which is not a simple attribute type (such as *Number* or *String*). The most useful way to express that a *Cashier* uses a *Register* is with an association, not with an attribute..

The attributes in a domain model should preferably be simple attributes or data types.
Very common attribute data types include: *Boolean, Date, Number, String (Text), Time*
Other common types include: *Address, Color, Geometries (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types*

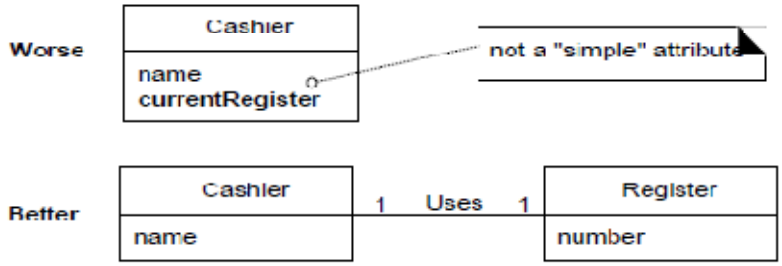


Figure 12.2 Relate with associations, not attributes

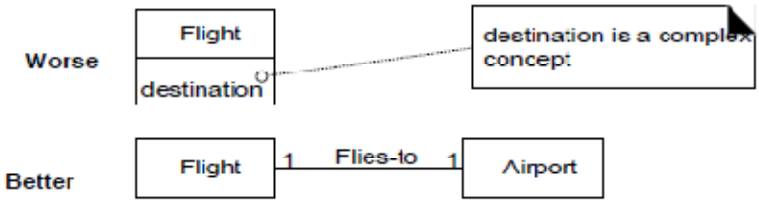


Figure 12.3 Avoid representing complex domain concepts as attributes; use associations.

Conceptual vs. Implementation Perspectives: What About Attributes in Code?:
The restriction that attributes in the domain model be only of simple data types does *not* imply that C++ or Java attributes (data members, instance fields) must only be of simple, primitive data types. The domain model focuses on pure conceptual statements about a problem domain, not software components.
Later, during design and implementation work, it will be seen that the associations between objects expressed in the domain model will often be implemented as attributes that reference other complex software objects.

Data Types:
Attributes should generally be **data types.** all primitive types (number, string) are UML data types, but not all data types are primitives. For example, *PhoneNumber* is a non-primitive data type. These data type values are also known as **value objects.**

> If in doubt, define something as a separate conceptual class rather than as an attribute.

• The *address* attribute should be a non-primitive *Address* class because it

Modeling Attribute Quantities and Units:
Most numeric quantities should not be represented as plain numbers. Consider price or velocity. These are quantities with associated units, and it is common to require knowing the unit, and to support conversions. The NextGen POS software is for an international market and needs to support prices in multiple currencies. In the general case, the solution is to represent *Quantity* as a distinct conceptual class, with an associated *Unit* [Fowler96]. Since quantities are considered data types (unique identity of instances is not important), it is acceptable to collapse their illustration into the attribute section of the class box (see Figure 12.6). It is also common to show *Quantity* specializations. *Money* is a kind of quantity whose units are currencies. *Weight* is a quantity with units such as kilograms or pounds.

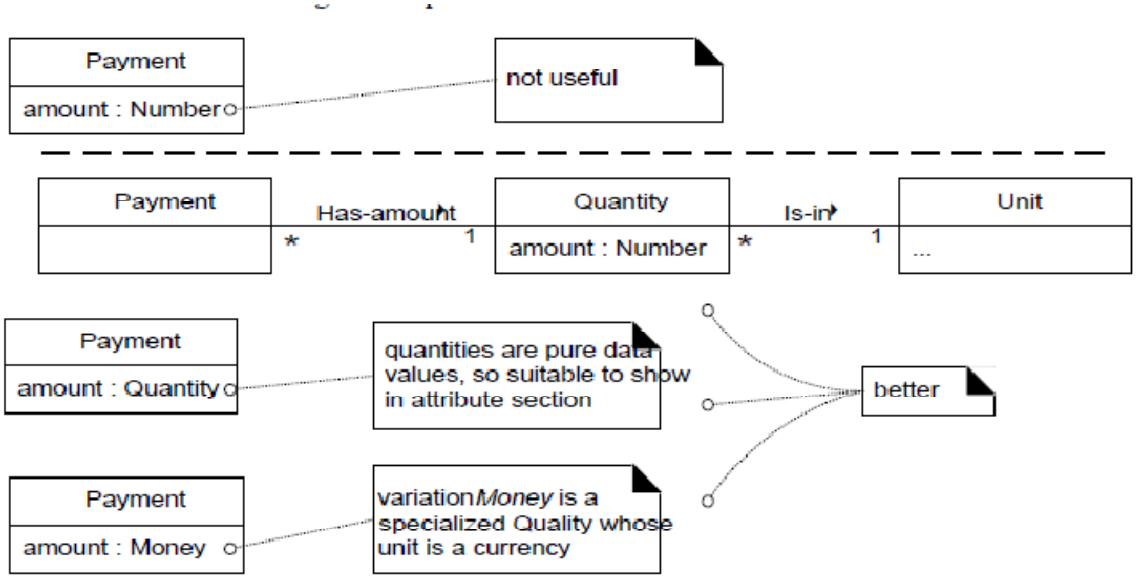


Figure 12.6 Modeling quantities.

Attributes in the NextGen Domain Model

The attributes chosen reflect the requirements for this iteration—the Process *Sale* scenarios of this iteration.

Payment
Product-Specification
Sale
SalesLineItem
Store

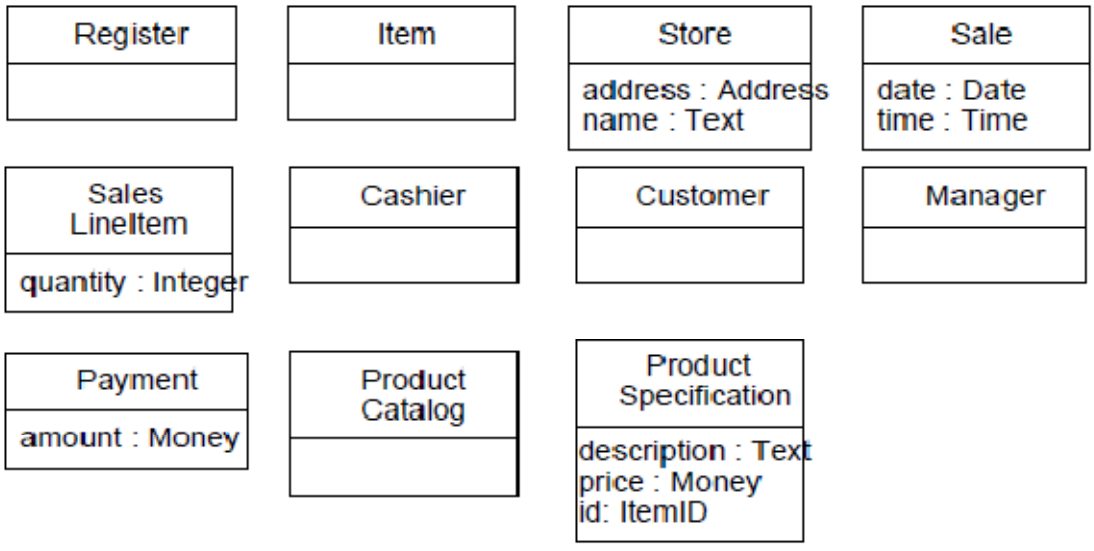


Figure 12.7 Domain model showing attributes.

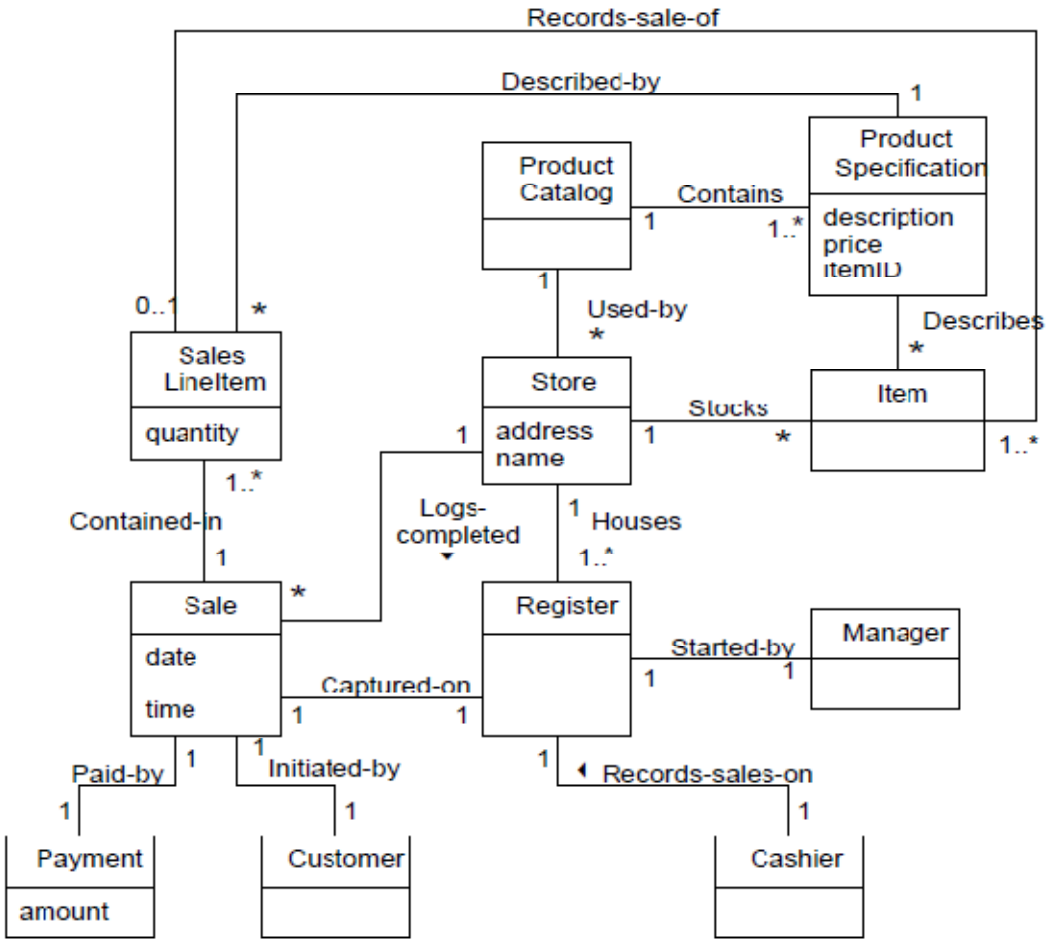


Figure 12.9 A partial domain model.

Interaction Diagrams:

The UML includes **interaction diagrams** to illustrate how objects interact via messages. The term *interaction diagram,* is a generalization of two more specialized UML diagram types; both can be used to express similar message interactions:

- collaboration diagrams
- sequence diagrams

Collaboration diagrams illustrate object interactions in a ***graph or network*** format, in which objects can be placed anywhere on the diagram, as shown in figure 15.1.

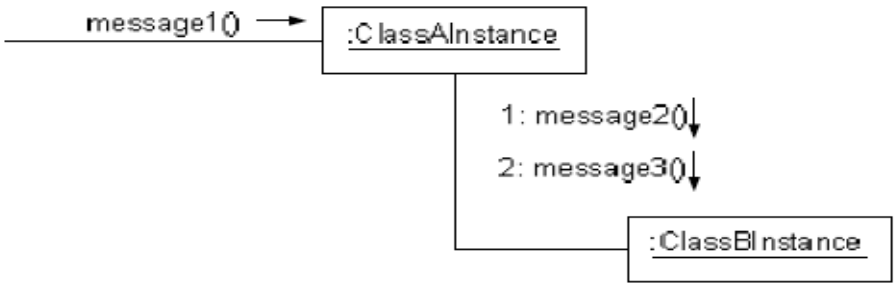


Figure 15.1 Collaboration diagram

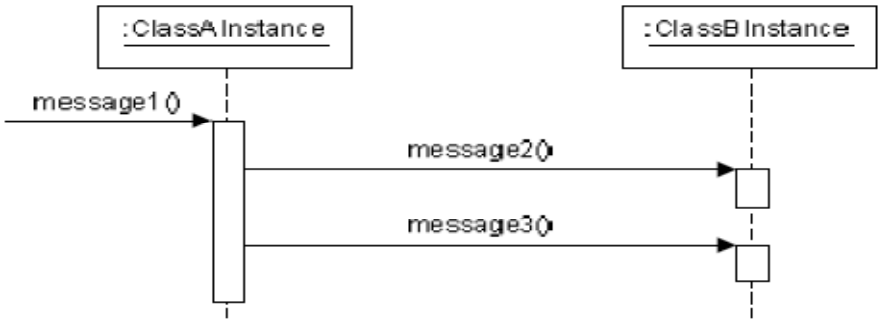


Figure 15.2 Sequence diagram.

Sequence diagrams illustrate interactions in a **fence** format, in which new object is added to the right.

Each type has strengths and weaknesses. When drawing diagrams to be published on pages of narrow width, collaboration diagrams have the advantage of allowing vertical expansion for new objects; additional objects in a sequence diagrams must extend to the right, which is limiting. On the other hand, collaboration diagram examples make it harder to easily see the sequence of messages.

Most prefer sequence diagrams when using a CASE tool to reverse engineer source code into an interaction diagram, as they clearly illustrate the sequence of messages.

Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages simple notation	forced to extend to the right when adding new objects; consumes horizontal space
collaboration	space economical—flexibility to add new objects in two dimensions better to illustrate complex branching, iteration, and concurrent behavior	difficult to see sequence of messages more complex notation

Example Collaboration Diagram: makePayment:

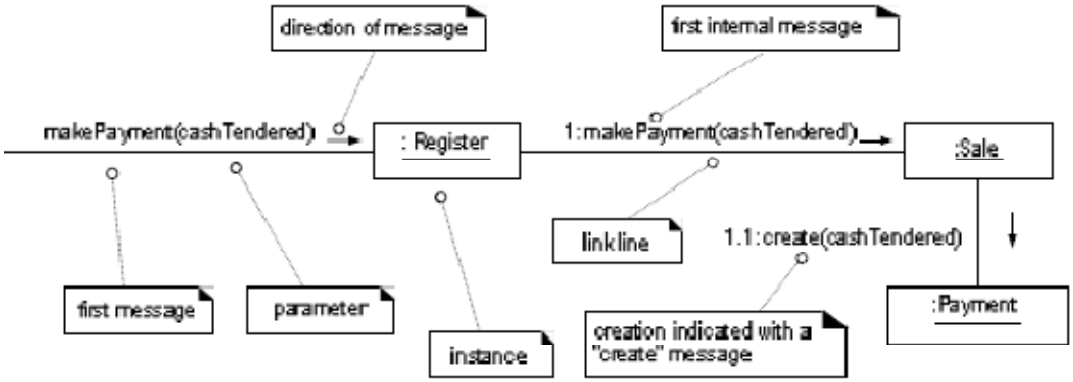


Figure 15.3 Collaboration diagram.

The collaboration diagram shown in Figure 15.3 is read as follows:
1. The message *makePayment is* sent to an instance of a *Register.* The sender is not identified.
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment.*

Example Sequence Diagram: makePayment:

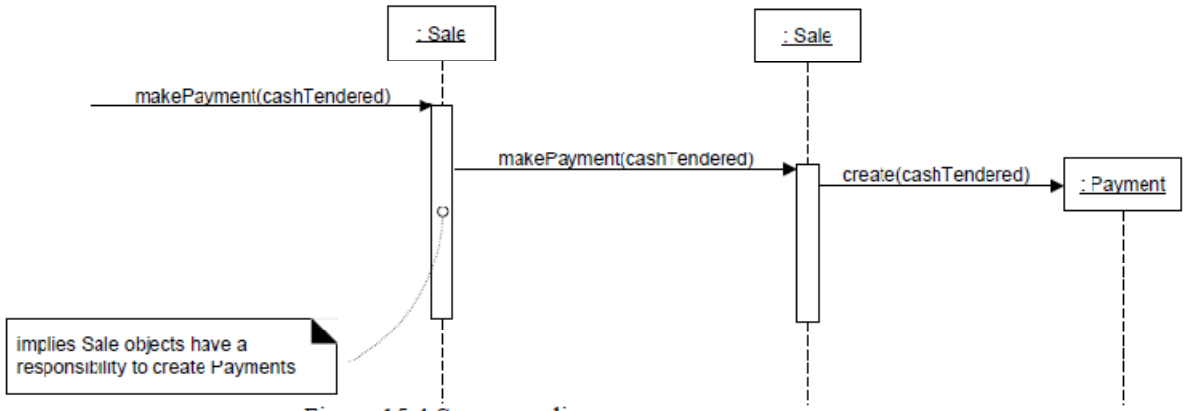


Figure 15.4 Sequence diagram.

The sequence diagram shown in Figure 15.4 has the same intent as the prior collaboration diagram.

Note that creation of use cases, domain models, and other artifacts is easier than the assignment of responsibilities and the creation of well-designed interaction diagrams.

Common Interaction Diagram Notation:

Classes and Instances:

The UML has adopted a simple and consistent approach to illustrate **instances** vs. classifiers (see Figure 15.5):, an instance uses the same graphic symbol as the type, but the designator string is underlined.

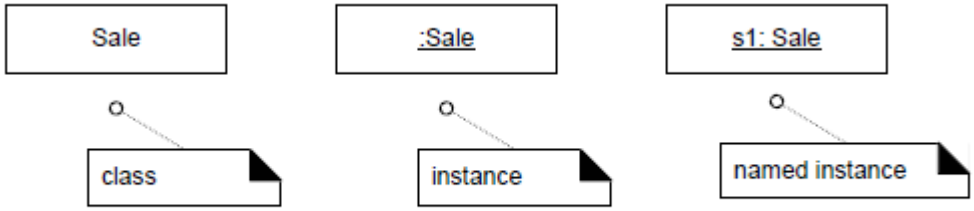


Figure 15.5 Class and instances.

Basic Message Expression Syntax:
The UML has a standard syntax for message expressions:
return := message(parameter : parameterType) : returnType
Type information may be excluded if obvious or unimportant. For example:
spec := getProductSpect(id)
spec := getProductSpect(id:ItemID)
spec := getProductSpect(id:ItemID) ProductSpecification
Basic Collaboration Diagram Notation

Basic Collaboration Diagram notation:

Links:
A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible (see Figure 15.6). More formally, a link is an instance of an association. For example, there is a link.or path of navigation.from a *Register* to a *Sale,* along which messages may flow, such as the *makePayment* message.
Note that multiple messages, and messages both ways, can flow along the same single link.

Messages:
Each message between objects is represented with a message expression and small arrow indicating the direction of the message. Many messages may flow along this link (Figure 15.7). A sequence number is added to show the sequential order of messages in the current thread of control.

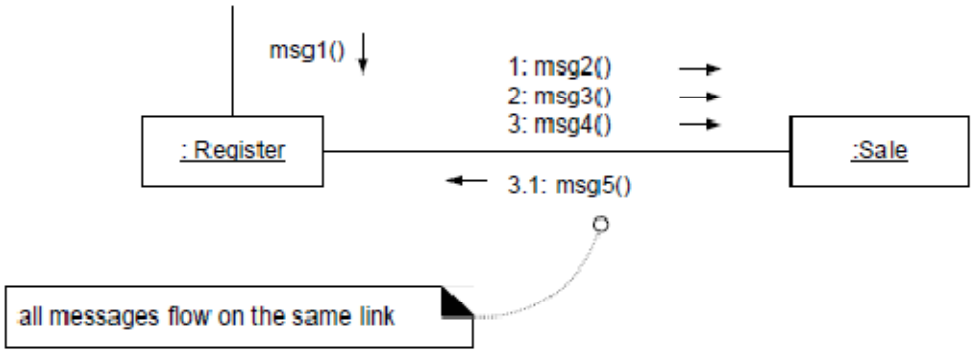


Figure 15.7 Messages.

Messages to "self" or "this"
A message can be sent from an object to itself (Figure 15.8). This is illustrated by a link to itself, with messages flowing along the link.

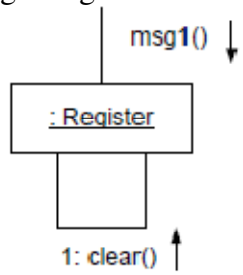


Figure 15.8 Messages to "this."

Creation of Instances:
Any message can be used to create an instance, but there is a convention in the UML to use a message named *create* for this purpose. If another (perhaps less obvious) message name is used, the message may be annotated with a special feature called a UML stereotype, like so: *«create».*

The *create* message may include parameters, indicating the passing of initial values. This indicates, for example, a constructor call with parameters in Java. Furthermore, the UML property *{new}* may optionally be added to the instance box to highlight the creation`

`

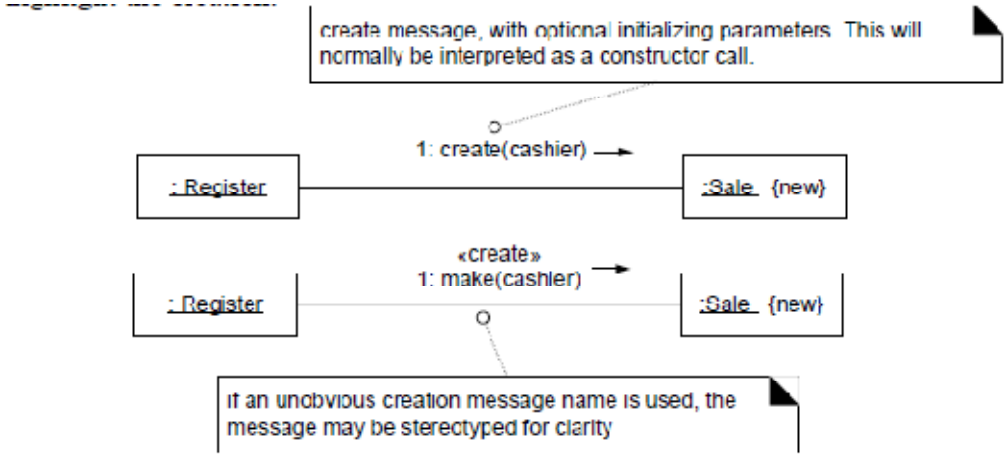
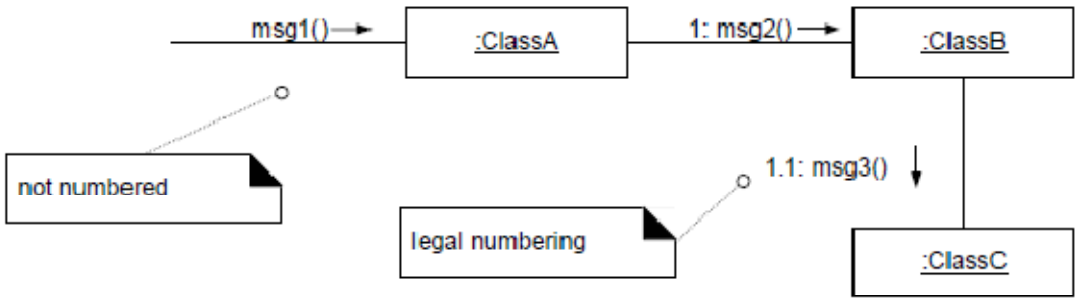


Figure 15.9 Instance creation.

Message Number Sequencing

The order of messages is illustrated with **sequence numbers,** as shown in Figure 15.10. The numbering scheme is:

1. The first message is not numbered. Thus,*msg1()* is unnumbered.
2. The order and nesting of subsequent messages is shown with a legal nubering scheme in which nested messages have a number appended to them. Nesting is denoted by prepending the incoming message number to the out going message number.



In Figure 15.11 a more complex case is shown.

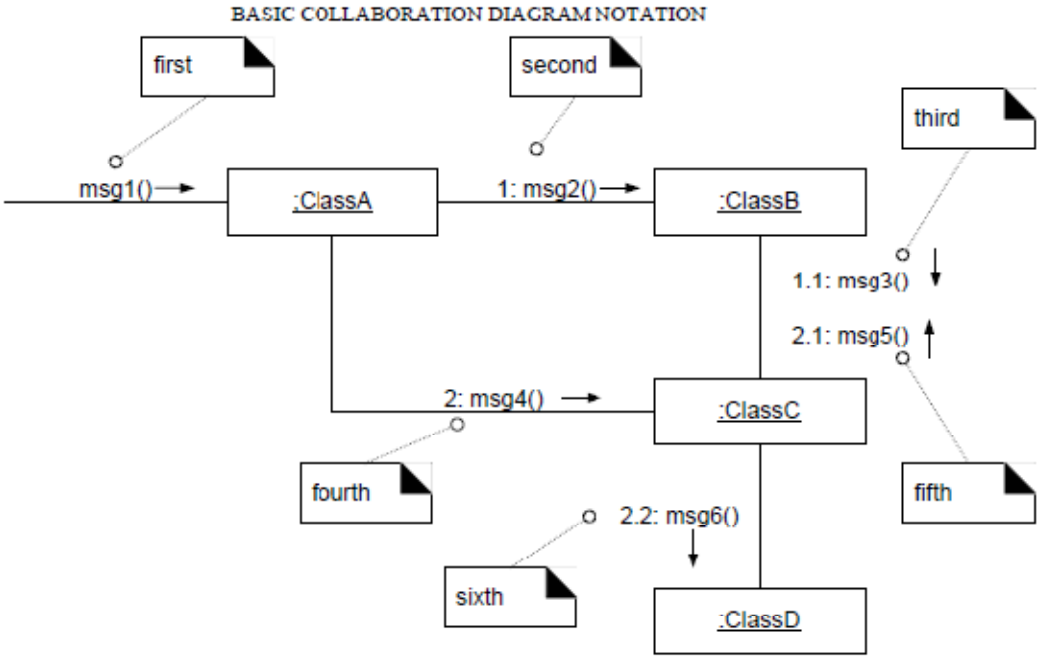


Figure 15.11 Complex sequence numbering.

Conditional Messages:

A conditional message (Figure 15.12) is shown by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to *true*.

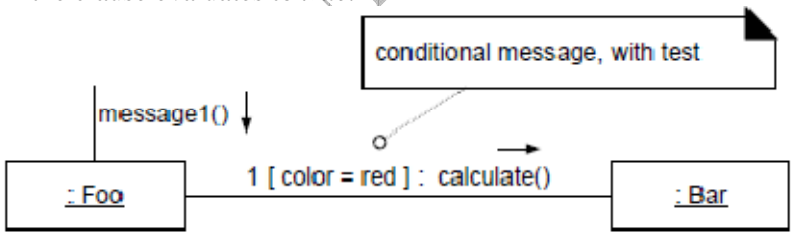


Figure 15.12 Conditional message.

Mutually Exclusive Conditional Paths:

The example in Figure 15.13 illustrates the sequence numbers with mutually exclusive conditional paths.

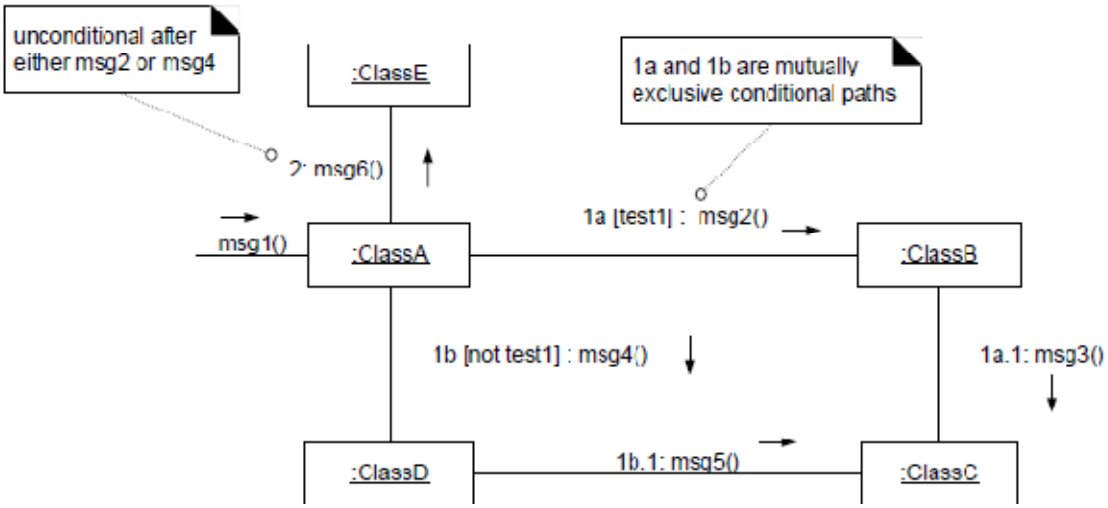


Figure 15.13 Mutually exclusive messages.

Figure 15.13 states that
either *1a* or *1b* could execute after *msg1*. Both are sequence number 1 since either could be the first internal message.
Note that subsequent nested messages are still consistently prepended with their outer message sequence. Thus *Ib. 1* is nested message within *Ib.*

Iteration or Looping:

Iteration notation is shown in Figure 15.14. If the details of the iteration clause are not important to the modeler, a simple '*' can be used.

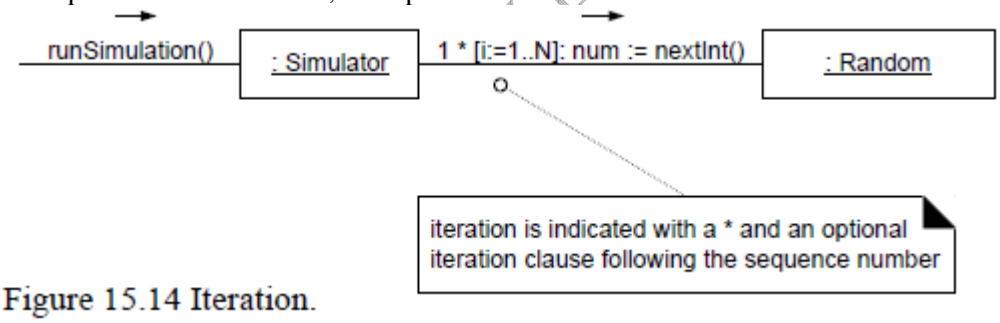


Figure 15.14 Iteration.

Iteration Over a Collection (Multiobject):

A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each. Often, some kind of iterator object is ultimately used, such as an implementation of *java.util.Iterator* or a C++ standard library iterator. In the UML, the term **multiobject** is used to denote a set of instances.a collection. In collaboration diagrams, this can be summarized as shown in Figure 15.15.

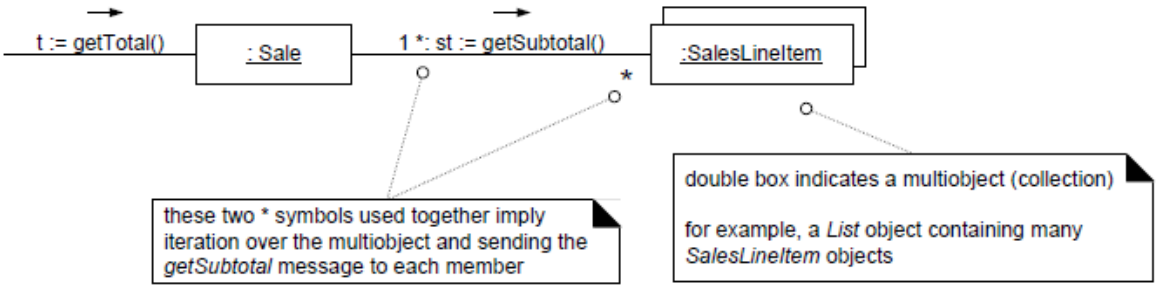


Figure 15.15 Iteration over a multiobject.

The "*" multiplicity marker at the end of the link is used to indicate that the message is being sent to each element of the collection, rather than being repeatedly sent to the collection object itself.

Messages to a Class Object:

Messages may be sent to a class itself, rather than an instance, to invoke class or **static methods.** A message is shown to a class box whose name is not underlined, indicating the message is being sent to a class rather than an instance (see Figure 15.16).

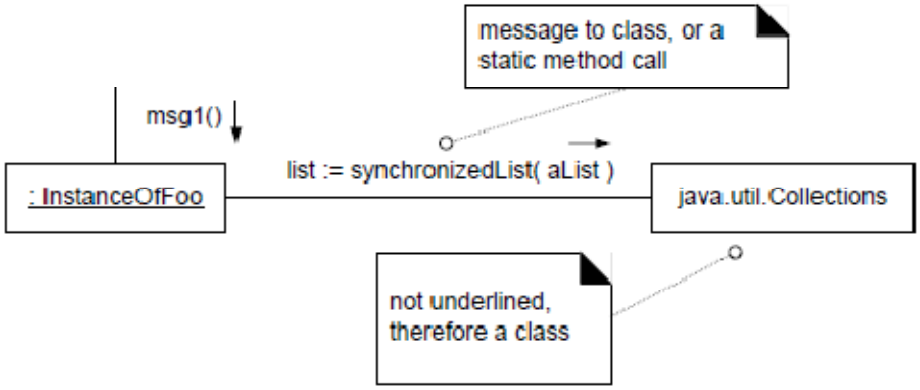


Figure 15.16 Messages to a class object (static method invocation).

Consequently, it is important to be consistent in underlining your instance names when an instance is intended, otherwise messages to instances versus classes may be incorrectly interpreted.

Basic Sequence Diagram Notation:

Links

Unlike collaboration diagrams, sequence diagrams do not show links.

Messages:

Each message between objects is represented with a message expression on an arrowed line between the objects (see Figure 15.17). The time ordering is organized from top to bottom.

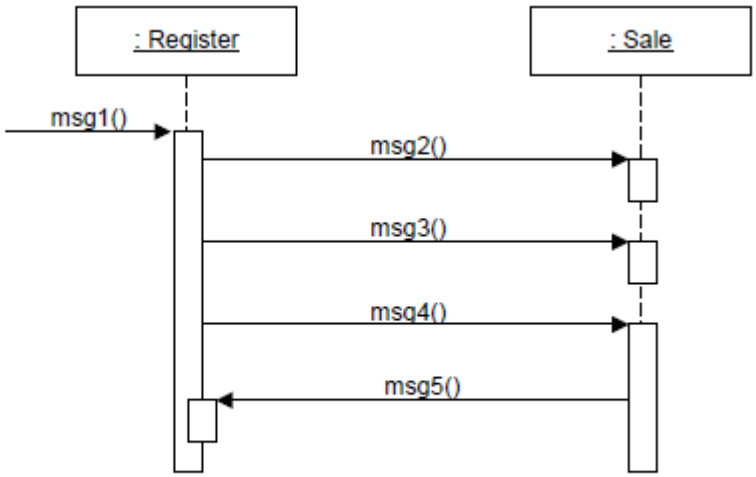


Figure 15.17 Messages and focus of control with activation boxes.

Focus of Control and Activation Boxes:
As illustrated in Figure 15.17, sequence diagrams may also show the focus of control (that is, in a regular blocking call, the operation is on the call stack) using an **activation box.**
The focus of control is a rectangular box , representing the period during which an element is performing the operation.

Illustrating Returns
A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box (see Figure 15.18

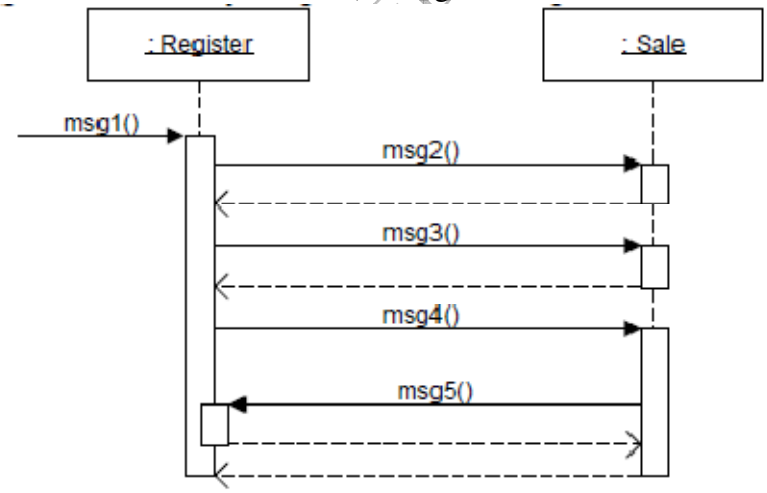


Figure 15.18 Showing returns.

Messages to "self" or "this"
A message can be illustrated as being sent from an object to itself by using a nested activation box (see Figure 15.19).

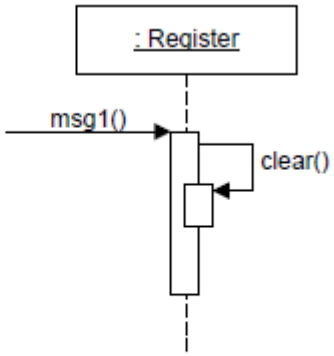


Figure 15.19 Messages to "this."

Creation of Instances

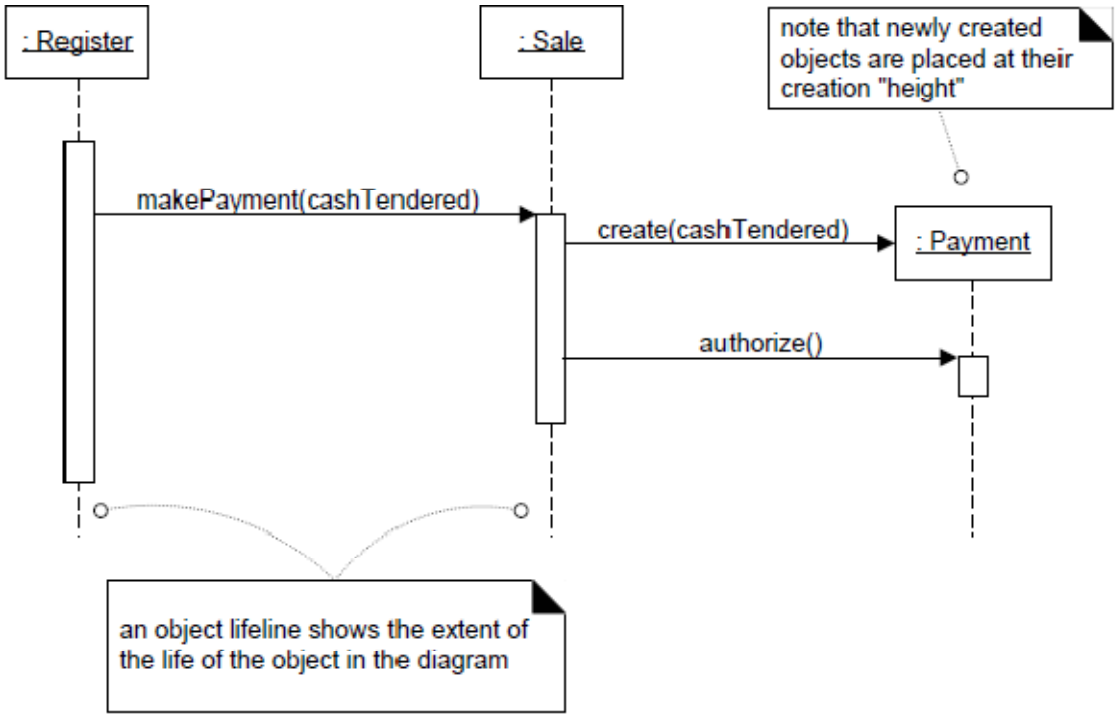


Figure 15.20 Instance creation and object lifelines.

Object Lifelines and Object Destruction:

Figure 15.20 also illustrates **object lifelines**.the vertical dashed lines underneath the objects. These indicate the time that an object exists

In some circumstances it is desirable to show explicit destruction of an object (as in C++, which does not have garbage collection); the UML lifeline notation provides a way to express this destruction (see Figure 15.21).

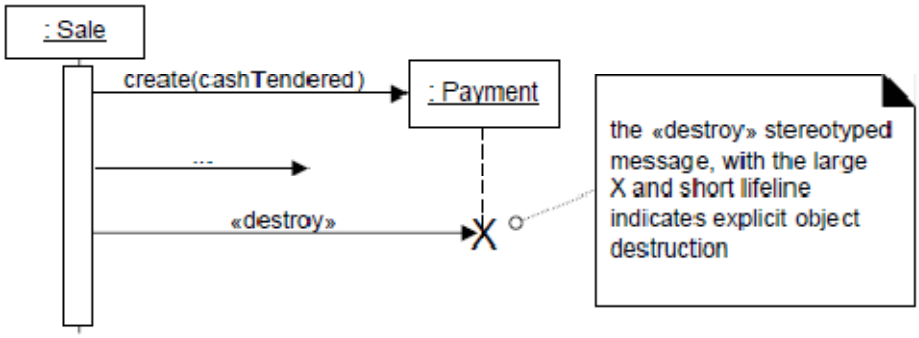


Figure 15.21 Object destruction

Conditional Messages:

A conditional message is shown in Figure 15.22.

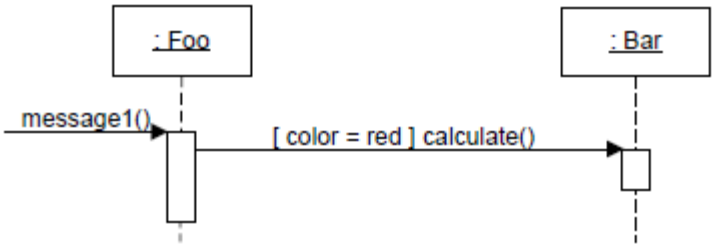


Figure 15.22 A conditional message.

Mutually Exclusive Conditional Messages:

The notation for this case is a kind of angled message line emerging from a common point, as illustrated in Figure 15.23.

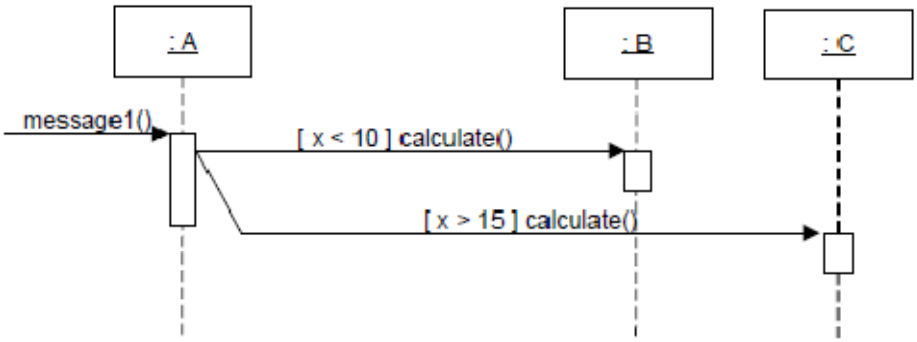


Figure 15.23 Mutually exclusive conditional messages.

Iteration for a Single Message:

Iteration notation for one message is shown in Figure 15.24.

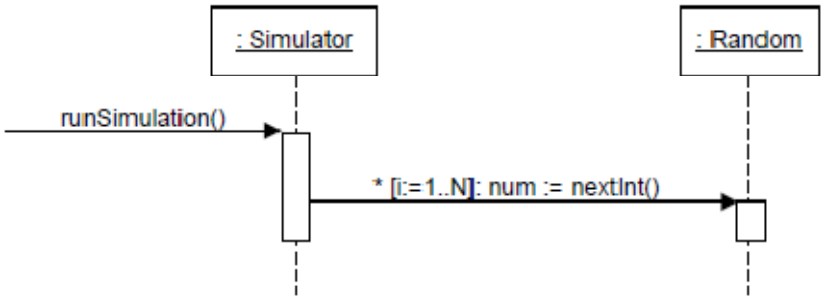


Figure 15.24 Iteration for one message.

Iteration of a Series of Messages
Notation to indicate iteration around a series of messages is shown in Figure 15.25.

Iteration Over a Collection (Multiobject):
In sequence diagrams, iteration over a collection is shown in Figure 15.26. With collaboration diagrams the UML specifies a '*' multiplicity marker at the end of the role (next to the multiobject) to indicate sending a message to each element rather than repeatedly to the collection itself. However, the UML does not specify how to indicate this with sequence diagrams.

Messages to Class Objects:
As in a collaboration diagram, class or static method calls are shown by not underlining the name of the classifier, which signifies a class object rather than an instance (see Figure 15.27).

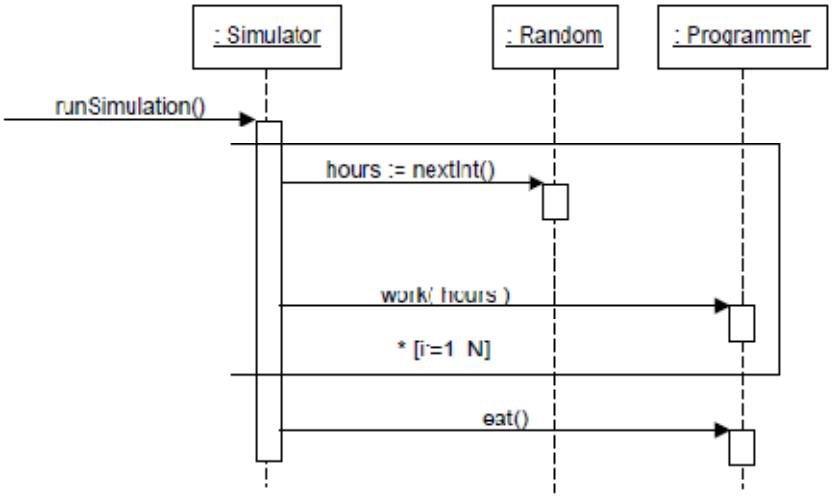


Figure 15.25 Iteration for a sequence of

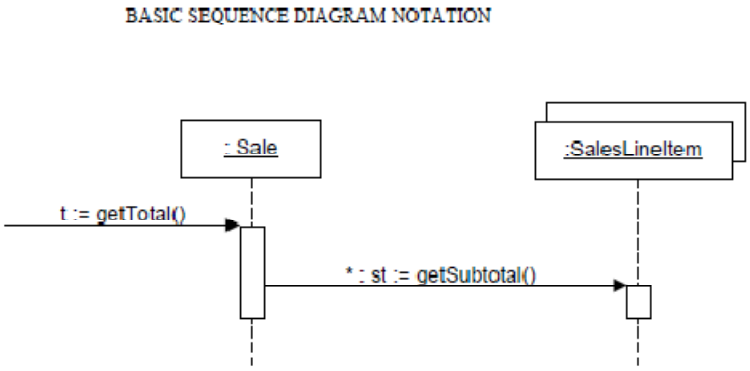


Figure 15.26 Iteration over a multiobject

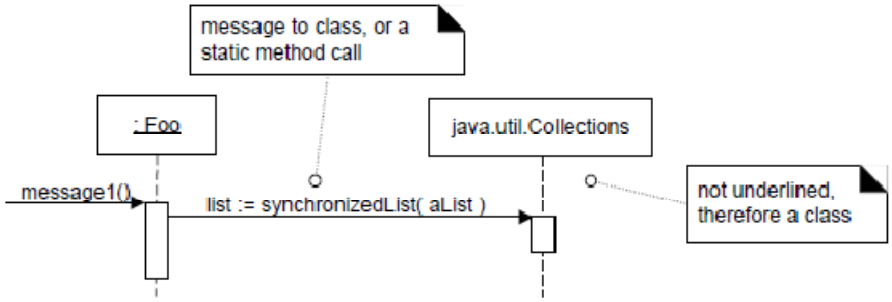


Figure 15.27 Invoking class or static methods

Introduction to GRASP design Patterns:

GRASP: General Responsibility Assignment Software Patterns
Objectives:

Define patterns.
Learn to apply five of the GRASP patterns.

Introduction:
After identifying your requirements and creating a domainmodel, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.

Deciding what methods belong where, and how the objects should interact. this is a critical step.this is at the heart of what it means to develop an object-oriented system.

GRASP as a Methodical Approach to Learning Basic Object Design The skillful assignment of responsibilities during the creation of interaction diagrams.

Responsibilities and Methods:

The UML defines a **responsibility** as "a contract or obligation of a classifier". Basically, these responsibilities are of the following two types:

- knowing
- doing

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects during object design. For example, I may declare that "a *Sale* is responsible for creating *SalesLineItems"* (a doing), or "a *Sale* is responsible for knowing its total" (a knowing).

Responsibilities and Interaction Diagrams:

The fundamental principles of GRASP patterns are assigning responsibilities to objects. This Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams (which are part of the UP Design Model),

Patterns:

A pattern is a recurring solution to a standard problem, in a particular context.

(or)

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design.

(or)

In OO design, a pattern is a named description of a problem and solution that can be applied in new contexts.

For example, the structure of a pattern is as follows:

Pattern Name
Problem
Solution

Patterns Have Names:

Naming a pattern has the following advantages:

. understanding and memorize the pattern.

. It facilitates communication.

.reducing a complex form to a simple one by eliminating detail.

GRASP is an acronym that stands for ***General Responsibility Assignment Software Patterns*** The name was chosen to suggest the importance of *grasp ing* these principles to successfully design object-oriented software.

GRASP Patterns:
The following sections present the first five GRASP patterns:

- ***Creator***
- ***Information Expert***
- ***Controller***
- ***Low Coupling***
- ***High Cohesion***
- ***Polymorphism***
- ***Protected variations***
- ***Indirection(Discussed in 4th unit)***
- ***Pure fabrication(Discussed in 4th unit)***

these five address very basic, common questions and fundamental design issues.

Creator:

Problem: Creation of objects is one of the most common activities in an object-oriented system. Who should be responsible for creating a new instance of some class?. Which class is responsible for creating objects?. Assign creation responsibilities in such a way that it can support low coupling, increased clarity, encapsulation, and reusability.

In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

- Assign class B the responsibility to create an instance of class A if one or more of
- the following is true:
- . B *aggregates* A objects.
- . B *contains* A objects.
- . B *records* instances of A objects.
- . B *closely uses* A objects.
- . B *has the initializing data* that will be passed to A when it is created (thus B
- is an Expert with respect to creating A).
- B is a *creator* of A objects.

Example: In the POS application, who should be responsible for creating a *SalesLineItem* instance?. we should look for a class that aggregates, contains *SalesLineItem* instances. Consider the partial domain model in Figure 16.7.

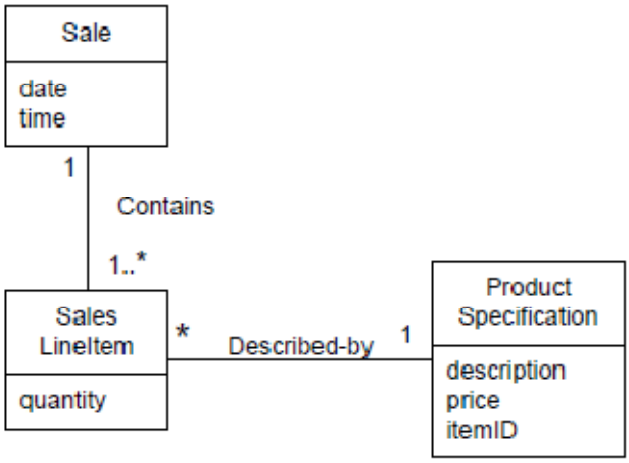


Figure 16.7 Partial domain model.

Since a Sate contains (in fact, aggregates) many *SalesLineItem* objects, the Creator pattern suggests that *Sale* is a good candidate to have the responsibility of creating *SalesLineItem* instances.

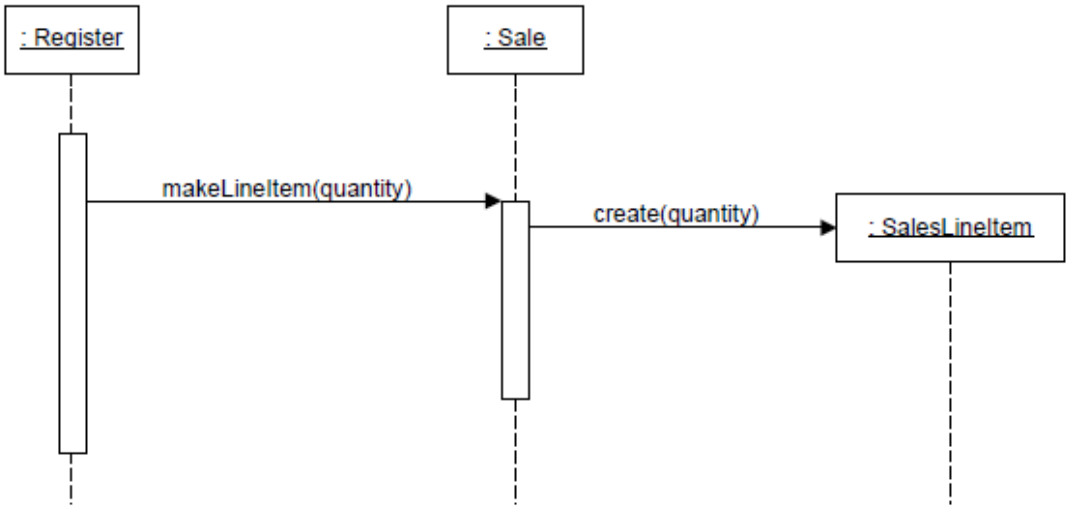


Figure 16.8 Creating a SalesLineItem.

This assignment of responsibilities requires that a *makeLineItem* method be defined in Sale.

Information Expert (or Expert):

Problem What is a general principle of assigning responsibilities to objects?. An application may require hundreds or thousands of responsibilities to be fulfilled. When the interactions between objects are defined, we assign the responsibilities to software classes. Well assigned responsibilities tend to be easier to understand, maintain, and extend, and there is more opportunity to reuse components in future applications.

These responsibilities include methods, computed fields, and so on.

Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

Solution Assign a responsibility to the information expert. The class that has the i*nformation* necessary to fulfill the responsibility, and then determine where that information is stored.

Example In the NextGEN POS application, some class needs to know the grand total of a sale.
the statement is: *Who should be responsible for knowing the grand total of a sale"?*
By *Information Expert,* we should look for that class of objects that has the information needed to determine the total.

consider the partial Domain Model in Figure

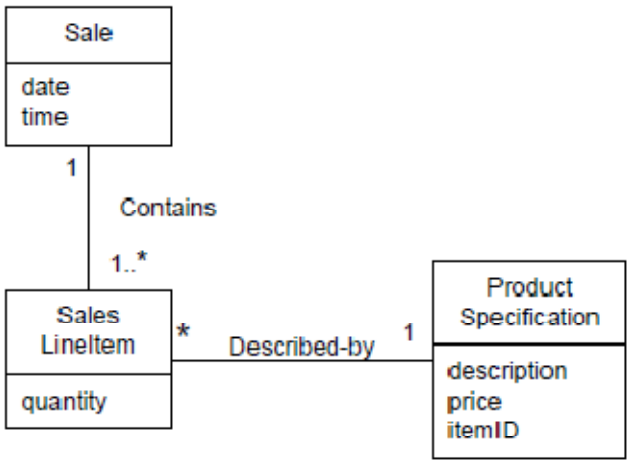


Figure 16.3 Associations of Sale.

What information is needed to determine the grand total? It is necessary to know about all the *SalesLineItem* instances of a sale and the sum of their subtotals. A *Sale* instance contains these. Therefore, by the guideline of Information Expert, *Sale* is a suitable class of object for this responsibility; it is an *information expert* for the work.

What information is needed to determine the line item subtotal?. *SalesLineItem.quantity* and *ProductSpecification.price* are needed. The *SalesLineItem* knows its quantity and its associated *ProductSpecification;* therefore, by Expert, *SalesLineItem* should determine the subtotal; it is the *information expert.*

In terms of an interaction diagram, this means that the *Sale* needs to send *get-Subtotal* messages to each of the *SalesLineItems* and sum the results; this design is shown in Figure 16.5.

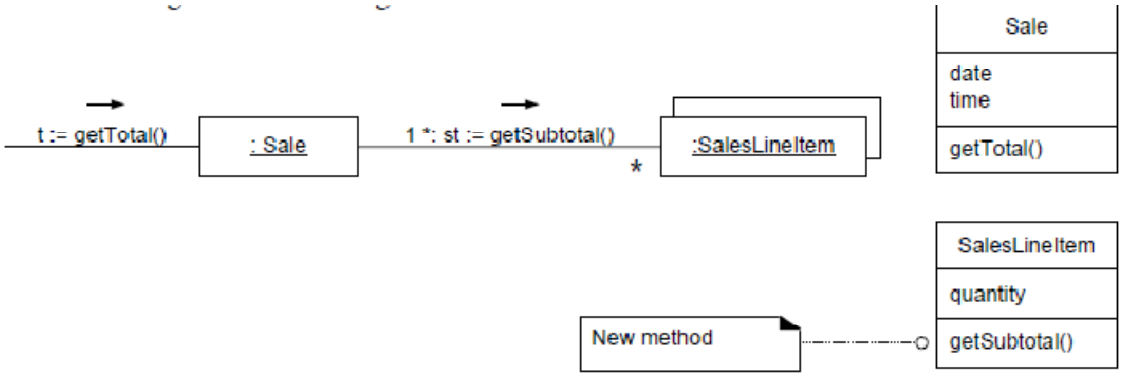


Figure 16.5 Calculating the Sale total

To fulfill the responsibility of knowing and answering its subtotal, a *Sales-Lineltem* needs to know the product price.
The *ProductSpecification* is an information expert on answering its price; therefore, a message must be sent to it asking for its price. The design is shown in Figure 16.6.
In conclusion, to fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects as follows.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

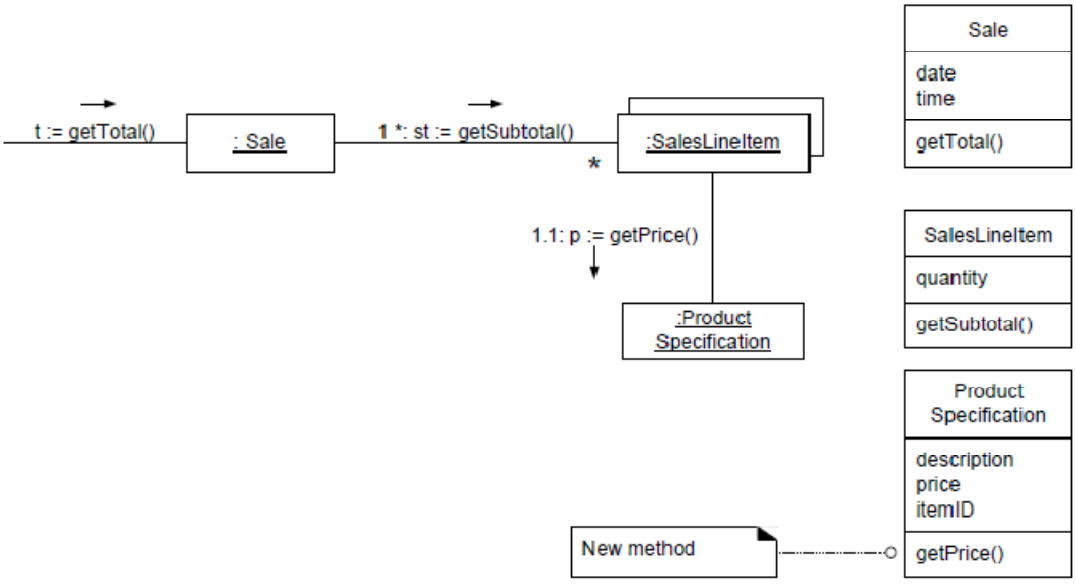


Figure 16.6 Calculating the *Sale* total.

Controller:

Problem Who should be responsible for handling an input system event?. An input **system event** is an event generated by an external actor. They are associated with **system operations**. operations of the system in response to system events.

Solution A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem *(facade controller).*
- Represents a use case scenario within which the system event occurs, often named *use-case or session controller.*
- Use the same controller class for all system events in the same use case scenario.
- Informally, a session is an instance of a conversation with an actor. Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

Example In the NextGen application, there are several system operations, as illustrated in Figure 16.13, showing the system itself as a class or component (which is legal in the UML).

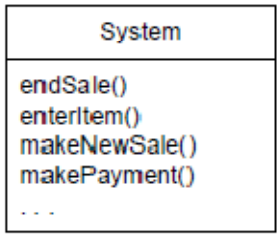


Figure 16.13 System operations associated with the system events.

During analysis, system operations may be assigned to the class *System*, to indicate they are system operations. However, this does *not* mean that a software class named *System* fulfills them during design. Rather, during design, a Controller class is assigned the responsibility for system operations (see Figure 16.14).

Who should be the controller for system events such as *enterItem* and *endSale*

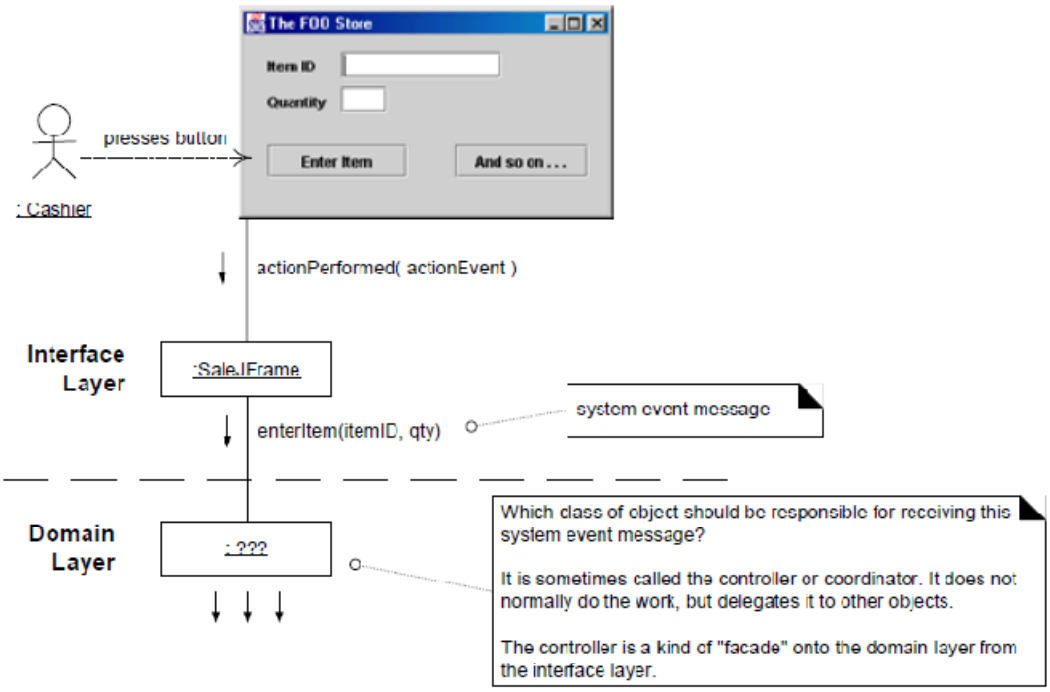


Figure 16.14 Controller for enterItem?

By the Controller pattern, here are some choices:

represents the overall "system," device, or subsystem	*Register, POSSystem*
represents a receiver or handler of all system events of a use case scenario	*ProcessSaleHandler, ProcessSaleSestsion*

In terms of interaction diagrams, it means that one of the examples in Figure 16.15 may be useful.

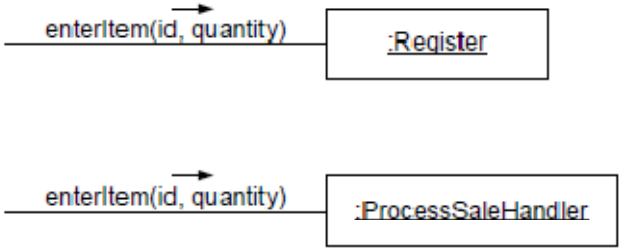


Figure 16.15 Controller choices.

Facade controllers are suitable when there are not "too many" system events, or it is not possible for the user interface (UI) to redirect system event messages to alternating controllers, such as in a message processing system. If a use-case controller is chosen, then there is a different controller for each use case. Note that this is not a domain object; it is an

artificial construct to support the system (a *Pure Fabrication* in terms of the GRASP patterns). For example, if the NextGen application contains use cases such as *Process Sale* and *Handle Returns,* then there may be a *ProcessSaleHandler* class and so forth. When should you choose a use-case controller? It is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling, typically when the facade controller is becoming "bloated" with excessive responsibilities. A use-case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes, and also provides a basis for knowing and reasoning about the state of the current scenario in progress. In the UP and Jacobson's older Objectory method, there are the (optional) concepts of boundary, control, and entity classes. **Boundary objects** are abstractions of the interfaces, **entity objects** are the application-independent (and typically persistent) domain software objects, and **control objects** are use case handlers as described in this Controller pattern.

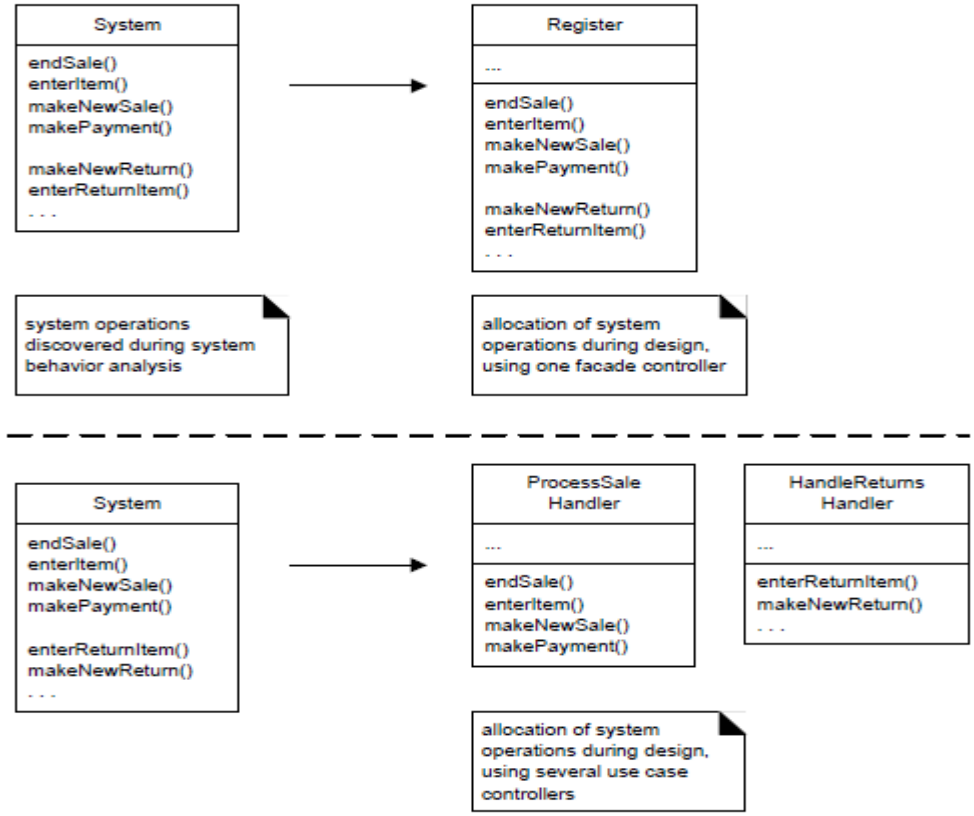


Figure 16.16 Allocation of system operations.

Bloated Controllers

Poorly designed, a controller class will have low cohesion. unfocused and handling too many areas of responsibility; this is called a bloated controller.

Signs of bloating include:

. There is only a *single* controller class receiving *all* system events in the sys tem, and there are many of them. This sometimes happens if a facade controller is chosen.

. The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.

. A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

There are several cures to a bloated controller, including:
1. Add more controllers a system does not have to have only one. Instead of facade Controllers, use use-case controllers. For example, consider an application with many system events, such as an airline reservation system.
It may contain the following controllers:

Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.

Message Handling Systems and the Command Pattern
Some applications are message-handling systems or servers that receive requests from other processes. A telecommunications switch is a common example. In such systems, the design of the interface and controller is somewhat different. The details are explored in a later chapter, but in essence, a common solution is to use the Command pattern and Command Processor pattern.

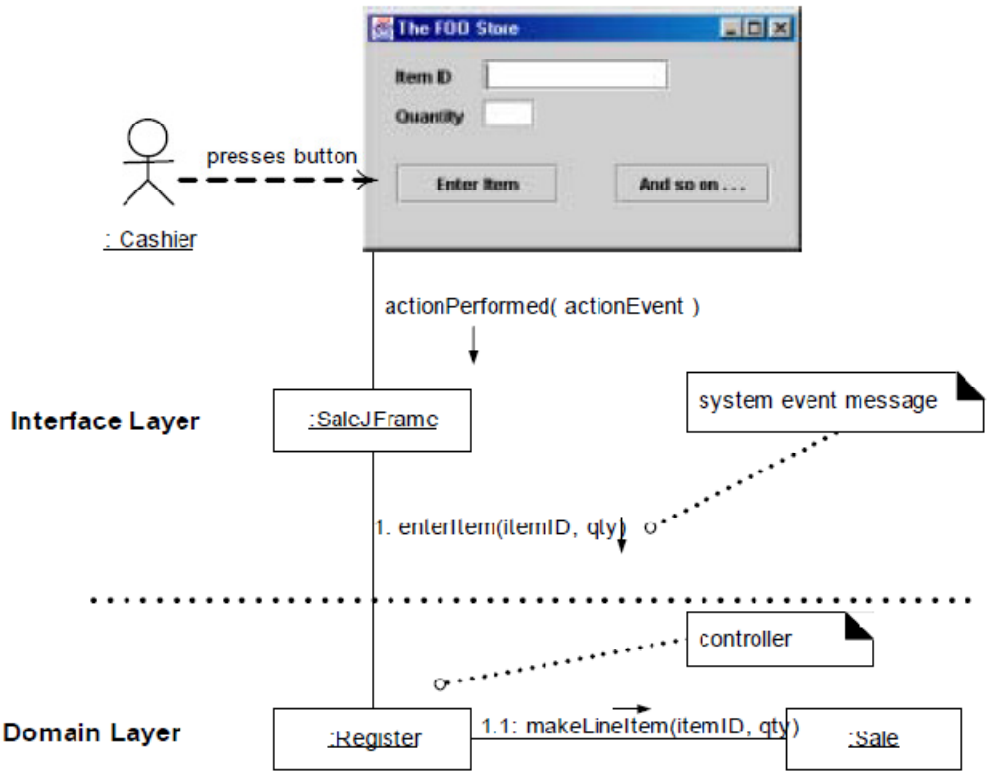


Figure 16.17 Desirable coupling of interface layer to domain layer.

Low Coupling:

Problem How to support low dependency, low change impact, and increased reuse?. Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. **Low coupling** is an evaluative pattern that dictates how to assign responsibilities to support:

Solution Assign a responsibility based on the following factors, so that coupling remains low.

- lower dependency between the classes,
- change in one class having lower impact on other classes,
- Higher reuse potential.

Example Consider the following partial class diagram from a NextGen case study:

Assume we have a need to create a *Payment* instance and associate it with the *Sale.* What class should be responsible for this? Since a *Register* "records" a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment.* The *Register* instance could then send an *addPayment* message to the *Sale,* passing along the new *Payment* as a parameter.

A possible partial interaction diagram reflecting this is shown in Figure 16.9.

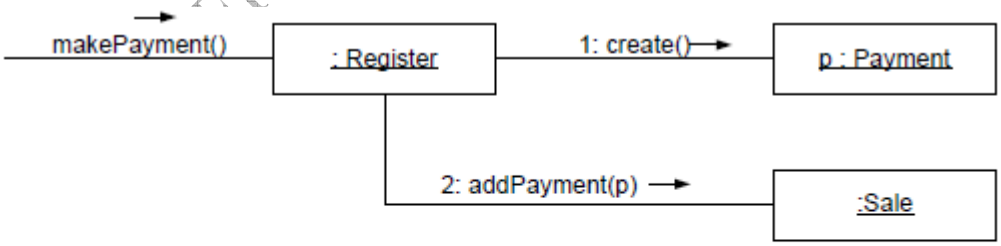


Figure 16.9 Register creates Payment.

This assignment of responsibilities couples the *Register* class to knowledge of the *Payment* class.

An alternative solution to creating the *Payment* and associating it with the *Sale* is shown in Figure 16.10.

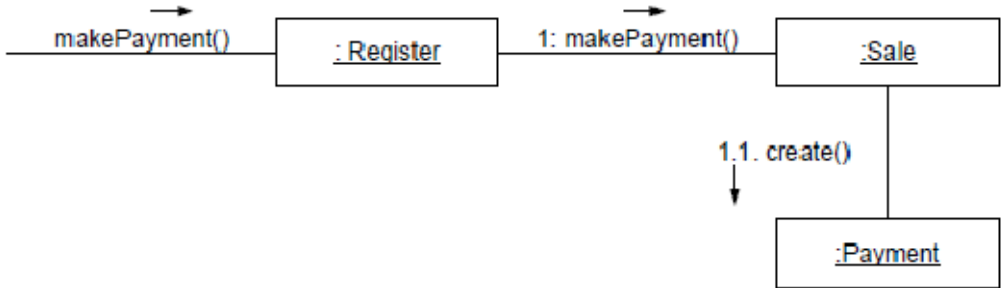


Figure 16.10 Sale creates Payment.

In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion. Nevertheless, it is one factor to consider in improving a design.

Benefits 1. not affected by changes in other components
2. simple to understand in isolation
3. convenient to reuse

High Cohesion:

Problem How to keep complexity manageable?
In terms of object design, **cohesion** (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on

Solution Assign a responsibility so that cohesion remains high. .A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
.-hard to reuse
.-hard to maintain
-delicate; constantly effected by change
-Low cohesion classes often represent a very "large grain" of abstraction, or have taken on responsibilities that should have been delegated to other objects.

Example The same example problem used in the Low Coupling pattern can be analyzed for High Cohesion.
Assume we have a need to create a (cash) *Payment* instance and associate it with the *Sale.* What class should be responsible for this? Since *Register* records a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment.* The *Register* instance could then send an *addPayrnent* message to the *Sale,* passing along the new *Payment* as a parameter, as shown in Figure 16.11.

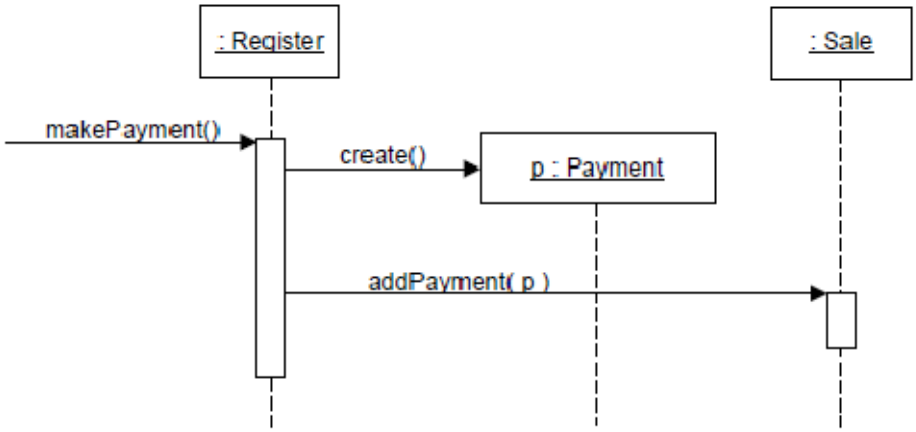


Figure 16.11 Register creates Payment.

This assignment of responsibilities places the responsibility for making a payment in the *Register.* The *Register* is taking on part of the responsibility for fulfilling the *makePayment* system operation.

In this isolated example, this is acceptable; but if we continue to make the *Register* class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

Imagine that there were fifty system operations, all received by *Register.* If it did the work related to each, it would become a "bloated" incohesive object. The point is not that this single *Payment* creation task in itself makes the *Register* incohesive, but as part of a larger picture of overall responsibility assignment, it may suggest a trend toward low cohesion.

And most important in terms of developing skills as an object designer, regardless of the final design choice, the valuable thing is that at least a developer knows to consider the impact on cohesion.

By contrast, as shown in Figure 16.12, the second design delegates the payment creation responsibility to the *Sale,* which supports higher cohesion in the Since the second design supports both high cohesion and low coupling, it is desirable.

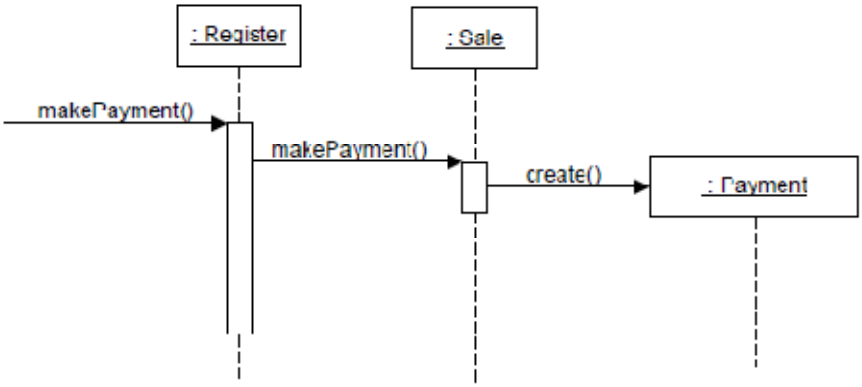


Figure 16.12 Sale creates Payment

In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.

Benefits . Clarity and ease of comprehension of the design is increased.
. Maintenance and enhancements are simplified.
. Low coupling is often supported.
. The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose.

Polymorphism:

Problem How to handle alternatives based on type? How to create pluggable software components?
Solution When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.

Alternatives based on type—Conditional variation is a fundamental theme in programs. If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic. This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places—wherever the conditional logic exists,
Pluggable software components—Viewing components in client-server relationships, how can you replace one server component with another, without affecting the client?
Example In the NextGen POS application, there are multiple external third-party tax calculators that must be supported (such as Tax-Master and Good-As-Gold Tax-Pro); the system needs to be able to integrate with different ones. Each tax calculator has a different interface, and so there is similar but varying behaviour to adapt to each of these external fixed interfaces or APIs. One product may support a raw TCP socket protocol, another may offer a SOAP interface, and a third may offer a Java RMI interface. What objects should be responsible for handling these varying external tax calculator interfaces?
1. Polymorphism has several related meanings. In this context, it means "giving the same name to services in different objectswhen the services are similar or related. The different object types usually implement a common interface or are related in an implementation hierarchy with a common superclass, but this is language- dependent; for example, dynamic binding languages such as Smalltalk do not Since the behavior of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator (or calculator adapter) objects themselves, implemented with a polymorphic *getTaxes* operation (see Figure 22.1).
These calculator adapter objects are not the external calculators, but rather, local software objects that represent the external calculators, or the adapter for the calculator. By sending a message to the local object, a call will ultimately be made on the external calculator in its native API. Each *getTaxes* method takes the *Sale* object as a parameter, so that the calculator can analyze the sale. The implementation of each *getTaxes* method will be different: *TaxMasterAdapter* will adapt the request to the API of Tax-Master, and so on.

TaxMasterAdapter

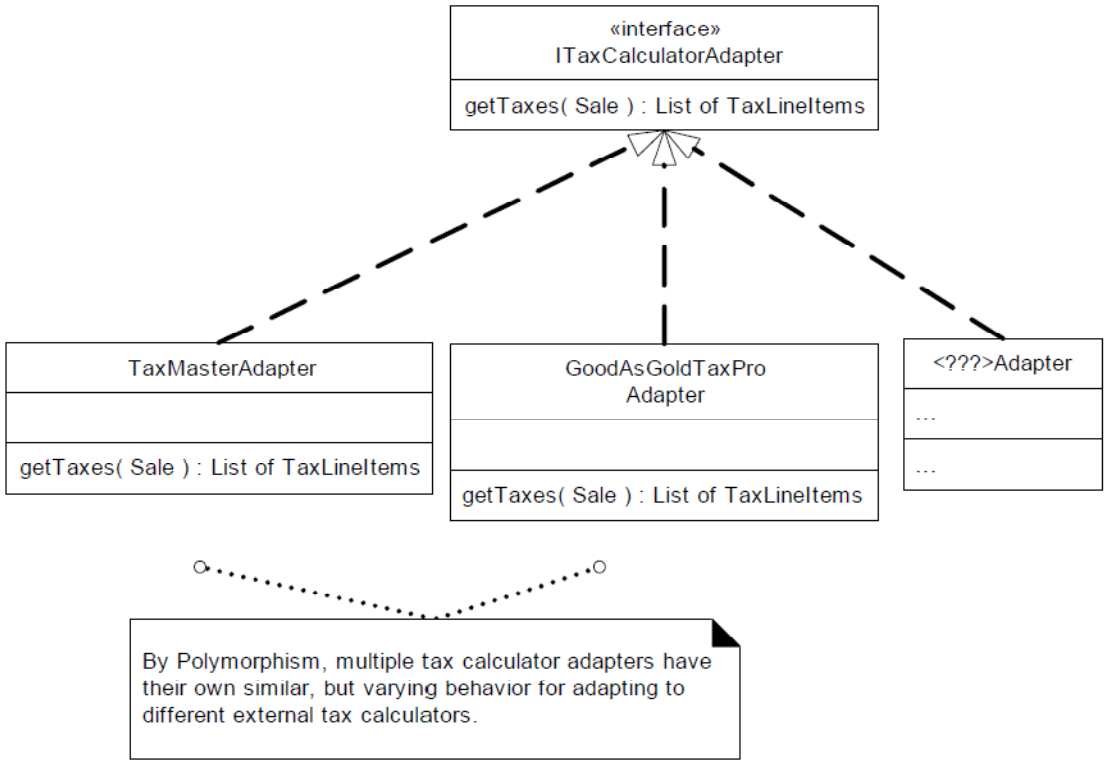


Figure 22.1 Polymorphism in adapting to different external tax calculators.

UML notation—Figure 22.1 introduces some new UML notation for specifying **interfaces** (a descriptor of operations without implementation), interface implementation, and for "collection" return types; Figure 22.2 elaborates. A UML **stereotype** is used to indicate an interface; a stereotype is a mechanism to categorize an element in some way. A stereotype name is surrounded by guillemets symbols, as in «interface». Guillemets are special *single-character* brackets most widely known by their use in French typography to indicate a quote; but to quote Rumbaugh, "the typographically challenged could substitute two angle brackets (« ») if necessary" .

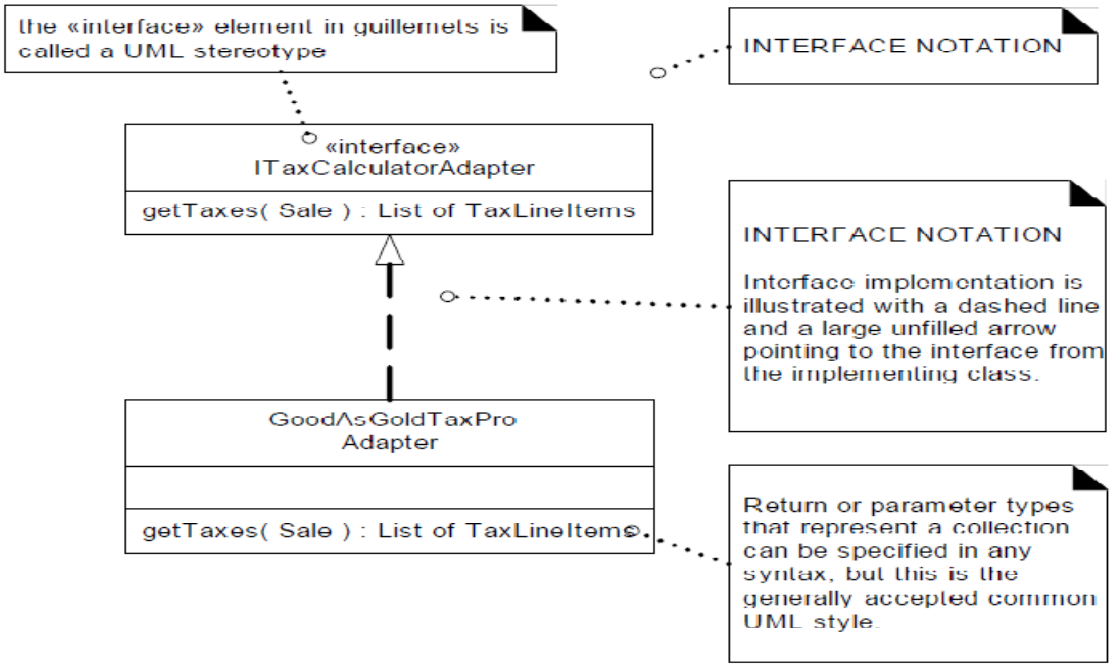


Figure 22.2 UML notation for interfaces and return types.

Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations. A design based on assigning responsibilities by Polymorphism can be easily extended to handle new variations. For example, adding a new calculator adapter class with its own polymorphic *getTaxet* method will have minor impact on the existing design. Sometimes, developers design systems with interfaces and polymorphism for speculative "future-proofing" against an unknown possible variation. If the variation point is definitely motivated by an immediate or very probable variability then the effort of adding the flexibility through polymorphism is of course rational. But critical evaluation is required, because it is not uncommon to see unnecessary effort being applied to future-proofing a design with polymorphism at variation points that in fact are improbable and will never actually arise. Be realistic about the true likelihood of variability before investing in increased flexibility.

• Extensions required for new variations are easy to add.

• New implementations can be introduced without affecting clients.

Protected Variations• A number of popular GoF design patterns rely on polymorphism, including Adapter, Command, Composite, Proxy, State, and Strategy. Choosing Message, Don't Ask "What Kind?"

Protected Variations:

Problem How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

Solution Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them. Note: The term "interface" is used in the broadest sense of an access view; it does not literally only mean something like a Java or COM interface.

Example For example, the prior external tax calculator problem and its solution with Polymorphism illustrate Protected Variations (Figure 22.1). The point of instability or variation is the different interfaces or APIs of external tax calculators. The POS system needs to be able to integrate with many existing tax calculator systems, and also with future third-party calculators not yet in existence. By adding a level of indirection, an interface, and using polymorphism with various *ITaxCalculatorAdapter* implementations, protection within the system from variations in external APIs is achieved. Internal objects collaborate with a stable interface; the various adapter implementations hide the variations to the external systems.

Object Design and CRC Cards:

Although not formally part of the UML, another device sometimes used to help assign responsibilities and indicate collaboration with other objects are **CRC cards** (Class-Responsibility-Collaborator cards)

CRC cards are index cards, one for each class, upon which the responsibilities of the class are briefly written, and a list of collaborator objects to fulfill those responsibilities. They are usually developed in a small group session. The GRASP patterns may be applied when considering the design while using CRC cards.

CRC cards are one approach to recording the results of responsibility assignment and collaborations. The recording can be enhanced with the use of interaction and class diagrams. The real value is not the cards or the diagrams, but the consideration of responsibility assignment.

Design Model: Use case realizations with GRASP patterns:

Objectives

• Design use-case realizations.
• Apply the GRASP patterns to assign responsibilities to classes.
• Use the UML interaction diagram notation to illustrate the design of objects.

Now we explore how to create a design of collaborating objects with responsibilities.

Note:

> The assignment of responsibilities and design of collaborations are very important and creative steps during design, either while diagraming or while programming.

Use-Case Realizations:

A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects. More precisely, a designer can describe the design of one or more *scenarios* of a use case; each of these is called a use-case realization. Use-case realization is a UP term or concept used to remind us of the connection between the requirements expressed as use cases, and the object design that satisfies the requirements.

UML interaction diagrams are a common to illustrate use-case realizations.

Interaction Diagrams and Use-Case Realizations:
consider various scenarios and system events such as:
• *Process Scale: makeNewSale, enterItem, endSale, makePayment*
If collaboration diagrams are used to illustrate the use-case realizations, a different collaboration diagram will be required to show the handling of each system event message. For example (Figure 17.1):

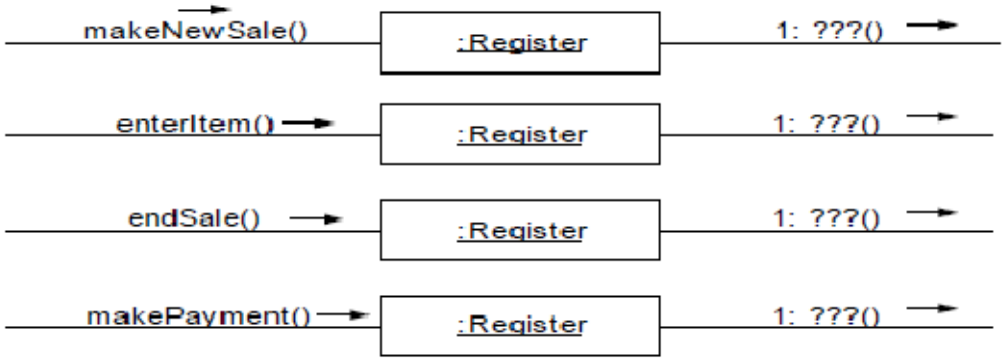


Figure 17.1 Collaboration diagrams and system event message handling.

On the other hand, if sequence diagrams are used, it *may* be possible to fit all system event messages on the same diagram, as in Figure 17.2.

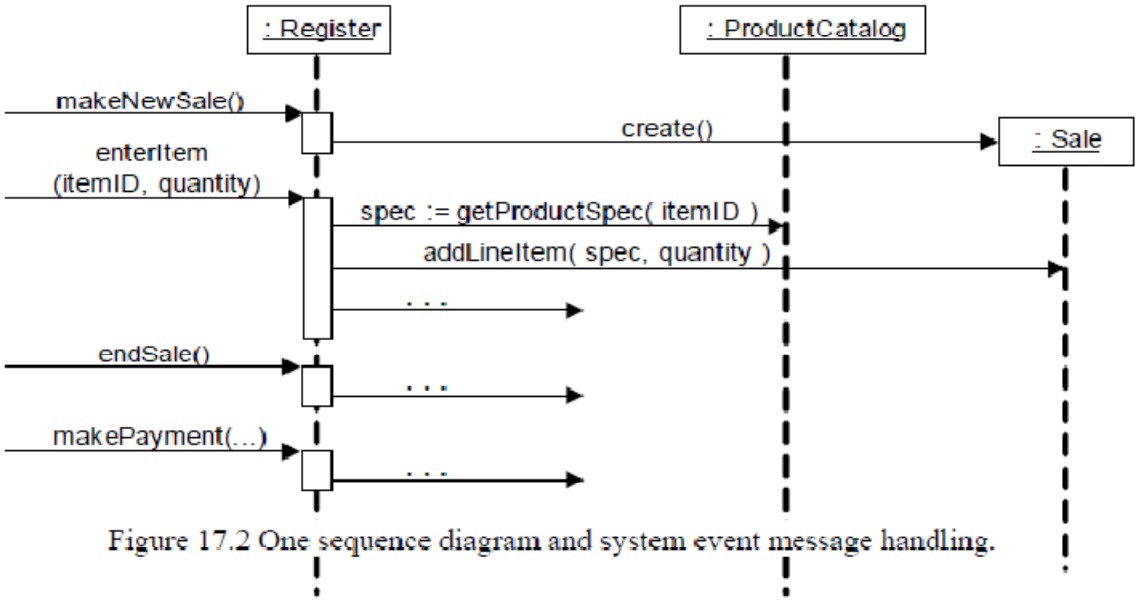


Figure 17.2 One sequence diagram and system event message handling.

However, it is often the case that the sequence diagram is then too complex or long. It is legal, as with interaction diagrams, to use a sequence diagram for each system event message, as in Figure 17.3.

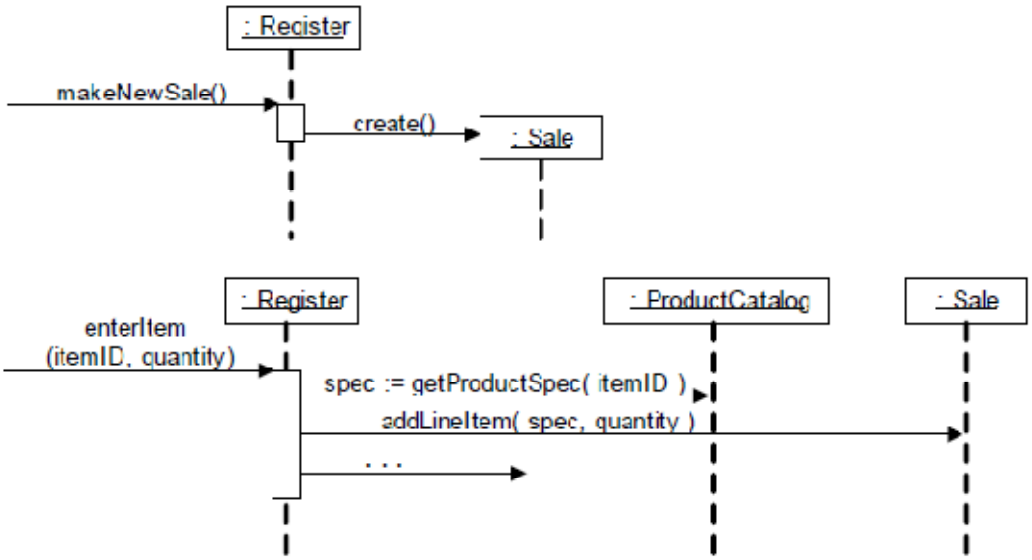


Figure 17.3 Multiple sequence diagrams and system event message handling.

Contracts and Use-Case Realizations:

it may be possible to design use-case realizations directly from the use case text. In addition, for some system operations, contracts may have been written that add greater detail or specificity. For example:

Contract CO2: enterItem

Operation: Cross References:	enterItem(itemID : ItemID, quantity : integer) Use Cases: Process Sale There is a sale underway.
Preconditions:	- A SalesLineItem instance sli was created (instance creation).
Postconditions:	

In conjunction with contemplating the use case text, for each contract, we work through the postcondition state changes and design message interactions to satisfy the requirements. For example, given this partial *enterItem* system opera- tion, a partial interaction diagram is shown in Figure 17.4 that satisfies the state change of *SalesLineItem* instance creation.

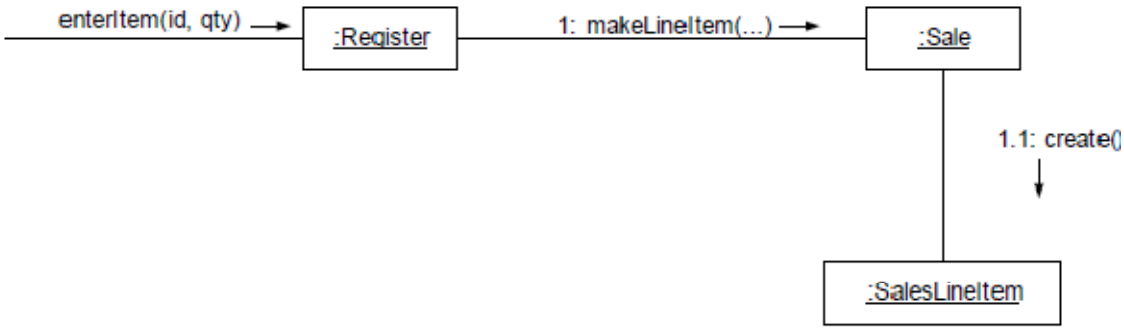


Figure 17.4 Partial interaction diagram.

The Domain Model and Use-Case Realizations:

Some of the software objects that interact via messages in the interaction diagrams are inspired from the Domain Model, such as a *Sale* conceptual class and *Sale* design class.

Use-Case Realizations for the NextGen Iteration:

Creating a New Sale

A software *Sale* object must be created, and the GRASP Creator pattern suggests assigning the responsibility for creation to a class that aggregates, contains, or records the object to be created.

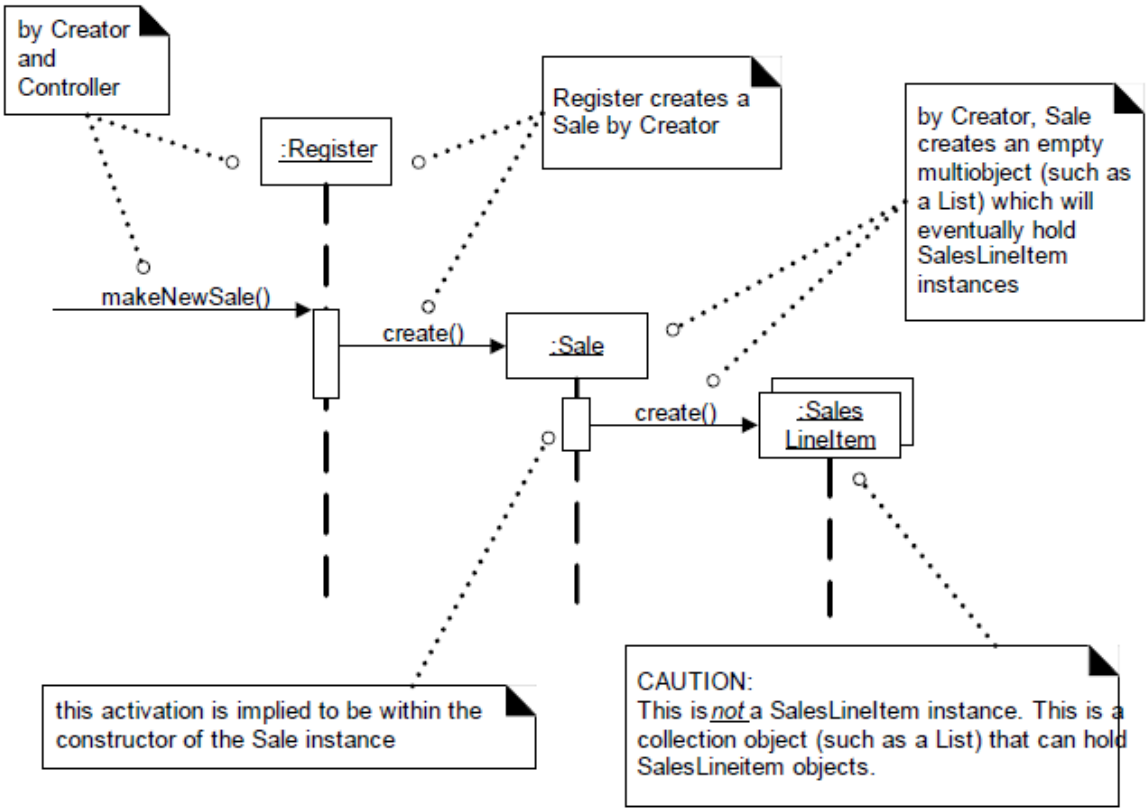


Figure 17.7 Sale and multiobject creation.

Analyzing the Domain Model reveals that a *Register* may be thought of as recording a *Sale;* indeed, the word "register" in business has for many years meant the thing that recorded (or registered) account transactions, such as sales.

Thus, *Register* is a reasonable candidate for creating a *Sale.* And by having the *Register* create the *Sale,* the *Register* can easily be associated with it over time, so that during future operations within the session, the *Register* will have a reference to the current *Sale* instance.

In addition to the above, when the *Sale* is created, it must create an empty collection (container, such as a Java *List)* to record all the future *SalesLineItem* instances that will be added. This collection will be contained within and maintained by the *Sale* instance, which implies by Creator that the *Sale* is a good candidate for creating it.

Therefore, the *Register* creates the *Sale,* and the *Sale* creates an empty collection, represented by a multiobject in the interaction diagram. Hence, the interaction diagram in Figure 17.7 illustrates the design.

Calculating the Sale Total

Consider this fragment of the *Process Sale* use case:

Main Success Scenario:

1. Customer arrives ...
2. Cashier tells System to create a new sale.
3. Cashier enters item identifier.
4. System records sale line item and ...

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
In step 5, a total is presented (or displayed). Because of the Model-View Separation principle, we should not concern ourselves with the design of how the sale total will be displayed, but it is necessary to ensure that the total is known. Note that no design class currently knows the sale total, so we need to create a design of object interactions that satisfies this requirement.
As always, Information Expert should be a pattern to consider unless it is a controller or creation problem (which it is not).
It is probably obvious the *Sale* itself should be responsible for knowing its total, but just to make the reasoning process to find an Expert crystal clear—with a simple example—please consider the following analysis.
1. State the responsibility:
o Who should be responsible for knowing the sale total?
2. Summarize the information required:
o The sale total is the sum of the subtotals of all the sales line-items.
o sales line-item subtotal := line-item quantity * product description price
3. List the information required to fulfill this responsibility and the classes that know this information.

Information Required for Sale Total	**Information Expert**
ProductSpecification.price	*ProductSpecification*
SalesLineItem. quantity	*SalesLineItem*
all the *SalesLineItems* in the current Sale	*Sale*

A detailed analysis follows:
• Who should be responsible for calculating the *Sale* total? By Expert, it should be the *Sale* itself, since it knows about all the *SalesLineItem*
instances whose subtotals must be summed to calculate the sale total. Therefore, *Sale* will have the responsibility of knowing its total, imple mented as a *getTotal* method. • For a *Sale* to calculate its total, it needs the subtotal for each *SalesLineItem.* Who should be responsible for calculating the *SalesLineItem* subtotal? By Expert, it should be the *SalesLineItem* itself, since it knows the quantity and the *ProductSpecification* it is associated with. Therefore, *SalesLineItem* will have the responsibility of knowing its subtotal, implemented as a *get- Subtotal* method.
• For the *SalesLineItem* to calculate its subtotal, it needs the price of the *ProductSpecification.* Who should be responsible for providing the *Product-Specification* price? By Expert, it should be the *ProductSpecification* itself, since it encapsulates the price as an attribute. Therefore, *Product-Specification* will have the responsibility of knowing its price, implemented as *agetPrice* operation.
Although the above analysis is trivial in this case, and the degree of excruciating elaboration presented is uncalled for in actual design practice, the same reasoning strategy to find an Expert can and should be applied in more difficult situations. You will find that once you learn these principles you can quickly perform this kind of reasoning mentally.

Object Design: makePayment:

The *makePayment* system operation occurs when a cashier enters the amount of cash tendered for payment. Here is the complete contract:

Contract CO4: makePayment

Operation: Cross References:	makePayment(amount: Money) Use Cases: Process Sale There is an
Preconditions:	underway sale.
Postconditions:	- A Payment instance p was created (instance creation). - p.amountTendered became amount (attribute modification). - p was associated with the current Sale (association formed). - The current Sale was associated with the Store (associa tion formed); (to add it to the historical log of completed sales).

A design will be constructed to satisfy the postconditions of *makePayment*.

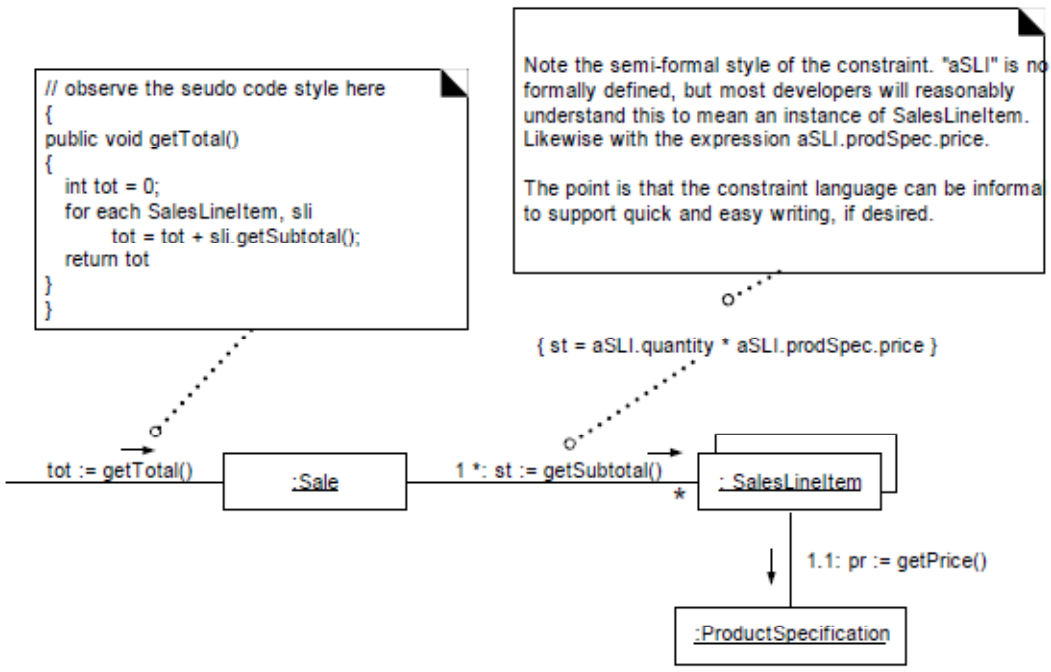


Figure 17.12 Algorithm notes and constraints.

17.13.

Connecting the UI Layer to the Domain Layer:

As has been briefly discussed, applications are organized into logical layers that separate the major concerns of the application, such as the UI layer (for UI concerns) and a "domain" layer (for domain logic concerns). Common designs by which objects in the UI layer obtain visibility to objects in the domain layer include the following:

• An initializing routine (for example, a Java *main* method) creates both a UI and a domain object, and passes the domain object to the UI.
• A UI object retrieves the domain object from a well-known source, such as a factory object that is responsible for creating domain objects.
The sample code shown before is an example of the first approach:

```
public class Main
{

{
Store store = new Store();
Register register = store.getRegister();
ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
...
}
}
```

Once the UI object has a connection to the *Register* instance (the facade controller in this design), it can forward system event messages to it, such as the *enter-Item* and *endSale* message (see Figure 17.18).

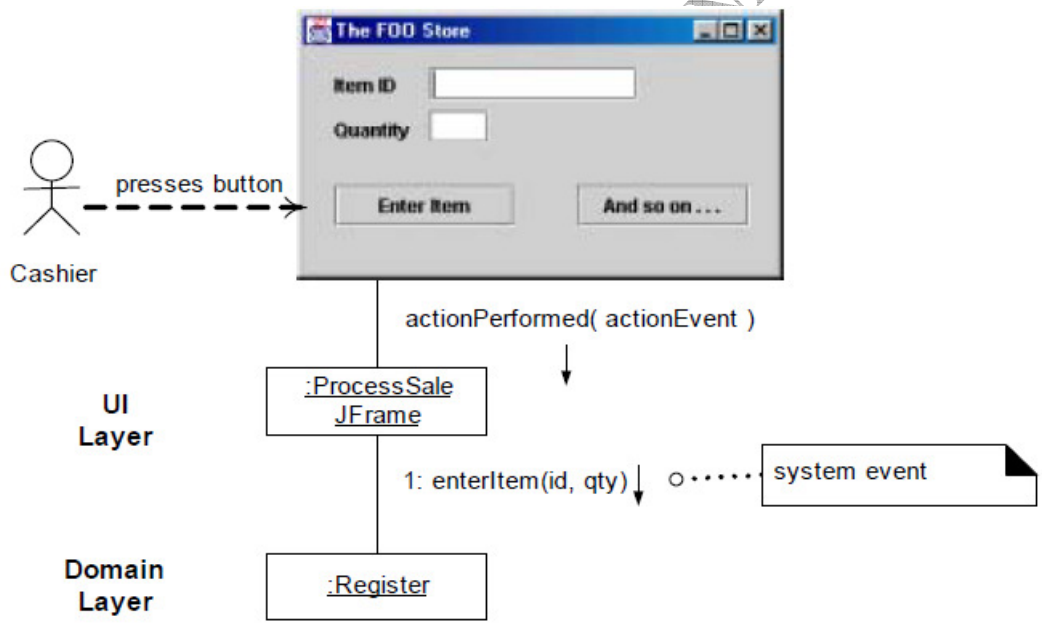


Figure 17.18 Connecting the UI and domain layers.

In the case of the *enterItem* message, the window needs to show the running total after each entry. There are several design solutions:
• Add a *getTotal* method to the *Register.* The UI sends the *getTotal* message to the *Register,* which forwards it to the *Sale.* This has the possible advantage of maintaining lower coupling from the UI to the domain layer—the UI only knows of the *Register* object. But it starts to expand the interface of the Register object, making it less cohesive.
• A UI asks for a reference to the current *Sale* object, and then when it needs the total (or any other information related to the sale), it directly sends mes sages to the *Sale.* This design increases the coupling from the UI to the domain layer. However, as was explored in the Low Coupling GRASP pat tern discussion, higher coupling in and of itself is not a problem; rather, it is especially coupling to unstable things that is a problem. Assume we decide the *Sale* is a stable object that will be an integral part of the design—which is very reasonable. Then,

coupling to the *Sale* is not a problem. As illustrated in Figure 17.19, this design follows the second approach.
Notice in these diagrams that the Java window *(ProcessSaleJFrame),* which is part of the UI layer, is not responsible for handling the logic of the application. It forwards requests for work (the system operations) to the domain layer, via the *Register.* This leads to the following design principle:

VISIBILITY:

In common usage, **visibility** is the ability of an object to "see" or have a reference to another object. There are four common ways that visibility can be achieved from object A to object *B:*
.**Attribute visibility**—B is an attribute of A.
• **Parameter visibility**—B is a parameter of a method of A.
• **Local visibility**—B is a (non-parameter) local object in a method of A.
• **Global visibility**—B is in some way globally visible.

Illustrating Visibility in the UML:
The UML includes notation to show the kind of visibility in a collaboration diagram (see Figure 18.6). These adornments are optional and not normally called for; they are useful when clarification is needed.

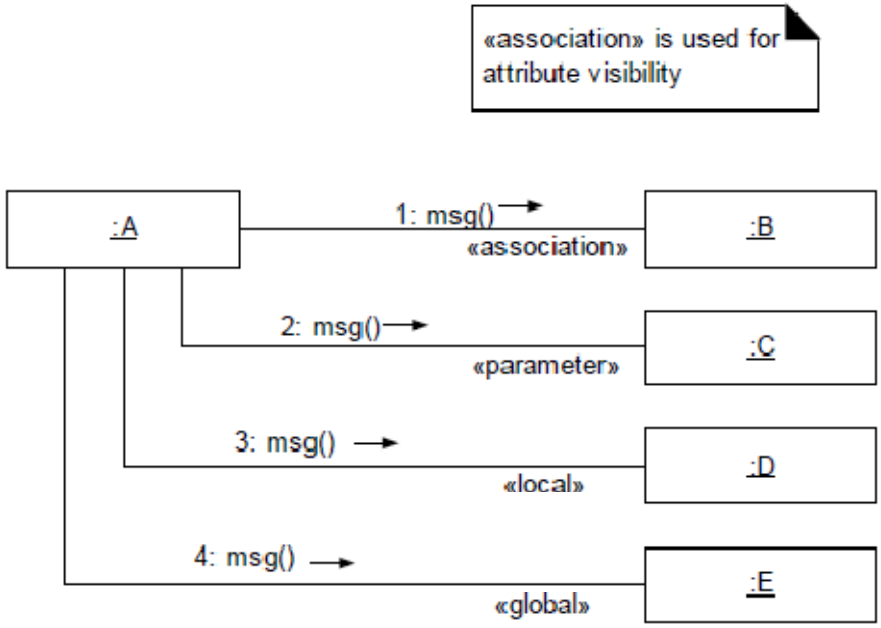


Figure 18.6 Implementation stereotypes for visibility.

DESIGN MODEL: CREATING DESIGN CLASS DIAGRAMS

Objectives

Create design class diagrams (DCDs).
Identify the classes, methods, and associations to show in a DCD.

Introduction:

With the completion of interaction diagrams for use-case realizations, it is possible to identify the specification for the software classes (and interfaces) that participate in the software solution with design details, such as methods.

- The UML has notation for showing design details in class diagrams

DCD and UP Terminology:

A design class diagram (DCD) illustrates the specifications for software classes and interfaces (for example, Java interfaces) in an application. Typical information includes:

• classes, associations and attributes
• interfaces, with their operations and constants
• methods
• attribute type information
• navigability
• dependencies

In contrast to conceptual classes in the Domain Model, design classes in the DCDs show definitions for software classes rather than real-world concepts. The UP defines the Design Model, which contains several diagram types, including interaction, package, and class diagrams. The class diagrams in the UP Design Model contain "design classes" in UP terms. Hence, it is common to speak of "design class diagrams and implies, "class diagrams in the Design Model."

Domain Model vs. Design Model Classes:

in the UP Domain Model, a *Sale* does not represent a software definition; rather, it is an abstraction of a real-world concept about which we are interested in making a statement. By contrast, DCDs express—for the software application—the definition of classes as software components. In these diagrams,
a *Sale* represents a software class (see Figure 19.2).

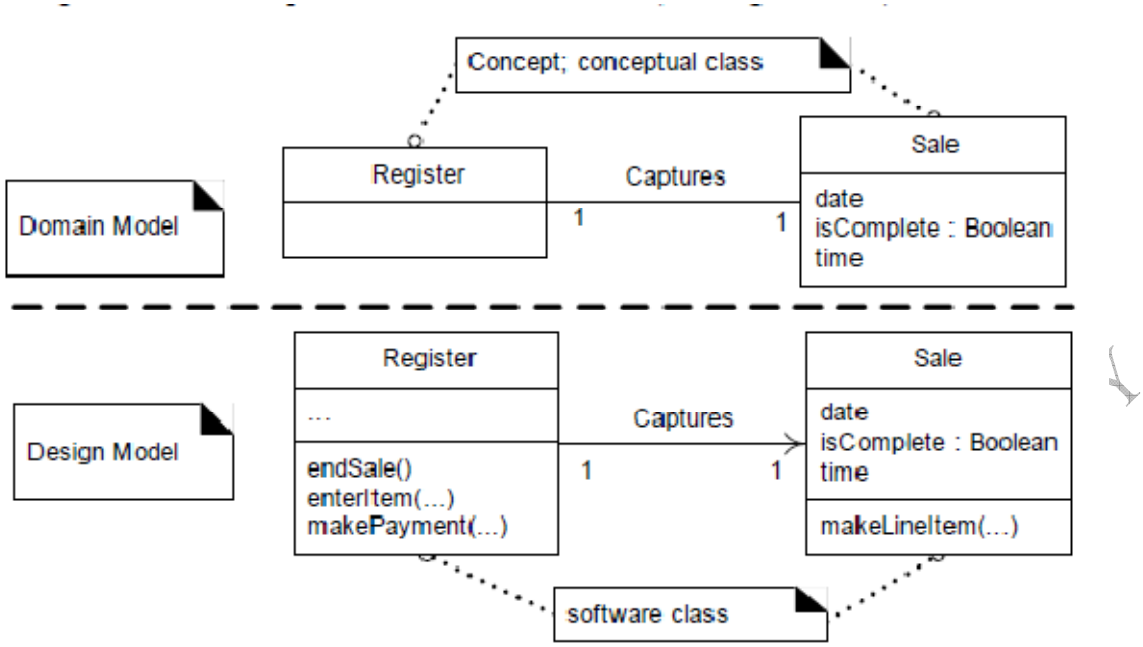


Figure 19.2 Domain model vs. Design Model classes.

Creating a NextGen POS DCD:

The first step in the creation of DCDs is to identify those classes that participate in the software solution.

For the POS application, these are:

Register	Sale
ProductCatalog	ProductSpecification
Store Payment	SalesLineItem

The next step is to draw a class diagram for these classes and include the attributes previously identified in the Domain Model that are also used in the design (see Figure 19.3). Note that some of the concepts in the Domain Model, such as *Cashier,* are not present in the design. There is no need—f or the current iteration—to represent them in software. However, in later iterations, as new requirements and use cases are tackled, they may enter into the design. For example, when security and log-in requirements are implemented, it is likely that a software class named *Cashier* will be relevant.

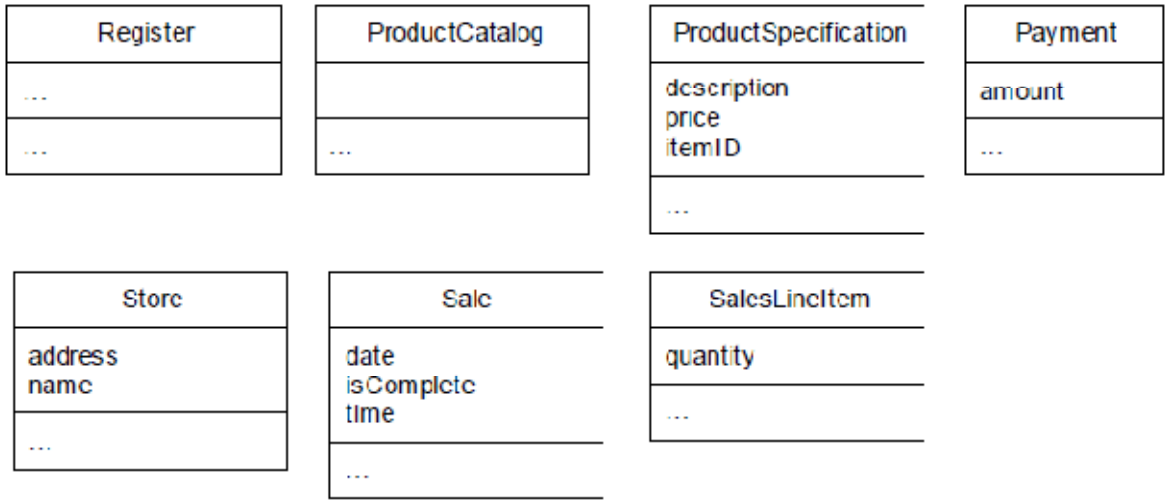


Figure 19.3 Software classes in the application.

Add Method Names:

The methods of each class can be identified by analyzing the interaction diagrams. For example, if the message *makeLineItem* is sent to an instance of class *Sale,* then class *Sale* must define a *makeLineItem* method (see Figure 19.4).

In general, the set of all messages sent to a class X across all interaction diagrams indicates the majority of methods that class X must define.

Inspection of all the interaction diagrams for the POS application yields the allocation of methods shown in Figure 19.5.

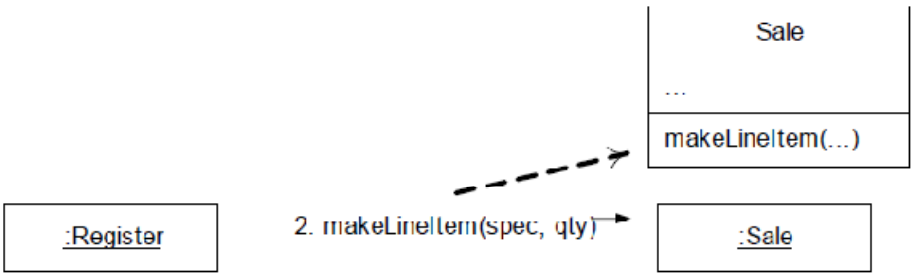


Figure 19.4 Method names from interaction diagrams.

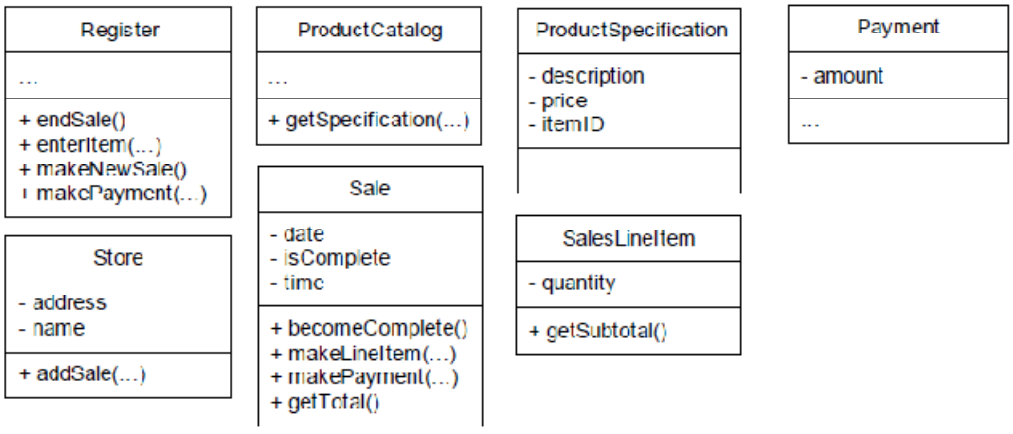


Figure 19.5 Methods in the application.

Adding More Type Information:
The types of the attributes, method parameters, and method return values may all optionally be shown. The question as to whether to show this information or not should be considered in the following context:

The DCD should be created by considering the audience.

- If it is being created in a CASE toolwith automatic code generation, full and exhaustive details are necessary.
- If it is being created for software developers to read, exhaustive low-level detail may adversely affect the noise-to-value ratio.

For example, is it necessary to show all the parameters and their type information? It depends on how obvious the information is to the intended audience. The design class diagram in Figure 19.7 shows more type information.

Adding Associations and Navigability:
Each end of an association is called a role, and in the DCDs the role may be decorated with a navigability arrow. Navigability is a property of the role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class. Navigability implies visibility—usually attribute visibility (see Figure 19.8).

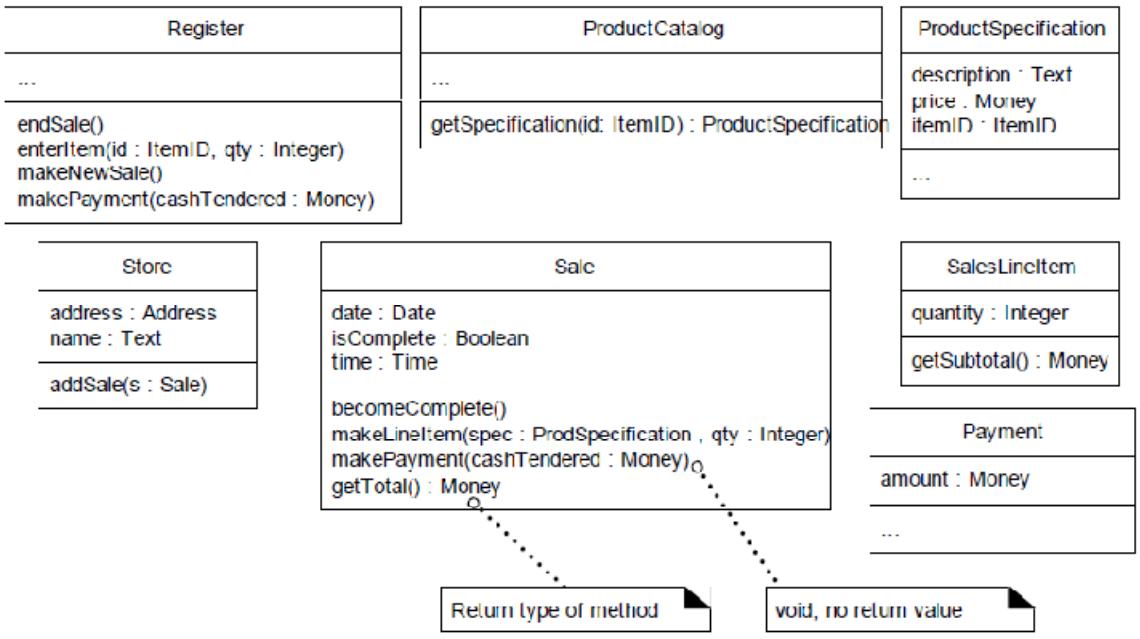


Figure 19.7 Adding type information.

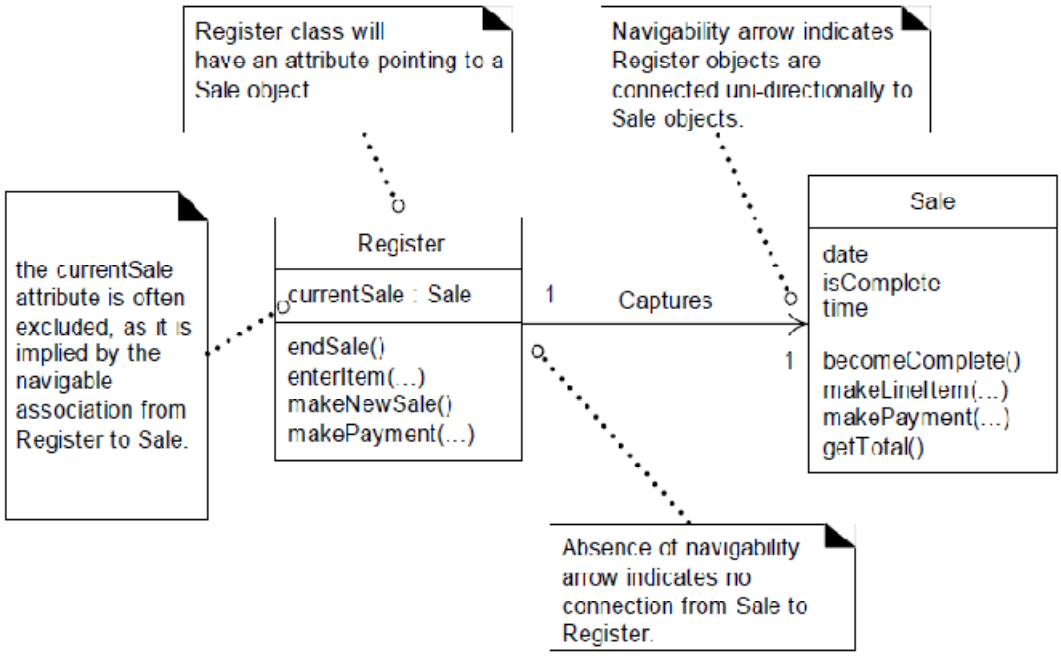


Figure 19.8 Showing navigability, or attribute visibility.

The usual interpretation of an association with a navigability arrow is attribute visibility from the source to target class. During implementation in an object-oriented programming language it is usually translated as the source class having an attribute that refers to an instance of the target class.

Adding Dependency Relationships:

The UML includes a general **dependency relationship,** which indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element. It is illustrated with a dashed arrow line. In class diagrams the dependency relationship is useful to depict non-attribute visibility between classes; in other words, parameter, global, or locally

declared visibility. By contrast, plain attribute visibility is shown with a regular association line and a navigability arrow. For example, the *Register* software object receives a return object of type *ProductSpecification* from the specification message it sent to a *ProductCatalog*. Thus *Register* has a short-term locally declared visibility to *ProductSpecifications.* And *Sale* receives a *ProductSpecification* as a parameter in the *makeLineItem* message; it has parameter visibility to one. These non-attribute visibilities may be illustrated with the dashed arrow line indicating a dependency relationship (see Figure 19.11). There is no significance in the curving of the dependency lines; it is graphically convenient.

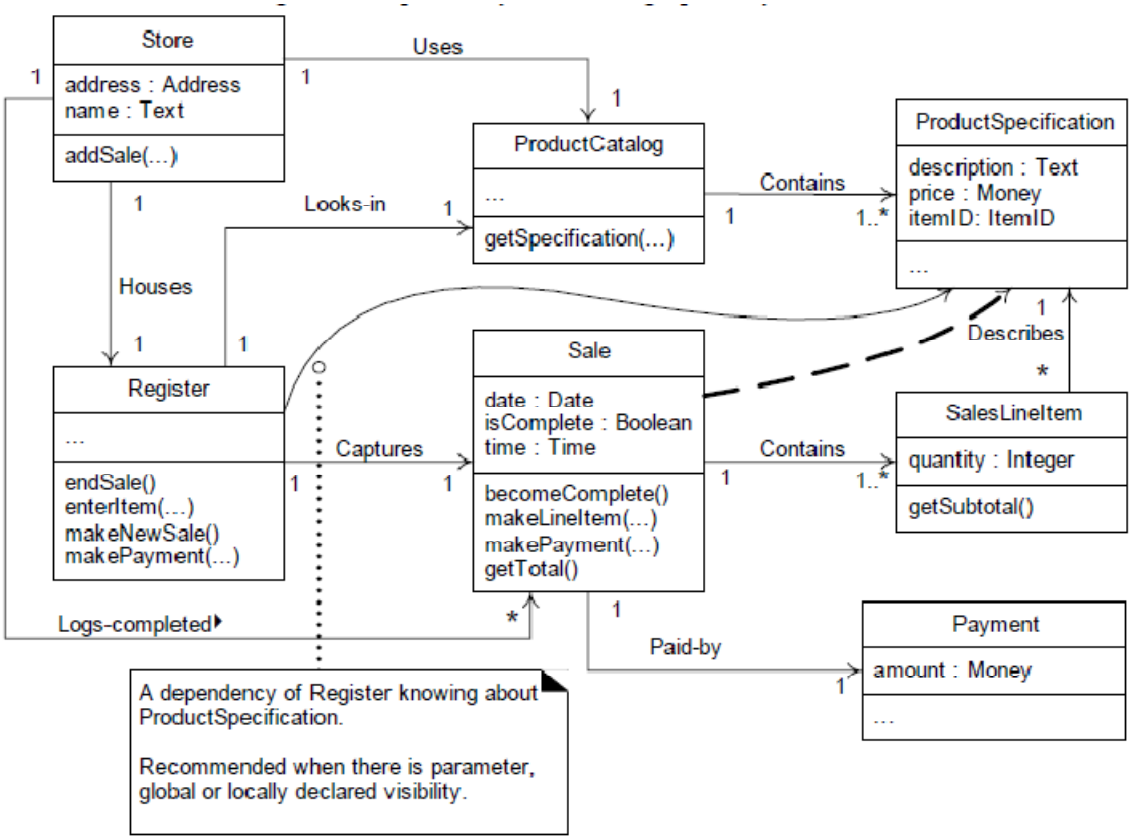


Figure 19.11 Dependency relationships indicating non-attribute visibility.

Visibility Defaults in the UML?

If no explicit visibility marker is shown for an attribute or method, what is the default? Answer: there isn't a default. If none is shown, it implies "not specified" in the UML. However, there is a common convention to assume that attributes are private and methods public, unless otherwise noted. The current iteration of the NextGen POS design class diagram (see Figure 19.13) does not have many interesting member details; all attributes are private and all methods public.

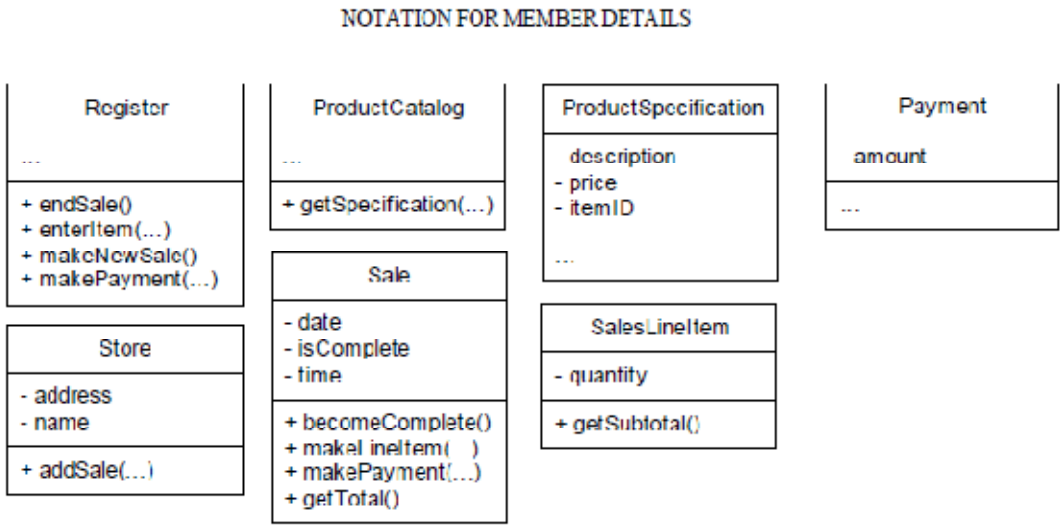
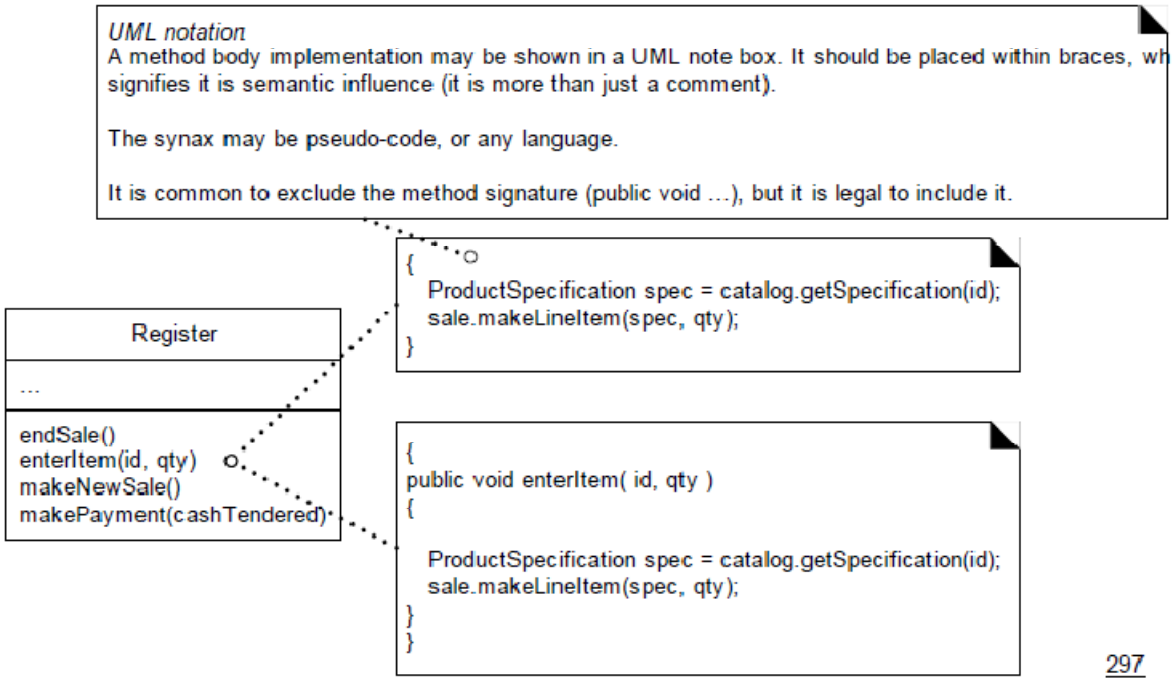


Figure 19.13 Member details in the POS class diagram.

Notation for Method Bodies in DCDs (and Interaction Diagrams):

A method body can be shown as illustrated in Figure 19.14 in both a DCD and an interaction diagram.



297

Figure 19.14 Method body notation.

DCDs Within the UP

DCDs are part of the use-case realizations and thus members of the UP Design

Model.

Discipline	Artifact Iteration->	Incep. Il	Elab. El. .En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specifications	s	r		
	Glossary	s	r		
Design	*Design Model*		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 19.1 Sample UP artifacts and timing. s - start; r - refine

Phases

Inception—The Design Model and DCDs will not usually be started until elaboration because it involves detailed design decisions, which are premature during inception.

Elaboration—During this phase, DCDs will accompany the use-case realization interaction diagrams; they may be created for the most architecturally significant classes of the design.

Note that CASE tools can reverse-engineer (generate) DCDs from source code. It is recommended to generate DCDs regularly from the source code, to visualize the static structure of the system.

Construction—DCDs will continue to be generate d from the source code as an aid in isualizing the static structure of the system.

Artifact influence emphasizing the DCDs is shown in Figure 19.15.

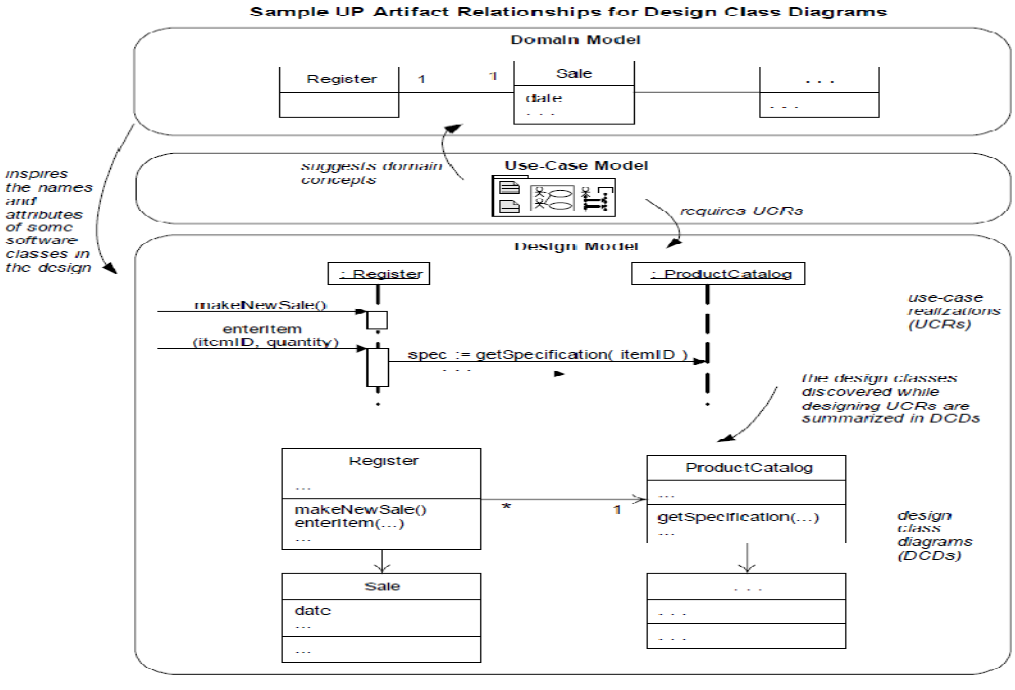


Figure 19.15 Sample UP artifact influence

IMPLEMENTATION MODEL: MAPPING DESIGNS TO CODE:

Objectives

Map design artifacts to code in an object-oriented language.

Introduction

With the completion of interaction diagrams and DCDs for the current iteration of the NextGen application, there is sufficient detail to generate code for the domain layer of objects.

The UML artifacts created during the design work—the interaction diagrams and DCDs—will be used as input to the code generation process.

The UP defines the Implementation Model. This contains the implementation artifacts such as the source code, database definitions, JSP/XML/HTML pages, and so forth.

Code Changes and the Iterative Process:

A strength of an iterative and incremental development process is that the results of a prior iteration can feed into the beginning of the next iteration (see Figure 20.1). Thus, subsequent analysis and design results are continually being refined and enhanced from prior implementation work. For example, when the code in iteration N deviates from the design of iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+1.

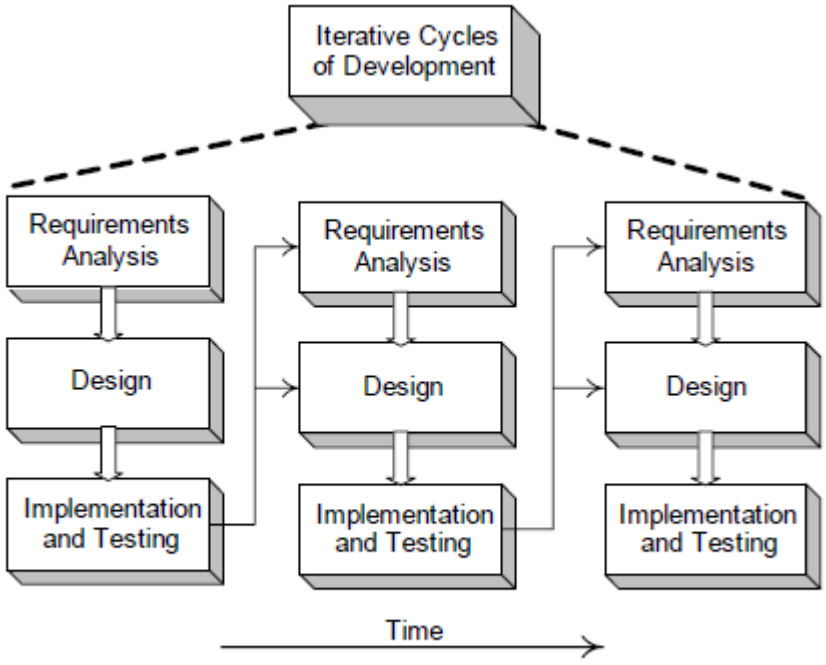


Figure 20.1 Implementation in an iteration influences later design.

An early activity within an iteration is to synchronize the design diagrams; the earlier diagrams of iteration N will not match the final code of iteration N, and they need to be synchronized before being extended with new design results. *Code Changes, CASE Tools, and Reverse-Engineering* It is desirable for the diagrams generated during design to be

semi-automati-cally updated to reflect changes in the subsequent coding work. Ideally this should be done with a CASE tool that can read source code and automatically generate, for example, package, class, and sequence diagrams.
This is an aspect of **reverse-engineering**—the activity of generating diagrams from source (or sometimes, executable) code.

Mapping Designs to Code:
Implementation in an object-oriented programming language requires writing source code for:
• class and interface definitions
• method definitions
The following sections discuss their generation in Java (as a typical case).

Creating Class Definitions from DCDs:
At the very least, DCDs depict the class or interface name, superclasses, method signatures, and simple attributes of a class. This is sufficient to create a basic class definition in an object-oriented programming language. Later discussion will explore the addition of interface and namespace (or package) information, among other details.
Defining a Class with Methods and Simple Attributes
From the DCD, a mapping to the basic attribute definitions (simple Java instance fields) and method signatures for the Java definition of *SalesLineItem* is straightforward, as shown in Figure 20.2.

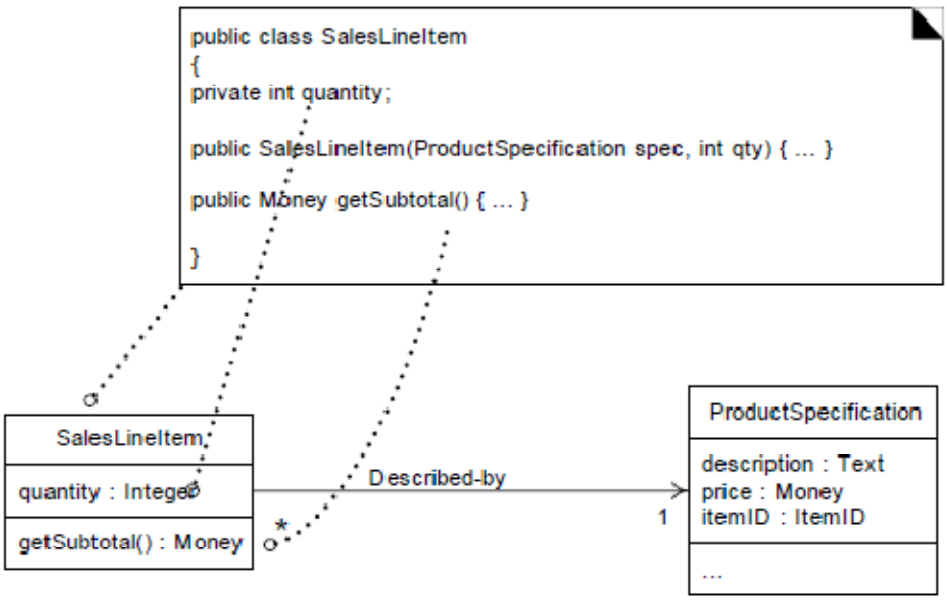


Figure 20.2 SalesLineItem in Java.

Adding Reference Attributes:
A reference attribute is an attribute that refers to another complex object, not to a primitive type such as a String, Number, and so on. The reference attributes of a class are suggested by the associations and navigability in a class diagram.
For example, a *SalesLineItem* has an association to a *ProductSpecification,* with navigability to it. It is common to interpret this as a reference attribute in class *SalesLineItem* that refers to a *ProductSpecification* instance (see Figure 20.3). In Java, this means that an instance field referring to a *ProductSpecification* instance is suggested.

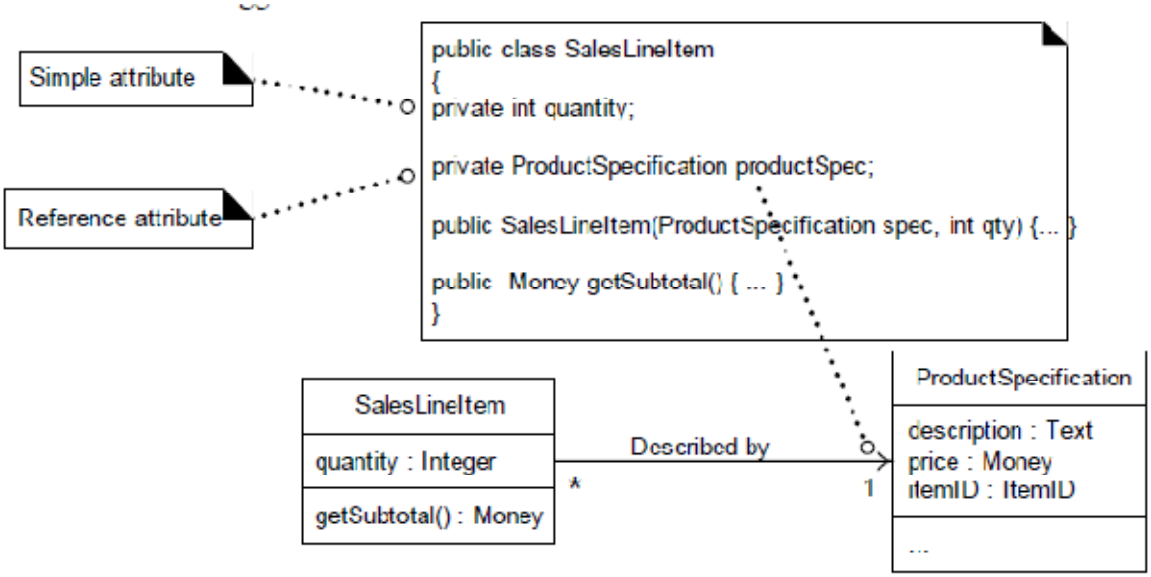


Figure 20.3 Adding reference attributes.

Note that reference attributes of a class are often implied, rather than explicit, in a DCD. For example, although we have added an instance field to the Java definition of *SalesLineItem* to point to a *ProductSpecification,* it is not explicitly declared as an attribute in the attribute section of the class box. There is a *suggested* attribute visibility—indicated by the association and navigability—which is explicitly defined as an attribute during the code generation phase.

Reference Attributes and Role Names:

The next iteration will explore the concept of role names in static structure diagrams. Each end of an association is called a role. Briefly, a **role name** is a name that identifies the role and often provides some semantic context as to the nature of the role. If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation, as shown in Figure 20.4.

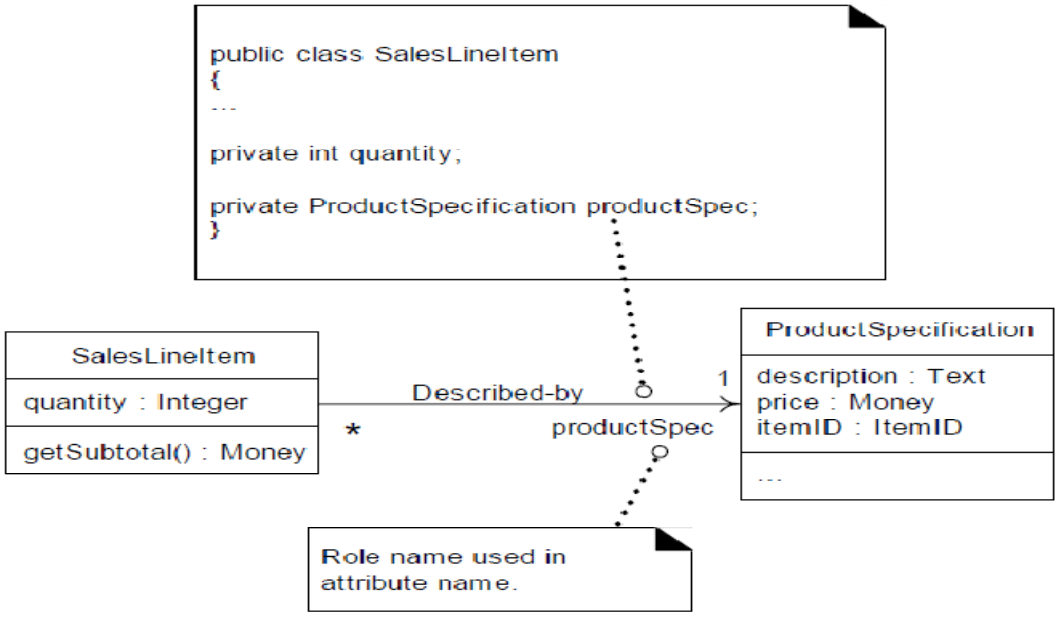


Figure 20.4 Role names may be used to generate instance variable names.

Mapping Attributes:
The *Sale* class illustrates that in some cases one must consider the mapping of attributes from the design to the code in different languages. Figure 20.5 illustrates the problem and its resolution.

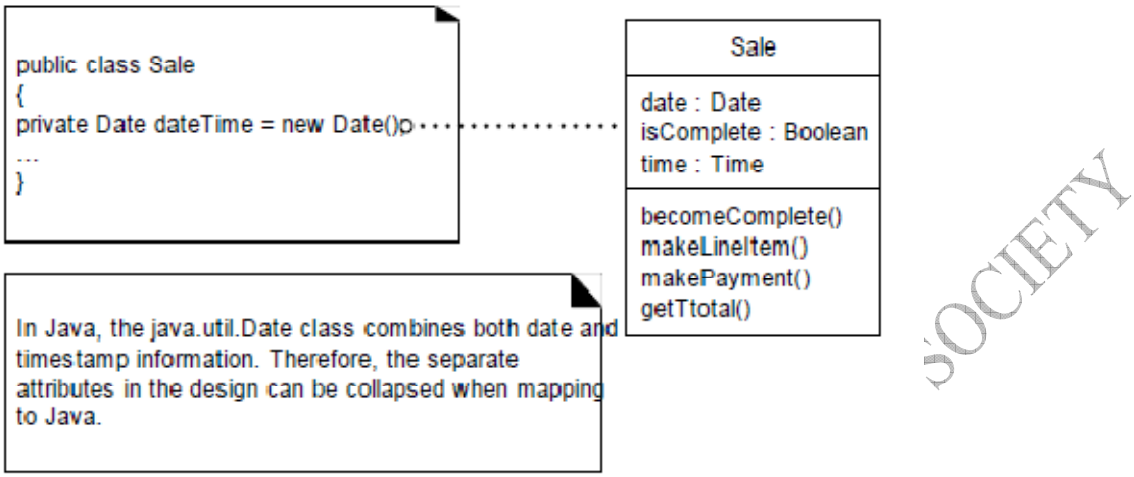


Figure 20.5 Mapping date and time to Java.

Creating Methods from Interaction Diagrams:
An interaction diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in the method definition. The *enterItem* interaction diagram in Figure 20.6 illustrates the Java definition of the *enterItem* method. In this example, the *Register* class will be used. A Java definition is shown in Figure 20.7.

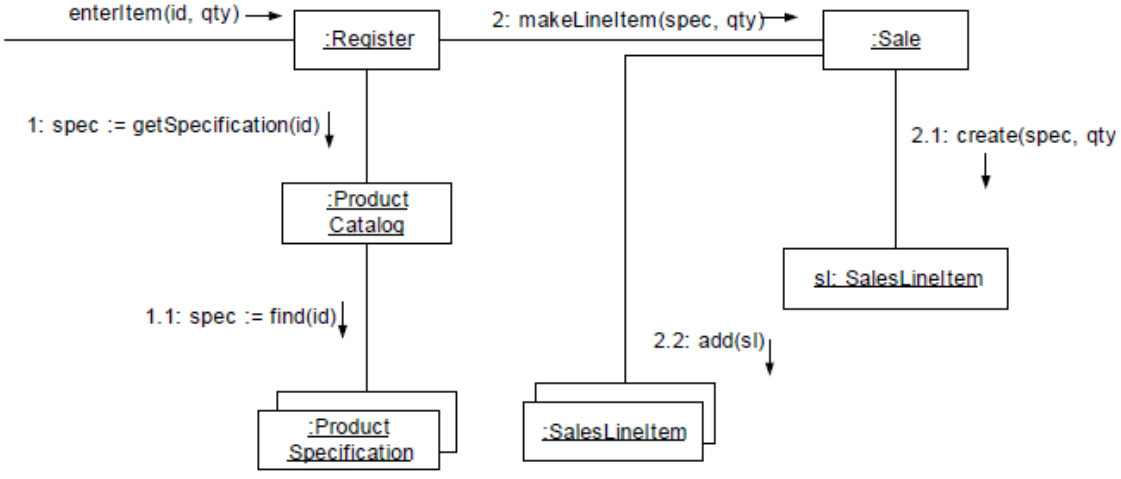


Figure 20.6 The enterItem interaction diagram.

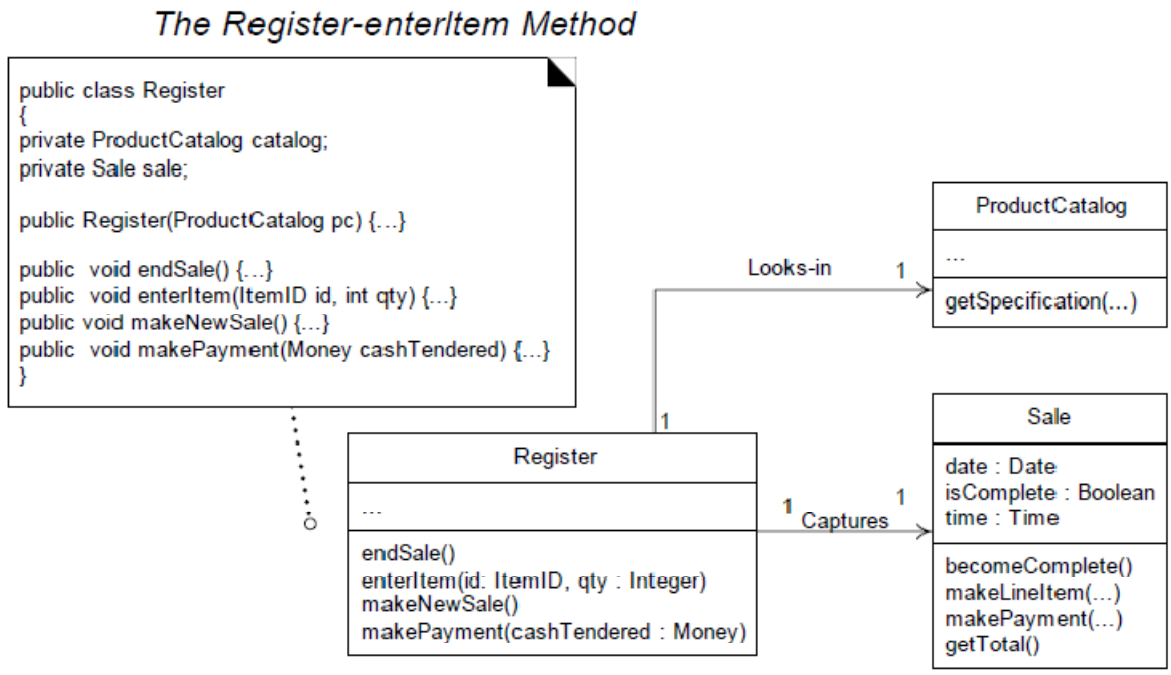


Figure 20.7 The Register class.

The *enterItem* message is sent to a *Register* instance; therefore, the *enterItem* method is defined in class *Register.* public void enterItem (ItemID itemID, int qty)
Message 1: *A getSpecification* message is sent to the *ProductCatalog* to retrieve a *ProductSpecification.*
ProductSpecif ication spec = catalog. getSpecif ication(itemID);
Message 2: The *makeLineItem* message is sent to the *Sale.* sale .makeLineItemf spec, qty);
In summary, each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method. The complete *enterItem* method and its relationship to the interaction diagram is shown in Figure 20.8.

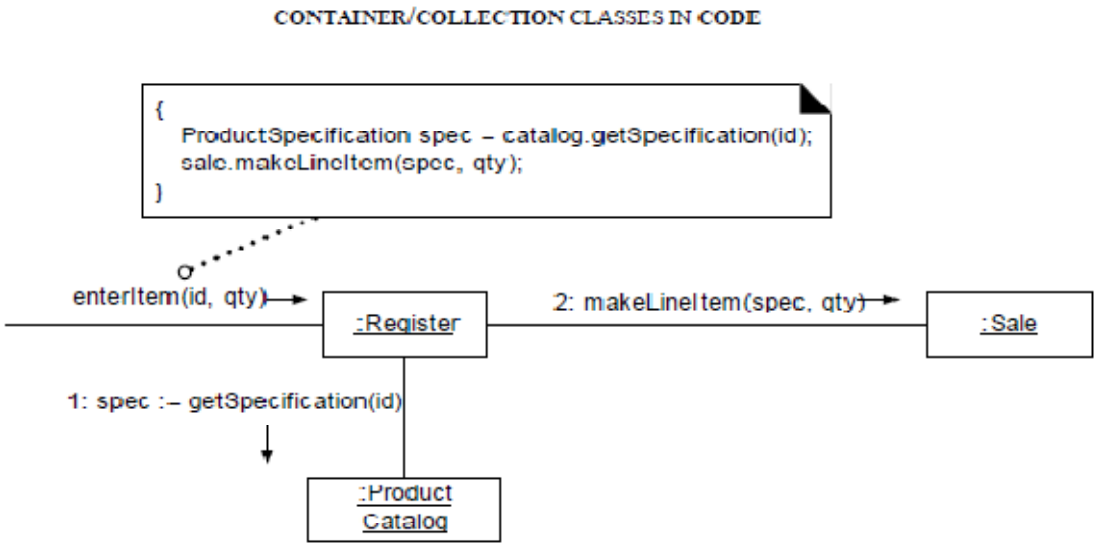


Figure 20.8 The enterItem method.

Container/Collection Classes in Code:

It is often necessary for an object to maintain visibility to a group of other objects; the need for this is usually evident from the multiplicity value in a class diagram—it may be greater than one. For example, a *Sale* must maintain visibility to a group of *SalesLineItem* instances, as shown in Figure 20.9.

In OO programming languages, these relationships are often implemented with the introduction of a intermediate container or collection. The one-side class defines a reference attribute pointing to a container/collection instance, which contains instances of the many-side class.

For example, the Java libraries contain collection classes such as *ArrayList* and *HashMap*, which implement the *List* and *Map* interfaces, respectively. Using *ArrayList,* the *Sale* class can define an attribute that maintains an ordered list *of SalesLineItem* instances.

The choice of collection class is of course influenced by the requirements; key-based lookup requires the use of a *Map,* a growing ordered list requires a *List,* and so on.

Exceptions and Error Handling:

Exception handling has been ignored so far in the development of a solution. This was intentional to focus on the basic questions of responsibility assignment and object design. However, in application development, it is wise to consider exception handling during design work, and certainly during implementation.

Briefly, in the UML, exceptions are illustrated as asynchronous messages in interaction diagrams. This is examined in Chapter 33.

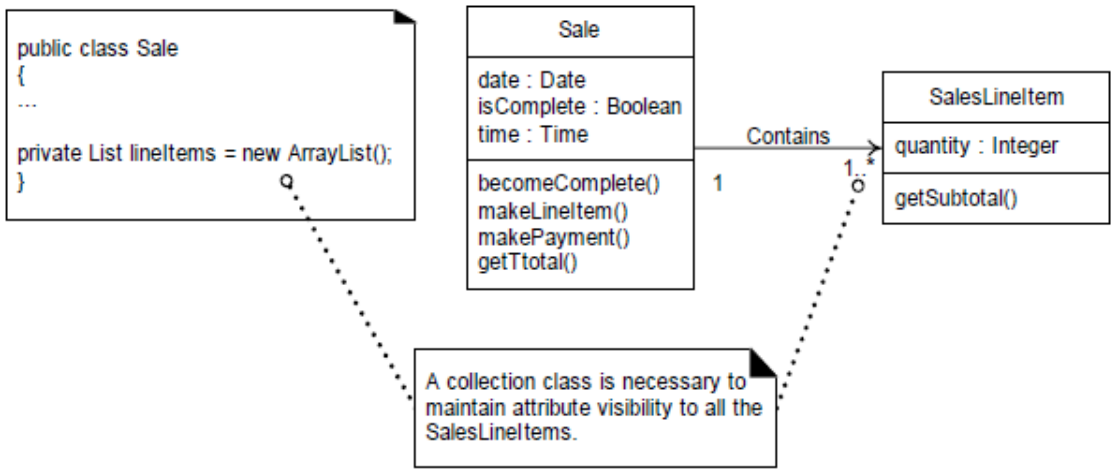


Figure 20.9 Adding a collection.

Test-First Programming

An excellent practice promoted by the Extreme Programming (XP) method and applicable to the UP is **test-first programming.**

In this practice, unit testing code is written *before* the code to be tested, and the developer writes unit testing code for *all* production code. The basic rhythm is to write a little test code, then write a little production code, make it pass the test, then write some more test code, and so forth. Advantages include:

• **The unit tests actually get written**—Human (or at least programmer) nature is such that avoidance of writing unit tests is very common, if left as an afterthought.

• **Programmer satisfaction**—If a developer writes the production code, informally debugs it, and then as an afterthought adds unit tests, it does not feel very satisfying. However, if the

tests are written first, and then production code is created and refined to pass the tests, there is some feeling of accomplishment—of passing a test. The psychological aspects of develop ment can't be ignored—programming is a human endeavor.

• **Clarification of interface and behavior**—Often, the exact public interface and behavior of a class is not perfectly clear until programming it. By writing the unit test for it first, one clarifies the design of the class.

• **Provable verification**—Obviously, having hundreds or thousands of unit tests provides some meaningful verification of correctness.

• **The confidence to change things**—In test-first programming, there are hundreds or thousands of unit tests, and a unit test class for each produc tion class. When a developer needs to change existing code—written by themselves or others—there is a unit te st suite that can be run, providing immediate feedback if the change caused an error.

Design class diagrams for case study and skeleton code:

This section presents a sample domain object layer program solution in Java for this iteration. The code generation is largely derived from the design class diagrams and interaction diagrams defined in the design work, based on the principles of mapping designs to code as previously explored.

The main point of this listing is to show that there is a translation from design artifacts to a foundation of code. This code defines a simple case; it is not meant to illustrate a robust, fully developed Java program with synchronization, exception handling, and so on.

Class Payment

```
public class Payment {
private Money amount;
public Payment( Money cashTendered ){ amount = cashTendered; }
public Money getAmount() { return amount; } }
```

Class ProductCatalog

```
public class ProductCatalog {
private Map productSpecifications = new HashMap();

public ProductCatalog() {
// sample data
ItemID idl = new ItemID( 100 );
ItemID id2 = new ItemID( 200 );
Money price = new Money( 3 );
ProductSpecification ps;
ps = new ProductSpecification( idl, price, "product 1" );
productSpecifications.put( idl, ps );
ps = new ProductSpecification( id2, price, "product 2" );
ProductSpecifications.put( id2, ps ); }
public ProductSpecification getSpecification( ItemID id ) {
return (ProductSpecification)productSpecifications.get( id );
}
}
```

Class Register

```
public class Register {
```

```
private ProductCatalog catalog;
private Sale sale;
public Register( ProductCatalog catalog ) {
this.catalog = catalog; }
public void endSaleO {
sale.becomeComplete();
}
public void enterltem( ItemID id, int quantity ) {
ProductSpecification spec = catalog.getSpecification( id );
sale.makeLineItem( spec, quantity ); }
public void makeNewSale() {
sale = new Sale(); }
public void makePayment( Money cashTendered ) {
sale.makePayment( cashTendered ); }

Class ProductSpecification
public class ProductSpecification {
private ItemID id;
private Money price;
private String description;
public ProductSpecification
( ItemID id. Money price. String description ) {
this.id = id;
this.price = price;
this.description = description; }
public ItemID getltemlDO { return id;}
public Money getPrice() { return price; }
public String getDescription() { return description; }
}
Class Sale
public class Sale
{
private List lineltems = new ArrayListO;
private Date date = new Date();
private boolean isComplete = false;
private Payment payment;
public Money getBalanceO {
return payment.getAmount().minus( getTotal() ); }
public void becomeComplete() { isComplete = true; }
public boolean isComplete() { return isComplete; }
public void makeLineltem
( ProductSpecification spec, int quantity ) {
lineltems.add( new SalesLineltem( spec, quantity ) ); }
public Money getTotal()
{
Money total = new MoneyO;
Iterator i = lineltems.iterator( ) ;
while ( i.hasNextO )
{
SalesLineltem sli = (SalesLineltem) i.nextO;
```

```
total.add( sli.getSubtotal() );
}
return total; }

public void makePayment( Money cashTendered )
{
payment = new Payment( cashTendered ); } }
Class SalesLineItem
public class SalesLineItem {
private int quantity;
private ProductSpecification productSpec;
public SalesLineItem (ProductSpecification spec, int quantity )
{
this.productSpec = spec;
this.quantity = quantity; }
public Money getSubtotal() {
return productSpec.getPrice().times( quantity );
} }
Class Store
public class Store
{
private ProductCatalog catalog = new ProductCatalog();
private Register register = new Register( catalog );
public Register getRegister() { return register; } }
```

UNIT-IV

MORE UML DIAGRAMS

Unit IV Syllabus:

More Design Patterns: Fabrication, Indirection, Singleton, Factory, Facade, Publish-Subscribe

FABRICATION or PURE FABRICATION:

Problem: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

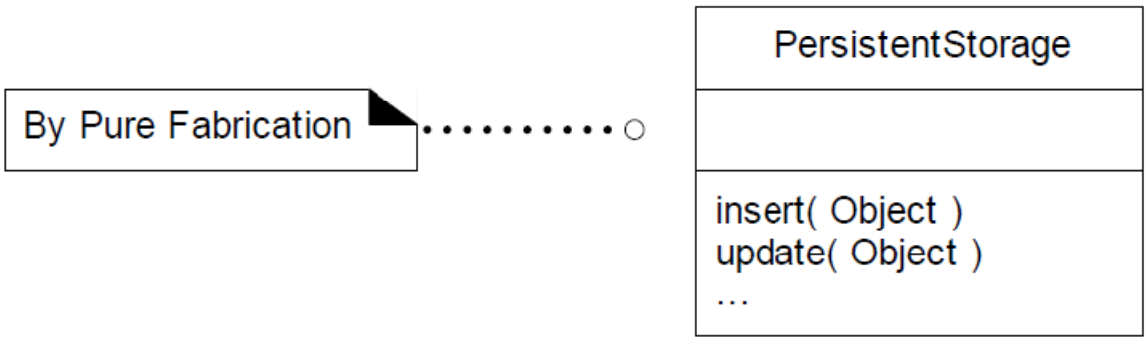
Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.

Such a class is *a fabrication* of the imagination. Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, *or pure*—hence a pure fabrication.

Assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.

Example: suppose that support is needed to save *Sale* instances in a relational database. By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, because the sale has the data that needs to be saved.

• The *Sale* class has to be coupled to the relational database interface (such as JDBC in Java technologies), so its coupling goes up. Placing these responsibilities in the *Sale* class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.
A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium, such as a relational database; call it the *PersistentStorage.* This class is a Pure Fabrication.



This Pure Fabrication solves the following design problems:
• The *Sale* remains well-designed, with high cohesion and low coupling.
• The *PersistentStorage* class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium.
• The *PersistentStorage* class is a very generic and reusable object. Creating a pure fabrication in this example is exactly the situation in which their use is called for—eliminating a bad design based on Expert, with poor cohesion and coupling, with a good design in which there is greater potential for reuse. Note that, as with all the GRASP patterns, the emphasis is on where responsibilities should be placed. In this example the responsibilities are shifted from the *Sale* class (motivated by Expert) to a Pure Fabrication.

The design of objects can be broadly divided into two groups:
1. Those chosen by **representational decomposition.**
2. Those chosen by **behavioral decomposition.**

For example, the creation of a software class such as *Sale* is by representational decomposition; the software class is related to or represents a thing in a domain. Representational decomposition is a common strategy in object design and supports the goal of reduced representational gap. But sometimes, we desire to assign responsibilities by grouping behaviors or by algorithm, without any concern for creating a class with a name or purpose that is related to a real-world domain concept.
These convenience classes are usually designed to group together some common behavior, and are thus inspired by behavioral rather than representational decomposition.

Many existing object-oriented design patterns are examples of Pure Fabrications: Adapter, Strategy, Command, and so on.

Related Patterns :
• Low Coupling.
• High Cohesion.
• A Pure Fabrication usually takes on responsibilities from the domain class that would be assigned those responsibilities based on the Expert pattern.
• All GoF design patterns [GHJV95], such as Adapter, Command, Strategy, and so on, are Pure Fabrications.
• Virtually all other design patterns are Pure Fabrications.

INDIRECTION:

Problem Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Solution Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an *indirection* between the other components.

Examples TaxCalculatorAdapter
These objects act as intermediaries to the external tax calculators. Via polymorphism, they provide a consistent interface to the inner objects and hide the variations in the external APIs. By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces (see Figure 22.3).

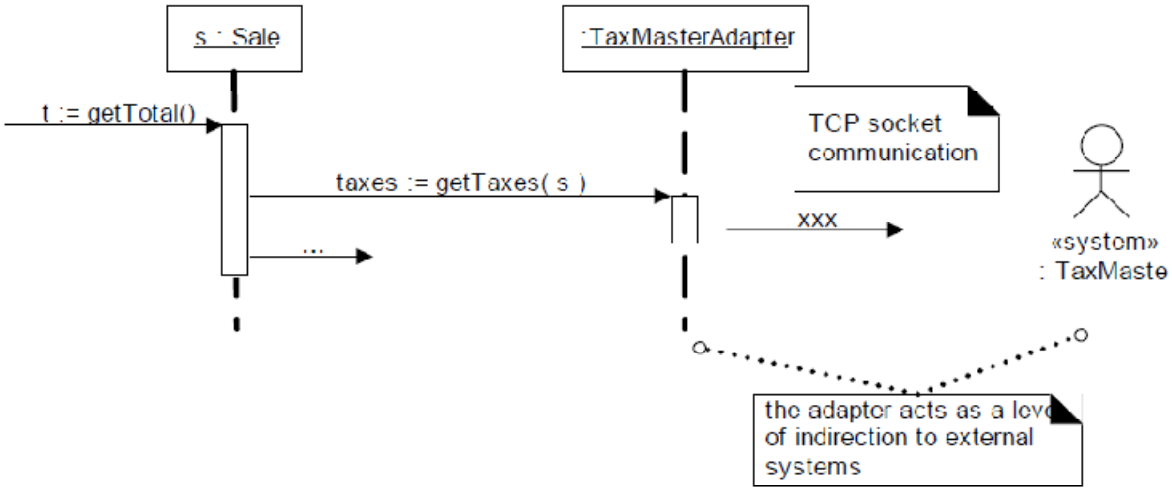


Figure 22.3 Indirection via the adapter.

PersistentStorage

The Pure Fabrication example of decoupling the *Sale* from the relational database services through the introduction of a *PersistentStorage* class is also an example of assigning responsibilities to support Indirection.

The *PersistentStorage* acts as a intermediary between the *Sale* and the database. "Most problems in computer science can be solved by another level of indirection"

many existing design patterns are specializations of Pure Fabrication, many are also specializations of Indirection. Adapter, Facade, and Observer are examples . In addition, many Pure Fabrications are generated because of Indirection. The motivation for Indirection is usually Low Coupling; an intermediary is added to decouple other components or services.

Related patterns:

- Lower coupling between components.
- Protected Variations
- Low Coupling
- Many GoF patterns, such as Adapter, Bridge, Facade, Observer, and Mediator.
- Many Indirection intermediaries are Pure Fabrications.

The Gang-of-Four Patterns

These additional patterns presented drawn from 23 *Design Patterns,* useful during object design. Since the book was written by four authors, these patterns have become known as the "Gang-of-Four"—or "GoF"—patterns.

A Shared Vocabulary

Factory (GoF):

The adapter raises a new problem in the design: the Adapter pattern solution for external services with varying interfaces, who creates the adapters?. And how to determine which class of adapter to create, such as *TaxMaster-Adapter* or *GoodAsGoldTaxProAdapter*

If some domain object creates them, the responsibilities of the domain object are going beyond pure application logic (such as sales total calculations) and into other concerns related to connectivity with external software components.

This point underscores another fundamental design principle (usually considered an architectural design principle): Design to maintain a **separation of concerns.** That is, modularize or separate distinct concerns into different areas, so that each has a cohesive purpose. For example, the domain layer of software objects emphasizes relatively pure application logic responsibilities, whereas a different group of objects is responsible for the concern of connectivity to external systems. Therefore, choosing a domain object (such as a *Register)* to create the adapters does not support the goal of a separation of concerns, and lowers its cohesion.

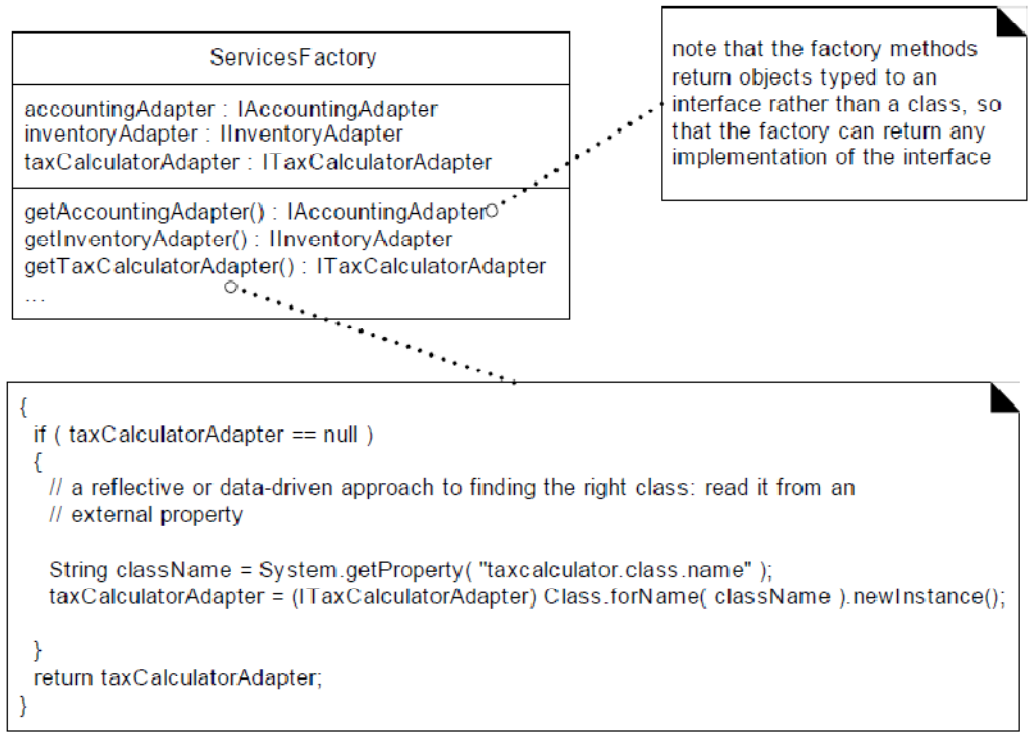


Figure 23.4 The Factory pattern.

UML notation—Observe the style in the UML diagram of Figure 23.4 that includes a note showing detailed pseudocode for the *getTaxCalculator-Adapter*. This style allows one to include dynamic algorithm details on a static class diagram such that it may lessen the need for interaction diagrams.

A common alternative in this case is to apply the **Factory** (or **Concrete Factory)** pattern, in which a Pure Fabrication "factory" object is defined to create objects.

Factory objects have several advantages:

- Separate the responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic.
- Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

Context/Problem:

Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution:
Create a Pure Fabrication object called a Factory that handles the creation. A Factory solution is illustrated in Figure 23.4.

Note that in the *ServicesFactory,* the logic to decide which class to create is resolved by reading in the class name from an external source (for example, via a system property if Java is used) and then dynamically loading the class. This is an example of a partial **data-driven design.** This design achieves Protected Variations with respect to changes in the implementation class of the adapter. Without changing the source code in this factory class, we can create instances of new adapter classes by changing the property value and ensuring the new class is visible in the Java class path for loading.

Related Patterns: Factories are often accessed with the Singleton pattern.

Singleton (GoF):

The *ServicesFactory* raises another new problem in the design: who creates the factory itself, and how is it accessed?
First, observe that only one instance of the factory is needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services. Thus, there is a visibility problem:

how to get visibility to this single *ServicesFactory* instance? One solution is pass the *ServicesFactory* instance around as a parameter to wherever a visibility need is discovered for it, or to initialize the objects that need visibility to it, with a permanent reference. This is possible but inconvenient; an alternative is the **Singleton** pattern.

Occasionally, it is desirable to support global visibility or a single access point to a single instance of a class rather than some other form of visibility. This is true for the *ServicesFactory* instance.

Context/Problem:
Exactly one instance of a class is allowed—it is a "singleton." Objects need a global and single point of access.
Solution:
Define a static method of the class that returns the singleton. For example, Figure 23.5 shows an implementation of the Singleton pattern.

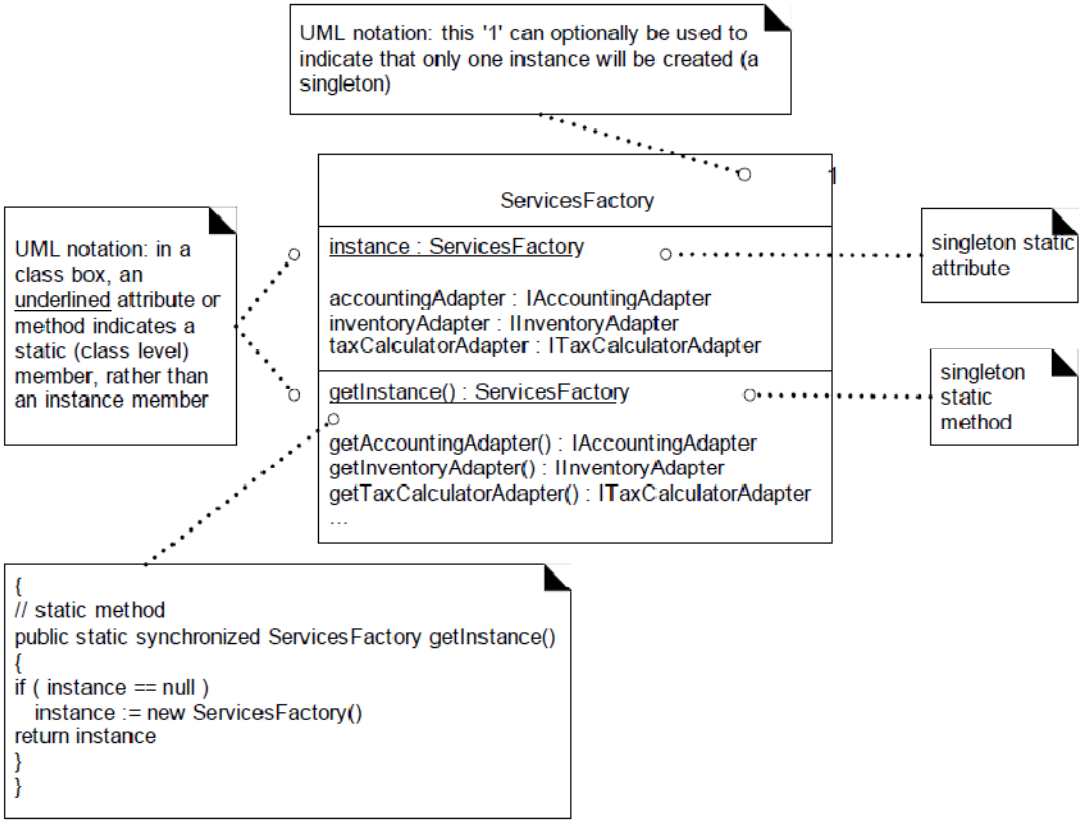


Figure 23.5 The Singleton pattern in the ServicesFactory class.

Thus, the key idea is that class X defines a static method *getlnstance* that itself provides a single instance of X. With this approach, a developer has global visibility to this single instance, via the static *getlnstance* method of the class, as in this example:

```
public class Register {
public void initialize()
{
... do some work ...
// accessing the singleton Factory via the getInstance call
accountingAdapter =
ServicesFactory.getInstance().getAccountingAdapter();
... do some work ... }
// other methods... }
```

Since visibility to public classes is global in scope (in most languages), at any point in the code, in any method of any class, one can write *SingletonClass.get-InstanceO* in order to obtain visibility to the singleton instance, and then send it a message, such as *SingletonClass.getInstance().doFoo().* It's hard to beat the feeling of being able to globally *doFoo.*

UML Shorthand for Singleton Access in Interaction Diagrams:
A UML notation that implies—but does not explicitly show—the *getlnstance* message in an interaction diagram is to add a «singleton» stereotype to the instance, as in Figure 23.6. This approach avoids having to explicitly show the (uninteresting) *getlnstance* message to the class before sending a message to the singleton instance.

Implementation and Design Issues:
A Singleton *getInstance* method is often frequently called. In multi-threaded applications, the creation step of the **lazy initialization** logic is a critical section requiring thread concurrency control. Thus, assuming the instance is lazy initialized, it is common to wrap the method with concurrency control. In Java,

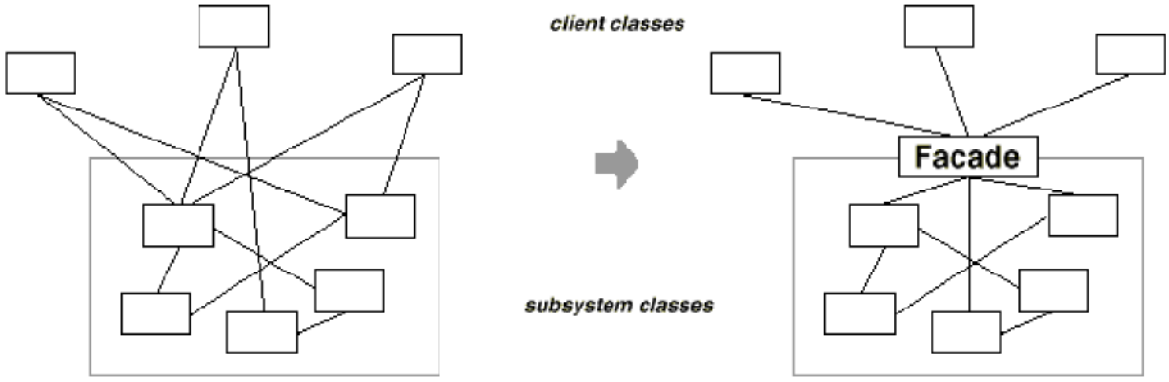
for example:

```
public static synchronized ServicesFactory getInstance()
{
if ( instance == null ) {
// critical section if multithreaded application
instance = new ServicesFactory(); }
return instance; }
```

FACADE (GOF):

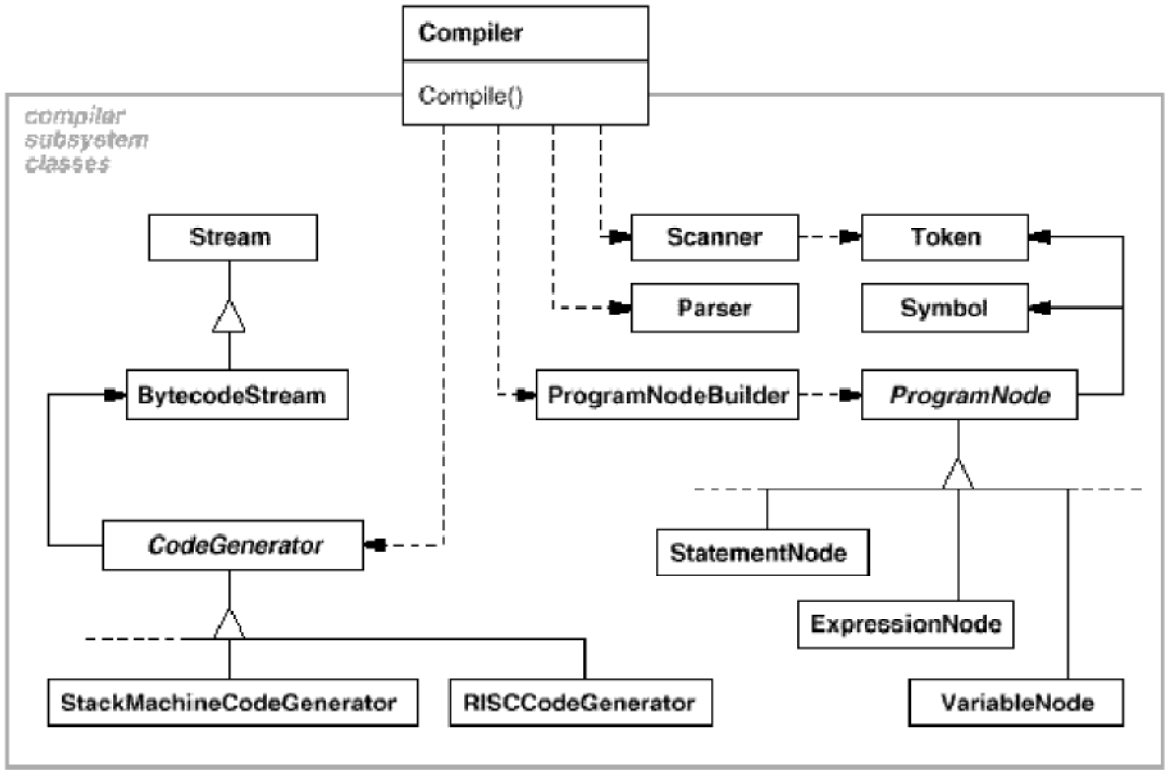
Intent:
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but low-level interfaces in the compiler subsystem only complicate their task. To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality. The Compiler class acts as a facade:

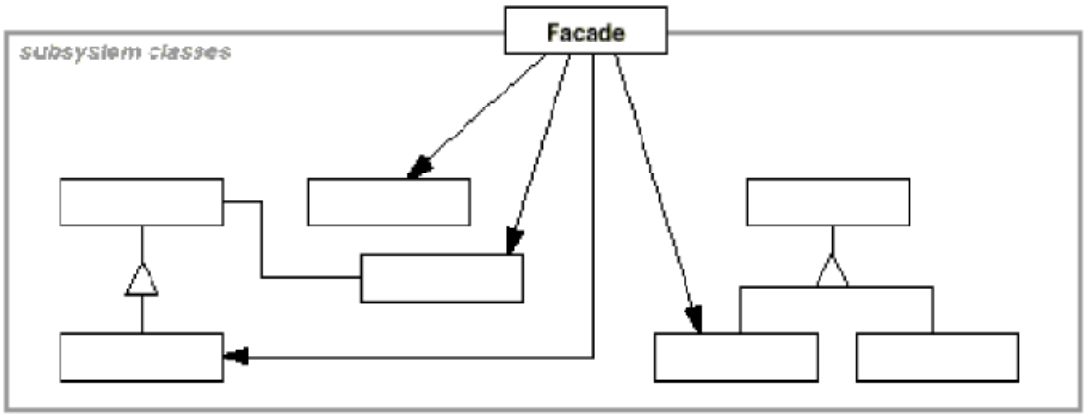
It offers clients a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely. The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it.



Applicability:

Use the Facade pattern when · you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
· there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability. · you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants:

· Facade (Compiler)
o knows which subsystem classes are responsible for a request.
o delegates client requests to appropriate subsystem objects.
· subsystem classes (Scanner, Parser, ProgramNode, etc.)
o implement subsystem functionality.
o handle work assigned by the Facade object.
o have no knowledge of the facade; that is, they keep no references to it.

Collaborations:

Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work. of its own to translate its interface to subsystem interfaces. · Clients that use the facade don't have to access its subsystem objects directly.

Consequences:

The Facade pattern offers the following benefits:
1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use. 2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.

Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others. 3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality. Implementation Consider the following issues when implementing a facade:

1. *Reducing client-subsystem coupling.* The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

2. *Public versus private subsystem classes.* A subsystem is analogous to a class in that both have interfaces, and both encapsulate something a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem. The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.
Making subsystem classes private would be useful, but few object-oriented languages support it. Both C++ and Smalltalk traditionally have had a globalname space for classes. Recently, however, the C++ standardization committee added name spaces to the language [Str94], which will let you expose just the public subsystem classes.

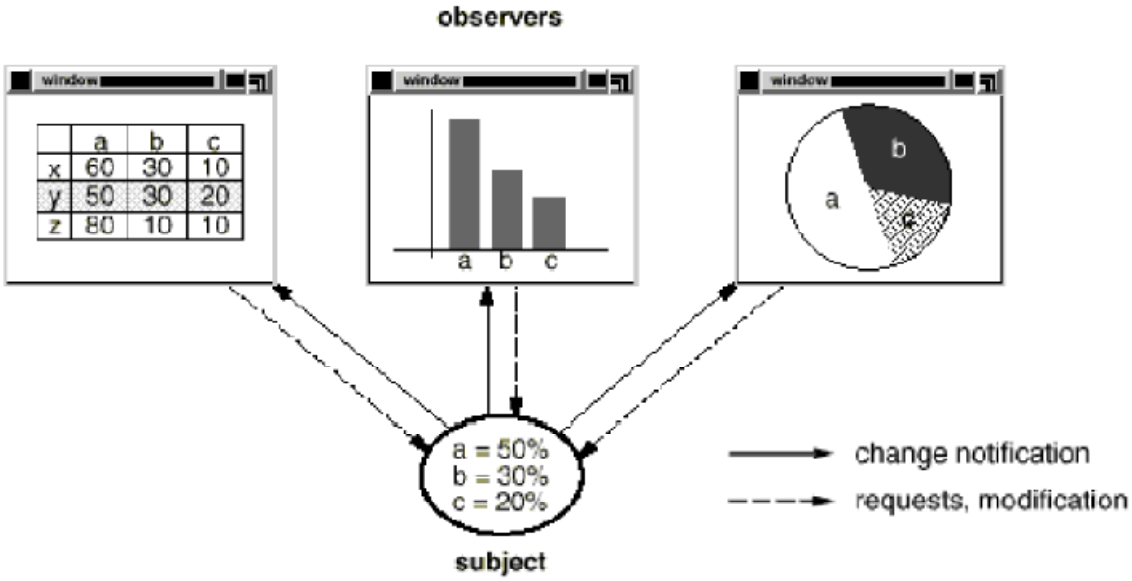
Related Patterns:

Abstract Factory
Mediator
Singletons

OBSERVER / PUBLISH-SUBSCRIBE:

Intent:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Also Known As Dependents, Publish-Subscribe.

Motivation:
A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



This behavior implies that the spreadsheet and bar chart are dependenton the data object and therefore should be notified of any change inits state. And there's no reason to limit the number of dependentobjects to two; there may be any number of different user interfacesto the same data.

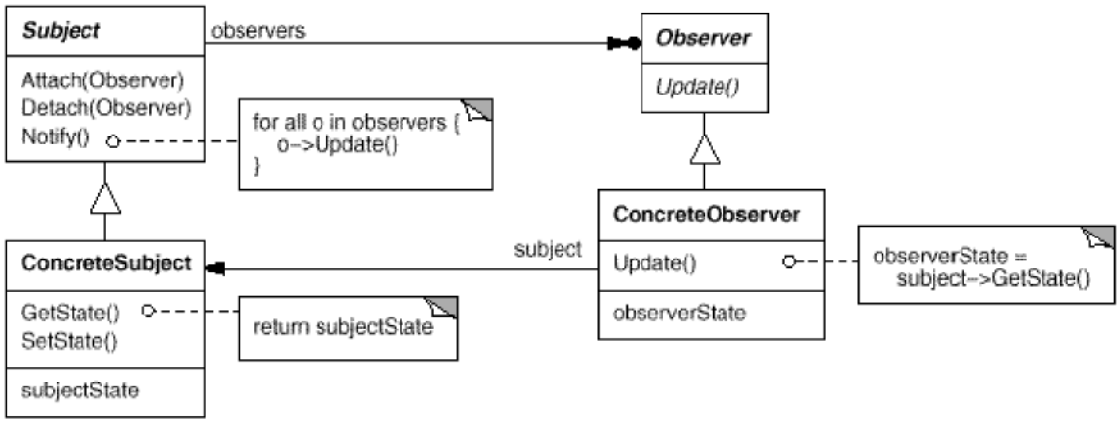
The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependen tobservers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Applicability:

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure



Participants:

- Subject

o knows its observers. Any number of Observer objects may observe a subject.
o provides an interface for attaching and detaching Observer objects.

- Observer

o defines an updating interface for objects that should be notified of changes in a subject.

- Concrete Subject

o stores state of interest to ConcreteObserver objects.
o sends a notification to its observers when its state changes.

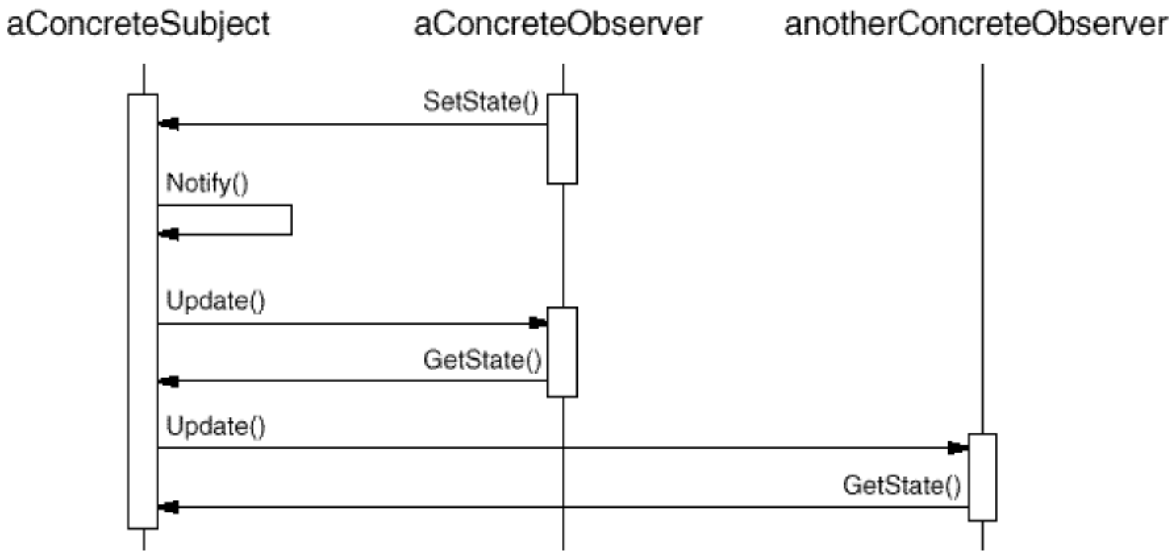
- Concrete Observer

o maintains a reference to a ConcreteSubject object.
o stores state that should stay consistent with the subject's.
o implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- Concrete Subject notifies its observers whenever a changeoccurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a Concrete Observer object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject. The following interaction diagram illustrates the collaborations

between a subject and two observers:



Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject. Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely. The Implementation section discusses some common variations.

Consequences:

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

Because Subject and Observer aren't tightly coupled, they can belong todifferent layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (andviolate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

2. *Support for broadcast communication* .Unlike an ordinary request, the notification that a subject sendsneedn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they canbe blind to the ultimate cost of changing the subject. A seeminglyinnocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject.

Without additional protocol to help observers discover what changed, they maybe forced to work hard to deduce the changes.

Implementation:

Several issues related to the implementation of the dependency mechanism are discussed here.

1. *Mapping subjects to their observers.* The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur storage overhead. On the other hand, this approach increases the cost of accessing the observers.

2. *Observing more than one subject.* It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source. It's necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.

*Who triggers the update?*The subject and its observers rely on the notification mechanism tostay consistent. But what object actually calls Notify to trigger theupdate? Here are two options: a. Have state-setting operations on Subject call Notify after theychange the subject's state. The advantage of this approach is thatclients don't have to remember to call Notify on the subject.

The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient. b. Make clients responsible for calling Notify at the right time. The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, there by avoiding needless intermediate updates. The disadvantage is that clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

4. *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make thesubject notify its observers as it is deleted so that they can resettheir reference to it. In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

5. *Making sure Subject state is self-consistent before notification.* It's important to make sure Subject state is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state.

This self-consistency rule is easy to violate unintentionally when Subject subclass operations call inherited operations.

For example, the notification in the following code sequence is trigged when the subject is in an inconsistent state:

```
void MySubject::Operation (int newValue) {
BaseClassSubject::Operation(newValue);
// trigger notification
_myInstVar += newValue;
// update subclass state (too late!)
}
```

You can avoid this pitfall by sending notifications from template methods (Template Method (360)) in abstract Subject classes. Define a primitive operation for subclasses to override, and makeNotify the last operation in the template method, which will ensure thatthe object is self-consistent when subclasses override Subject operations.

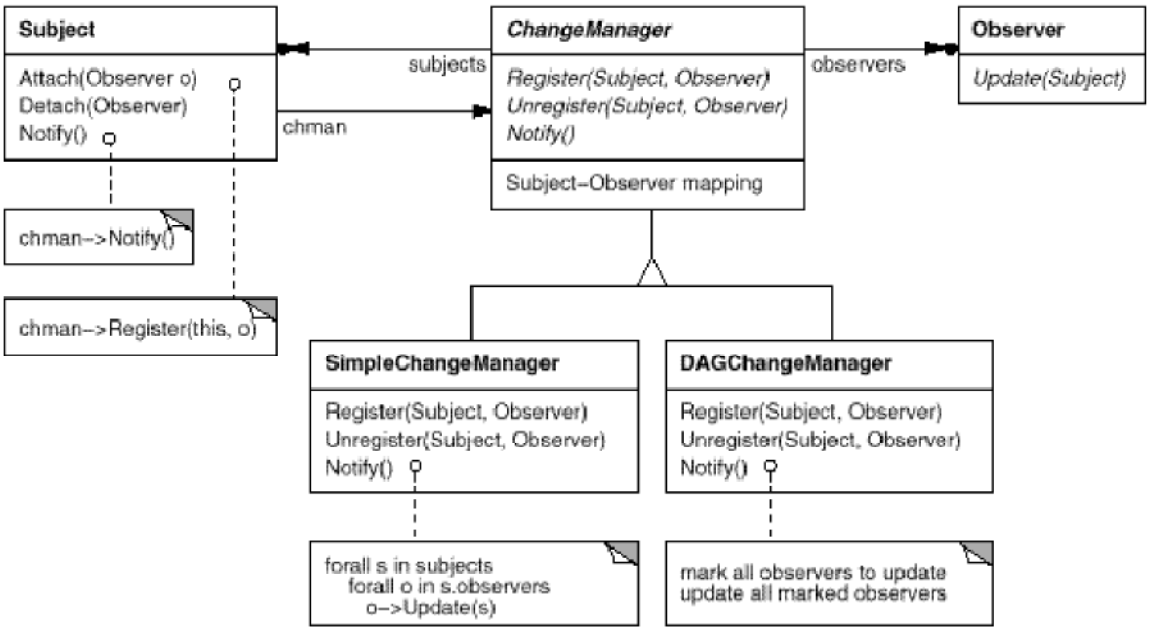
```
void Text::Cut (TextRange r)
```

{ReplaceRange(r); // redefined in subclasses
Notify();
}
By the way, it's always a good idea to document which Subject operations trigger notifications.
6. *Avoiding observer-specific update protocols: the push and pull models.* Implementations of the Observer pattern often havethe subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amountof information may vary widely.
At one extreme, which we call the push model, the subjects ends observers detailed information about the change, whether they want it or not. At the other extreme is the pull model; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter. The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.
7. *Specifying modifications of interest explicitly.* You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this uses the notion of aspects for Subject objects. To register interest in particular events, observers are attached to their subjects using

void Subject::Attach(Observer*, Aspect& interest);

where interest specifies the eventof interest. At notification time, the subject supplies the changed aspect to its observers as a parameter to the Update operation.
For example:
void Observer::Update(Subject*, Aspect& interest);
8. *Encapsulating complex update semantics.* When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a Change Manager. Itspurpose is to minimize the work required to make observers reflect a change in their subject.
For example, if an operation involves changes to several interdependent subjects, you might have to ensure that their observers are notified only after *all* the subjects have been modified to avoid notifying observers more than once.
Change Manager has three responsibilities:
1. It maps a subject to its observers and provides an interface
To maintain this mapping. This eliminates the need for subjects to Maintain references to their observers and vice versa.
2. It defines a particular update strategy.
3. It updates all dependent observers at the request of a subject.
The following diagram depicts a simple ChangeManager-based implementation ofthe Observer pattern. There are two specialized ChangeManagers.SimpleChangeManager is naive in that it always updates all observers ofeach subject. In contrast, DAGChangeManager handles directed-acyclicgraphs of dependencies between subjects and their observers. ADAGChangeManager is preferable to a SimpleChangeManager when an observerobserves more than one subject. In that case, a change in two or moresubjects might cause redundant

updates. The DAGChangeManagerensuresthe observer receives just one update. SimpleChangeManager is fine when multiple updates aren't an issue.



ChangeManager is an instance of the Mediator pattern. In general there is only one ChangeManager, and it is knownglobally. The Singleton pattern would beuseful here.

9. *Combining the Subject and Observer classes.* Class libraries written in languages that lack multiple inheritance(like Smalltalk) generally don't define separate Subject and Observer classes but combine their interfaces in one class. That lets youdefine an object that acts as both a subject and an observer without multiple inheritance. In Smalltalk, for example, the Subject and Observer interfaces are defined in the root class Object, making themavailable to all classes.

Known Uses:

The first and perhaps best-known example of the Observer pattern appears in Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment . MVC's Model class plays the role of Subject, while View is the base class for observers. Smalltalk,ET++, and the THINK class library provide ageneral dependency mechanism by putting Subject and

Observer interface sin the parent class for all other classes in the system. Other user interface toolkits that employ this pattern are InterViews ,the AndrewToolkit , and Unidraw . InterViewsdefines Observer and Observable (for subjects) classes explicitly.Andrew calls them "view" and "data object," respectively. Uni draw splits graphical editor objects into View (for observers) and Subject parts.

Related Patterns:

Mediator
Singleton

UNIT-V

MORE UML DIAGRAMS

Unit V Syllabus:

More UML diagrams : State-Chart diagrams, Activity diagrams, Component Diagrams, Deployment diagrams, Object diagrams.

STATE-CHART DIAGRAMS:

Objectives:
Create statechart diagrams for classes and use cases..

Introduction:
The UML includes statechart diagram notation to illustrate the events and states of things—transactions, use cases, people, and so forth. The most important notational features are shown, but there are others not covered in this introduction.

The use of statechart diagrams is emphasized for showing system events in use cases, but they may additionally be applied to any class.

Events, States, and Transitions:
An **event** is a significant or noteworthy occurrence.
For example: A telephone receiver is taken off the hook.
A state is the condition of an object at a moment in time—the time between events.
For example: • A telephone is in the state of being "idle" after the receiver is placed on the hook and until it is taken off the hook.
A transition is a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state.
For example:• When the event "off hook" occurs, transition the telephone from the "idle" to "active" state.

Statechart Diagrams:
A UML statechart diagram, as shown in Figure 29.1, illustrates the interesting events and states of an object, and the behavior of an object in reaction to an event. Transitions are shown as arrows, labeled with their event. States are shown in rounded rectangles. It is common to include an initial pseudo-state, which automatically transitions to another state when the instance is created.

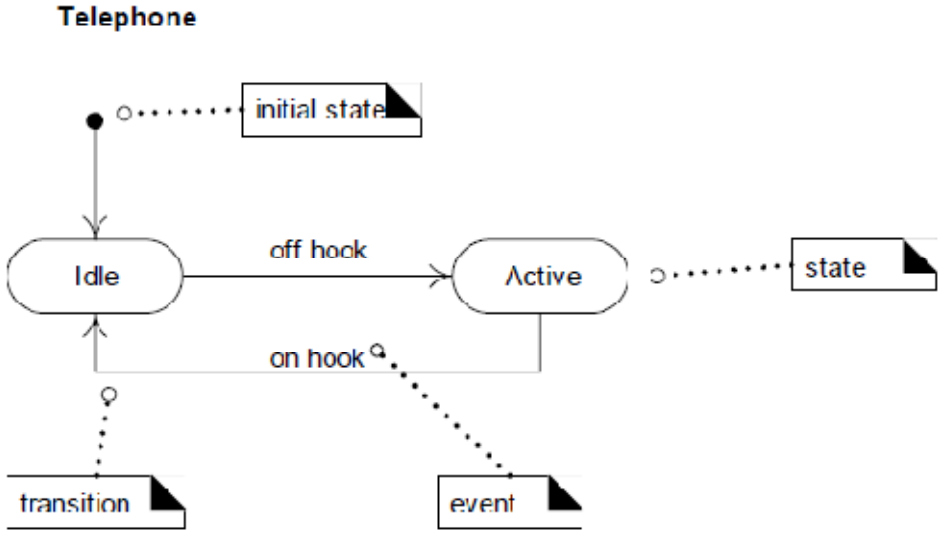


Figure 29.1 Statechart diagram for a telephone.

A statechart diagram shows the lifecycle of an object: what events it experiences, its transitions, and the states it is in between these events. It need not illustrate every possible event; if an event arises that is not represented in the diagram, the event is ignored as far as the statechart diagram is concerned.
Therefore, we can create a statechart diagram that describes the lifecycle of an object at arbitrarily simple or complex levels of detail, depending on our needs.

Subject of a Statechart Diagram:
A statechart diagram may be applied to a variety of UML elements, including:
• classes (conceptual or software)
• use cases
Since an entire "system" may be represented by a class, it too may have its own statechart diagram.

Statechart Diagrams in the UP?:
There is not one model in the UP called the "state model." Rather, any element in any model (Design Model, Domain Model, and so forth) may have a statechart to better understand or communicate its dynamic behavior in response to events. For example, a statechart associated with the *Sale* design class of the Design Model is itself part of the Design Model.

Use Case Statechart Diagrams:
A useful application of statechart diagrams is to describe the legal sequence of external system events that are recognized and handled by a system in the context of a use case.
For example:
→ During the *Process Sale* use case in the NextGen POS application,it is not legal to perform the *makeCreditPayment* operation until the *endSale* event has happened.
→ During the *Process Document* use case in a word processor, it is not legal to perform the File-Save operation until the File-New or File-Open event has happened.A statechart diagram that depicts the overall system events and their sequence within a use case is a kind of **use case statechart diagram.** The use case statechart diagram in Figure 29.2 shows a simplified version of the system events for the *Process Sale* use case in the POS application. It illustrates that it is not legal to generate a *makePayment* event if an *endSale* event has not previously caused the system to transition to the *WaitingForPayment* state.

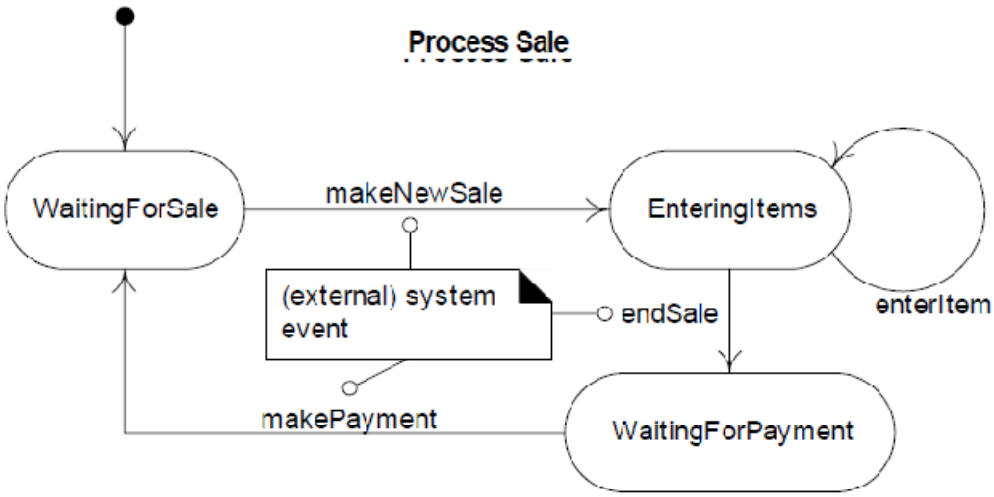


Figure 29.2 Use case statechart diagram for *Process Sale*.

Utility of Use Case Statechart Diagrams:

The number of system events and their legal order for the *Process Sale* use case are (so far) relatively trivial, thus the use of a statechart diagram to show legal sequence may not seem compelling. But for a complex use case with myriad system events—such as when using a word processor—a statechart diagram that illustrates the legal order of external events is helpful.

Here's how: During design and implementation work, it is necessary to create and implement a design that ensures no out-of-sequence events occur, otherwise an error condition is possible. For example, the system should not be allowed to receive a payment unless a sale is complete; code must be written to guarantee that.

Given a set of use case statechart diagrams, a designer can methodically develop a design that ensures correct system event order.

Possible design solutions include: hard-coded conditional tests for out-of-order events use of the *State* pattern (discussed in a subsequent chapter) disabling widgets in active windows to disallow illegal events (a desirable approach) a state machine interpreter that runs a state table representing *a* use case statechart diagram In a domain with many system events, the conciseness and thoroughness of use case statechart diagrams help a designer ensure that nothing is missed.

Use Case Statechart Diagrams for the POS Application

Process Sale:

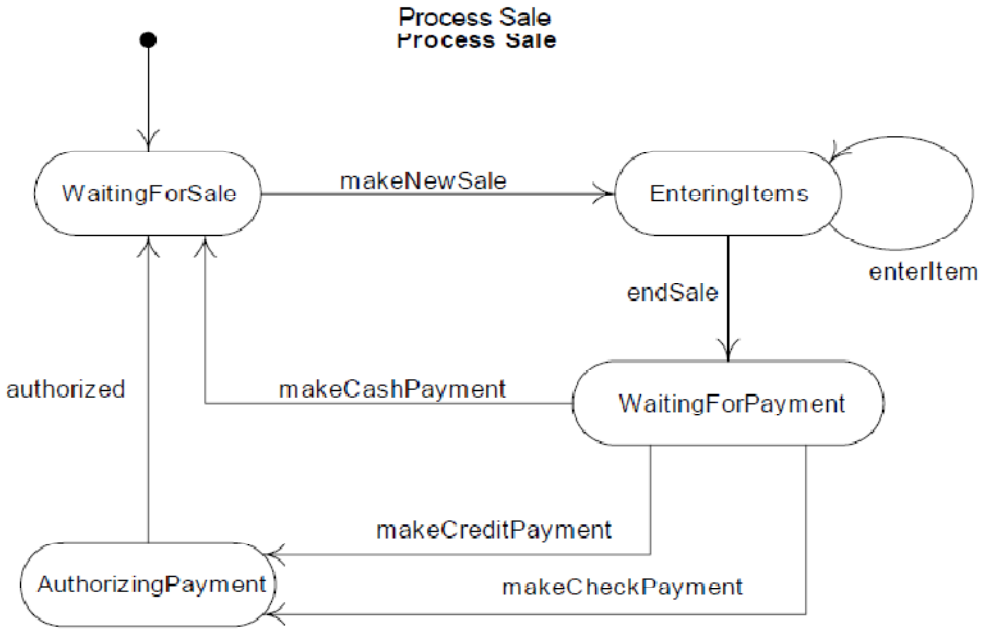


Figure 29.3 A sample statechart.

Classes that Benefit from Statechart Diagrams:
In addition to statechart diagrams for use cases or the overall system, they may be created for virtually any type or class.

State-Independent and State-Dependent Objects:
If an object always responds the same way to an event, then it is considered **state-independent** (or modeless) with respect to that event. For example, if an object receives a message, and the responding method always does the same thing—the method will typically have no conditional logic. The object is state-independent with respect to that message. If, for all events of interest, an object always reacts the same way, it is a **state-independent object.**
By contrast, **state-ependent objects** react differently to events depending on their state.
Create statecharts for state-dependent objects with complex behavior.
In general, business information systems have a minority of interesting state-dependent classes. By contrast, process control and telecommunication domains often have many state-dependent objects.

Common State-dependent Classes:
Following is a list of common objects which are usually state-dependent, and for which it may be useful to create a statechart diagram:

- ✓ **Use cases**
  Viewed as a class, the *Process Sale* use case reacts differently to the *endSale* event dependent of if a sale is underway or not.
- ✓ **Stateful sessions**—These are server-side software objects representing ongoing sessions or conversations with a client; for example, EJB stateful session objects.

Another very common example is server-side handling of web client application and presentation flow logic; for example, a Java technology servlet helper or "controller" that

remembers the state of the session with a Web client, and controls the transitions to new web pages, or the modified display of the current web page, based upon the state of the session or conversation.

o A stateful session can usually be viewed as a software class representing a use case. Recall that one of the GRASP Controller pattern variants is a use case controller, which is a use case stateful session object.

- ✓ **Systems**—This is a class representing the overall application or system. o The *"POS system."*
- ✓ **Windows**
  The Edit-Paste action is only valid if there is something in the "clipboard" to paste.
- ✓ **Controllers**—These are GRASP controller objects.
  The *Register* class, which handles the *enterItem* and *endSale* system events.
- ✓ **Transactions**—These are ways a transaction (a sale, order, payment) reacts to an event is often dependent on its current state within its overall lifecycle.
  If a *Sale* received a *makeLineItem* message after the *endSale* event, it should either raise an error condition or be ignored.
- ✓ **Devices**

TV, microwave oven: they react differently to a particular event depending upon their current state.

- ✓ **Role Mutators**—These are classes that change their role.
  A *Person* changing roles from being a civilian to a veteran.

Illustrating External and Interval Events:

Event Types:

It is useful to categorize events as follows:

 External event—Also known as a system event, is caused by something (for example, an actor) outside our system boundary. SSDs illustrate externalevents. Noteworthy external events precipitate the invocation of system operations to respond to them.

o When a cashier presses the "enter item" button on a POS terminal, an external event has occurred.

 Internal event—Caused by something inside our system boundary. In terms of software, an internal event arises when a method is invoked via a message or signal that was sent from another internal object. Messages in interaction diagrams suggest internal events.

o When *a Sale* receives a *makeLineItem* message, an internal event has occurred.

 Temporal event—Caused by the occurrence of a specific date and time or passage of time. In terms of software, a temporal event is driven by a realtime or simulated-time clock.

o Suppose that after an *endSale* operation occurs, a *makePayment* operation must occur within five minutes, otherwise the current sale is automatically purged.

Statechart Diagrams for Internal Events:

A statechart diagram can show *internal* events that typically represent messages received from other objects. Since interaction diagrams also show messages and their reactions (in terms of other messages), why use a statechart diagram to illustrate internal events and object design? The object design paradigm is that of objects that collaborate via messages to fulfill tasks; the UML interaction diagrams directly illustrates that paradigm. It is somewhat incongruous to use a statechart diagram to show a design of object messaging and interaction.1

Consequently, I have reservations about recommending the use of statechart diagrams that show internal events for the purpose of creative object design.2 However, they may be useful to summarize the results of a design, after it is complete.
By contrast, as the previous discussion on use case statechart diagrams explained, a statechart diagram for *external* events can be a helpful and succinct tool.
Prefer using statechart diagrams to illustrate external and temporal events, and the reaction to them, rather than using them to design object behaviour based on internal events.

Additional Statechart Diagram Notation:
The UML notation for statechart diagrams contains a rich set of features that are not exploited in this introduction. Three significant features are:

- transition actions transition
- guard conditions nested states

Transition Actions and Guards:
A transition can cause an action to fire. In a software implementation, this may represent the invocation of a method of the class of the statechart diagram. A transition may also have a conditional guard—or boolean test. The transition only occurs if the test passes.

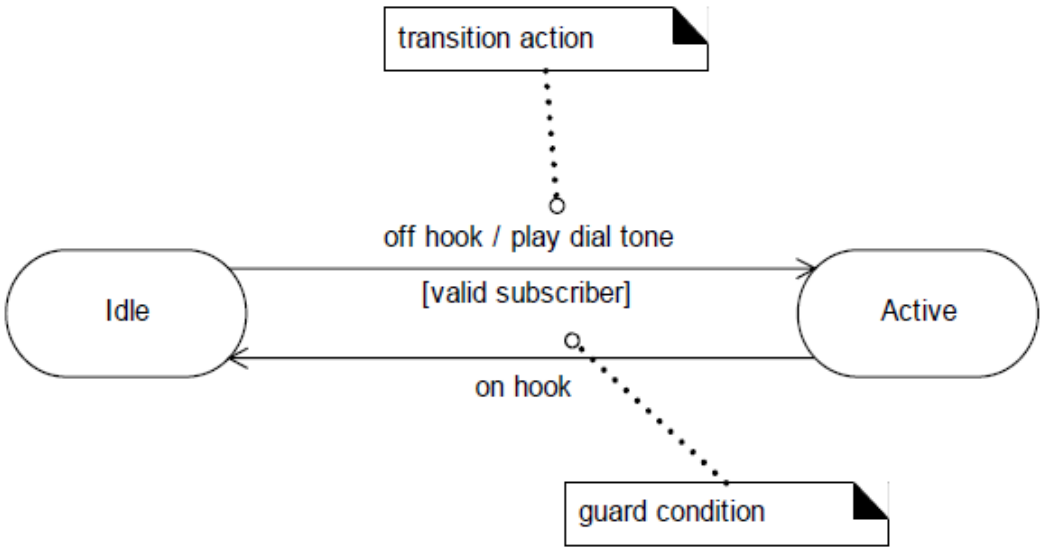


Figure 29.4 Transition action and guard notation.

Nested States
A state allows nesting to contain substates; a substate inherits the transitions of its superstate (the enclosing state). This is a key contribution of the Harel state-chart diagram notation that UML is based on, as it leads to succinct statechart diagrams. Substates may be graphically shown by nesting them in a superstate box.

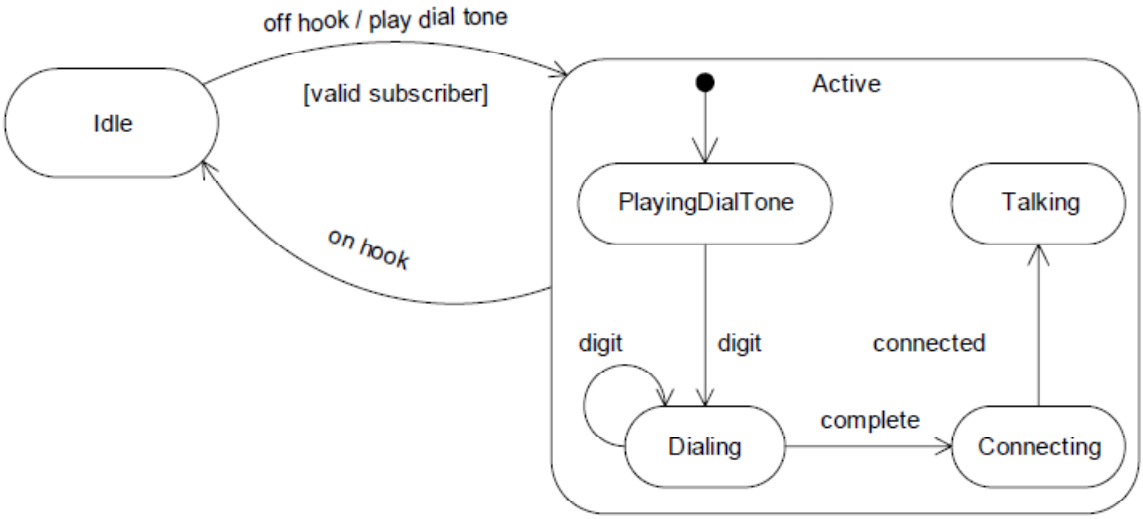


Figure 29.5 Nested states.

For example, when a transition to the *Active* state occurs, creation and transition into the *PlayingDialTone* substate occurs. No matter what substate the object is in, if the *on hook* event related to the *Active* superstate occurs, a transition to the *Idle* state occurs.

Activity Diagrams:

A UML activity diagram offers rich notation to show a sequence of activities. It may be applied to any purpose (such as visualizing the steps of a computer algorithm), but is considered especially useful for visualizing business workflows and processes, or use cases.

One of the UP workflows (disciplines) is **Business Modeling;** its purpose is to understand and communicate "the structure and the dynamics of the organization in which a system is to be deployed" . A key artifact of the Business Modeling discipline is the **Business Object Model** (a superset of the UP Domain Model), which essentially visualizes how a business works, using UML class, sequence, and activity diagrams.
Thus, activity diagrams are especially applicable within the Business Modeling discipline of the UP.
Some of the outstanding notation includes parallel activities, swimlanes, and action-object flow relationships, as illustrated in Figure 38.7. Formally, an activity diagram is considered a special kind of UML statechart diagram in which the states are actions, and event transition is automatically triggered by action completion.

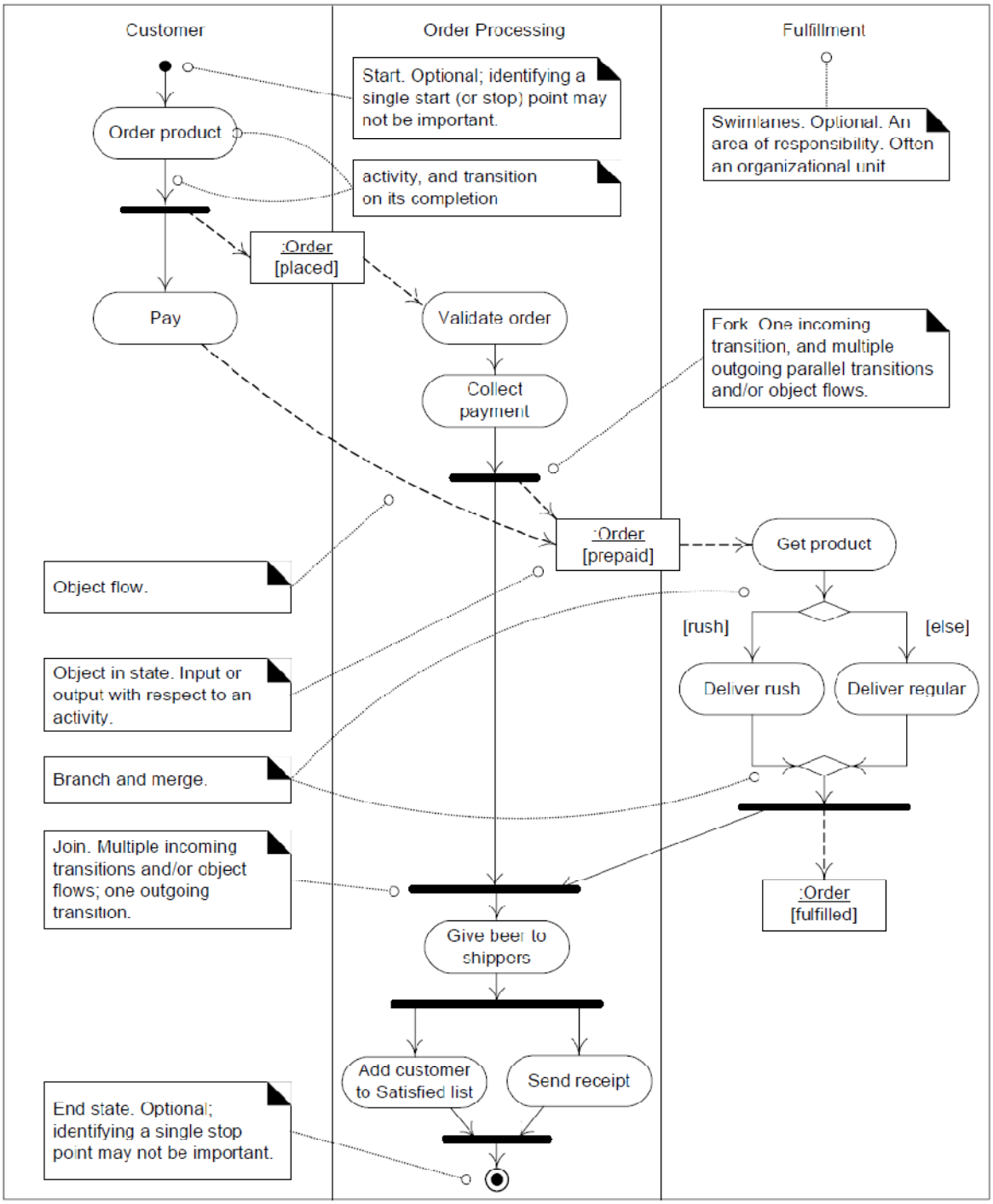


Figure 38.7 Activity diagram.

Implementation Diagrams

The UML defines several diagrams that can be used to illustrate implementation details. The most commonly used is a deployment diagram, to illustrate the deployment of components and processes to processing nodes.

Component Diagrams

To quote: A **component** represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [OMG01]. It may, for example, be source code, binary, or executable. Examples include executables such as a browser or HTTP server, a database, a DLL, or a JAR file (such as for an Enterprise Java Bean). UML components are usually shown within deployment diagrams, rather than on their own. Figure 38.4 illustrates some common notation.

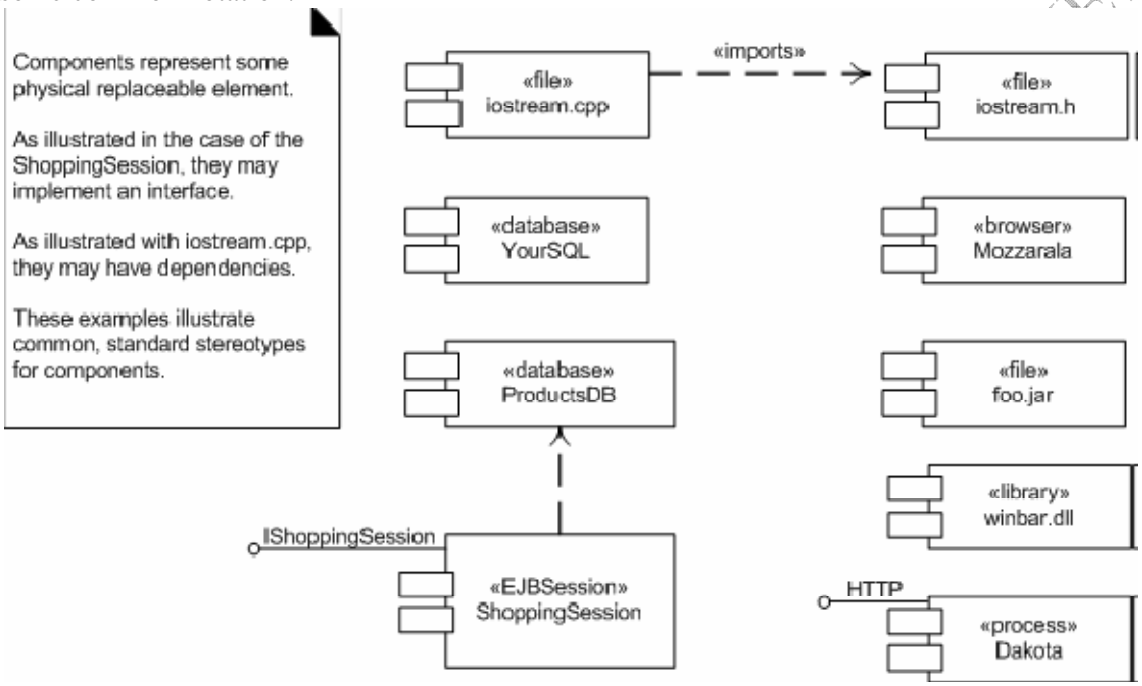


Figure 38.4 UML components.

Deployment Diagrams

A deployment diagram shows how instances of components and processes are configured for run-time execution on instances of processing **nodes** (something with memory and processing services; see Figure 38.5).

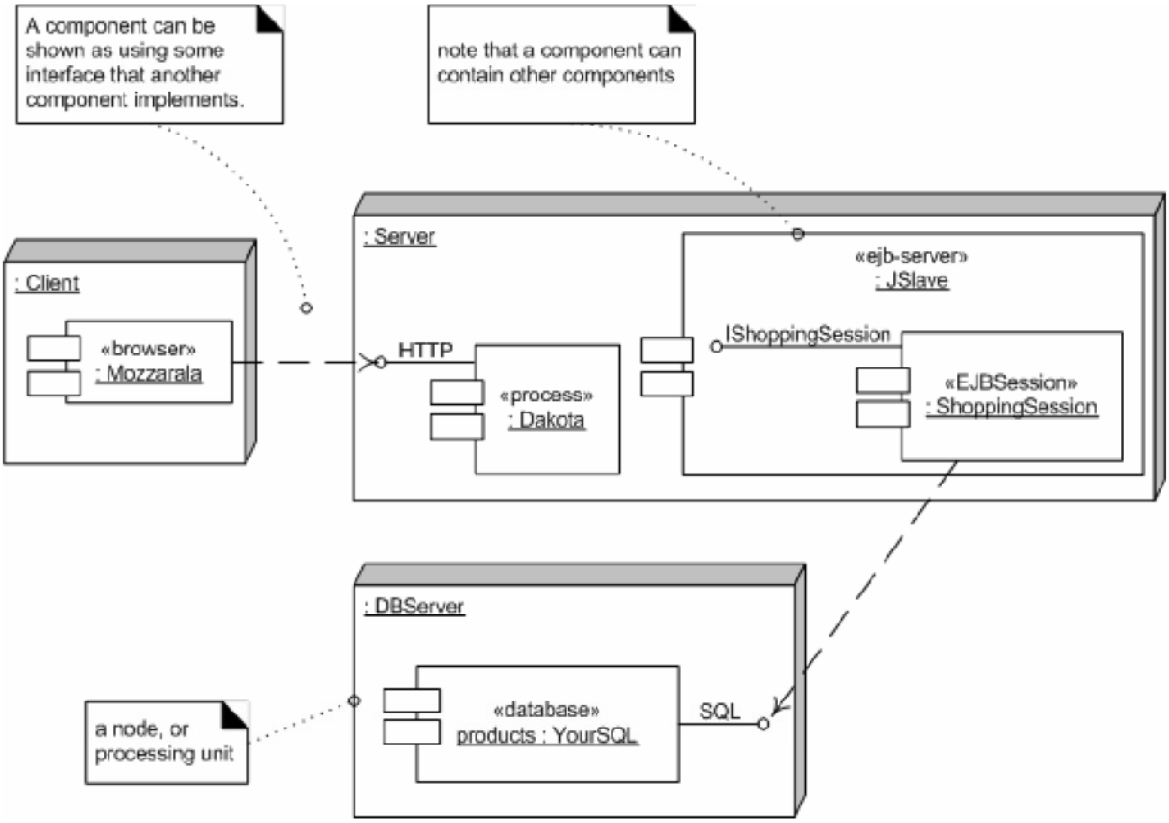


Figure 38.5 A deployment diagram.

UNIT-VI

Advanced concepts in OOAD

Unit VI Syllabus:

Advanced concepts in OOAD : Use case relationships, Generalizations Domain Model refinements, Architecture, Packaging model elements.

USECASE RELATIONSHIPS:

Objectives:
• Relate use cases with ***include*** **and** ***extend*** associations.

Introduction:
Use cases can be related to each other. For example, a subfunction use case such as *Handle Credit Payment* may be part of several regular use cases, such as *Process Sale* and *Process Rental.*

Organizing use cases into relationships has no impact on the behavior or requirements of the system. Rather, it is simply an organization mechanism to (ideally) improve communication and comprehension of the use cases, reduce duplication of text, and improve management of the usecase documents.

Caution:
In some organizations working with use cases, unproductive time has been spent debating how to relate use cases in a use case diagram, rather than the important use case work: writing text. Specifying the requirements is done by writing, not by organizing use cases, which is an optional step to possibly improve their comprehension or reduce duplication. If a team starts off use-case modeling by spending hours (or worse, days) discussing

The include Relationship:

This is the most common and important relationship. It is common to have some partial behavior that is common across several use cases.

For example, the description of paying by credit occurs in several use cases, including *Process Sale, Process Rental, Contribute to Lay-away Plan,* and so forth. Rather than duplicate this text, it is desirable to separate it into its own subfunction use case, and indicate its inclusion. This is simply refactoring and linking text to avoid duplication.

For example: **UC1:**

3. System receives payment approval and signals approval to Cashier. 4. ... Extensions: 2a. System detects failure to collaborate with external system: 1 . System signals error to Cashier. 2. Cashier asks Customer for alternate payment.

This is the **include** relationship.A slightly shorter (and thus perhaps preferred) notation to indicate an included use case is simply to underline it or highlight it in some fashion.

For example:

Process Sale

Main Success Scenario: 1 . Customer arrives at a POS checkout with goods and/or services to purchase. 7. Customer pays and System handles payment. Extensions: 7b. Paying by credit: Include *Handle Credit Payment. 7c.* Paying by check: Include *Handle Check Pavment.*

UC7: Process Rental

Extensions: 6b. Paying by credit: Include *Handle Credit Payment.*

UC12: Handle Credit Payment

Level: Subfunction Main Success Scenario: 1. Customer enters their credit account information. 2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

UC1: Process Sale

Extensions: 7b. Paying by credit: *Handle Credit Payment.* *7c.* Paying by check: *Handle Check Payment.*

The *Extensions* section of the *Process Sale* use case, but was factored out to avoid duplication. Also note that the same *Main Success* and *Extensions* structures are used in the subfunction use case as in the regular elementary business process use cases such as *Process Sale.*

Use *include* when you are repeating yourself in two or more separate use cases and you want to avoid repetition.
Another motivation is simply to decompose long use case into subunits to improve comprehension.

Using include with Asynchronous Event Handling:
Yet another use of the include relationship is to describe the handling of an asynchronous event, such as when a user is able to\ select or branch to a particular window, function, or web page, or within a range of steps.

The basic notation is to use the *a*, b*,* ... style labels in the Extensions section. Recall that these imply an extension or event that can happen at any time. A minor variation is a range label, such as *3-9,* to be used when the asynchronous event can occur within a relatively large range of the use case steps, but not all.

UC1: Process FooBars

Main Success Scenario:
1. ... Extensions:
a*. At any time. Customer selects to edit personal information: *Edit Personal Information.*
b*. At any time. Customer selects printing help: *Present Printing Help.* 2-1 1 . Customer cancels: *Cancel Transaction Confirmation.*

The include relationship can be used for most use case relationship problems.

When to use include relationship:
Factor out subfunction use cases and use the *Include* relationship when:
• They are duplicated in other use cases.
• A use case is *very* complex and long, and separating it into subunits aids comprehension.

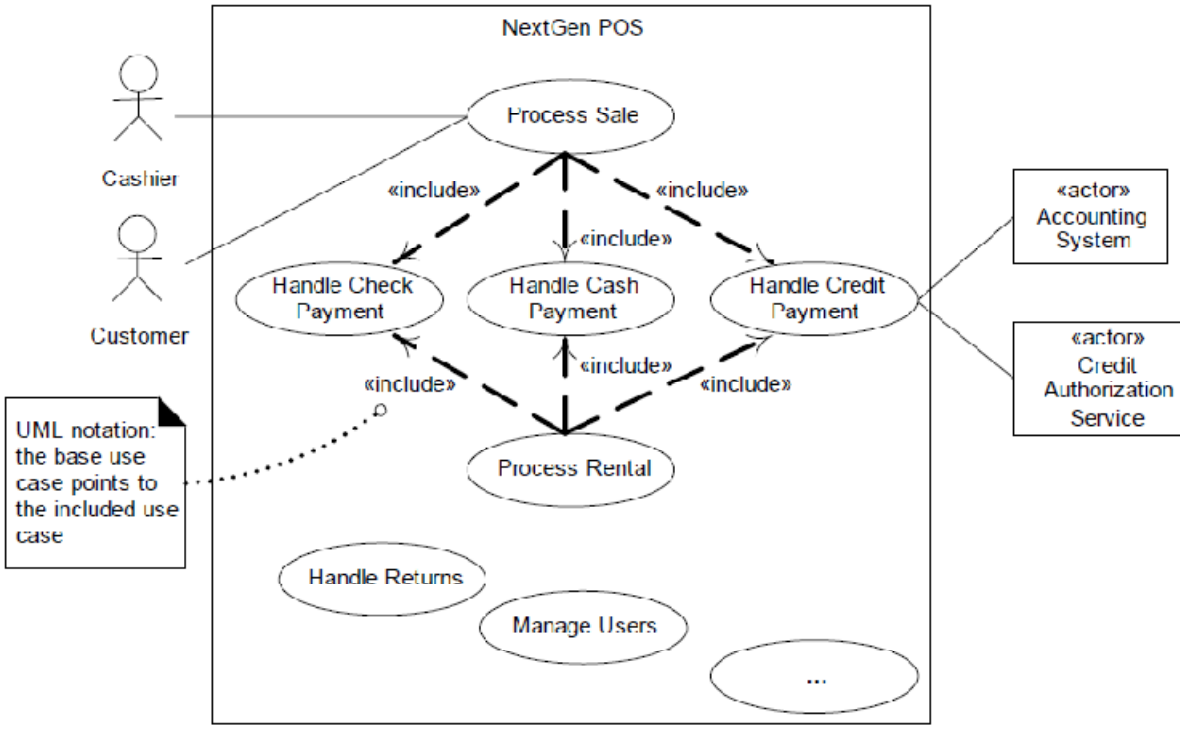


Figure 25.1 Use case include relationship in the Use-Case Model.

The extend Relationship:

Suppose a use case's text should not be modified for some reason. The use case has been baselined as a stable artifact, and can't be touched. How to append to the use case without modifying its original text?

The **extend** relationship provides an answer. The idea is to create an extending or addition use case, and within it, describe where and under what condition it extends the behavior of some base use case.
For example:

UC1: Process Sale (the base use case)

Extension Points: *VIP Customer,* step 1 . *Payment,* step 7. Main Success Scenario:
1 . Customer arrives at a POS checkout with goods and/or services to purchase.
...
7. Customer pays and System handles payment.
...

UC15: Handle Gift Certificate Payment (the extending use case)

...
Trigger: Customer wants to pay with gift certificate.
Extension Points: Payment in Process Sale. Level:
Subfunction Main Success Scenario:
1 . Customer gives gift certificate to Cashier. 2.
Cashier enters gift certificate ID.
...

Extension points are labels in the base use case which the extending use case references as the point of extension, so that the step numbering of the base use case can change without affecting the extending use case.

Extend alternative is an option when the base use case is closed to modification. Note that a signature quality of the extend relationship is that the base use case *(Process Sale)* has no reference to the extending use case *(Handle Gift Certificate Payment),* and therefore, does not define or control the conditions under which the extensions trigger.

Process Sale is complete and whole by itself, without knowing about the extending use case. Observe that this *Handle Gift Certificate Payment* addition use case could alternatively have been referenced within *Process Sale* with an include relationship, as with *Handle Credit Payment.* That is often suitable.
But this example was

motivated by the constraint that the *Process Sale* use case was not to be modified, which is the situation in which to use extend rather than include.

Some use case guidelines recommend using extending use cases and the extend relationship to model conditional or optional behavior inserted into the base use case. This is not inaccurate, but it misses the point that optional and conditional behavior can simply be recorded as text in the *Extensions* section of the base use case.

What most practically motivates using the extend technique is when it is undesirable for some reason to modify the base use case.

The extend relationship notation is illustrated in Figure 25.2.

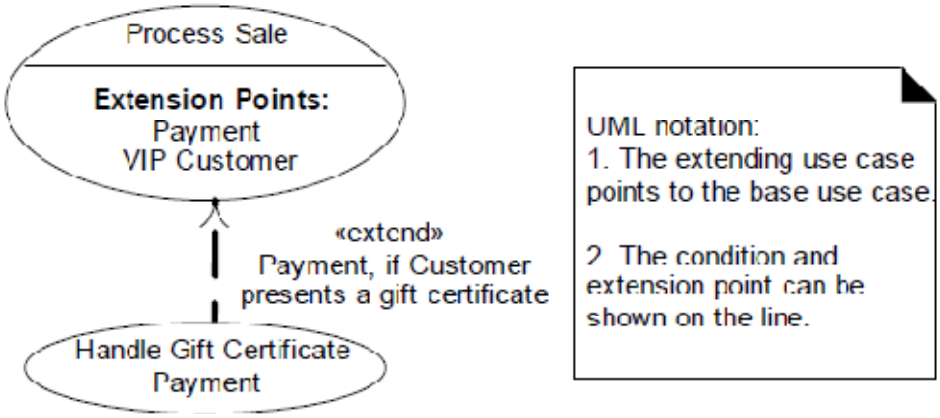


Figure 25.2 The extend relationship.

Concrete, Abstract, Base, and Addition Use Cases:

A concrete use case is initiated by an actor and performs the entire behaviour desired by the actor. These are the elementary business process use cases.
For example, *Process Sale* is a concrete use case. By contrast, an **abstract use case** is never instantiated by itself; it is a subfunction use case that is part of another use case.
Handle Credit Payment is abstract; it doesn't stand on its own, but is always part of another story, such as *Process Sale.*

A use case that includes another use case, or that is extended or specialized by another use case is called a **base use case.** *Process Sale* is a base use case with respect to the included *Handle Credit Payment* subfunction use case.

On the other hand, the use case that is an inclusion, extension, or specialization is called an **addition use case.** *Handle Credit Payment* is the addition use case in the include relationship to *Process Sale.* Addition use cases are usually abstract. Base use cases are usually concrete.

Generaliztions:

The concepts *CashPayment, CreditPayment,* and *Check Payment* are all very similar.

In this situation, useful to organize them into a **generalization-specialization class hierarchy** (or simply **class hierarchy)** in which the **superclass** *Payment* represents a more general concept, and the **subclasses** more specialized ones.

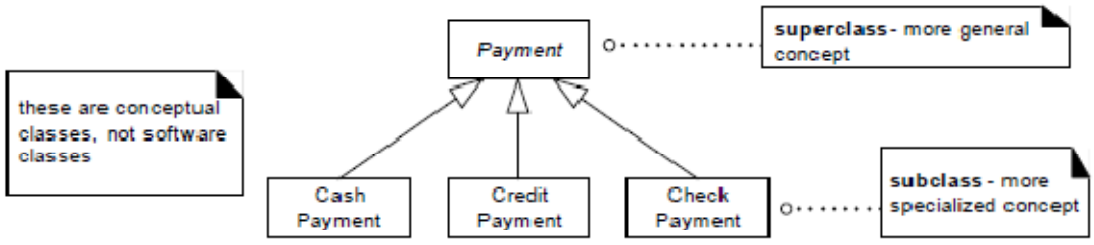


Figure 26.1 Generalization-specialization hierarchy.

Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships. It is a way to

construct taxonomic classifications among concepts which are then illustrated in class hierarchies.

Identifying a superclass and subclasses is of value in a domain model because their presence allows us to understand concepts in more general, refined and abstract terms. It leads to economy of expression, improved comprehension and a reduction in repeated information.

Thus, Identify domain superclasses and subclasses relevant to the current investigation, and illustrate them in the Domain Model.

UML notation:
in the UML the generalization relationship between elements is indicated with a large hollow triangle pointing to the more general element from the more specialized one (see Figure 26.2).

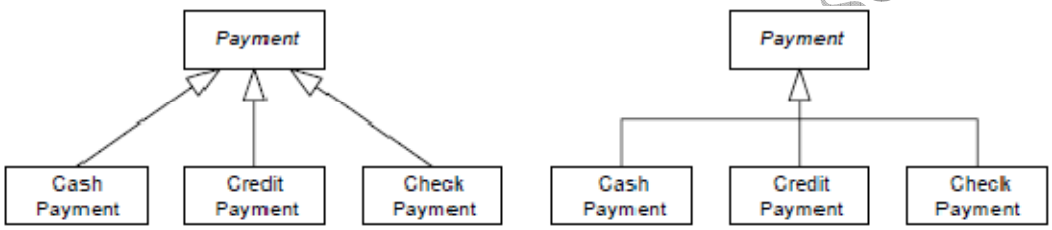


Figure 26.2 Class hierarchy with separate and shared arrow notations.

Defining Conceptual Superclasses and Subclasses:
Since it is valuable to identify conceptual super- and subclasses, it is useful to clearly and precisely understand generalization, superclasses, and subclasses in terms of class definition and class sets.

A conceptual superclass definition is more general or encompassing than a subclass definition.

For example, consider the superclass *Payment* and its subclasses *(CashPayment,*and so on).

Assume the definition *of Payment* is that it represents the transaction of transferring money (not necessarily cash) for a purchase from one party to another, and that all payments have an amount of money transferred. The model corresponding to this is shown in Figure 26.3.

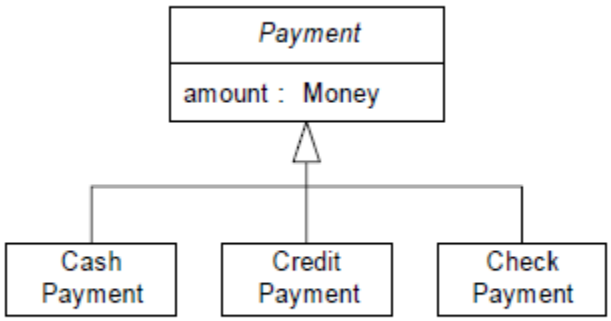


Figure 26.3 Payment class hierarchy.

A *CreditPayment* is a transfer of money via a credit institution which needs to be authorized. My definition *of Payment* encompasses and is more general than my definition of *CreditPayment.*

Generalization and Class Sets:
Conceptual subclasses and superclasses are related in terms of set membership. All the members of a conceptual subclass set are members of their superclass set.

For example, in terms of set membership, all instances of the set *CreditPayment* are also members of the set *Payment.* In a Venn diagram, this is shown as in Figure 26.4.

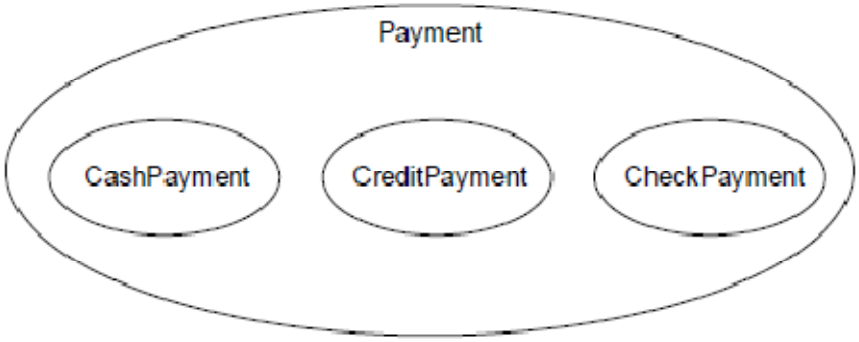


Figure 26.4 Venn diagram of set relationships.

Conceptual Subclass Definition Conformance:
When a class hierarchy is created, statements about superclasses that apply to subclasses are made. For example, Figure 26.5 states that all *Payments* have an *amount* and are associated

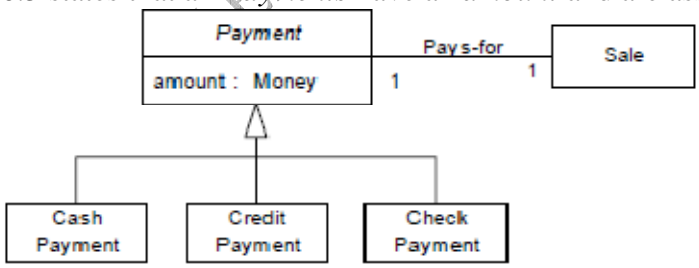


Figure 26.5 Subclass conformance.

with a *Sale.*
All *Payment* subclasses must conform to having an amount and paying for a *Sale.* In general, this rule of conformance to a superclass definition is the *100% Rule:*

100% Rule
100% of the conceptual superclass's definition should be applicable to the sub-class. The subclass must conform to 100% of the superclass's:

- attributes
- associations

Conceptual Subclass Set Conformance:
A conceptual subclass should be a member of the set of the superclass. Thus, *CreditPayment* should be a member of the set *of Payments.* Informally, this expresses the notion that the conceptual subclass *is a kind of* superclass. *CreditPayment is a kind of Payment.* More tersely, *is-a-kind-of* is called *is-a.*

This kind of conformance is the *Is-a Rule:*

> ***Is-a Rule***
>
> All the members of a subclass set must be members of their superclass set. In natural language, this can usually be informally tested by forming the statement: *Subclass is a Superclass*

For instance, the statement *CreditPayment is a Payment* makes sense, and conveys the notion of set membership conformance.

What Is a Correct Conceptual Subclass?

From the above discussion, apply the following tests to define a correct subclass when constructing a domain model:

> A potential subclass should conform to the:
>
> - 100% Rule (definition conformance)
> - Is-a Rule (set membership conformance)

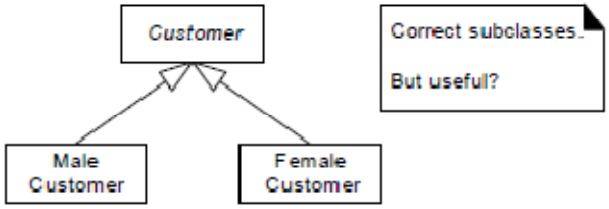


Figure 26.6 Legal conceptual class partition, but is it useful in our domain?

When to define conceptual Subclasses:

Create a conceptual subclass of a superclass when:
1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.
4. The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.

When to Define a Conceptual Superclass:

Generalization into a common superclass is usually advised when commonality is identified among potential subclasses.

Create a conceptual superclass in a generalization relationship to subclasses when:
• The potential conceptual subclasses represent variations of a similar concept.
• The subclasses will conform to the 100% and Is-a rules.
• All subclasses have the same attribute which can be factored out and expressed in the superclass.

• All subclasses have the same association which can be factored out and related to the superclass.

NextGen POS Conceptual Class Hierarchies

Payment Classes:

Based on the above criteria for partitioning the *Payment* class, it is useful to create a class hierarchy of various kinds of payments. The justification for the superclass and subclasses is shown in Figure 26.7.

Authorization Service Classes:

Credit and check authorization services are variations on a similar concept, and have common attributes of interest. This leads to the class hierarchy in Figure 26.8.

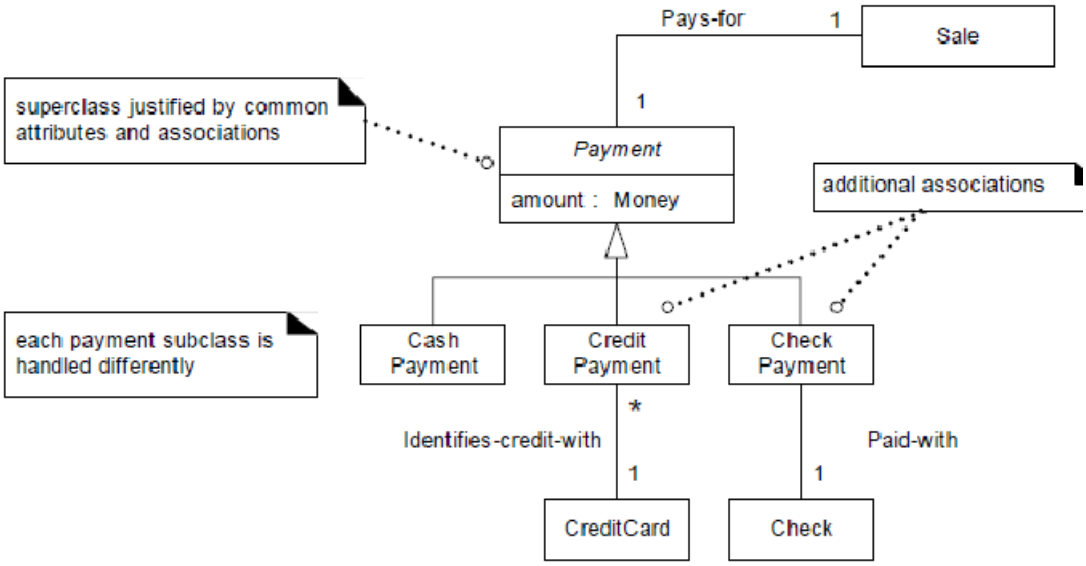


Figure 26.7 Justifying Payment subclasses.

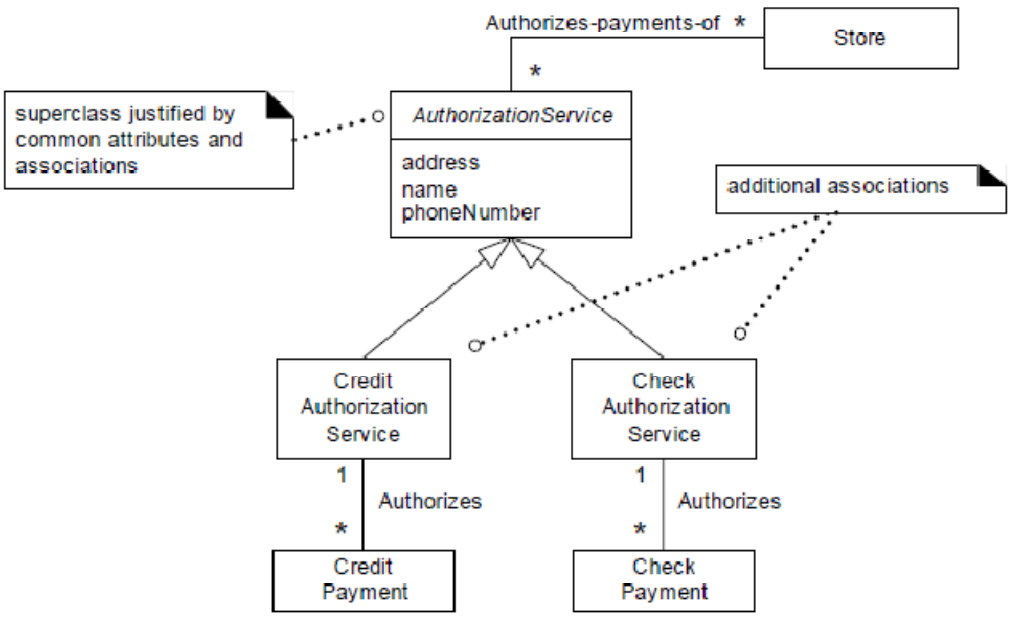


Figure 26.8 Justifying the AuthorizationService hierarchy.

Abstract Conceptual Classes:

It is useful to identify abstract classes in the domain model because they constrain what classes it is possible to have concrete instances of, thus clarifying the rules of the problem domain.

If every member of a class C must also be a member of a subclass, then class C is called an abstract conceptual class.

For example, assume that every *Payment* instance must more specifically be an instance of the subclass *CreditPayment, CashPayment,* or *CheckPayment.* This is illustrated in the Venn diagram of Figure 26.11 (b).

Since every *Payment* member is also a member of a subclass, *Payment* is an abstract conceptual class by definition.

By contrast, if there can be *Payment* instances that are not members of a subclass, it is not an abstract class, as illustrated in Figure 26.11 (a).

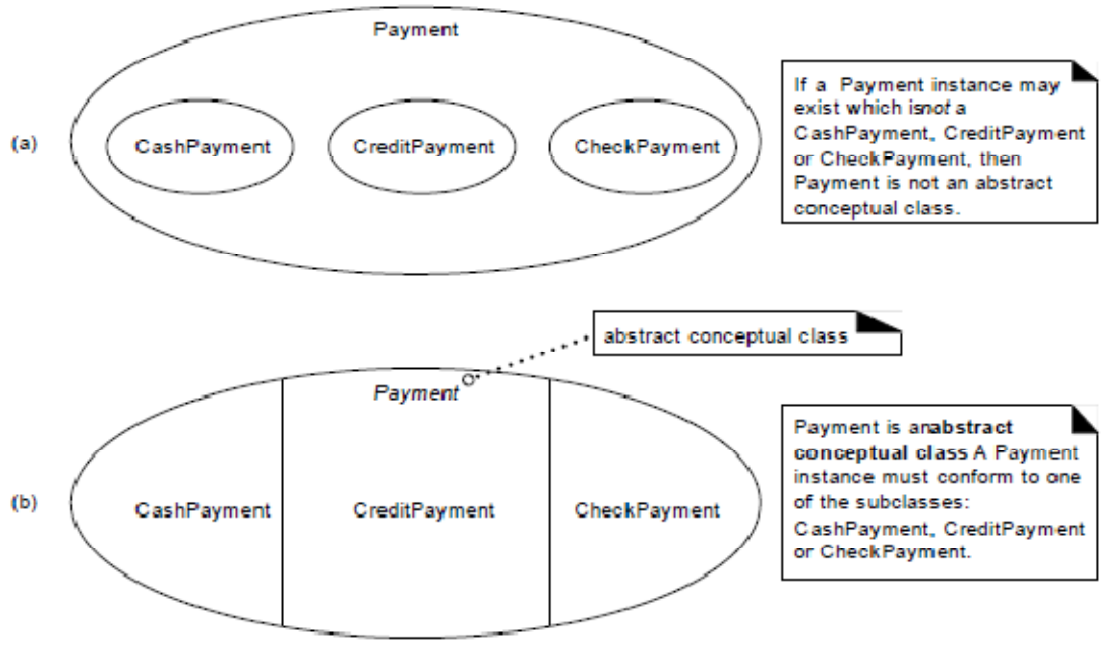


Figure 26.11 Abstract conceptual classes.

In the POS domain, every *Payment* is really a member of a subclass. Figure 26.11 (b) is the correct depiction of payments; therefore, *Payment* is an abstract conceptual class.

Abstract Class Notation in the UML:

To review, the UML provides a notation to indicate abstract classes—the class name is italicized (see Figure 26.12).

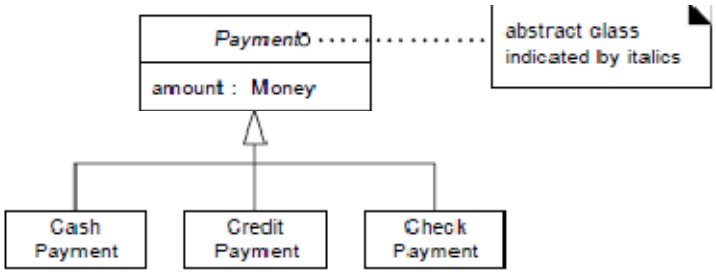


Figure 26.12 Abstract class notation.

Identify abstract classes and illustrate them with an italicized name in the Domain Model.

Modeling Changing States:

Assume that a payment can either be in an unauthorized or authorized state, and it is meaningful to show this in the domain. As shown in Figure 26.13, one modeling approach is to define subclasses of *Payment: UnauthorizedPayment* and *AuthorizedPay-ment.* However, note that a payment does not stay in one of these states; it typically transitions from unauthorized to authorized. This leads to the following guideline:

Do not model the states of a concept X as subclasses of X. Rather, either: • Define a state hierarchy and associate the states with X, or • Ignore showing the states of a concept in the domain model; show the states in state diagrams instead.

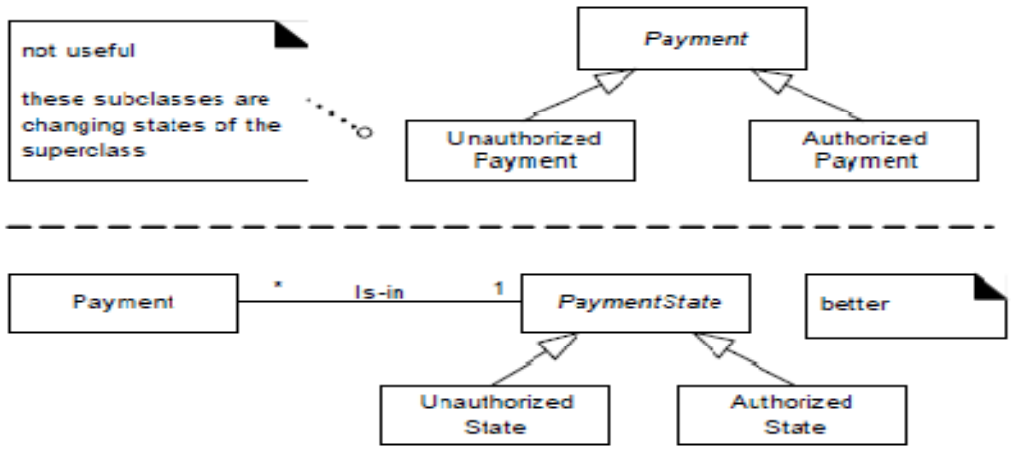


Figure 26.13 Modeling changing states.

Class Hierarchies and Inheritance in Software:

In an object-oriented programming language, a software subclass **inherits** the attribute and operation definitions of its superclasses by the creation **of software class hierarchies. Inheritance** is a software mechanism to make superclass things applicable to subclasses. It supports refactoring code from subclasses and pushing it up class hierarchies.

Therefore, inheritance has no real part to play in the discussion of the domain model, although it most definitely does when we transition to the design and implementation view.

The conceptual class hierarchies generated here may or may not be reflected in the Design Model. For example, the hierarchy of authorization service transaction classes may be collapsed or expanded into alternate software class hierarchies, depending on language features and other factors. For instance, C++ templatized classes can sometimes reduce the number of classes.

REFINING THE DOMAIN MODEL:

Objectives:

- Add association classes to the Domain Model.
- Add aggregation relationships.
- Model the time intervals of applicable information.
- Choose how to model roles.
- Organize the Domain Model into packages.

Association Classes:

The following domain requirements set the stage for association classes:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.

Furthermore, a store has a different merchant ID for each service. Placing *merchantID* in *Store* is incorrect because a *Store* can have more than one value for *merchantID.* The same is true with placing it in *Authorization-Service* (see Figure 27.1).

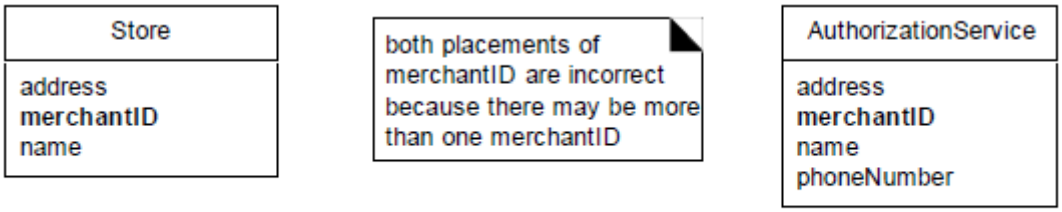


Figure 27.1 Inappropriate use of an attribute.

This leads to the following modelling principle:

In a domain model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another class that is associated with C.
For example:
- *APerson* may have many phone numbers. Place phone number in another class, such as *PhoneNumber* or *ContactInformation,* and associate many of these to *Person.*

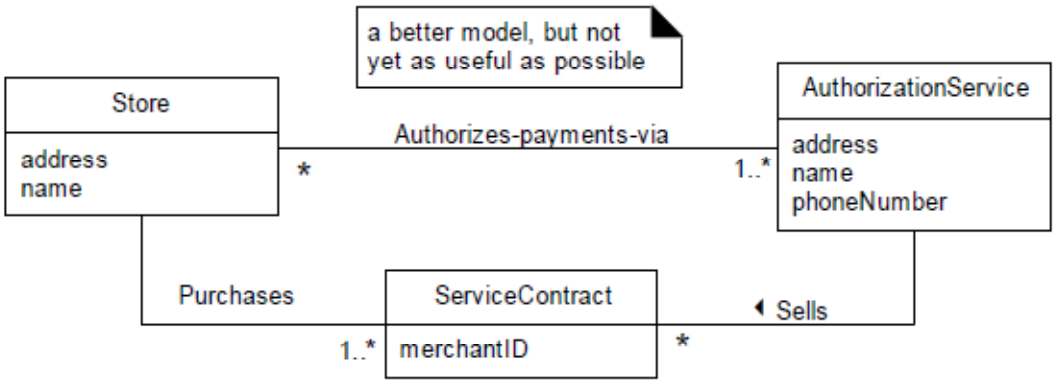


Figure 27.2 First attempt at modeling the merchantID problem.

The above principle suggests that something like the model in Figure 27.2 is more appropriate.

This leads to the notion of an **association class,** in which we can add features to the association itself. *ServiceContract* may be modeled as an association class related to the association between *Store* and *AuthorizationService.*

In the UML, this is illustrated with a dashed line from the association to the association class. Figure 27.3 visually communicates the idea that a *Service-Contract* and its attributes are related to the association between a *Store* and *AuthorizationService,* and that the lifetime of the *ServiceContract* is dependent on the relationship.

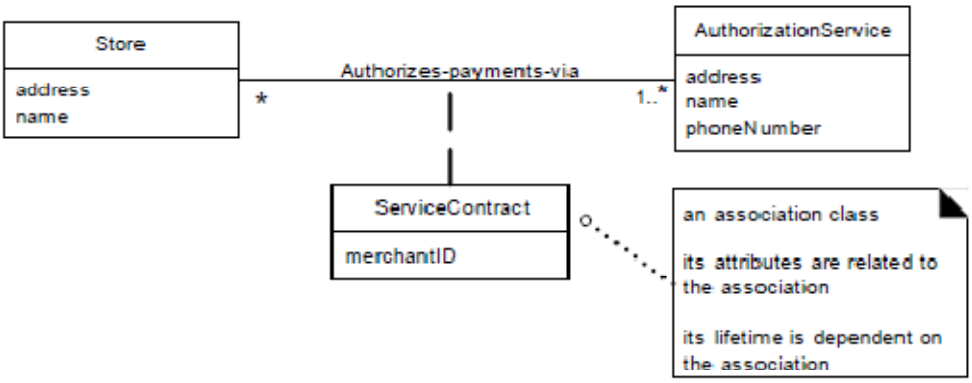


Figure 27.3 An association class.

Guidelines for adding association classes include the following:

Clues that an association class might be useful in a domain model:

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- There is a many-to-many association between two concepts, and information associated with the association itself.

The presence of a many-to-many association is a common clue that a useful association class is lurking in the background somewhere; when you see one, consider an association class. Figure 27.4 illustrates some other examples of association classes.

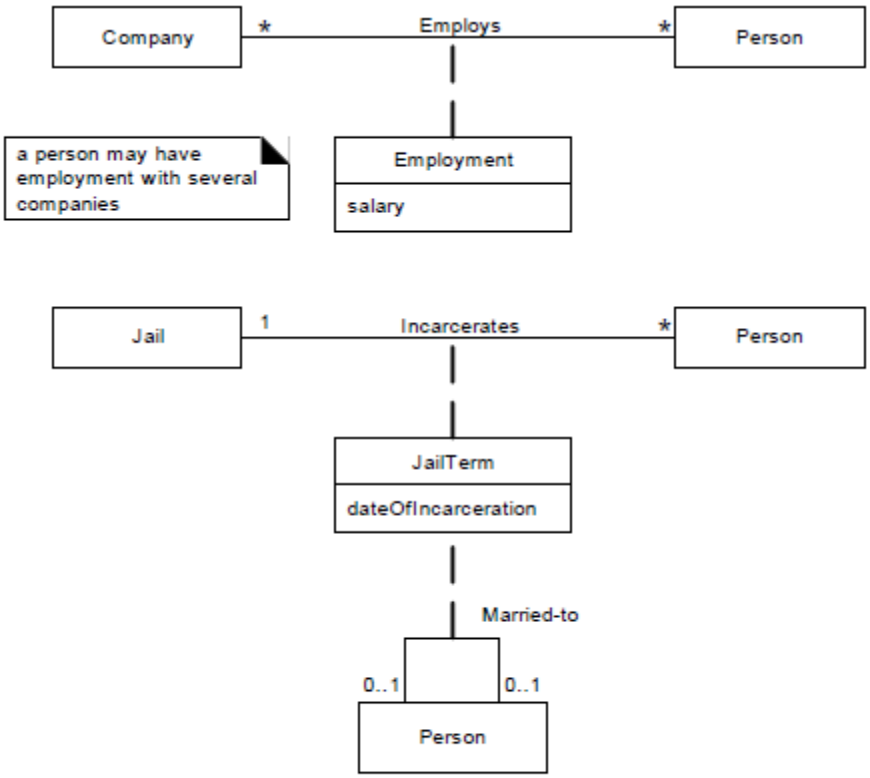


Figure 27.4 Association classes.

Aggregation and Composition:

Aggregation is a kind of association used to model whole-part relationships between things. The whole is called the **composite.**

For instance, physical assemblies are organized in aggregation relationships, such as a *Hand* aggregates *Fingers.*

Aggregation in the UML:

Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of a whole-part association (see Figure 27.5).

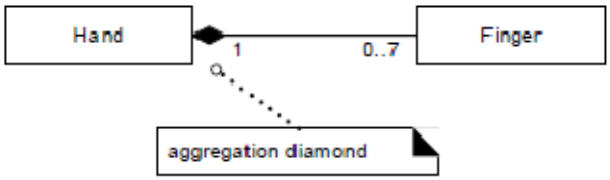


Figure 27.5 Aggregation notation.

Aggregation is a property of an association role.

The association name is often excluded in aggregation relationships since it is typically thought of as *Has-part.* However, one may be used to provide more semantic detail.

Composite Aggregation—Filled Diamond:

Composite aggregation, or composition, means that the part is a member of only one composite object, and that there is an existence and disposition dependency of the part on the composite.

For example, a hand is in a composition relationship to a finger.

In the Design Model, composition and its existence dependency implication indicates that composite software objects create (or caused the creation of) the part software objects (for example, *Sale* creates *SalesLineItem).*

But in the Domain Model, since it does not represent software objects, the notion of the whole creating the part is seldom relevant (a real sale does not create a real sales line item). However, there is still an analogy.

For example, in a "human body" domain model, one thinks of the hand as including the fingers.

Composition is signified with a filled diamond. It implies that the composite solely owns the part, and that they are in a tree structure parts hierarchy; it is the most common form of aggregation shown in models. For example, a finger is a part of at most one hand (we hope!), thus the aggregation diamond is filled to indicate composite aggregation (see Figure 27.6).

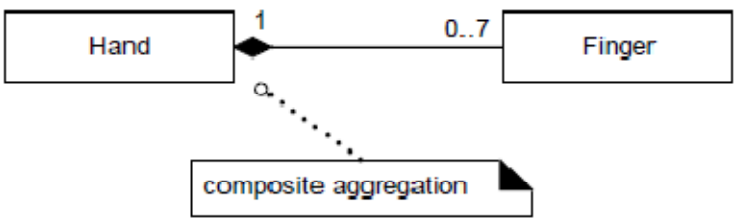


Figure 27.6 Composite aggregation.

If the multiplicity at the composite end is exactly one, the part may *not* exist separate from some composite. For example, if the finger is removed from one hand, it must be

immediately attached to another composite object (another hand, a foot, ...); at least, that is what the model is declaring, regardless of the medical merits of this idea!

If the multiplicity at the composite end is 0..1, then the part may be removed from the composite, and still exist apart from membership in any composite. So, if you want fingers floating around by themselves, use 0..1.

Shared Aggregation—Hollow Diamond:

Shared aggregation means that the multiplicity at the composite end may be more than one, and is signified with a hollow diamond.

It implies that the part may be simultaneously in many composite instances. Shared aggregation seldom) exists in physical aggregates, but rather in nonphysical concepts.

For instance, a UML package may be considered to aggregate its elements. But an element may be referenced in more than one package (it is owned by one, and referenced in others), which is an example of shared aggregation (see Figure 27.7).

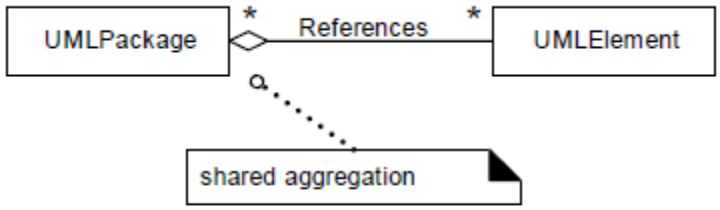


Figure 27.7 Shared aggregation.

How to Identify Aggregation

In some cases, the presence of aggregation is obvious—usually in physical assemblies. But sometimes, it is not clear.

On aggregation: If in doubt, leave it out.

Here are some guidelines that suggest when to show aggregation:

Consider showing aggregation when:

- The lifetime of the part is bound within the lifetime of the composite — there is a create-delete dependency of the part on the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as the location.
- Operations applied to the composite propagate to the parts, such as destruction, movement, recording.

Other than something being an obvious assembly of parts, the next most useful clue is the presence of a create-delete dependency of the part on the whole.

A Benefit of Showing Aggregation:

Discover and show aggregation because it has the following benefits, most of which relate to the design rather than the analysis.

• It clarifies the domain constraints regarding the eligible existence of the part independent of the whole. In composite aggregation, the part may not exist outside of the lifetime of the whole.

During design work, this has an impact on the create-delete dependencies between the whole and part software classes and database elements (in terms of referential integrity and cascading delete paths).

• It assists in the identification of a creator (the composite) using the GRASP Creator pattern.

• Operations—such as copy and delete—applied to the whole often propagate to the parts.

Aggregation in the POS Domain Model:

In the POS domain, the *SalesLineItems* may be considered a part of a composite *Sale;* in general, transaction line items are viewed as parts of an aggregate transaction (see Figure 27.8). In addition to conformance to that pattern, there is a create-delete dependency of the line items on the *Sale*—their lifetime is bound within the lifetime of the *Sale.* By similar justification, *ProductCatalog* is an aggregate of *Product-Specifications.*

Figure 27.8 Aggregation in the point-of-sale application.

No other relationship is a compelling combination that suggests whole-part semantics, a create-delete dependency, and "If in doubt, leave it out."

Roles as Concepts vs. Roles in Associations:

In a domain model, a real-world role—especially a human role—may be modeled in a number of ways, such as a discrete concept, or expressed as a role in an association.

For example, the role of cashier and manager may be expressed in at least the two ways illustrated in Figure 27.11.

The first approach may be called "roles in associations"; the second "roles as concepts."

Both approaches have advantages.

Roles in associations are appealing because they are a relatively accurate way to express the notion that the same instance of a person takes on multiple (and dynamically changing) roles in various associations. I, a person, simultaneously or in sequence, may take on the role of writer, object designer, parent, and so on.

On the other hand, roles as concepts provides ease and flexibility in adding unique attributes, associations, and additional semantics. Furthermore, the implementation of roles as separate classes is easier because of limitations of current popular object-oriented programming languages—it is not convenient to dynamically mutate an instance of one class into another, or dynamically add behavior and attributes as the role of a person changes.

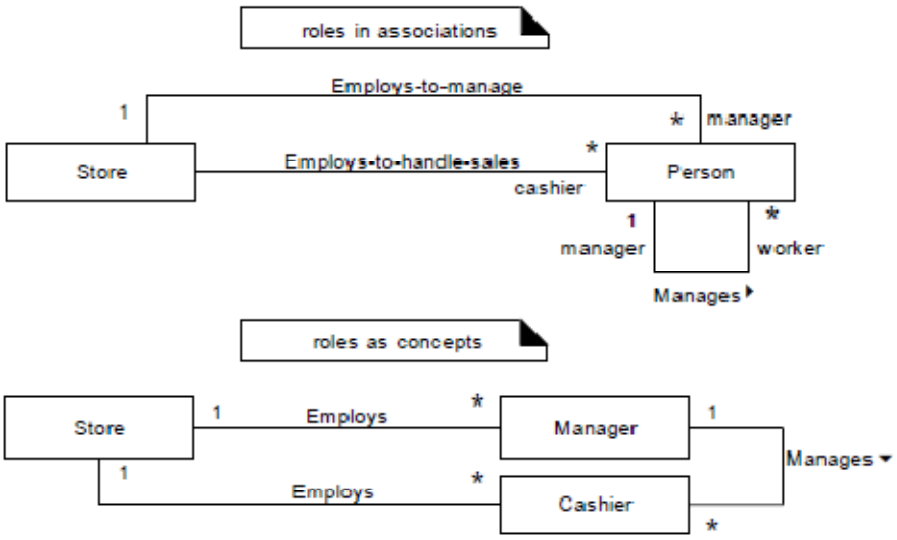


Figure 27.11 Two ways to model human roles.

Derived Elements:

A derived element can be determined from others. Attributes and associations are the most common derived elements. When should derived elements be shown?

Avoid showing derived elements in a diagram, since they add complexity without new information. However, add a derived element when it is prominent in the terminology, and excluding it impairs comprehension.

For example, a *Sale total* can be derived from *SalesLineItem* and *Product-Specification* information (see Figure 27.12). In the UML, it is shown with a "/" preceding the element name.

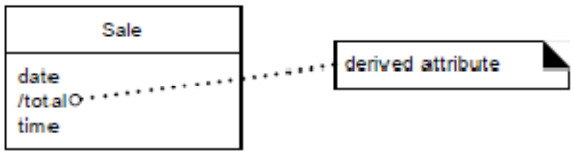


Figure 27.12 Derived attribute.

Association Role Names:

Each end of an association is a role, which has various properties, such as:

- name
- multiplicity

A role name identifies an end of an association and ideally describes the role played by objects in the association. Figure 27.10 shows role name examples. An explicit role name is not required—it is useful when the role of the object is not clear. It usually starts with a

lowercase letter. If not explicitly present, assume that the default role name is equal to the related class name, though starting with a lowercase letter.

Qualified Associations:

A qualifier may be used in an association; it distinguishes the set of objects at the far end of the association based on the qualifier value. An association with a qualifier is a **qualified association.**

For example, *ProductSpecifications* may be distinguished in a *ProductCatalog* by their *itemID,* as illustrated in Figure 27.14 (b). As contrasted in Figure 27.14 (a) vs. (b), qualification reduces the multiplicity at the far end from the qualifier, usually down from many to one.

Depicting a qualifier in a domain model communicates how, in the domain, things of one class are distinguished in relation to another class. They should not, in the domain model, be used to express design decisions about lookup keys, although that is suitable in other diagrams illustrating design decisions.

Qualifiers do not usually add useful new information, they can sharpen understanding about the domain.

The qualified associations between *Product-Catalog* and *ProductSpecification* provide a reasonable example of a value-added qualifier.

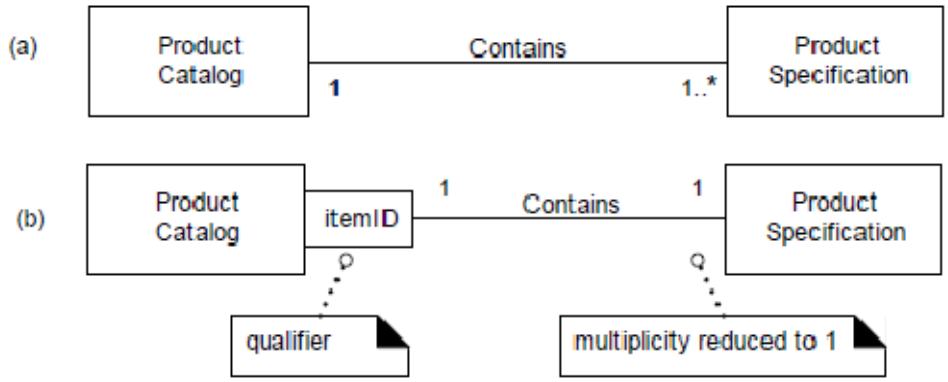


Figure 27.14 Qualified association.

Reflexive Associations:

A concept may have an association to itself; this is known as a **reflexive association** (see Figure 27.15).

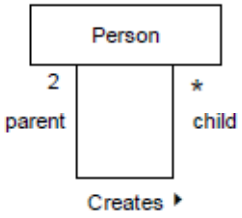


Figure 27.15 Reflexive association.

Ordered Elements:

If associated objects are ordered, this can be shown as in Figure 27.16. For example, the *SalesLineItems* must be maintained in the order entered.

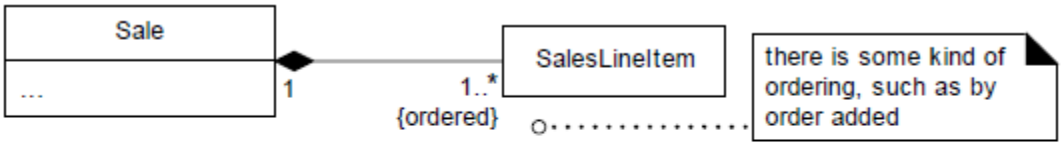


Figure 27.16 Ordered elements.

Using Packages to Organize the Domain Model: (Packaging Elements)

A domain model can easily grow large enough that it is desirable to factor it into packages of strongly related concepts, as an aid to comprehension and parallel analysis work in which different people do domain analysis within different sub-domains.

UML Package Notation:

To review, a UML package is shown as a tabbed folder (see Figure 27.17). Subordinate packages may be shown within it. The package name is within the tab if the package depicts its elements; otherwise, it is centered within the folder itself.

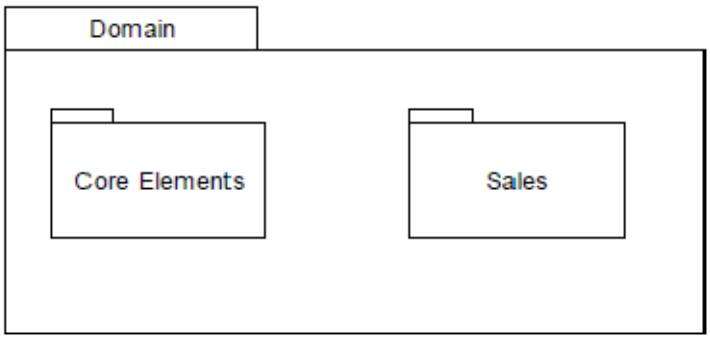


Figure 27.17 A UML package.

Ownership and References

An element is *owned* by the package within which it is defined, but may be *referenced* in other packages. In that case, the element name is qualified by the package name using the pathname format *PackageName::ElementName* (see Figure 27.18). A class shown in a foreign package may be modified with new associations, but must otherwise remain unchanged.

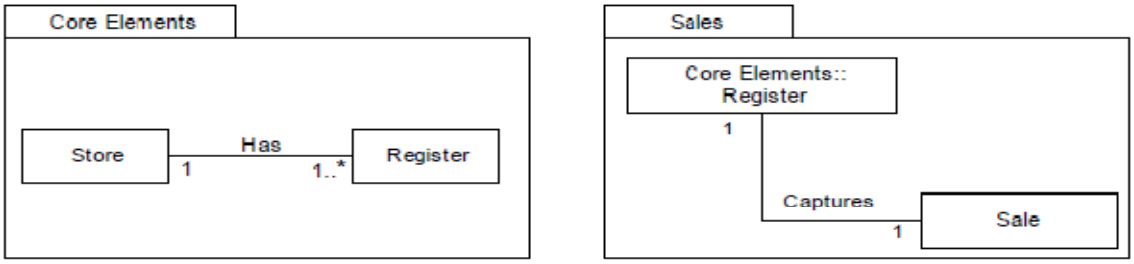


Figure 27.18 A referenced class in a package.

Package Dependencies:
If a model element is in some way dependent on another, the dependency may be shown with a dependency relationship, depicted with an arrowed line. A package dependency indicates that elements of the dependent package in some way know about or are coupled to elements in the target package.

For example, if a package references an element owned by another, a dependency exists. Thus, the *Sales* package has a dependency on the *Core Elements* package (see Figure 27.19).

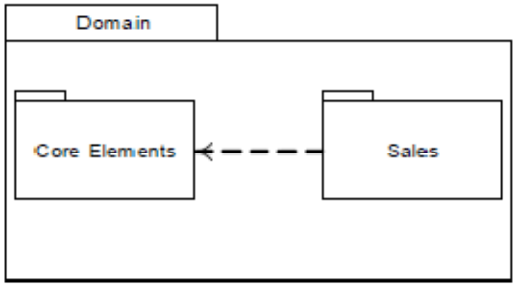


Figure 27.19 A package dependency.

Package Indication without Package Diagram
At times, it is inconvenient to draw a package diagram, but still desirable to indicate the package that the elements are a member of. In this situation, include a note (dog-eared note) on the diagram, as illustrated in Figure 27.20.

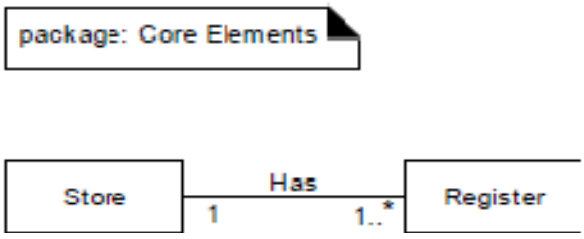


Figure 27.20 Illustrating package ownership with a note.

How to Partition the Domain Model:

How should the classes in a domain model be organized within packages? Apply the following general guidelines:

To partition the domain model into packages, place elements together that:

- are in the same subject area —closely related by concept or purpose
- are in a class hierarchy together
- participate in the same use cases
- are strongly associated

It is useful if all elements related to the domain model are rooted in a package called *Domain,* and all widely shared, common, core concepts are defined in a packaged named something

like *Core Elements* or *Common Concepts,* in the absence of any other meaningful package within which to place them.

POS Domain Model Packages

Based on the above criteria, the package organization for the POS Domain Model is shown in Figure 27.21.

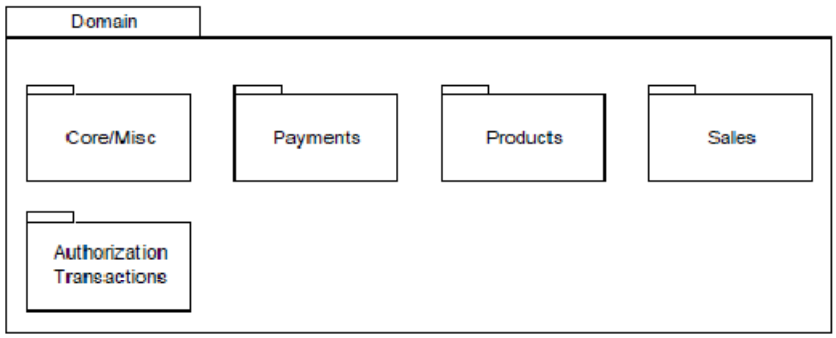


Figure 27.21 Domain concept packages.

Core/Misc Package:

A Core/Misc package (see Figure 27.22) is useful to own widely shared concepts or those without an obvious home. In later references, the package name will be abbreviated to *Core.* There are no new concepts or associations particular to this iteration in this package.

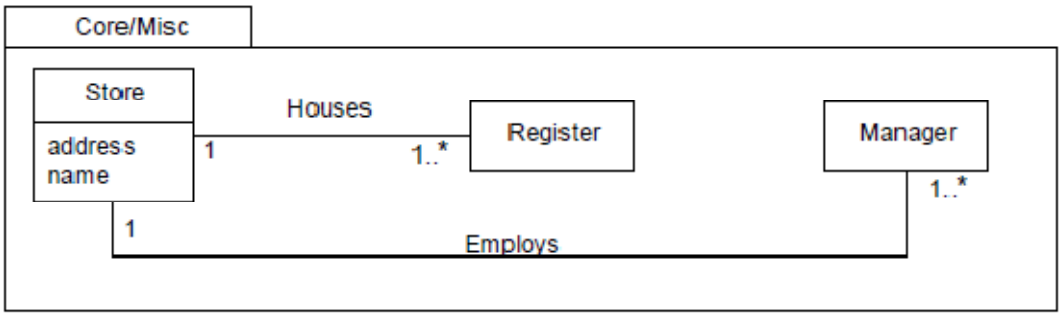


Figure 27.22 Core package.

Sales package in POS System:

With the exception of composite aggregation and derived attributes, there are no new concepts or associations particular to this iteration (see Figure 27.25).

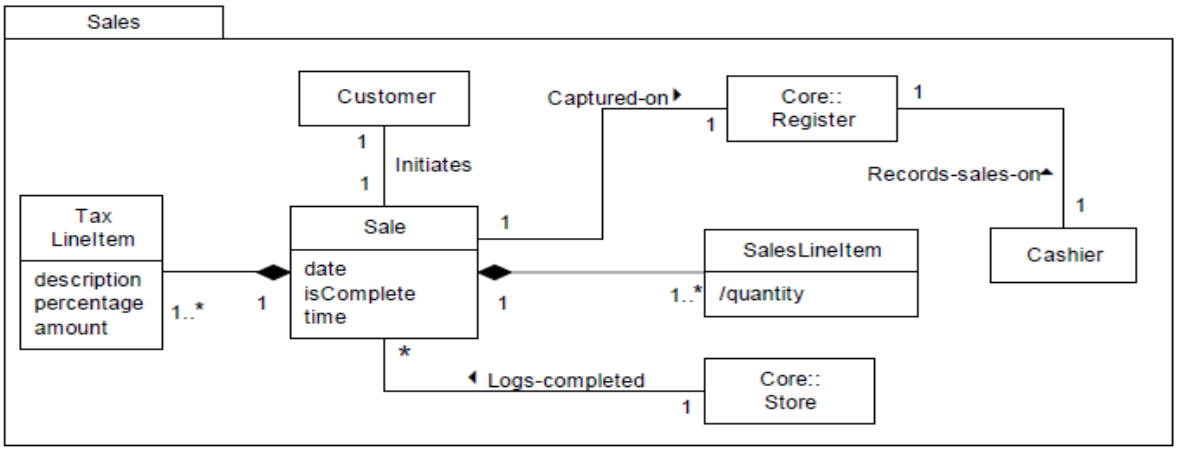


Figure 27.25 Sales package.

ARCHITECURE:

Objectives:

1.Design a logical architecture in terms of layers and partitions with the Layers pattern.
2.Illustrate the logical architecture using UML package diagrams.
3.Apply the Facade, Observer and Controller patterns.

Software Architecture:

One definition of **software architecture is:**
An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides ths organization---these elements and their interfaces, their collaborations, and their composition.

In software development, architecture is thought of as both a noun and a verb.

As a noun, the architecture includes—as the prior definition indicates—the organization and structure of the major elements of the system. Beyond this static definition, it includes the system behavior, especially in terms of large scale responsibilities of systems and subsystems, and their collaborations. In terms of a description, the architecture includes the *motivations* or rationale for

Why the system is designed the way it is.

As a verb, architecture is part investigation and part design work; for clarity, the term is best qualified, as in architectural investigation or architectural design.

Architectural investigation involves identifying those functional and (especially) non-functional requirements that have (or should have) a significant impact on the system design, such as market trends, performance, cost, maintainability, and points of evolution. Broadly, it is requirements analysis with a focus on those requirements that have special influence on the major system
design decisions.

Architectural design is the resolution of these forces and requirements in the design of the software, the hardware and networking, operations, policies, and so forth.

In the UP, architectural investigation and design are together called **architectural analysis.**

Architectural Dimensions and Views in the Unified Process:
The architecture of a system encompasses several dimensions.
For example:
The logical architecture, which describes the system in terms of its conceptual organization in layers, packages, major frameworks, classes, interfaces, and subsystems.
The deployment architecture, which describes the system in terms of the allocation of processes to processing units, and the network configuration.

The Unified Process suggests six views of the architecture we focuses on a logical view of the architecture.

Architectural Patterns and Pattern Categories:
There are well-known best practices in architectural design, especially regarding large-scale logical architecture, and these have been written as patterns, such as Layers. UP recommends *Pattern-Oriented Software Architecture* (POSA)
The POSA categorizes of patterns at different levels:
1. Architectural patterns—related to the large-scale and coarse-grained design, and typically applied during the early iterations (the elaboration phase) when the major structures and connections are established.
o The Layers patterns, which structures a system into major layers.

2. Design patterns—related to the small and medium-scale design of objects and frameworks. Applicable to designing a solution for connecting the large scale elements defined via architectural patterns, and during detailed design work for any local design aspect. Also known as micro-architectural patterns.
o The Facade pattern, which can be used to provide the interface from one layer to the next.
o The Strategy pattern, to allow pluggable algorithms.
3. Idioms—language or implementation-oriented low-level design solutions.o The Singleton pattern, to ensure global access to a single instance of a class.

There are other pattern categories. The POSA categories form a neat triad, and are useful for many patterns, but do not cover the entire gamut of published patterns. As the risk of oversimplification, a pattern is the repeating best practice of what works—in any domain. Other published categories of patterns include:
 organizational and software development process patterns
 user interface patterns
 testing patterns

Architectural Patterns/ Layers:

Solution The essential ideas of the Layers pattern are simple:
 Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.

 Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided.

A layer is a large-scale element, often composed of several packages or subsystems. The Layers pattern relates to the logical architecture; that is, it describes the conceptual organization of the design elements into groups, independent of their physical packaging or deployment.

Layers defines a general N-tier model for the logical architecture; it produces a **layered architecture.**

 Source code changes are rippling throughout the system—many parts of the systems are highly coupled.

 Application logic is intertwined with the user interface, and so cannot be reused with a different interface, nor distributed to another processing node.
 Potentially general technical services or business logic is intertwined with more application-specific logic, and so can not be reused, distributed to another node, or easily replaced with a different implementation. previous problems). It is thus difficult to divide the work along clear bound aries for different developers.

• Due to the high coupling and mixing of concerns, it is laborious and costly to evolve the application's functionality, scale up the system, or update it to use new technologies.

Example The purpose and number of layers varies across applications and application domains (information systems, operating systems, and so forth. Applied to information systems, typical layers are illustrated and explained in Figure 30.1.

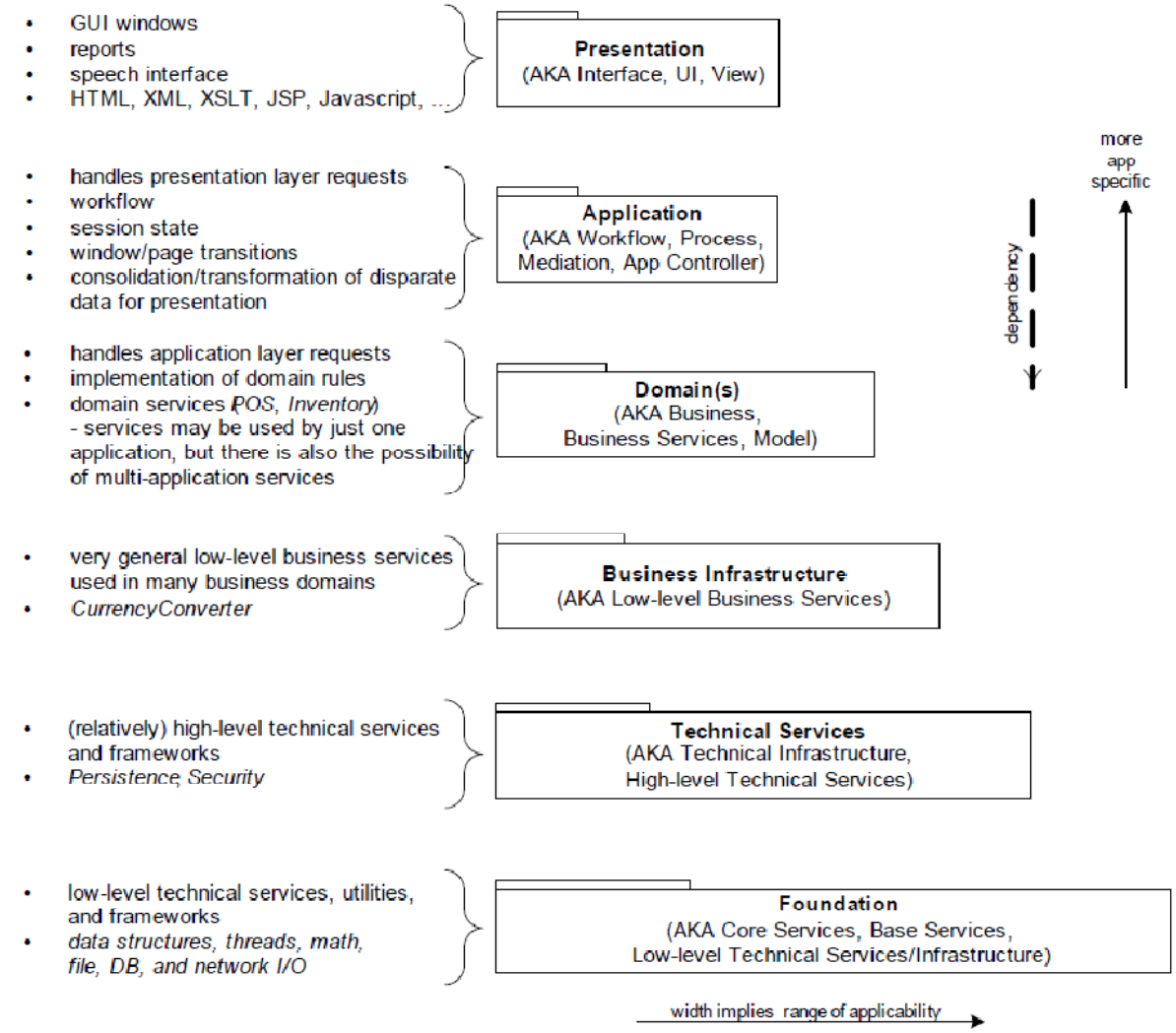


Figure 30.1 Common layers in an information system logical architecture. Based on these archetypes, Figure 30.2 illustrates a partial logical layered architecture for the NextGen application.

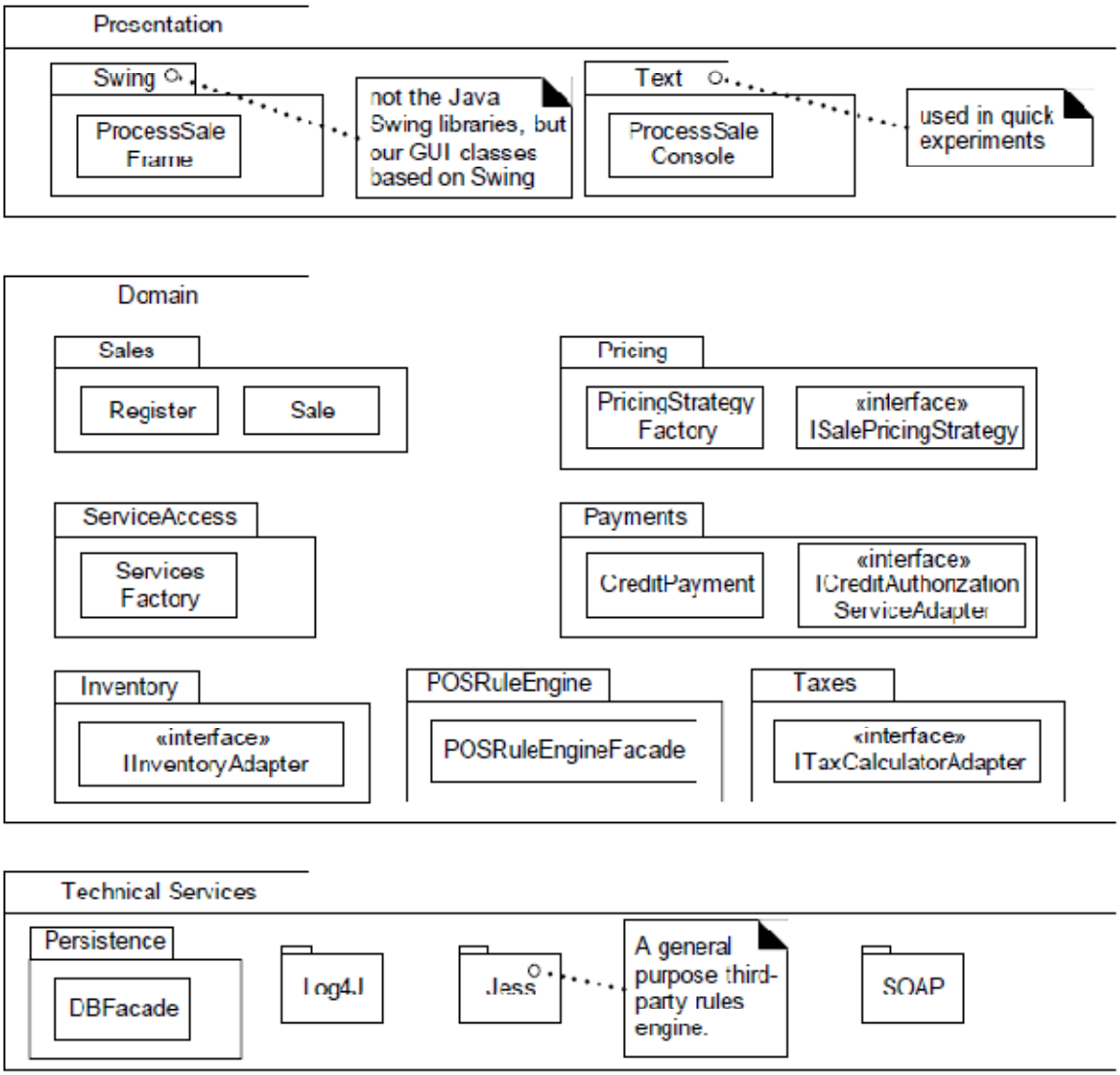


Figure 30.2 Partial logical view of layers in the NextGen application.

UML notation: Package diagrams are used to illustrate the layers. In the UML, a layer is simply a package.

Since this is iterative development, it is normal to create a design of layers that starts simple, and evolves over the iterations of the elaboration phase. One goal of this phase is to have the core architecture established (designed and implemented) by the end of the iterations in elaboration, but this does not mean doing a large up-front speculative architectural design before starting to program.

Rather, a tentative logical architecture is designed in the early iterations, and it evolves incrementally through the elaboration phase.

Observe that just a few sample types are present in this package diagram; this is not only motivated by limited page space in formatting this book, but is a signature quality of an **architectural view** diagram—it only shows a few noteworthy elements in order to concisely convey the major ideas of the architecturally significant aspects. The idea in a UP

architectural view document is to say to the reader, "I've chosen this small set of instructive elements to convey the big ideas."

Inter-Layer and Inter-Package Coupling

It is also informative to include a diagram in the logical view that illustrates noteworthy coupling between the layers and packages. A partial example is illustrated in Figure 30.3.

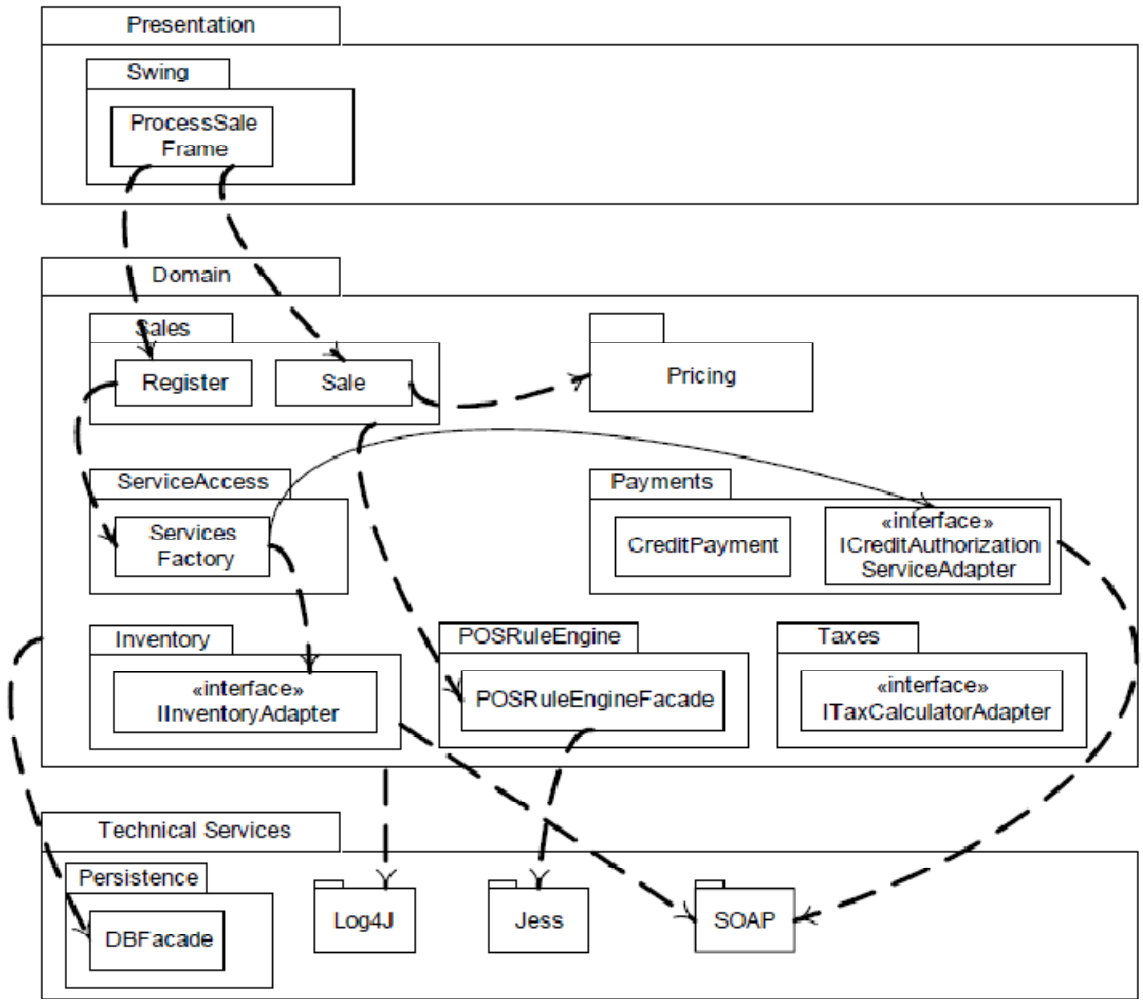


Figure 30.3 Partial coupling between packages.

UML notation:

Observe that dependency lines can be used to communicate coupling between packages or types in packages. Plain dependency lines are excellent when the communicator does not care to be more specific on the exact dependency (attribute visibility, subclassing, ...), but just wants to highlight general dependencies.

Note also the use of a dependency line emitting from a package rather than a particular type, such as from the *Sales* package to *POSRuleEngineFacade* class, and the *Domain* package to the *Log4J* package. This is useful when either the specific dependent type is not interesting, or the communicator wants to suggest that many elements of the package may share that dependency.

Another common use of a package diagram is to hide the specific types, and focus on illustrating the package-package coupling, as in the partial diagram of Figure 30.4.

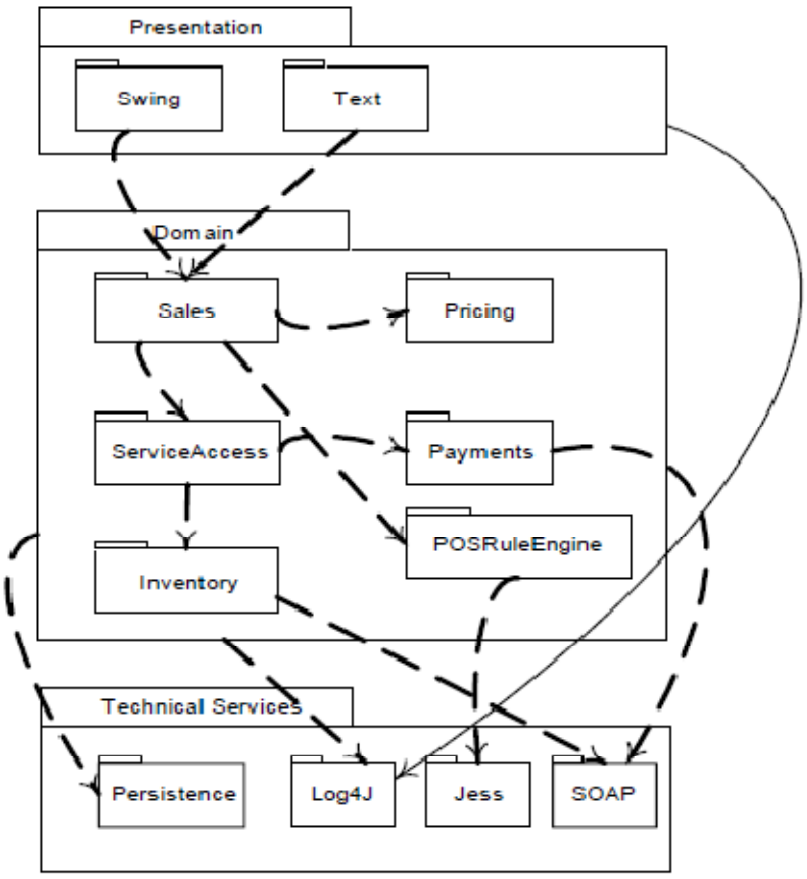


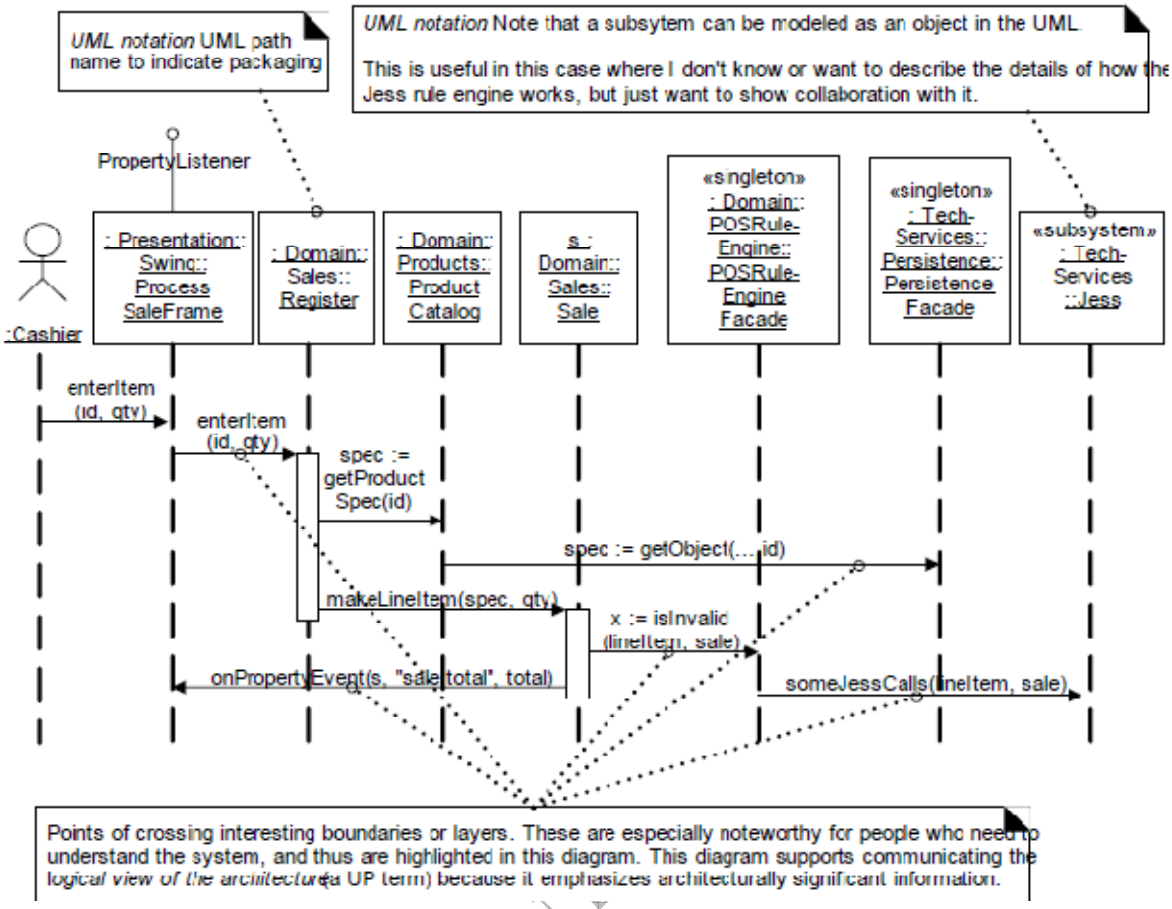
Figure 30.4 Partial package coupling.

In fact, Figure 30.4 illustrates probably the most common style of logical architecture diagram in the UML—a package diagram that shows between perhaps 5 to 20 major packages, and their dependencies.

Inter-Layer and Inter-Package Interaction Scenarios

Package diagrams show static information. To understand the dynamics of how objects across the layers connect and communicate, an interaction diagram is informative. In the spirit of an "architectural view" which hides uninteresting details, and emphasizes what the architect wants to convey, an interaction dia-cross layer and package boundaries. A set of interaction diagrams that illustrate **architecturally significant scenarios** (in the sense that they illustrate many aspects of the large-scale or big ideas in the design) is thus useful.

For example, Figure 30.5 illustrates part *of a Process Sale* scenario that emphasizes the connection points across the layers and packages.



Simple Packages vs. Subsystems

Some packages or layers are not just conceptual groups of things, but are true subsystems with behavior and interfaces. To contrast:

 The *Pricing* package is not a subsystem; it simply groups the factory and strategies used in pricing. Likewise with Foundation packages such as *java.util.*

 On the other hand, the *Persistence, POSRuleEngine,* and *Jess* packages are subsystems. They are discrete engines with cohesive responsibilities that do work.

In the UML, a subsystem can be identified with a stereotype, as in Figure 30.6.

Tiers, Layers, and Partitions

The original notion of a **tier** in architecture was a logical layer, not a physical node, but the word has become widely used to mean a physical processing node (or cluster of nodes), such as the "client tier" (the client computer). This presentation will avoid the term for clarity, but bear this in mind when reading architecture literature.

The **layers** of an architecture are said to represent the vertical slices, while **partitions** represent a horizontal division of relatively parallel subsystems of a layer. For example, the *Services* layer may be divided into partitions such as *Security* and *Reporting* (Figure 30.13).

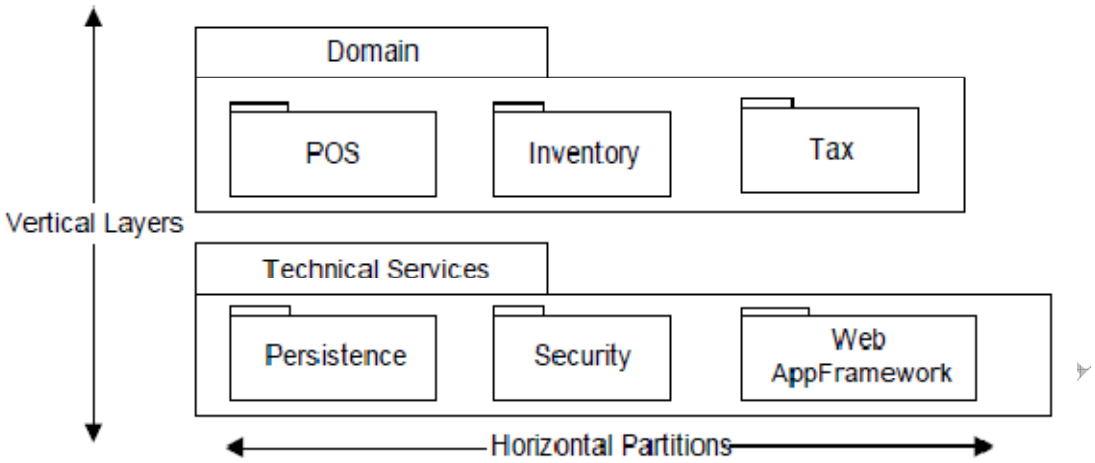


Figure 30.13 Layers and partitions.

Benefits of layering:

- In general, there is a separation of concerns, a separation of high from low-level services, and of application-specific from general services. This reduces coupling and dependencies, improves cohesion, increases reuse potential, and increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations. This is generally not possible for lower-level Technical Service or Foundation layers (e.g., *java.util),* but may be possible for Presentation, Application, and Domain layers.
- Lower layers contain reusable functions.
- Some layers (primarily the Domain and Technical Services) can be distributed.
- Development by teams is aided because of the logical segmentation.

Layers and Packages:

Part of the UP Implementation Model is the organization of the source code. For languages such as Java or C#, which provide easy package (namespace) support, the mapping from the logical packaging to the implementation packaging is sim-ilar, with notable exceptions when third-party libraries are used.3 In fact, it is only in the early stages of development, when packages have been speculatively drawn, but not implemented, that there are meaningful differences.

Over time, as the code base grows, it is common to abandon the early speculative drawings (such as the ones we have just seen), and instead use a reverse-engineering UML CASE tool that reads the source code and generates a package diagram. Then, these automatically generated package diagrams, which accurately reflect the code (the real design) become the basis for the logical view of the architecture.

To use Java as an example for mapping to implementation packages, the layers and packages illustrated in Figure 30.4 might map to Java package names as follows:

Information Systems: The Classic Three-Tier Architecture

An early influential description of a layered architecture for information systems that included a user interface and persistent storage of data was known as a **three-tier architecture** (Figure 30.14), described in the 1970s .

A classic description of the vertical tiers in a three-tier architecture is:

1. Interface—windows, reports, and so on.

2. Application Logic—tasks and rules that govern the process.
3. Storage—persistent storage mechanism.

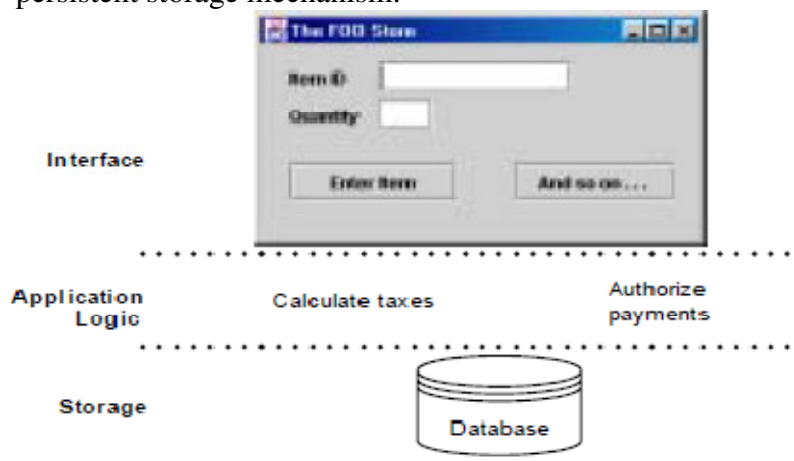


Figure 30.14 Classic view of a three-tier architecture.

The singular quality of a three-tier architecture is the separation of the application logic into a distinct logical middle tier of software.

The interface tier is relatively free of application processing; windows or web pages forward task requests to the middle tier.

The middle tier communicates with the back-end storage layer.

There was some misunderstanding that the original description implied or required a physical deployment on three computers, but the intended description was purely logical; the llocation of the tiers to compute nodes could vary from one to three. See Figure 30.15.

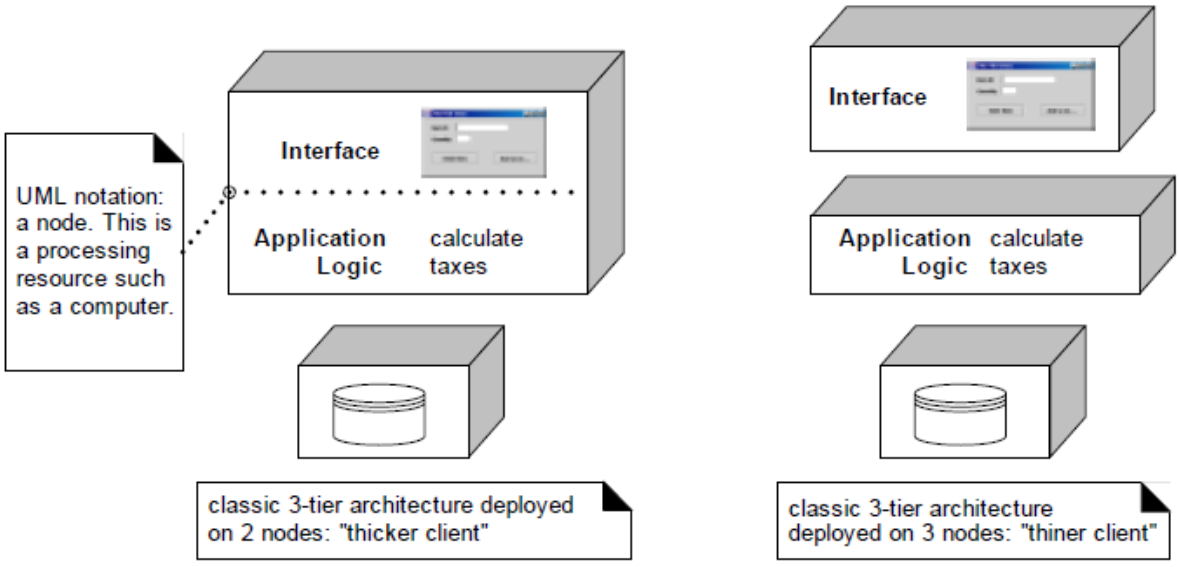


Figure 30.15 A three-tier logical division deployed in two physical architectures.

The three-tier architecture was contrasted by the Gartner Group with a **two-tier** design, in which, for example, application logic is placed within window definitions, which read and write directly to a database; there is no middle tier that separates out the application logic.

Two-tier client-server architectures became especially popular with the rise of tools such as Visual Basic and PowerBuilder.

Two-tier designs have (in some cases) the advantage of initial quick development, Nevertheless, there are applications that are primarily simple CRUD (create, retrieve,

update, delete) data intensive systems, for which this is a suitable choice.
 Indirection—layers can add a level indirection to lower-level services.
 Protected Variation—layers can protect against the impact of varying implementations.
 Low Coupling and High Cohesion—layers strongly support these goals.
 Its application specifically to object-oriented information systems.

The Model-View Separation Principle:

What kind of visibility should other packages have to the Presentation layer? How should non-window classes communicate with windows? It is desirable that there is no direct coupling from other components to window objects because the windows are related to a particular application, while (ideally) the non-windowing components may be reused in new applications or attached to a new interface. This is the Model-View Separation principle.

In this context, **model** is a synonym for the Domain layer of objects. **View** is a synonym for presentation objects, such as windows, applets and reports. The **Model-View Separation** principle4 states that model (domain) objects should not have *direct* knowledge of view (presentation) objects, at least as view objects.

So, for example, a *Register* or *Sale* object should not directly send a message to a GUI window object *ProcessSaleFrame,* asking it to display something, change color, close, and so forth.

The motivation for Model-View Separation includes:
 To support cohesive model definitions that focus on the domain processes, rather than on user interfaces.
 To allow separate development of the model and user interface layers.
 To minimize the impact of requirements changes in the interface upon the domain layer.
 To allow new views to be easily connected to an existing domain layer, without affecting the domain layer.
 To allow multiple simultaneous views on the same model object, such as both a tabular and business chart view of sales information.
 To allow execution of the model layer independent of the user interface layer, such as in a message-processing or batch-mode system.
 To allow easy porting of the model layer to another user interface framework. *Model-View Separation and "Upward" Communication.*

How can windows obtain information to display? Usually, it is sufficient for them to send messages to domain objects, querying for information which they then display in widgets—a **polling or pull-from-above** model of display updates.

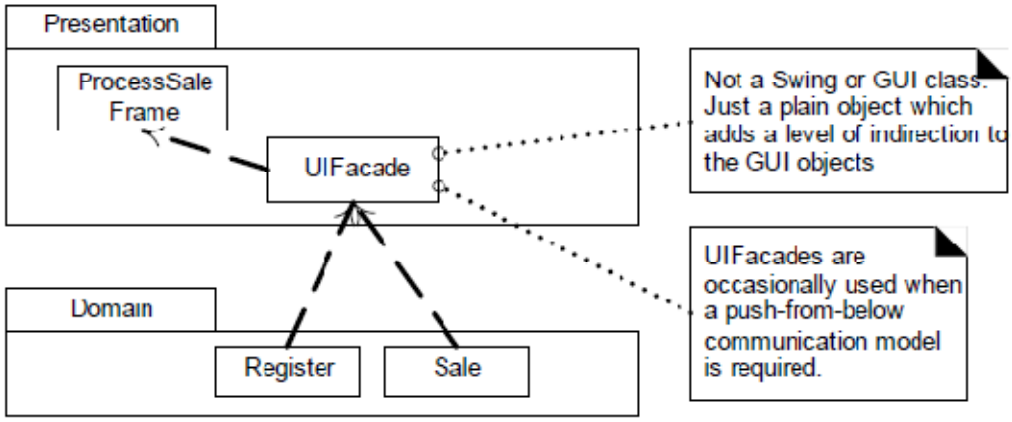


Figure 30.16 A Presentation layer UIFacade is occasionally used for push-from-below designs.

However, a polling model is sometimes insufficient. For example, polling every second across thousands of objects to discover only one or two changes, which are then used to refresh a GUI display, is not efficient. In this case it is more efficient for the few changing domain objects to communicate with windows to cause a display update as the state of domain objects changes. Typical situations of this case include:

- Monitoring applications, such as telecommunications network management.
- Simulation applications which require visualization, such as aerodynamics modeling.

In these situations, a **push-from-below** model of display update is required. Because of the restriction of the Model-View Separation pattern, this leads to the need for "indirect" communication from lower objects up to windows—pushing up notification to update from below.

There are two common solutions:

1. The Observer pattern, via making the GUI object simply appear as an object that implements an interface such as *PropertyListener*.
2. A Presentation facade object. That is, adding a facade within the Presenta tion layer that receives requests from below. This is an example of adding Indirection to provide Protected Variation if the GUI changes. For example, see Figure 30.16.