CS8592 OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT-I V-Sem-CSE 2017-Regulations

UNIT I UNIFIED PROCESS AND USECASE DIAGRAMS 9

Introduction to OOAD with OO Basics – Unified Process – UML diagrams – Use Case –Case study – the Next Gen POS system, Inception -Use case Modeling – Relating Use cases – include, extend and generalization – When to use Use-cases

Question Bank – UNIT-I

1) What is the critcal ability of an Object Oriented System?

A critical ability of **Object Oriented development** is to skillfully **assign responsibilities** to **software objects**. It is one activity that must be performed either while drawing a UML diagram or programming and it strongly influences the robustness, maintainability, and reusability of software components.

2) **What is Analysis and Design?**

Analysis emphasizes an ***investigation*** of the **problem and requirements**, rather than a **solution.** For example, if a new online trading system is desired, Analysis answers the following questions :

How will it be used?

What are its functions?

"Analysis" is a broad term, and it is refered as ***requirements analysis*** (an investigation of the requirements) or ***object-oriented analysis*** (an investigation of the domain objects).

Design emphasizes a ***conceptual solution*** (in software and hardware) that fulfills the **requirements**, rather than its **implementation**. For example, a description of a **database schema** and **software objects**. Design ideas often exclude low-level or "obvious" details obvious to the intended consumers. Ultimately, designs can be implemented, and the **implementation** (such as **code**) expresses the true and complete realized design. As with analysis, the term is best qualified, as in ***object-oriented design*** or *database design*.

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

3) **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 flight information system, some of the concepts include *Plane*, *Flight*, and *Pilot*.

During **object-oriented design** (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a *Plane* software object may have a *tailNumber* attribute and a *getFlightHistory* method (see Figure 1.1).

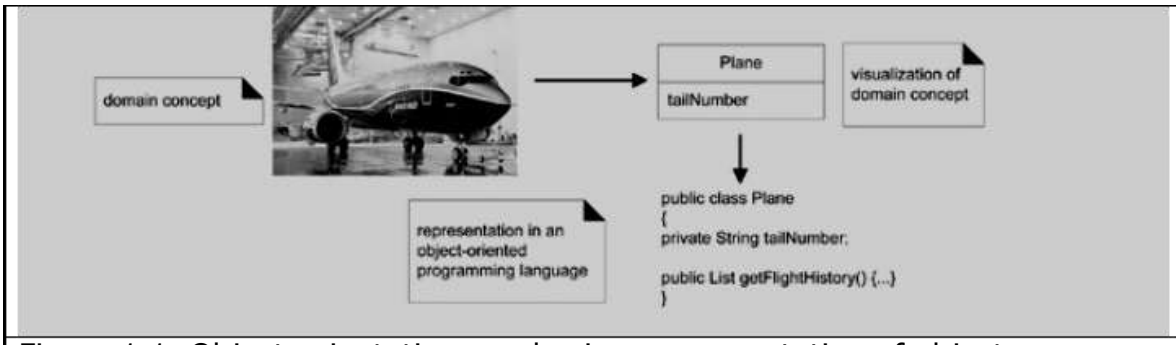


Figure 1.1. Object-orientation emphasizes representation of objects.

4) **Define a Domain Model**.

Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that are considered noteworthy.

The result can be expressed in a **domain model** that shows the *noteworthy* domain concepts or objects. For example, a partial domain model is shown in Figure 1.2.

. It can be noted that a domain model is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain and it is also called a **conceptual object model**.

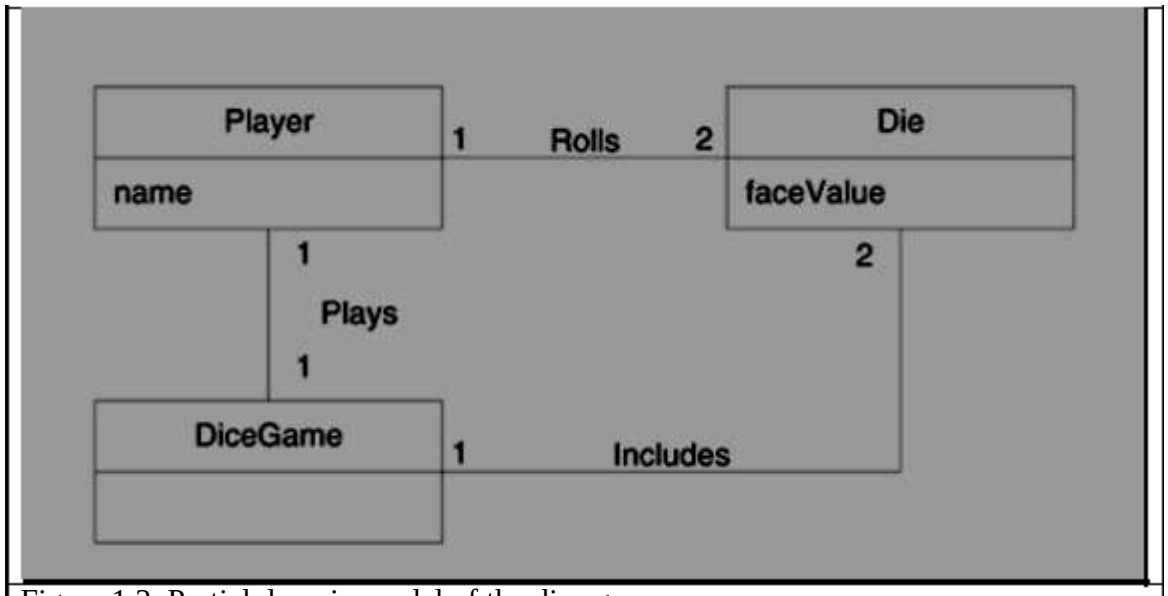


Figure 1.2. Partial domain model of the dice game

5) **What is the UML?**

The Unified Modeling Language(UML) is a visual language for specifying, constructing and documenting the artifacts of systems.

The word *visual* in the definition is a key point - the UML is the de facto standard *diagramming notation* for drawing or presenting pictures (with some text) related to software primarily OO software

6) **What are the three ways to apply UML?**

1) UML as sketch Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
2) UML as blueprint Relatively detailed design diagrams used either for
1) reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).

If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams. These "blueprints" can help the reader understand the bigpicture elements, structure, and collaborations.

Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying UML sketching).
3) UML as programming language - Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language." This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

Agile modeling emphasizes *UML as sketch*; this is a common way to apply the UML, often with a high return on the investment of time (which is typically short).

7) What are the three perspectives to apply UML?

Three Perspectives to Apply UML

1. **Conceptual perspective** the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
2. **Specification (software) perspective** the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
3. **Implementation (software) perspective** the diagrams describe software implementations in a particular technology (such as Java).

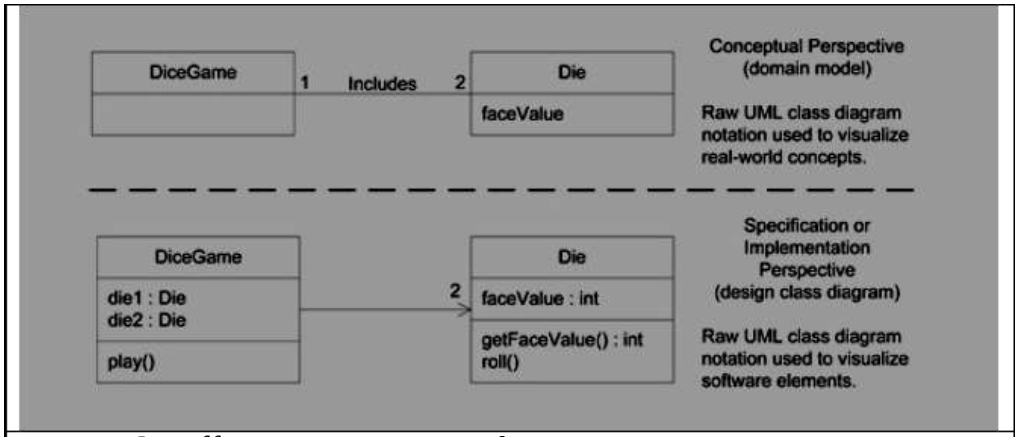


Figure 1.3. Different perspectives with UML.

8) **Define a) Conceptual Class,b) Software Class,c) Implementation Class.**

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.

Implementation class - a class implemented in a specific OO language such as Java.

9) **What is the Unified Process(UP)?**

A **software development process** describes an approach to building, deploying, and possibly maintaining software.

The **Unified Process** has emerged as a popular *iterative* 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 UP is very flexible and open, and encourages including skillful practices from other iterative methods, such as from **Extreme Programming** (**XP**), **Scrum**, and so forth. For example, XP's **test-driven development, refactoring** and **continuous integration** practices can fit within a UP project. So can Scrum's common project room ("war room") and daily Scrum meeting practice.

The UP combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented process description.

10) What is the importance of the Unified Process(UP)?

1) The UP is an *iterative* process. Iterative development influences how to introduce OOA/D , and to understand how it is best practiced.
2) UP practices provide an example *structure* for how to do and thus how to explain OOA/D.
3) The UP is flexible, and can be applied in a lightweight and *agile* approach that includes practices from other agile methods (such as XP or Scrum).

11) What is Iterative and Evolutionary Development?

A key practice in both the UP and most other modern methods is **iterative development**. In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called **iterations**; the outcome of each is a tested, integrated, and executable *partial* 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** (see Figure 2.1). Because feedback and adaptation evolve the specifications and design, it is also known as **iterative and evolutionary development**. Early iterative process ideas were known as spiral development and evolutionary development [Boehm]

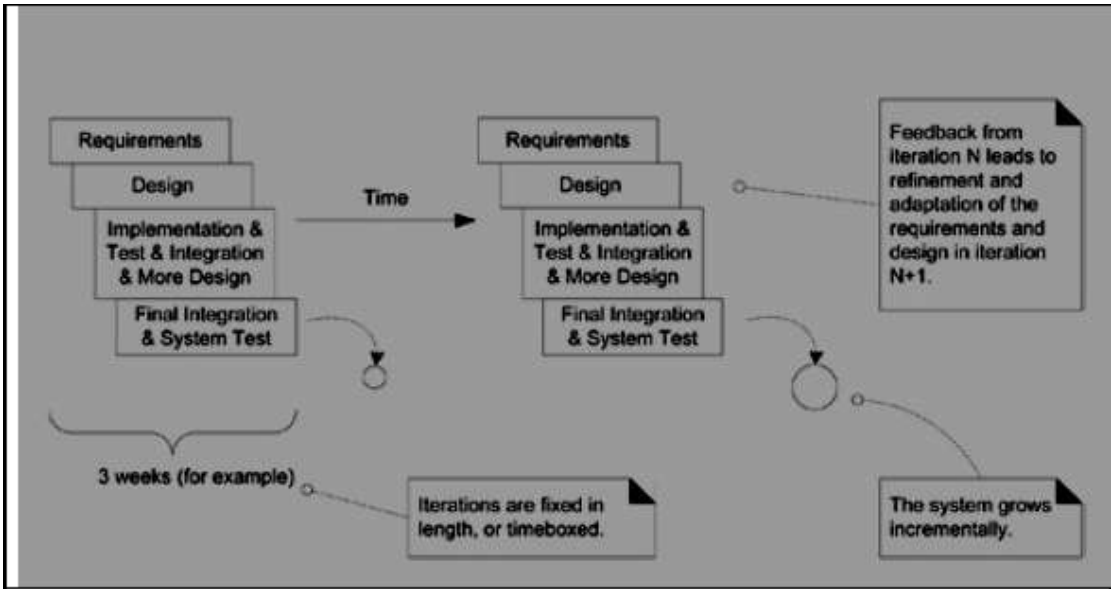


Figure 1.4. Iterative and evolutionary development.

.

12) **What are the benefits of iterative development?**

Benefits include:

- less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods
- 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

13) **Why is Waterfall is so failure prone?**

There isn't one simple answer to why the waterfall is so failure-prone, but it is strongly related to a key false assumption underlying many failed software projects that the specifications are predictable and stable and can be correctly defined at the start, with low change rates. This turns out to be far from accurate and a costly misunderstanding. A study by Boehm and Papaccio showed that a typical software project experienced a 25% change in requirements. And this trend was corroborated in another major study of thousands of software projects, with change rates that go even higher35% to 50% for large projects as illustrated in Figure 1.5.

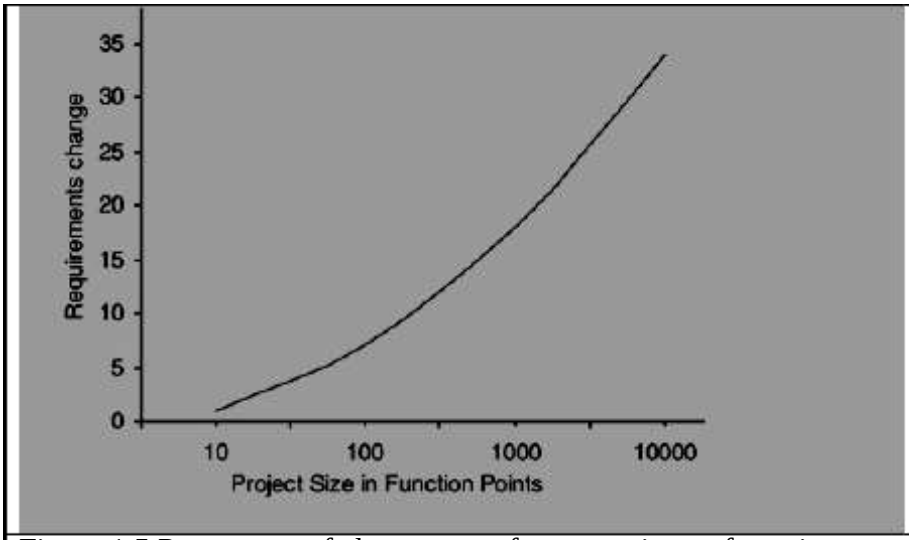


Figure 1.5 Percentage of change on software projects of varying sizes.

14) What is the need for feedback and adaptation?

The Need for Feedback and Adaptation

In complex, changing systems (such as most software projects) feedback and adaptation are key ingredients for success.

- Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.
- Feedback from tests and developers to refine the design or models.
- Feedback from the progress of the team tackling early features to refine the schedule and estimates.
- Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration

15) What are agile methods?

Agile development methods usually apply timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage *agility,*rapid and flexible response to change.

Agile methods share best practices like evolutionary refinement of plans, requirements, and design. In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams, and more.

16) Name any five agile principles.

a) Satisfy the customer through early and continuous delivery of valuable software.
b) Agile processes harness change for customer's competitive advantage.
c) Deliver working software frequently

d) Agile software promote sustainable development
e) The best,architecture,requirements,and designs emerge from self-organizing teams

17) **What is Agile Modeling?**

The very act of modeling can and should provide a way to better understand the problem or solution space. The purpose of doing UML is to quickly explore(more quickly than with code) alternatives and the path to a good OO design. Many agile methods,such as feature-driven Development,DSDM,and Scrum include significant modeling sessions, The purpose of modeling is primarily support understanding and communication ,not documentation. Defer simple or straightforward design problems until programming – solve them while programming and testing. Model and apply the UML for the smaller percentage of unusual,difficult,tricky parts of the design space. Prefer sketching UML on white boards,and capturing the diagrams with a digital camera.

Model in pairs(or traids) at the whiteboard – The purpose of modeling is to discover,understand and share the understanding.

Create models in parallel. For example,start sketching in one whiteboard, Dynamic View UML Interaction diagram,and in another whiteboard,the static view,the UML Class Diagram.

All prior diagrams are incomplete hints – throw-away explorations;only tested code demonstrates the true code.

18) What are the other Critical UP practices?

The idea for UP practice is short timeboxed iterative,evolutionary,and adaptive development. Some additional best practices and key ideas in UP are

- Tackle high-risk and high-value issue in eary iterations
- Continual evaluation,feedback and requirements from users
- Build cohesive,core architecture in ealy iterations
- Continuously verify quality;test early,often and realistically
- Practice Change Request and Configuration Management

19) What are the different UP Phases?

An UP Project organizes work and iterations across four major phases :

1) **Inception** – approximate Vision,Business case,Scope,vague estimates
2) **Elaboration** – Refined BVision,iterative implementation of the core architecture,resolution of high risks,identification of most requirements and scope,more realistic estimates
3) **Construction** – Iterative implementation of the remaining lower risk and easier elements,and preparation for deployment
4) **Transition** – beta tests,deployment

20) What are the UP disciplines?

In the UP,an artifact is the general term for any work product : Code,Web Graphics,Schema,Test Documents,diagrams,model and so on.

Some of the artifacts in the following Disciplines are :

a) Business Modeling – The Domain Model artifact,to visualize noteworthy concepts in the application domain
b) Requirements – The Use Case Model and Supplementary specification artifacts to capture functional and non-functional requirements
c) Design – The Design Model artifact,to design the software artifacts
d) Implementation – Programming and building the system,not deploying it

21) What is Inception?

Envision the product scope,vision,and business case.

Do the stakeholders have basic agreement on the vision of the project,and is it worth investing in serious investigations?

The purpose of inception stage is not to define all the requirements. The Up is not the waterfall and the first phase inception is not the time todo all requirements or create believable estimates or plans. That happens during elaboration.

22) How long is the Inception phase can be?

The intent of inception is to establish some initial common vision for the objectives of the project,determine if it is feasible,and decide if its is worth some serious investigation in elaboration. It can be brief.

23) **List any five inception artifacts**.

Artifact	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. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) 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.
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.

24) **Define Requirements.**

Requirements are capabilities and conditions to which the system – and more broadly,the project must conform.

The UP promotes a set of best practices,one of which is to manage requirements.

In the context of changing and and unclear stakeholder's wishes – Managing requirements means – a systematic approach to finding,documenting,organizing,and tracking the changing requirements of a system.

A prime challenge of requirements analysis is to find,communicate,and remember(Towrite down) what is really needed,in a formthat clearly speaks to the client and development team members.

25) **What are the types and categories of requirements?**

In the UP, requirements are categorized according to the FURPS+ model [Grady92], 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, configurability.

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** for example, a physical box.
- **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 of not considering some important facet of the system.

Some of these requirements are collectively called the **quality attributes, quality requirements**, or the "- ilities" of a system. These include usability, reliability, performance, and supportability. In common usage, requirements are categorized as **functional** (behavioral) or **non-functional** (everything else); some dislike this broad generalization [BCK98], but it is very widely used.

26) What are the key requirement artifacts?

The Key requirements artifacts are :

Use-Case Model - A set of typical scenarios of using a system. There are primarily for functional
(behavioral) requirements.

Supplementary Specification - Basically, everything not in the use cases. This artifact is primarily for all non-functional requirements, such as performance or licensing. It is also the place to record functional features not expressed (or expressible) as use cases; for example, a report generation.

Glossary - In its simplest form, the Glossary defines noteworthy terms. It also encompasses the concept of the data dictionary, which records requirements related to data, such as validation rules, acceptable values, and so forth. The Glossary can detail any element: an attribute of an object, a parameter of an operation call, a report layout, and so forth.

Vision - Summarizes high-level requirements that are elaborated in the Use-Case Model and Supplementary Specification, and summarizes the business case for the project. A short executive overview document for quickly learning the project's big ideas.

Business Rules - Business rules (also called Domain Rules) typically describe requirements or policies that transcend one software projectthey are required in the domain or business, and many applications may need to conform to them. An excellent example is government tax laws. Domain rule details *may* be recorded in the Supplementary Specification, but because they are usually more enduring and applicable than for one software project, placing them in a central Business Rules artifact (shared by all analysts of the company) makes for better reuse of the analysis effort.

27) **What are Use Cases?**

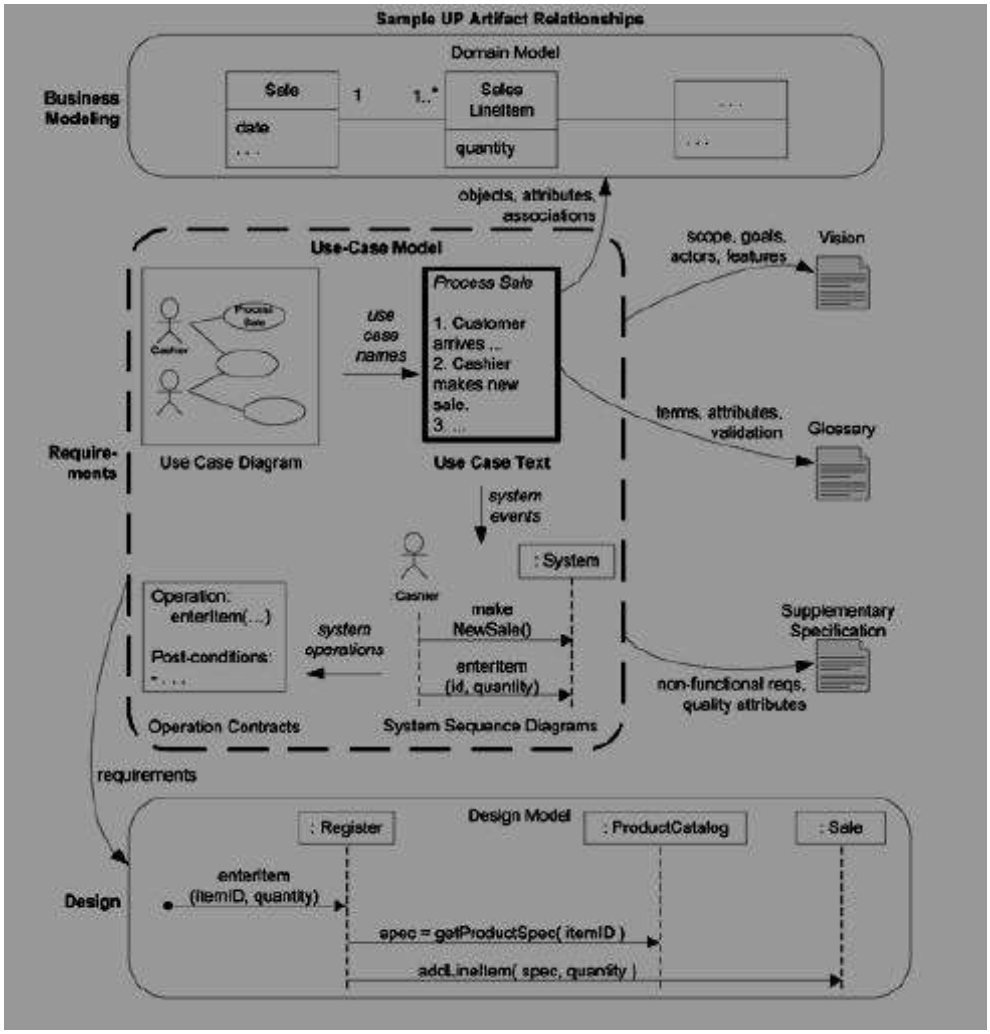
Informally, **use cases** are ***text stories*** of some **actor** using a **system** to meet **goals**.

Use Case Example :

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 i**nventory**. The customer receives a **receipt** from the system and then leaves with the items.

Notice that ***use cases are not diagrams, they are text .*** Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

Use cases often need to be more detailed or structured than this example, but the essence is **discovering and recording functional requirements by writing stories** of using a system to fulfill user goals; that is, *cases of use*.



28) Define a) Actors,b)Scenarios,and c) Use cases.

Definition: **What are Actors, Scenarios, and Use Cases?**

Informal definitions:

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; 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 payment denial.

Informally then, **a use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal

Definition of a use case 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 [RUP].

29) What is Use-case Modeling?

Use-Case Model is the set of all written use cases; it is a model of the system's functionality and environment. **Use cases** are **text documents, not diagrams**, and use-case modeling is primarily an act of **writing text**, not **drawing diagrams**.
The Use-Case Model is not the only requirement artifact in the UP. There are also the **Supplementary Specification, Glossary, Vision, and Business Rules**. These are all useful for requirements analysis, but secondary at this point.
The Use-Case Model may optionally include a **UML use case diagram** to show the **names** of **use cases and actors, and their relationships.** This gives a nice **context diagram** of a system and its environment. It also provides a quick way to list the use cases by name.
There is nothing object-oriented about use cases; we're not doing OO analysis when writing them. Use cases are a key requirements input to classic OOA/D.

30) What are the three kinds of Actors?

Definition: What are Three Kinds of Actors?
Actors are roles played not only by people, but by organizations, software, and machines.
There are three kinds of external actors in relation to the SuD:

1) **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.
   Why identify? To find user goals, which drive the use cases**.**
2) **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
   Why identify? To clarify external interfaces and protocols.
3) **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
   Why identify? To ensure that *all* necessary interests are identified and satisfied.

31) What are the common use case artifacts?

32) What are preconditions and postconditions?

Preconditions and Success Guarantees (Postconditions)

Preconditions state what *must always* be true before a scenario is begun 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, such as logging in, that has successfully completed.

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.

EXAMPLE :
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.

33) **How to find Use cases?**

Guideline: How to Find Use Cases

Use cases are defined to satisfy the goals of the primary actors. Hence, the basic procedure 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 actorsthose that have goals fulfilled through using services of the system.
3) Identify the goals for each primary actor.
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.

34) What does Use case Diagram represent? Give an example. Applying UML: Use Case Diagrams

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

Guideline

A simple use case diagram is drawn in conjunction with an actor-goal list.
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. A sample *partial* use case context diagram for the NextGen system is shown below.

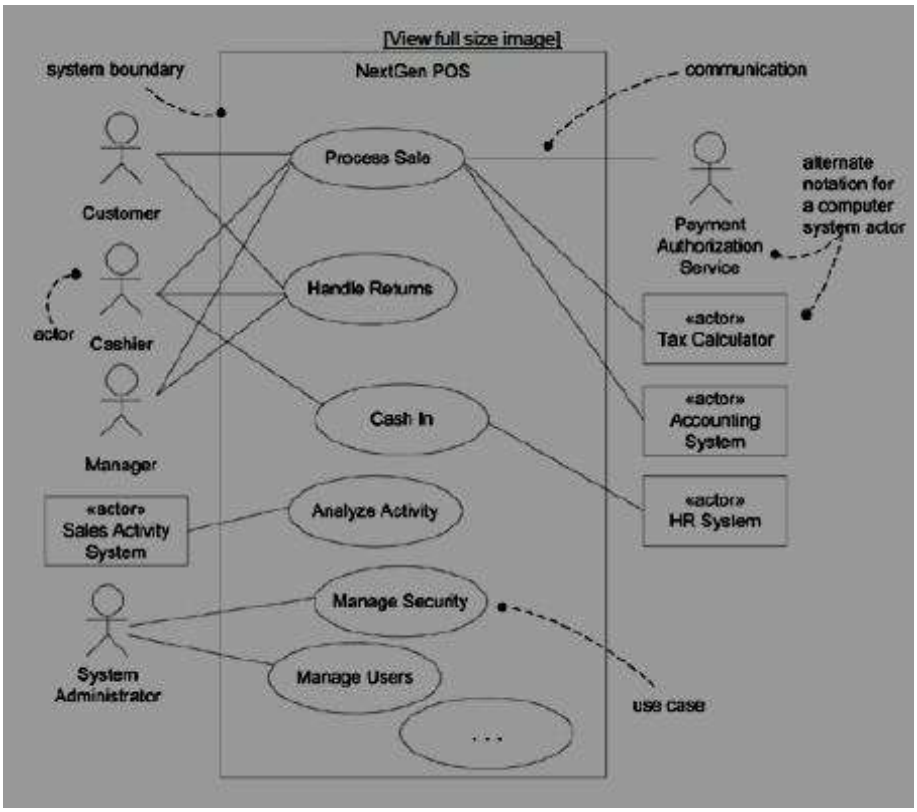


Figure 1.5. Partial use case context diagram.

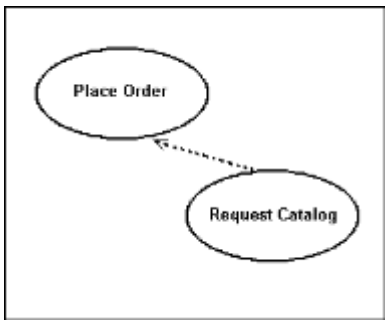
35) Define use case relationships a)Include b)Extend

Include Relationship

Use cases can be related to each other.

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.

The include relationship can be used for most use case relationship problems.



To summarize:

Factor out subfunction use cases and use the *Include* relationship when:

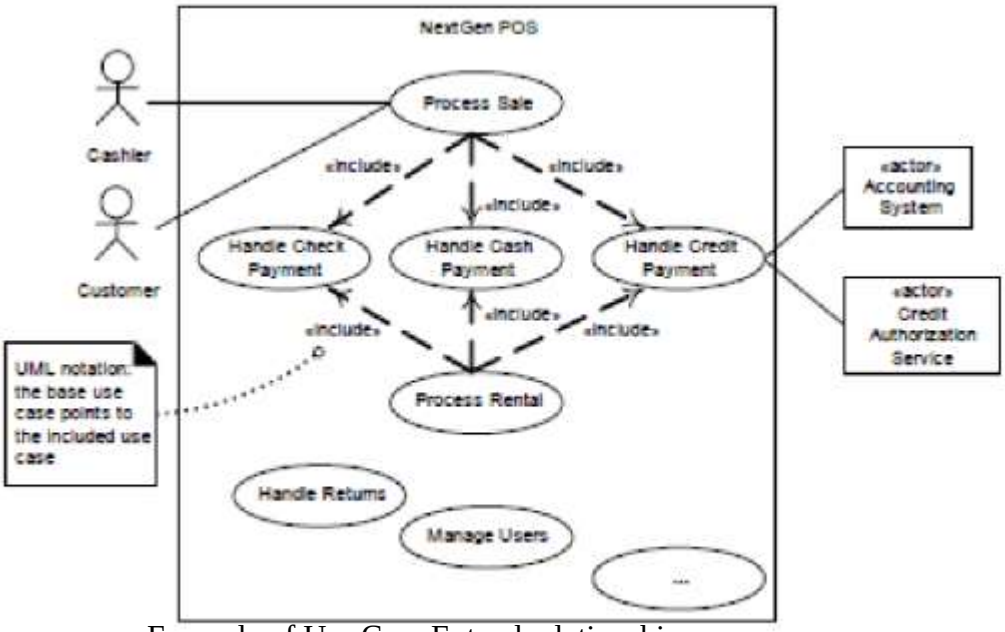
• They are duplicated in other use cases.

• A use case is *very* complex and long, and separating it into subunits aids comprehension.

Extend Relationship

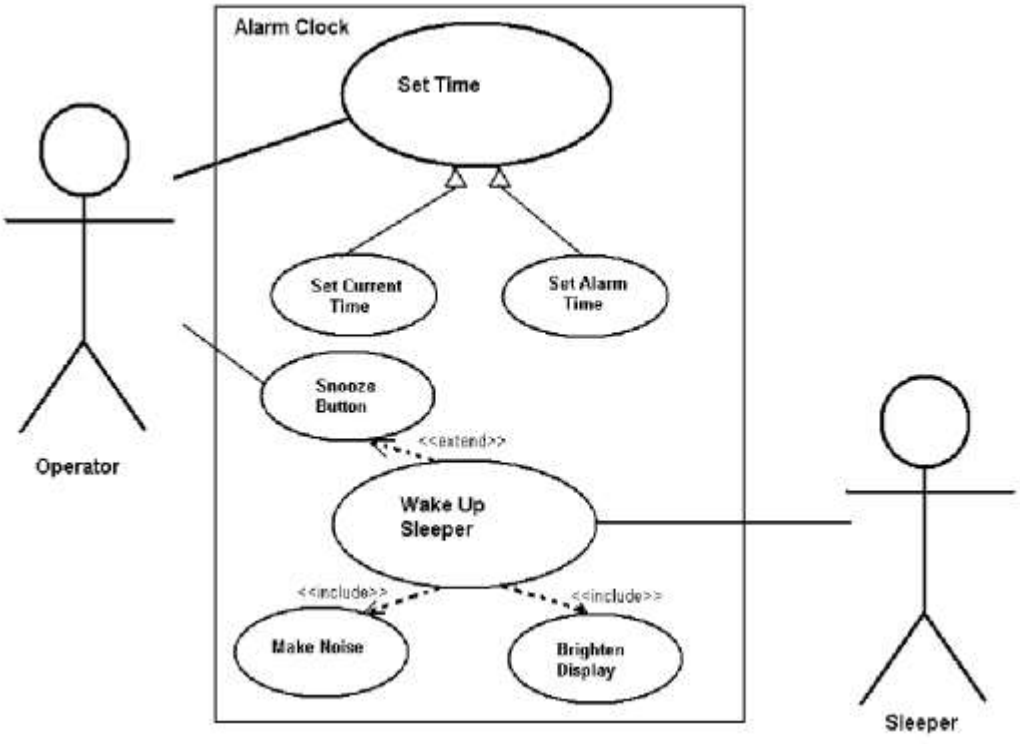
- Extend puts additional behavior in a use case that does not know about it.
- It is shown as a dotted line with an arrow point and labeled <<extend>>
- In this case, a customer can request a catalog when placing an order

The following diagram illustrates use case Include relationship



Example of Use Case Extend relationship

Figure 25.1 Use case include relationship in the Use-Case Model.



CS8592 Object Oriented Analysis and Design

V-Sem-CSE V-Sem-IT Anna University 2017 Regulations

UNIT II STATIC UML DIAGRAMS 9

Class Diagram— Elaboration – Domain Model – Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies – Aggregation and Composition – Relationship between sequence diagrams and use cases – When to use Class Diagrams

---

Question Bank
UNIT-II

1) Define a Domain Model.

Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that are considered noteworthy.

The result can be expressed in a **domain model** that shows the *noteworthy* domain concepts or objects. For example, a partial domain model is shown in Figure 1.2.

. It can be noted that a domain model is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain and it is also called a **conceptual object model**.

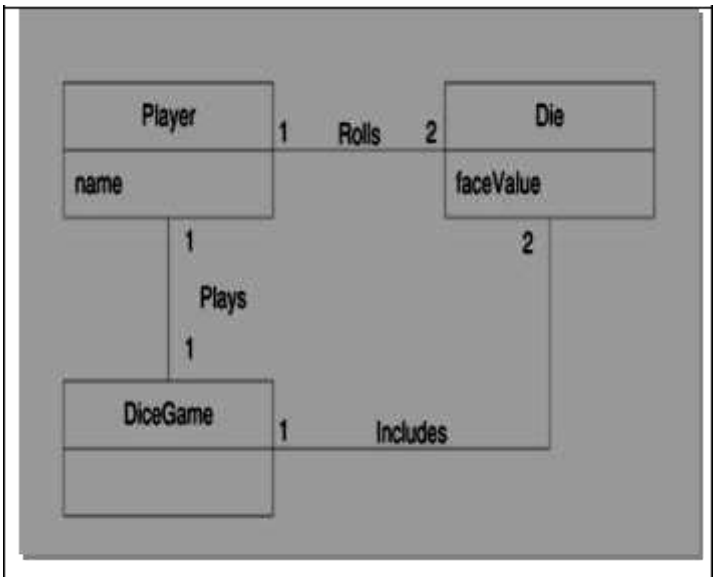


Figure 1.2. Partial domain model of the dice game

2) Define a) Conceptual Class,b) Software Class,c) Implementation Class.

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.
Implementation class - a class implemented in a specific OO language such as Java.

3) What are the different UP Phases?

An UP Project organizes work and iterations across four major phases :

1) **Inception** – approximate Vision,Business case,Scope,vague estimates
2) **Elaboration** – Refined BVision,iterative implementation of the core architecture,resolution of high risks,identification of most requirements and scope,more realistic estimates
3) **Construction** – Iterative implementation of the remaining lower risk and easier elements,and preparation for deployment
4) **Transition** – beta tests, deployment

4) What are the UP disciplines?

In the UP,an artifact is the general term for any work product :
Code,Web Graphics,Schema,Test Documents,diagrams,model and so on.
Some of the artifacts in the following Disciplines are :

a) Business Modeling – The Domain Model artifact,to visualize noteworthy concepts in the application domain
b) Requirements – The Use Case Model and Supplementary specification artifacts to capture functional and non-functional requirements
c) Design – The Design Model artifact,to design the software artifacts
d) Implementation – Programming and building the system,not deploying it

5) What is Inception?

Envision the product scope,vision,and business case.

Do the stakeholders have basic agreement on the vision of the project,and is it worth investing in serious investigations?

The purpose of inception stage is not to define all the requirements. The Up is not the waterfall and the first phase inception is not the time todo all requirements or create believable estimates or plans. That happens during elaboration.

6) How long is the Inception phase can be?

The intent of inception is to establish some initial common vision for the objectives of the project,determine if it is feasible,and decide if its is worth some serious investigation in elaboration. It can be brief.

7) List any five inception artifacts.

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. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) 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.
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.

8) Define Requirements.

Requirements are capabilities and conditions to which the system – and more broadly,the project must conform.

The UP promotes a set of best practices,one of which is to manage requirements.

In the context of changing and and unclear stakeholder's wishes – Managing requirements means – a systematic approach to finding,documenting,organizing,and tracking the changing requirements of a system.

A prime challenge of requirements analysis is to find,communicate,and remember(Towrite down) what is really needed,in a formthat clearly speaks to the client and development team members.

9) What are the types and categories of requirements?

In the UP, requirements are categorized according to the FURPS+ model [Grady92], 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, configurability.
- **Implementation** resource limitations, languages and tools, hardware, ...
- **Interface** constraints imposed by interfacing with external systems.
- **Operations** system management in its operational setting.
- **Packaging** for example, a physical box.
- **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 of not considering some important facet of the system.

Some of these requirements are collectively called the **quality attributes, quality requirements**, or the "- ilities" of a system. These include usability, reliability, performance, and supportability. In common usage, requirements are categorized as **functional** (behavioral) or **non-functional** (everything else); some dislike this broad generalization [BCK98], but it is very widely used.

10) What are the key requirement artifacts?

The Key requirements artifacts are :

Use-Case Model - A set of typical scenarios of using a system. There are primarily for functional
(behavioral) requirements.

Supplementary Specification - Basically, everything not in the use cases. This artifact is primarily for all non-functional requirements, such as performance or licensing. It is also the place to record functional features not expressed (or expressible) as use cases; for example, a report generation.

Glossary - In its simplest form, the Glossary defines noteworthy terms. It also encompasses the concept of the data dictionary, which records requirements related to data, such as validation rules, acceptable values, and so forth. The Glossary can detail any

element: an attribute of an object, a parameter of an operation call, a report layout, and so forth.

Vision - Summarizes high-level requirements that are elaborated in the Use-Case Model and Supplementary Specification, and summarizes the business case for the project. A short executive overview document for quickly learning the project's big ideas. **Business Rules** - Business rules (also called Domain Rules) typically describe requirements or policies that transcend one software projectthey are required in the domain or business, and many applications may need to conform to them. An excellent example is government tax laws. Domain rule details *may* be recorded in the Supplementary Specification, but because they are usually more enduring and applicable than for one software project, placing them in a central Business Rules artifact (shared by all analysts of the company) makes for better reuse of the analysis effort.

11) What are Use Cases?

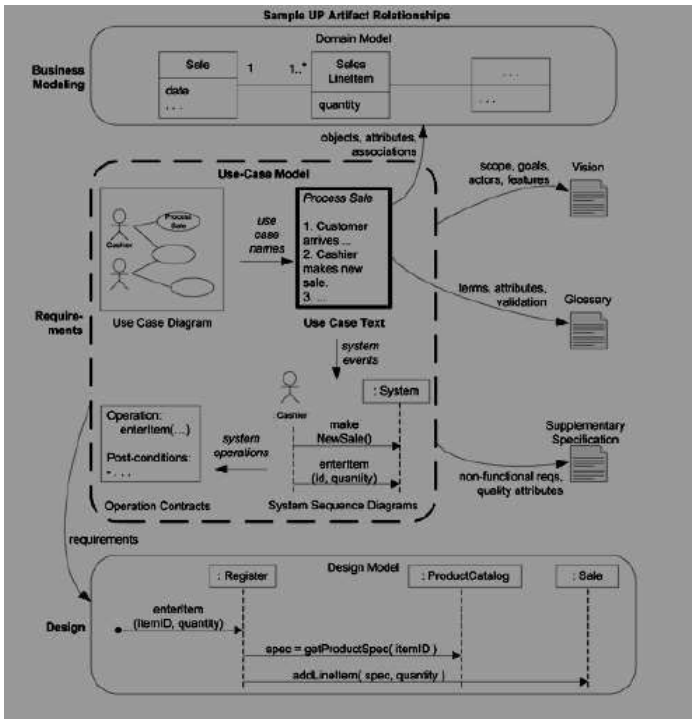
Informally, **use cases** are ***text stories*** of some **actor** using a **system** to meet **goals**.

Use Case Example :

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 i**nventory**. The customer receives a **receipt** from the system and then leaves with the items.

Notice that ***use cases are not diagrams, they are text.*** Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

Use cases often need to be more detailed or structured than this example, but the essence is **discovering and recording functional requirements by writing stories** of using a system to fulfill user goals; that is, *cases of use*.



12) What is Use-case Modeling?

Use-Case Model is the set of all written use cases; it is a model of the system's functionality and environment. **Use cases** are **text documents, not diagrams**, and use-case modeling is primarily an act of **writing text**, not **drawing diagrams**.
The Use-Case Model is not the only requirement artifact in the UP. There are also the **Supplementary Specification, Glossary, Vision, and Business Rules**. These are all useful for requirements analysis, but secondary at this point.
The Use-Case Model may optionally include a **UML use case diagram** to show the **names** of **use cases and actors, and their relationships.** This gives a nice **context diagram** of a system and its environment. It also provides a quick way to list the use cases by name.
There is nothing object-oriented about use cases; we're not doing OO analysis when writing them. Use cases are a key requirements input to classic OOA/D.

13) What are the three kinds of Actors?

Definition: What are Three Kinds of Actors?
Actors are roles played not only by people, but by organizations, software, and machines. There are three kinds of external actors in relation to the SuD:

1) **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.
   Why identify? To find user goals, which drive the use cases**.**
2) **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
   Why identify? To clarify external interfaces and protocols.
3) **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
   Why identify? To ensure that *all* necessary interests are identified and satisfied.

14) What are preconditions and postconditions?

Preconditions and Success Guarantees (Postconditions)

Preconditions state what *must always* be true before a scenario is begun 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, such as logging in, that has successfully completed.
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. EXAMPLE :

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.

15) How to find Use cases? Guideline:

How to Find Use Cases

Use cases are defined to satisfy the goals of the primary actors. Hence, the basic procedure 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 actorsthose that have goals fulfilled through using services of the system.
3) Identify the goals for each primary actor.
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.

16) What does Use case Diagram represent? Give an example.

Applying UML: Use Case Diagrams

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

Guideline

A simple use case diagram is drawn in conjunction with an actor-goal list.

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. A sample *partial* use case context diagram for the NextGen system is shown below.

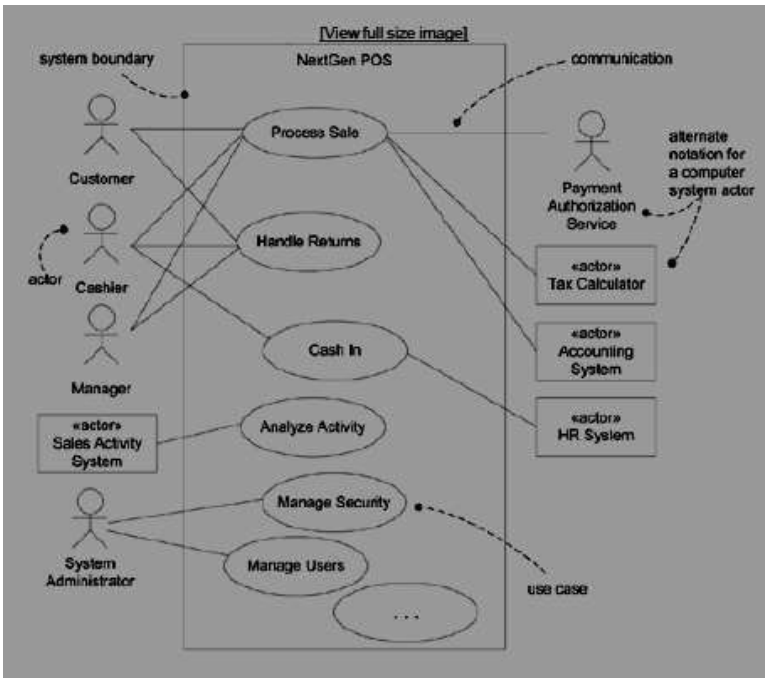


Figure 1.5. Partial use case context diagram.

17) Define use case relationships a)Include b) Extend c) Generalization

Include Relationship

Use cases can be related to each other.

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. The include relationship can be used for most use case relationship problems.

To summarize:

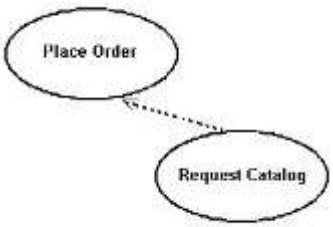
Factor out subfunction use cases and use the *Include* relationship when:

• They are duplicated in other use cases.

• A use case is *very* complex and long, and separating it into subunits aids comprehension.

Extend Relationship

- Extend puts additional behavior in a use case that does not know about it.
- It is shown as a dotted line with an arrow point and labeled <<extend>>
- In this case, a customer can request a catalog when placing an order



The following diagram illustrates use case Include relationship

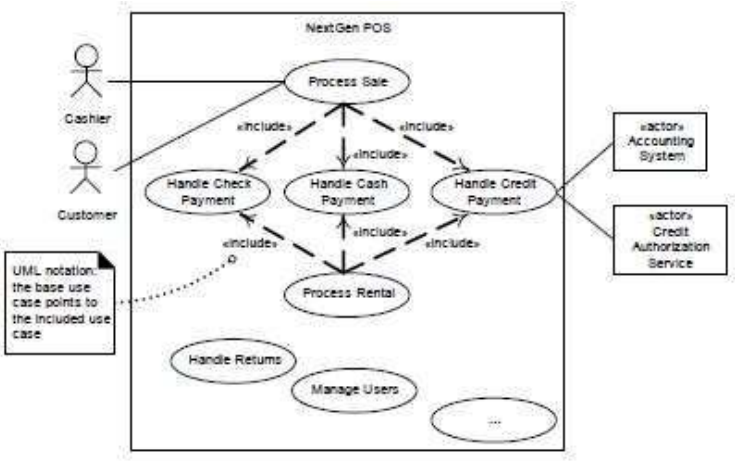
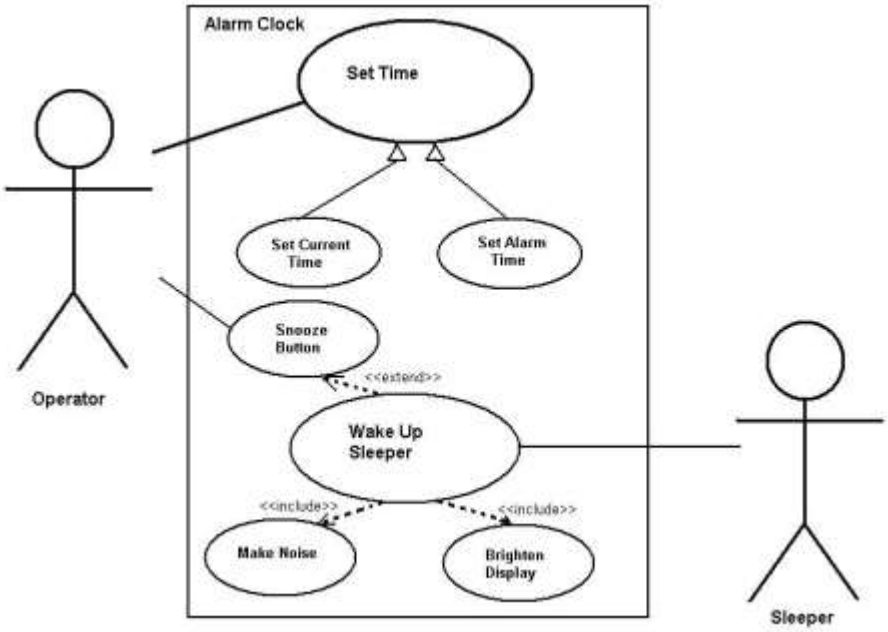


Figure 25.1 Use case include relationship in the Use-Case Model.

Example of Use Case Extend relationship



18) What is Elaboration?

- Elaboration often consists of two or more iterations(2 to 6 weeks duration)
- Each iteration is time-boxed(i.e. End Date fixed)
- Elaboration is not Design Phase(i.e. the model is not fully developed)
- Also it is not throw away Prototype;rather the code and design are production quality

In other words,Elaboration is the initial series of iterations during which

- The core ,risky software architecture is programmed and tested
- The majority of requirements are discovered and stabilized
- The major risks are mitigated or reduced

19) Define Elaboration.

Build the core architecture,resolve the high risk elements,Define most Requirements,and Estimate the overall schedule and resources

20) Define the first iteration is elaboration phase.

Iteration-1 of Elaboration Phase emphasizes fundamental and common OOA/D skills used in building OO Systems.

Example – NextGen POS

:

Iteration 1 Requirements

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

- 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.
- Nothing fancy or complex is handled, just a simple happy path scenario, and the design and implementation to support it.
- 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 ,database,are done(Not in detail)
- Subsequent iterations will grow on this foundation.

21) What happens in inception?

Inception is a short step to elaboration. It determines basic feasibility,risk and scope,to decide if the project is worth more serious investigation

22) What are the likely activities and artifacts in inception?

- A short requirement workshop
- Most actors,goals and Use Cases named.
- Most use cases written in brief format; 1—20% of use cases written in fully dressed format
- Most influential and risky quality requirements identified
- Supplementary specification written(Version One)
- Risk List
- Technical proof-of-Concept,User Interface-Oriented Prototypes
- Decision on components : to buy/build/reuse taken(Eg. To buy Tax calculation package
- High Level candidate Architecture made(not a detailed,final or correct one made)
- Used as starting point of investigation(Eg. A Java Client side application with no Application Server, and no Oracle for Database

23) What artifacts may start in Inception?

Slno	Artifact	Features
01	Domain Model	Visualization of Domain concepts
02	Design Model	• Set of Diagrams describing the Logical Design(Software Class Diagrams,Object Interaction Diagrams,and Package Diagrams)
03	Software Architecture Document	• A Learning Aid • Key Architectural Issues & their Resolution in summary form • Summary of outstanding Design Ideas & its motivation
04	Data Model	• Database Schemas • Mapping between Objects & Non-Object representations

05	Use Case Story Boards,UI Prototypes	➢ Describe UI,Navigation Paths,Usability Models

24) What is a Domain Model?

A domain Model is the most important and classic model in OO Analysis.

- It illustrates noteworthy Concepts in a Domain.
- Source of Inspiration for designing some software objects(which become inputs to several artifacts)

25) Give an example of domain model with UML class Diagram notation.

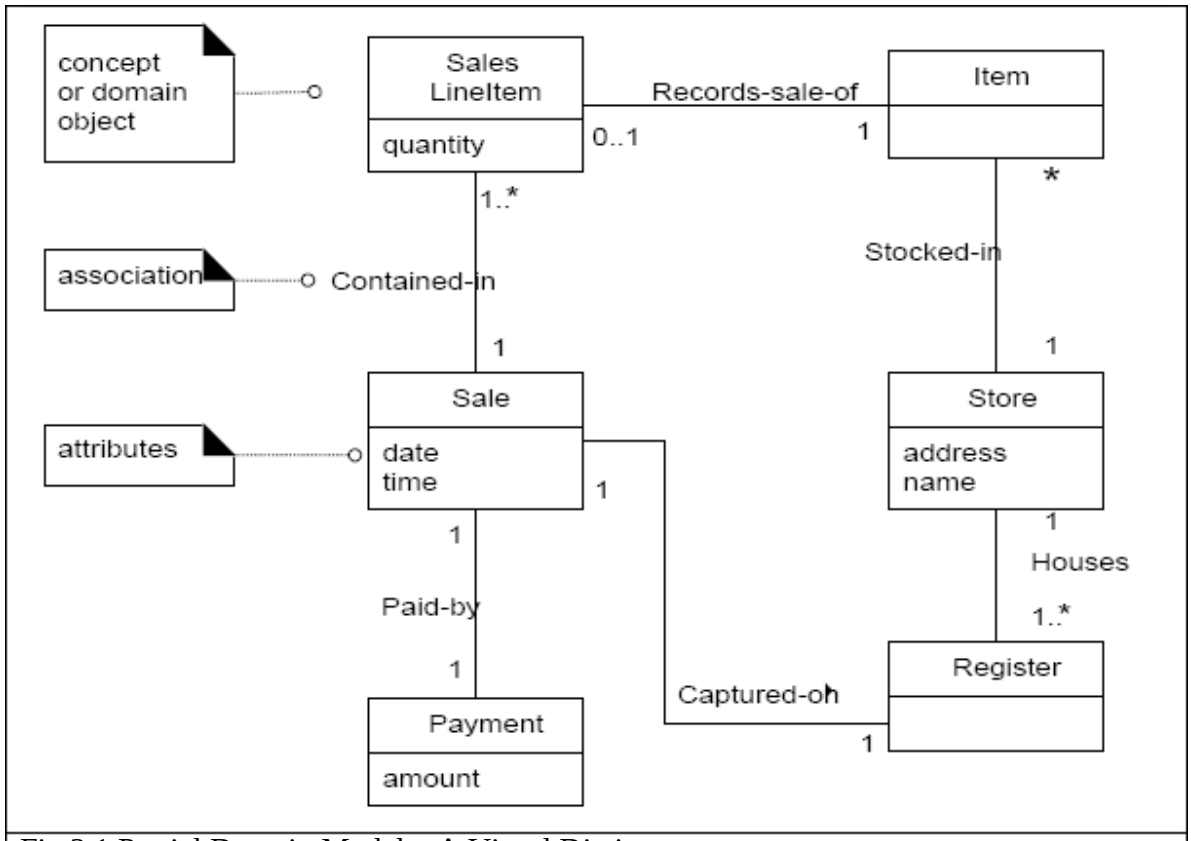


Fig 2.1 Partial Domain Model – A Visual Dictionary

Explanation

A partial Domain Model drawn with UML class diagram notation -:

- **Conceptual classes of** *Payment* and *Sale* are significant in this domain.
- **A** *payment* is related to *Sale* which is meaningful to note.
- The *Sale* has date and time (Attributes we care about)

26) What are the criteria in planning the next iteration during elaboration phase?

Risk	Technical complexity and factors such as uncertainity of effort or usability
Coverage	Major parts of the system are at least touched on early iterations
Criticality	Functions that client consider of high business value

27) Why call a domain model a visual dictionary?

A domain model is a visual dictionary of

- the noteworthy abstractions
- domain vocabulary,and
- information content

A domain model visualizes and relates words or concepts in the domain. It also shows an abstraction of the conceptual classes and shows how they relate to each other.

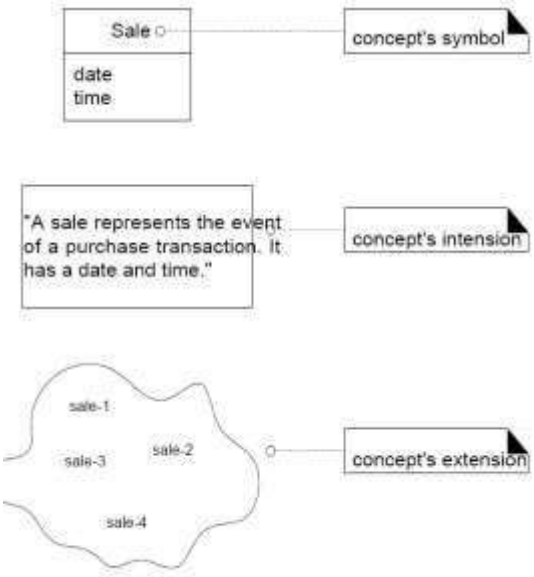
28) What are conceptual classes?

Conceptual Classes

Informally, a conceptual class is an idea, thing, or object.
More formally, 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.

For example, consider the conceptual class for the event of a purchase transaction. We may choose to name it by the symbol *Sale.* The intension of a *Sale* may state that it "represents the event of a purchase transaction, and has a date and time." The extension *of Sale* is all the examples of sales; in other words, the set of all sales.



A conceptual class has a symbol, intension, and extension

29) Are domain and Data Models are the same thing?

A domain model is not a **data model** (which by definition shows persistent data to be stored somewhere)

30) How a domain model is created?

Steps involved in creating a domain model :

- Find the conceptual classes
- Draw them as classes in a UML class diagram
- Add associations and attributes

31) What are the three strategies to find conceptual classes?

a) Reuse or modify existing models(First,Best,and easiest approach). There are published ,well crafted domain models and data models for many common domains ,such as inventory,finance,health,banking,and so forth.
b) Use a Category List
c) Identify noun phrases.

32) What is Conceptual Class Category List?

We can kick start the creation of a domain model by making a list of candidate conceptual classes. The following table contains many common categories(which are usually worth considering as meeting business information needs)

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*

Conceptual Class Category	Examples
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.

33) Explain the method of finding conceptual classes using Noun Phrase Identification.

Noun phrase identification is another useful technique which is based on **linguistic analysis.**

- It is based on identifying the **nouns** and **noun phrases** in textual descriptions of a domain, which can be considered 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.

For example,**Noun phrases** are identified (shown in bold) from **Process Sale Use case** as per the text description below :

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.

34) Explain with an example,the method of finding and drawing conceptual classes.

From the category list and known phrase analysis,a list is generated of candidate conceptual classes for the domain. The list is constrained to the requirements and a simplified version as for iteration-1. As an example the following are identified list of conceptal classes for Process Sale scenario :

Sale	Cashier
CashPayment	Customer
SalesLineItem	Store
Item	ProductDescription
Register	ProductCatalog
Ledger	

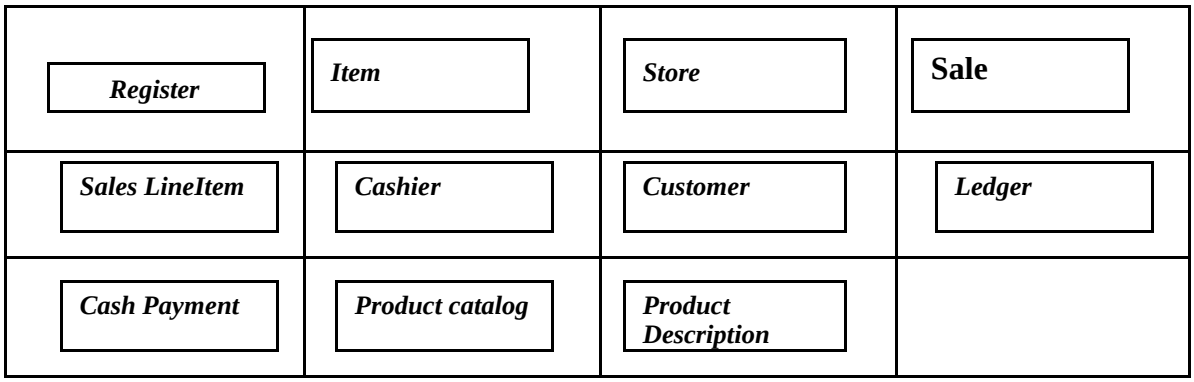


Fig. Initial POS Domain Model

35) What are Description classes? Give examples.

A Description class contains information that descrkibes something else. For example,a ProductDescription that records the price,picture,and text descriptions of an item.This was first named the ***Item-Descriptor pattern.***

The need for description classes

- An item instance represents a physical item in a store;it may have a serial number
- An item has description,price,and itemID
- A Product Description class records information about items
- Even if all inventoried items are sold and corresponding item software instances are deleted,the Product Description still remains.

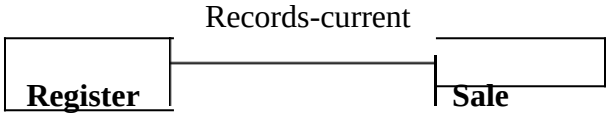
36) **What is Association? Explain with an example**.

An association is a relationship between classes(more precisely,instances of these classes) that indicates some meaningful and interesting connections.

37) Explain Association using UML notation.

UML Definition:

Associations are defined as semantic relationship between two or more classifiers that involve connections among their instances



38) What is multiplicity?

Multiplicity defines how many instances of class A can be associated with one instance of a class B.

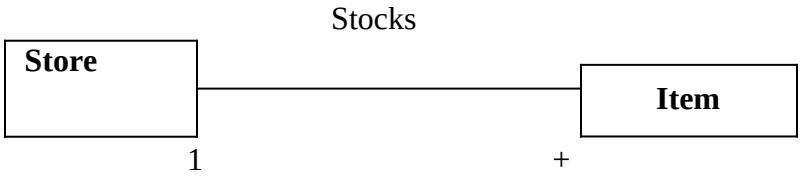


Fig. Multiplicity of an Association

39) Give the method of finding Associations using Common Association List.

Category	Examples
A is a transaction related to another transaction B	CashPayment – Sale Cancellation - Reservation
A is a line item of a transaction B	SalesLineItem - Sale
A is a product or service for a transaction B(or line item)	Item – SalesLineItem (or Sale) Flight - Reservation
A is a role related to a transaction B	Customer – Payment Passenger - Ticket
A is a physical or logical part of B	Drawer – Register Square - Board Seat - AirPlane
A is physically or logically contained in /or B	Register – Store Item-Shelf Square-Board Passenger - Airline
A is a description for B	ProductDescription – Item FlightDescription - Flight
A is known /logged/recorded/reported/captured in B	Sale – Register Piece – Square Reservation - Flightmanifest
A is a member of B	Cashier – Store Player – monnopolyGame Pilot - Airline
A is an organizational Subunit of B	Department – Store

	Maintenance - Airline
A uses or manages or owns B	Cashier – Register Player – Piece Pilot - Airplane
A is nest to B	SalesLineItem – SalesLineItem Square – Square City - City

40) Define an attribute. Explain with an example using UML notation. An attribute is a logical data value of an object.

Example

Sale needs a dateTime attribute
Store needs a name and address
Cahier needs an ID

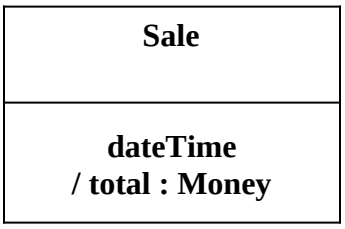
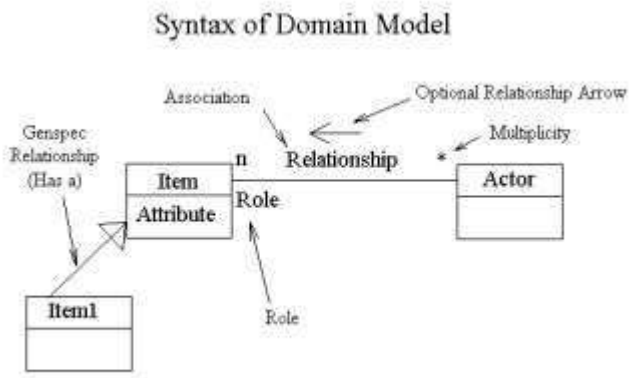
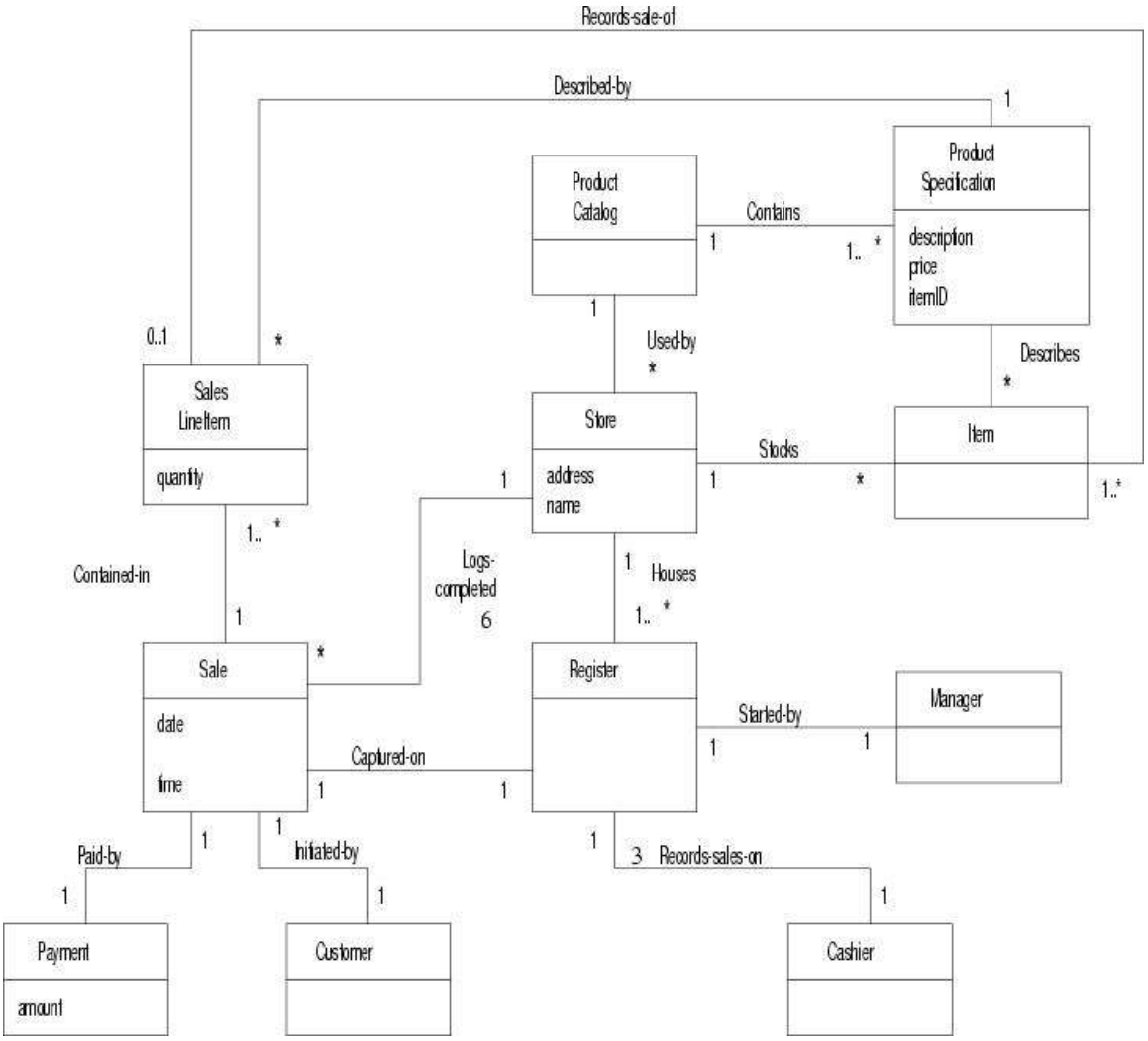


Fig**. Class and attributes**

41) How attributes are used in Domain Models? Explain using examples.





42) How domain model is further refined after the first iteration?

Generalization and specializations are fundamental concepts in domain modeling. Conceptual class hierarchies are often inspiration for Software class hierarchies that exploits inheritance and reduce duplication of code.

Packages are a way to organize large domain models into smaller units.

Domain model is further refined with Generalization,Specialization,Association classes,Time intervals,Composition and packages,usage of subclasses

43) How Domain Model is incrementally developed?

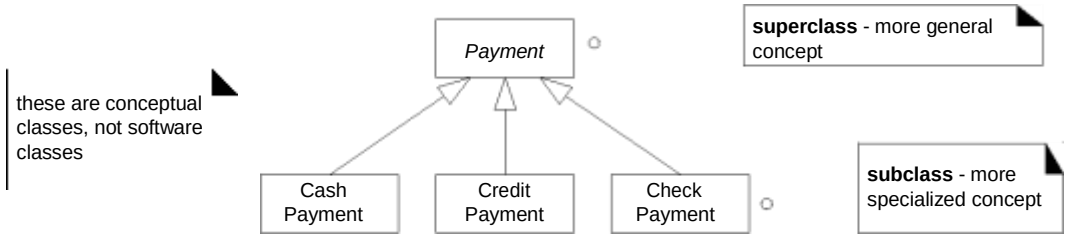
The domain model is incrementally developed by considering concepts in the requirements for this iteration.

44) Explain Generalization-Specialization hierarchy with an example.

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 illustrated in class hierarchies.

Identifying a superclass and subclasses leads to economy of expression,improved comprehension and a reduction in repeated information.

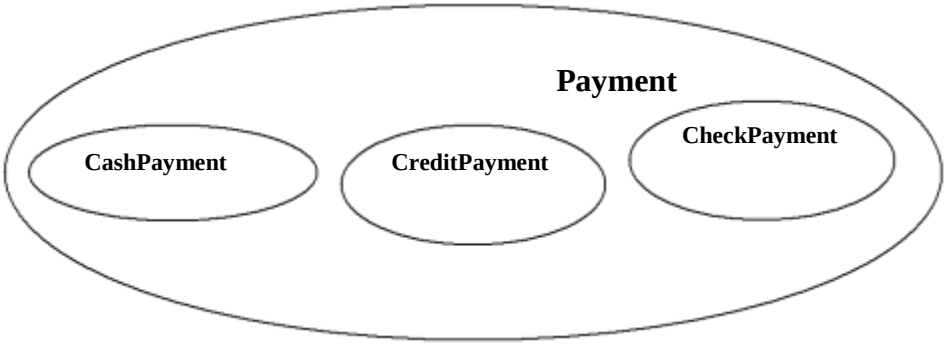
45) Explain Class Hierarchies with an example



- Identify superclasses and subclasses when they help us understand concepts in more general, abstract terms and reduce repeated information.
- Expand class hierarchy relevant to the current iteration and show them in the Domain Model.

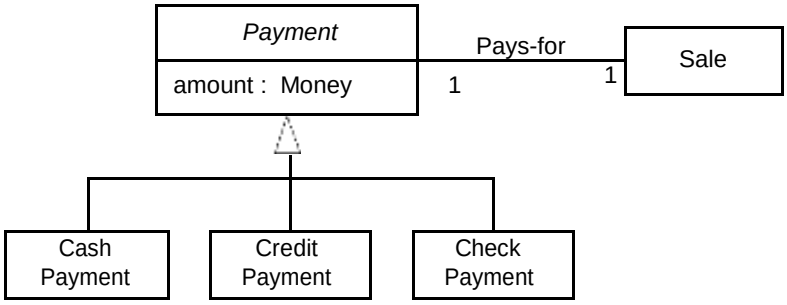
46) How Conceptual subclass and Super Classes are related in terms of set membership? Explain using Venn Diagram.

Conceptual subclasses and superclasses are related interms of set membership. By definition,all members of a conceptual subclass set are members of their superclass set. For example,interms of set membership,all instances of the set *CreditPayment* are also members of the set *Payment.* This is shown in the venn diagram shown below.



47) Explain Subclass Conformance.

When a class hierarchy is created,statements about superclasses that apply to subclasses are made. For example,the following fig. states that all P***ayments*** have an ***amount*** and are associated with a ***Sale***.



All Payment subclasses must **conform** to having an amount and paying for a Sale.

48) What is 100% Rule? Or What is the rule of Conformance to Superclass Definition?

100% of the conceptual Superclass's definition should be applicable to the subclass. The subclass must conform to 100% of the Superclass's:

1) **attributes**
2) **associations**

49) What are the strong motivation to partition a conceptual class with subclasses?

The following are the strong motivations to partition a class into 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

50) Give examples of motivations to partition a Conceptual Class into Subclasses

Conceptual Subclass Motivation	Examples
The subclass has additional attribute of interest	Payments – NA Library – Book,Subclass of LoanableResource,has an ISBN attribute
The Subclass has additional associations of interest	Payments – CreditPayment,subclass of Payment,is associated with CreditCard Library –video,subclass of LoanableResource,is associated with Director.
The subclass concept is operated upon,handled,reacted to,or manipulated differently than the super class or other subclasses,in ways that are of interest	Payments – CreditPayment,subclass of payment,is handled differently than than other kinds of payments in how it is authorized. Library-Software,subclass of LoanableResource,requires a depsosit before it may be loaned.
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	Payments – not applicable. Library – not applicable. Market Research – MaleHuman,subclass of Human,behaves differently than FwemaleHuman with respect to shopping habits.

51) Explain how a Conceptual Super Class is defined and when?

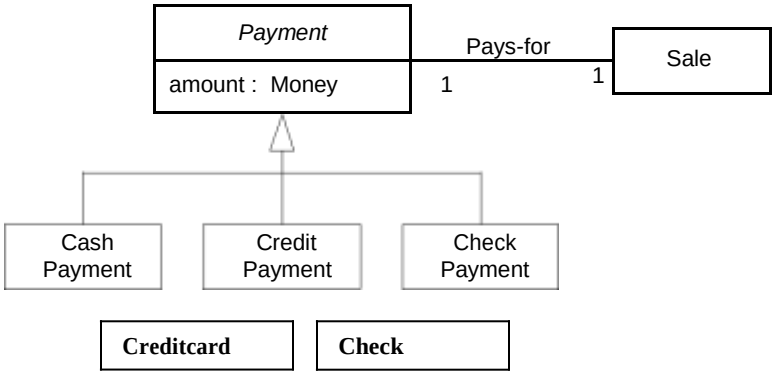
Generalization into a common Superclass is made when commonality is identified among potential subclasses.

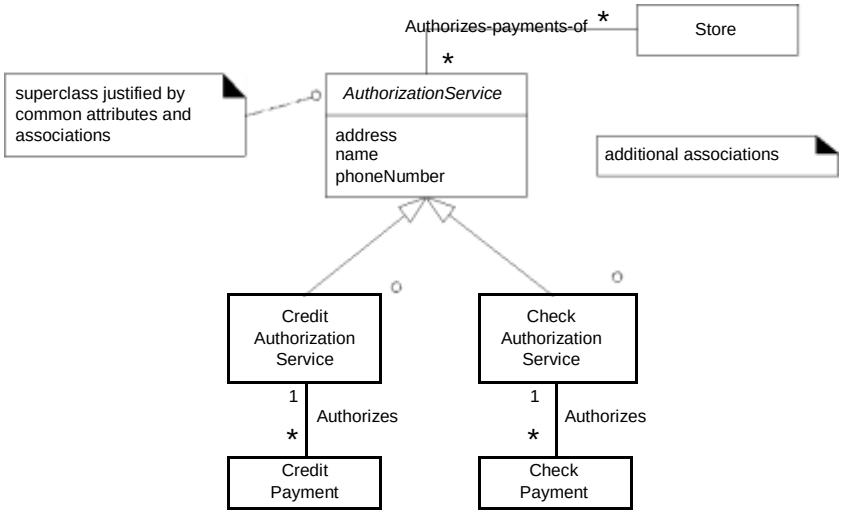
52) What are the guide lines followed in defining a Super Class?

GuideLine
Create a superclass in a generalization relationship to subclasses when :

- The potential conceptual subclasses repreent variations of a similar concept.
- The subclasses will confirm to the 100% and Is-a rules.
- All subclasses have the same attribute that can be factored out and expressed in the superclass
- All subclasses have the same association that can be factored out and related to the super class.

53) Explain in detail an example how a superclass-subclass hierarchies are defined and give justification.

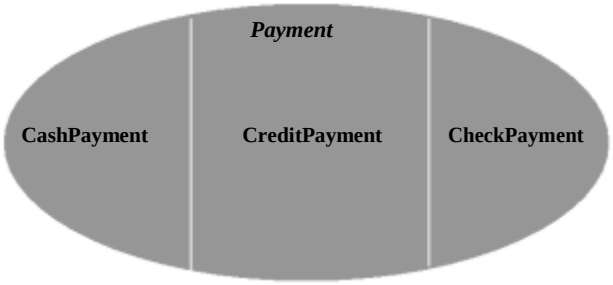




54) What are Abstract Conceptual Classes?

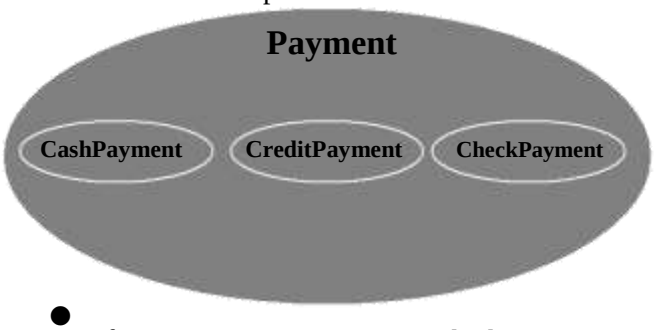
If every member of a class C must also be a member of a subclass C is called an **abstract conceptual class**.

For example,assume that every *Payment* instance must more specifically be an instance of the subclass *CreditPayment,CashPayment,or CheckPayment*. Since every *Payment* member is also a member of a subclass,*Payment* is an abstract conceptual class by definition.



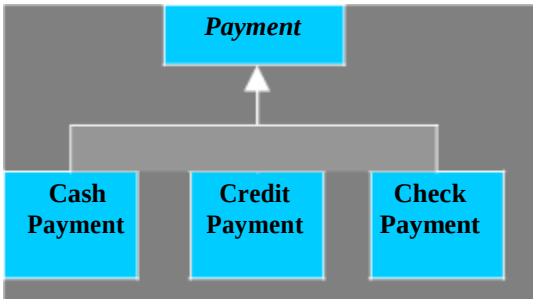
Abstract Classes

- Def.: If every instance of a class C must also be an instance of a subclass, then C is called an abstract conceptual class.

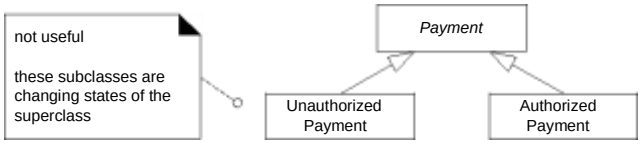


- If a Payment instance exists which is not a member of a subclass, then Payment is not abstract – it is concrete.

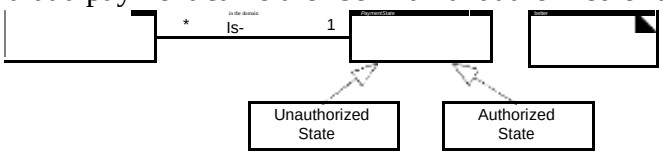
Abstract Classes



55) Explain with an example modeling of changing states.



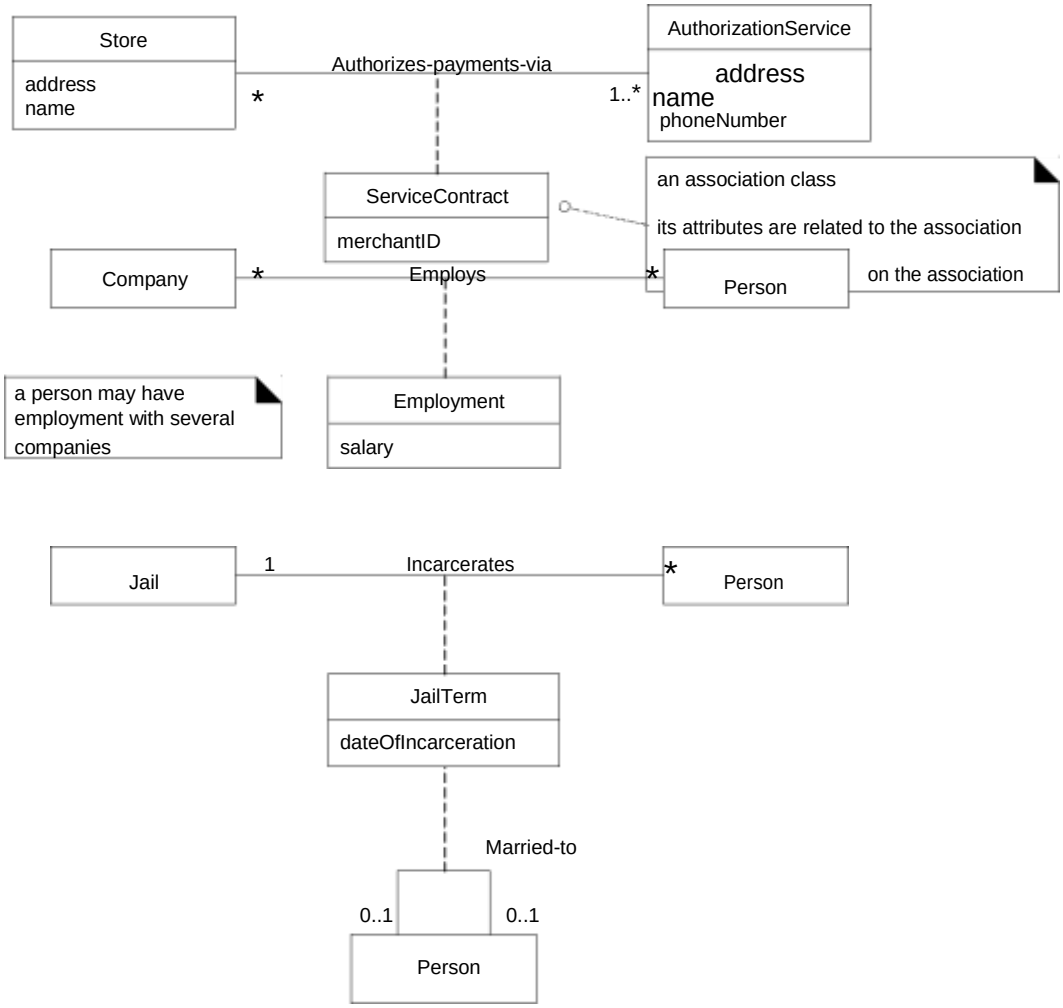
Assume that a payment can either be in an unauthorized or authorized state and it is Guideline.



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 in of a concept in the domain model,show the states in the state diagrams instead.

56) Explain with example Assocition classes.



57) Define a) Aggregation b) Composition.

Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships

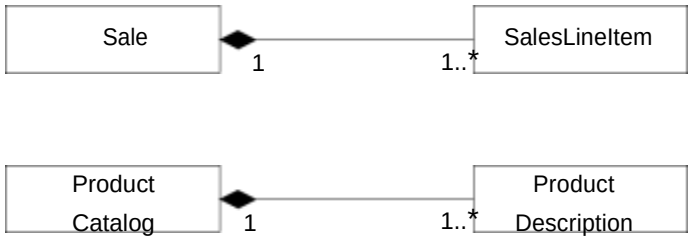
58) What are the guidelines for identifying composition?

Compositions is also known as composite aggregation,is a strong kind of whole-part aggregation and is useful to show in some models. A composition relationship implies that 1) an instance of the part(such as a *Square*) belongs to only one composite instance(such as a *Board*) at a time, 2) the part must always belong to a composite,and 3) the composite is responsible for the creation and deletion of its parts – either by itself creating/deleting the parts,or by collaborating

with other objects. Related to this constraint is that if the composite is destroyed,its parts must either be destroyed,or attached to another composite – no free floating Fingers allowed.

Example

In the POS domain,the *SalesLineItems* may be considered as part of a composite *Sale*; In general,transaction line items are viewed as parts of an aggregate transaction. In addition to conformance to this pattern,there is a create/delete dependency of the line items on the Sale – their lifetime is bound within the lifetime of the *Sale*. Figure. **Aggregation in the point-of-sale application**



.

59) What are the benefits of showing Composition?

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.

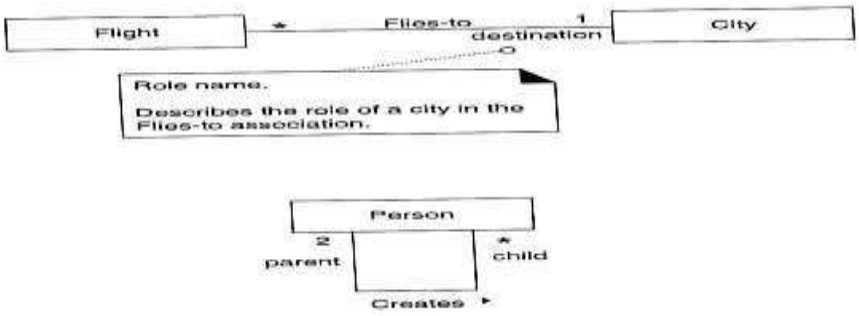
- During design work,this has an impact on the create-delete dependencies between the whole and part software classes and database elements(in terms of referential integrity and cascading delete paths)
- It consists in the identification of a creator(the composite) using the GRASP Creator Pattern.
- Operations – such as copy and delete – applied to the whole often propagate to the parts.

60) What are Association Role Names?

Each end of an association is a role,which has various properties,such as :

1. Name
2. Multiplicity

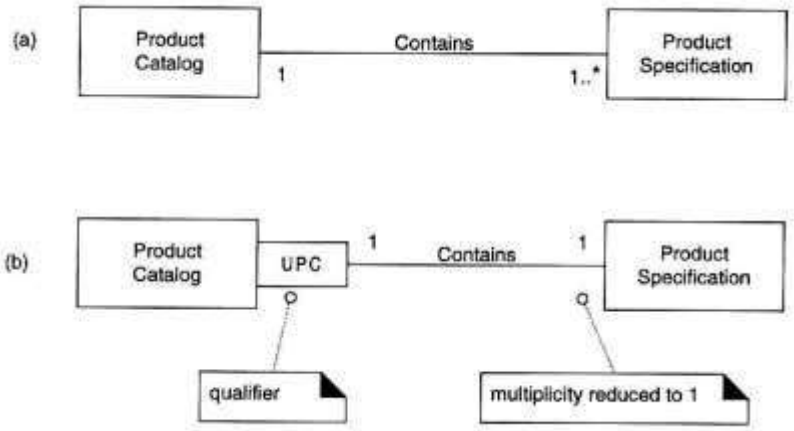
A role name identifies an end of an association and ideally describes the role played by objects in the association



61) What is qualified Association?

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,*ProductDescriptions* may be distinguished in a ProductCatalog by their itemID,as illustrated in the figure below.



- As contrasted in previous figure (a) vs. (b), qualification reduces the multiplicity at the far end from the qualifier, usually down from many to one.
- Qualifiers do not usually add compelling useful new information, and we can fall into the trap of ―design – think‖.
- However,they can sharpen understanding about the domain.

CS8592 OBJECT ORIENTED ANALYSIS AND DESIGN

VI-Sem-IT 2017-Regulations

UNIT III DYNAMIC AND IMPLEMENTATION UML DIAGRAMS 9

Dynamic Diagrams – UML interaction diagrams – System sequence diagram – Collaboration diagram – When to use Communication Diagrams – State machine diagram and Modelling –When to use State Diagrams – Activity diagram – When to use activity diagrams Implementation Diagrams – UML package diagram – When to use package diagrams – Component and Deployment Diagrams – When to use Component and Deployment diagrams

---

Question Bank – 2 marks

UNIT-III

1) What are UML state machine diagrams?

- *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.

2) Define events,states,and transitions?

Events, States, and Transitions

- An **event** is a significant or noteworthy occurrence.

For example:
A telephone receiver is taken off the hook.

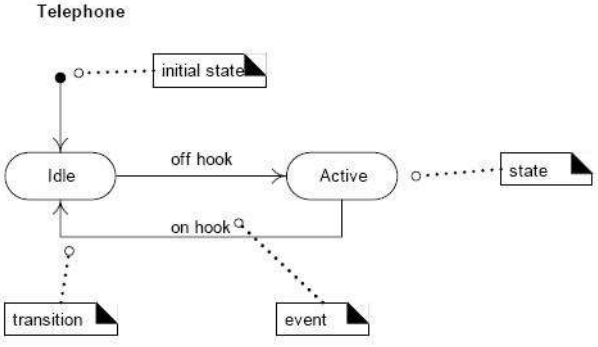
- **A state** is the condition of an object at a moment in time—the time between events.

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.

3) Explain state machine diagram with an example.



A statechart diagram shows the lifecycle of an object: what events it experiences, its transitions, and the states it is in between these events.

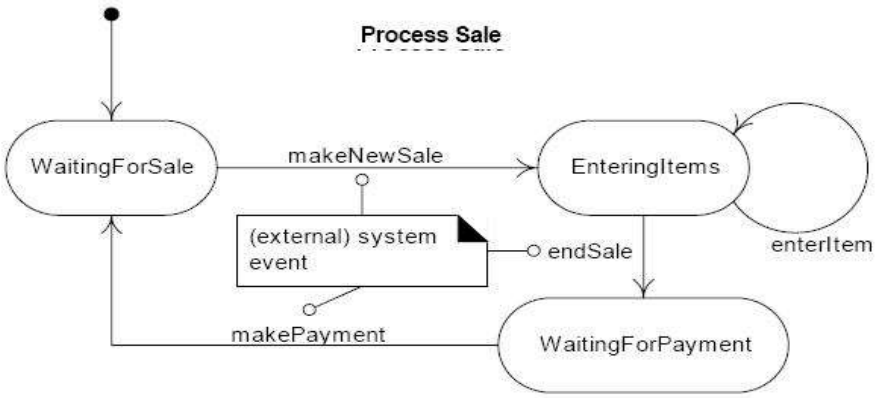


Figure 29.2 Use case statechart diagram for *Process Sale.*

4) What is transition action?

   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.

5) What is guard condition?

   A transition may also have a conditional guard—or boolean test. The transition only occurs if the test passes.

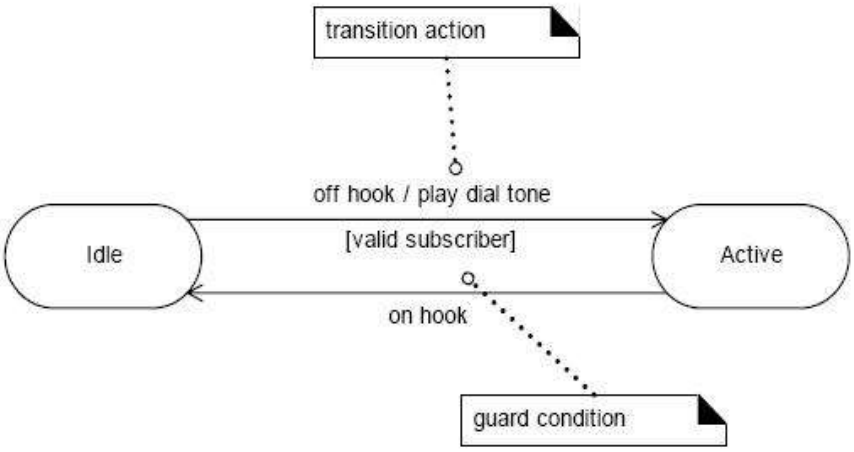


Figure 29.4 Transition action and guard notation.

6) What are nested states?

3

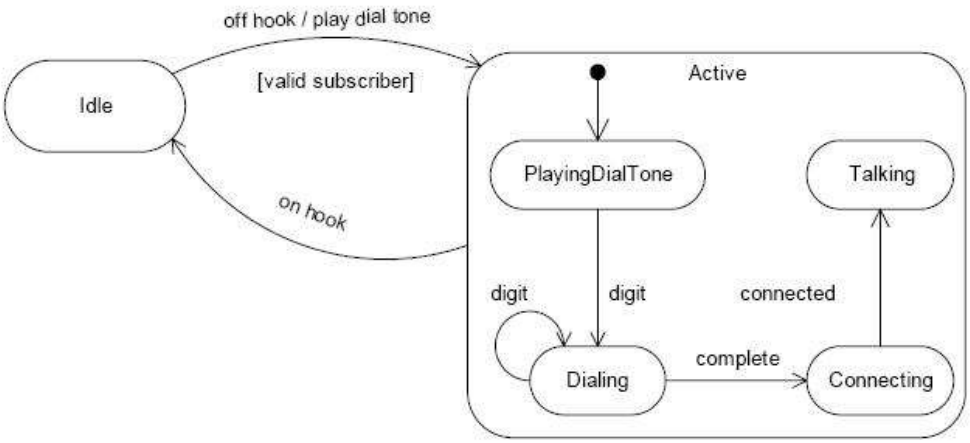


Figure 29.5 Nested states.

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.

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.

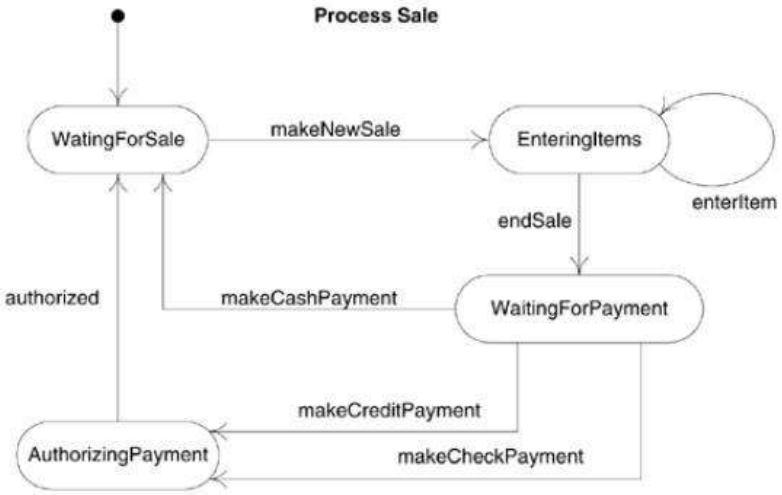
7) Explain with a diagram a sample state machine for legal sequence of use case operations.

The following state machine diagram shows legal sequencing of a use case operation :

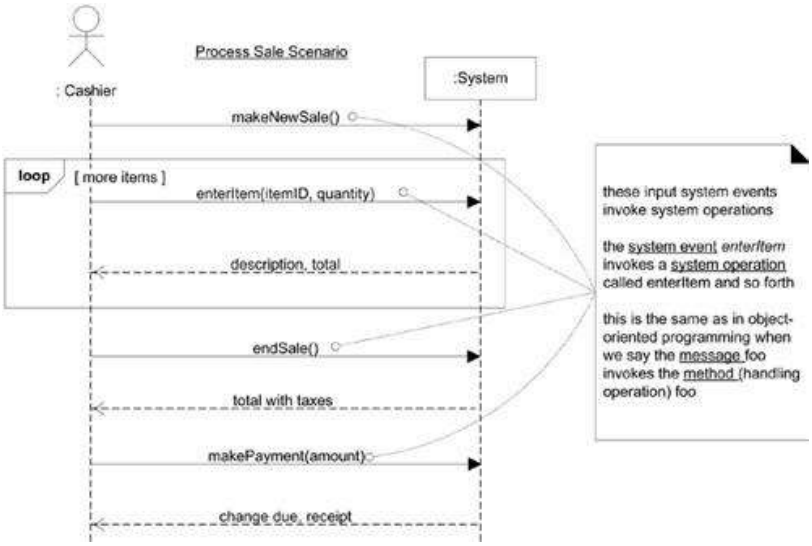
Example application :

Process Sale use case.

Figure 29.5. A sample state machine for legal sequence of use case operations.



8) What is a system operation?



Operation contracts may be defined for **system operations**operations that the system as a black box component offers in its public interface. System operations can be identified while sketching SSDs, as in Figure abobe. To be more precise, the SSDs show **system events**events or I/O messages relative to the system. Input system events imply the system has system operations to handle the events, just as an OO message (a kind of event or signal) is handled by an OO method (a kind of operation).

The entire set of system operations, across all use cases, defines the public **system interface**, viewing the system as a single component or class. In the UML, the system as a whole can be represented as one object of a class named (for xample) *System*.

9) What are operation contracts?

Use cases or system features are the main ways in the UP to describe system behavior, and are usually sufficient. Sometimes a more detailed or precise description of system behavior has value. Operation contracts

use a pre- and post-condition form to describe detailed changes to objects in a domain model, as the result of a system operation. A domain model is the most common OOA model, but operation contracts and state models (introduced on p. 485) can also be useful OOA-related artifacts.

Operation contracts may be considered part of the UP Use-Case Model because they provide more analysis detail on the effect of the system operations implied in the use cases.

10) Explain operation contract with an example.

Example

Here's an operation contract for the *enterItem* system operation. The critical element is the *postconditions*; the
other parts are useful but less important.

Contract CO2: enterItem
Operation: enterItem(itemID: ItemID, quantity: integer)
Cross References: Use Cases: Process Sale
Preconditions: There is a sale underway.
Postconditions:
- A SalesLineItem instance sli was created *(instance creation).*
- sli was associated with the current Sale *(association formed).*
- sli.quantity became quantity *(attribute modification).*
- sli was associated with a ProductDescription, based on itemID match *(association formed).*

The categorizations such as "*(instance creation)*" are a learning aid, not properly part of the contract.

11) What are post conditions?

- Describe changes in the state of objects in the domain model that are true when the operation has finished.
    - They are not actions to be performed during the operation
- Postconditions fall into these categories :
    - Instance creation and deletion
    - Attribute modification
    - Associations (UML links) formed and broken.

12) Explain guidelines for how to create and write contracts?

Apply the following advice to create contracts:

1. Identify system operations from the SSDs.
For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.

2. To describe the postconditions, use the following categories:
- instance creation and deletion
- attribute modification
- associations formed and broken

3. Writing Contracts

As mentioned, write the postconditions in a declarative, passive past tense form (*was* …) to emphasize the *observation* of a change rather than a design of how it is going to be achieved.
For example:
(better) A *SalesLineItem* **was** created.
(worse) Create a *SalesLineItem*.
Remember to establish an association between existing objects or those newly created.
For example,
it is not enough that a new *SalesLineItem* instance is created when the *enterItem* operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with *Sale*;
thus:
The *SalesLineItem* was associated with the *Sale* (association formed).

13) Discuss few examples on operation contracts.

Example: NextGen POS Contracts
System Operations of the Process Sale Use Case
Contract CO1: makeNewSale
Operation: makeNewSale()
Cross References:
Use Cases: Process Sale
Preconditions: none
Postconditions:
- A Sale instance s was created (instance creation).
- s was associated with a Register (association formed).
- Attributes of s were initialized.

Note the vague description in the last postcondition. If understandable, it's fine.
On a project, all these particular postconditions are so obvious from the use case that the *makeNewSale* contract should probably not be written.
Recall one of the guiding principles of healthy process and the UP: Keep it as light as possible, and avoid all artifacts unless they really add value.

Contract CO2: enterItem
Operation: enterItem(itemID: ItemID, quantity: integer)
Cross References: Use Cases: Process Sale
Preconditions: There is a sale underway.
Postconditions:
- A SalesLineItem instance sli was created (instance creation).
- sli was associated with the current Sale (association formed).
- sli.quantity became quantity (attribute modification).
- sli was associated with a ProductDescription, based on itemID match (association formed).

Contract CO3: endSale

14) What is an implementation model?

In UP terms, there exists an **Implementation Model**. This is all the implementation artifacts, such as the source code, database definitions, JSP/XML/HTML pages, and so forth. Thus, the code being created in this chapter can be considered part of the UP Implementation Model.

15) What are the steps involved in mapping design to code?

Mapping Designs to Code

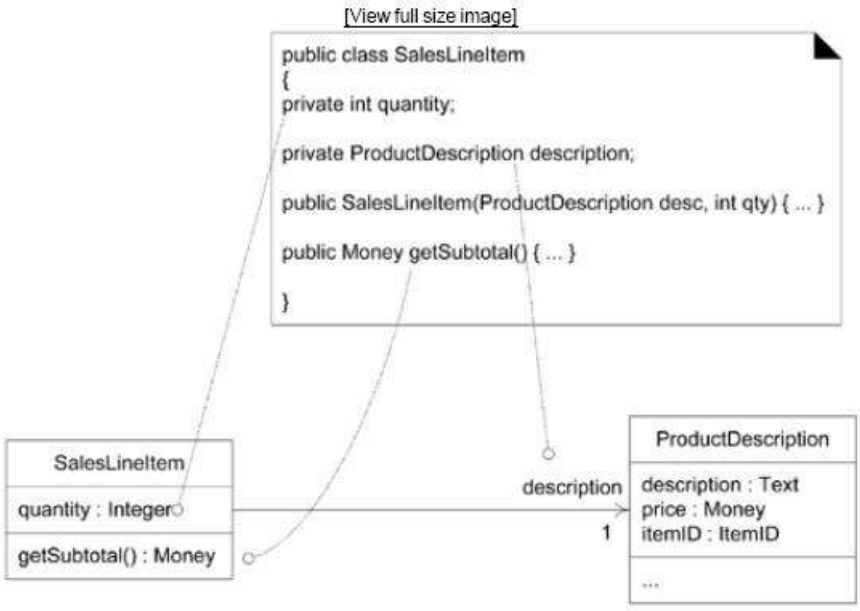
Implementation in an object-oriented language requires writing source code for:

- class and interface definitions
- method definitions

16) Explain with an example of creating class definitions from DCDs.

Creating Class Definitions from DCDs
At the very least, DCDs depict the class or interface name, superclasses, operation signatures, and attributes of a class. This is sufficient to create a basic class definition in an OO language. If the DCD was drawn in a UML tool, it can generate the basic class definition from the diagrams.
Defining a Class with Method Signatures and Attributes
From the DCD, a mapping to the attribute definitions (Java *fields*) and method signatures for the Java definition of *SalesLineItem* is straightforward, as shown in Figure below.

Figure 20.1. SalesLineItem in Java.

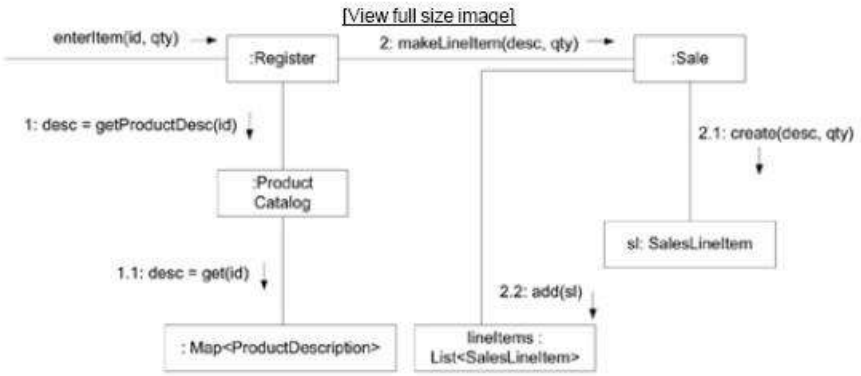


17) Explain in detail using an example how methods can be created from interaction diagrams?

Creating Methods from Interaction Diagrams

The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions. The *enterItem* interaction diagram in Figure 20.2 illustrates the Java definition of the *enterItem* method. For this example, we will explore the implementation of the *Register* and its *enterItem* method. A Java definition of the *Register* class is shown in Figure below.

Figure 20.2. The enterItem interaction diagram.



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 *getProductDescription* message is sent to the *ProductCatalog* to retrieve a *ProductDescription*.
ProductDescription desc = catalog.getProductDescription(itemID);
Message 2: The *makeLineItem* message is sent to the *Sale*.
currentSale.makeLineItem(desc, 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 20.4

The Register.enterItem Method

Figure 20.3. The Register class.

[View full size image]

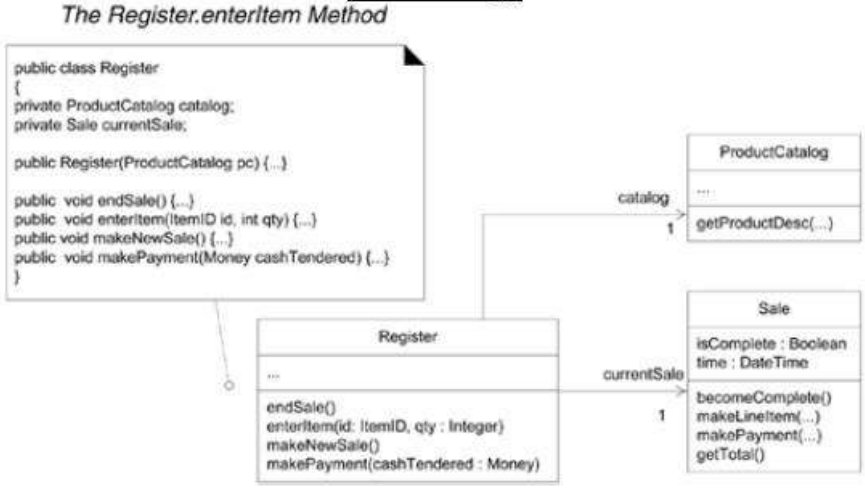
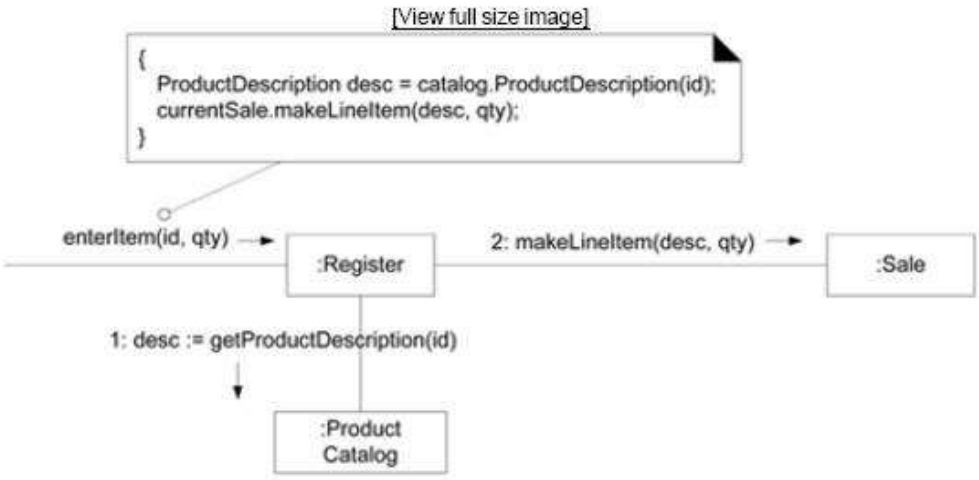


Figure 20.4. The enterItem method.

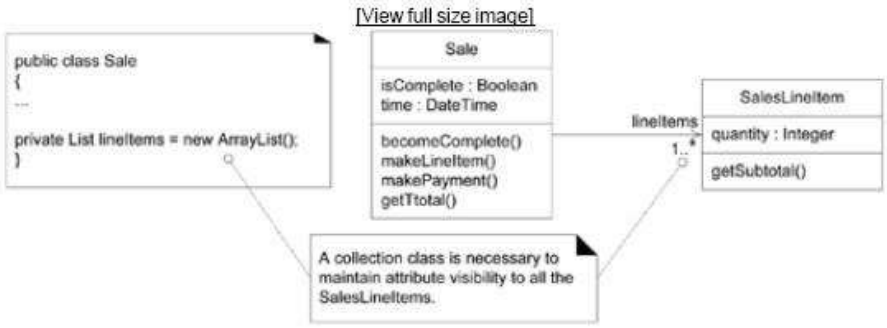


18) What are collection classes? Give examples.

Collection Classes in Code

One-to-many relationships are common. For example, a *Sale* must maintain visibility to a group of many *SalesLineItem* instances, as shown in Figure 20.5. In OO programming languages, these relationships are usually implemented with the introduction of a **collection** object, such as a *List* or *Map*, or even a simple array.

Figure 20.5. Adding a collection.



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.

As a small point, note that the *lineItems* attribute is declared in terms of its interface.

Guideline: If an object implements an interface, declare the variable in terms of the interface, not the concrete class.
For example, in Figure 20.5 the definition for the *lineItems* attribute demonstrates this guideline:
private List lineItems = new ArrayList();

19) Wite short notes on a) Exception and error handling
Exception and error handling has been ignored in the beginning. The main focus was on responsibility assignment and object design. But during implementation phase we need to consider the large-scale exception handling strategies during design modeling.
The operation syntax allows to specify how exceptions are to be handles :

Example :
+ getPlayer(name : string) : Player {exception IOException}
Public Player getplayer(string name_ throws exception

20) What is Test driven development?(TDD)

Test-Driven or Test-First Development
An excellent practice promoted by the Extreme Programming (XP) method [Beck00], and applicable to the UP and other iterative methods (as most XP practices are), is **test-driven development** (TDD) or **test-first development**. 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.

21) Explain in detail a sample application for mapping design to code.

This section presents a sample domain layer of classes 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. 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.
Comments excluded on purpose, in the interest of brevity, as the code is simple.

Class Payment

```
// all classes are probably in a package named
// something like:
package com.foo.nextgen.domain;
public class Payment
{
private Money amount;
public Payment( Money cashTendered ){ amount = cashTendered; }
public Money getAmount() { return amount; }
}
```

Class ProductCatalog

```
public class ProductCatalog
{
private Map<ItemID, ProductDescription>
descriptions = new HashMap()<ItemID, ProductDescription>;
public ProductCatalog()
{
// sample data
ItemID id1 = new ItemID( 100 );
ItemID id2 = new ItemID( 200 );
Money price = new Money( 3 );
ProductDescription desc;
desc = new ProductDescription( id1, price, "product 1" );
descriptions.put( id1, desc );
desc = new ProductDescription( id2, price, "product 2" );
descriptions.put( id2, desc );
}
public ProductDescription getProductDescription( ItemID id )
{
return descriptions.get( id );
}
}
```

Class Register

```
public class Register
{
private ProductCatalog catalog;
private Sale currentSale;
public Register( ProductCatalog catalog )
{
this.catalog = catalog;
}
public void endSale()
{
currentSale.becomeComplete();
}
public void enterItem( ItemID id, int quantity )
{
ProductDescription desc = catalog.getProductDescription( id );
currentSale.makeLineItem( desc, quantity );
}
public void makeNewSale()
{
currentSale = new Sale();
}
public void makePayment( Money cashTendered )
{
currentSale.makePayment( cashTendered );
}
}
```

Class ProductDescription

```
public class ProductDescription
{
private ItemID id;
private Money price;
private String description;
public ProductDescription
( ItemID id, Money price, String description )
{
this.id = id;
this.price = price;
this.description = description;
}
public ItemID getItemID() { return id; }
public Money getPrice() { return price; }
```

```
public String getDescription() { return description; }
}
```

Class Sale

```
public class Sale
{
private List<SalesLineItem> lineItems =
new ArrayList()<SalesLineItem>;
private Date date = new Date();
private boolean isComplete = false;
private Payment payment;
public Money getBalance()
{
return payment.getAmount().minus( getTotal() );
}
public void becomeComplete() { isComplete = true; }
public boolean isComplete() { return isComplete; }
public void makeLineItem
( ProductDescription desc, int quantity )
{
lineItems.add( new SalesLineItem( desc, quantity ) );
}
public Money getTotal()
{
Money total = new Money();
Money subtotal = null;
for ( SalesLineItem lineItem : lineItems )
{
subtotal = lineItem.getSubtotal();
total.add( subtotal );
}
return total;
}
public void makePayment( Money cashTendered )
{
payment = new Payment( cashTendered );
}
}
```

Class SalesLineItem

```
public class SalesLineItem
{
private int quantity;
private ProductDescription description;
public SalesLineItem (ProductDescription desc, int quantity )
{
this.description = desc;
this.quantity = quantity;
}
public Money getSubtotal()
{
return description.getPrice().times( quantity );
}
}
```

Class Store

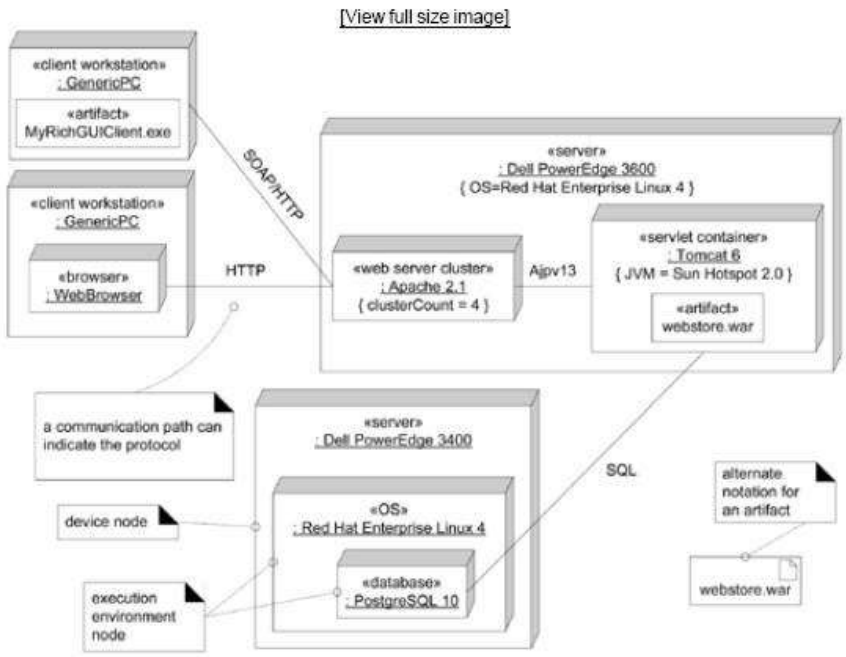
```
public class Store
{
private ProductCatalog catalog = new ProductCatalog();
private Register register = new Register( catalog );
public Register getRegister() { return register; }
}
```

22) What are deployment diagrams?

Deployment Diagrams

A deployment diagram shows the assignment of concrete software artifacts (such as executable files) to computational nodes (something with processing services). It shows the deployment of software elements to the **physical architecture** and the communication (usually on a network) between physical elements. See Figure
37.1. Deployment diagrams are useful to communicate the physical or deployment architecture.

Figure 37.1. A deployment diagram.



23) What are the basic elements of deployment diagrams?

The basic element of a deployment diagram is a **node**,
of two types:
device node (or **device**) A physical (e.g., digital electronic) computing resource with processing and memory services to execute software, such as a typical computer or a mobile phone. **execution environment node** (EEN) This is a *software* computing resource that runs within an outer node (such as a computer) and which itself provides a service to host and execute other executable software elements. For example:
an *operating system* (OS) is software that hosts and executes programs
a *virtual machine* (VM, such as the Java or .NET VM) hosts and executes programs a *database engine* (such as PostgreSQL) receives SQL program requests and executes them, and hosts/executes internal stored procedures (written in Java or a proprietary language)

a *Web browser* hosts and executes JavaScript, Java applets, Flash, and other executable technologies
a *workflow engine*
a *servlet container* or *EJB container*
As the UML specification suggests, many node types may show stereotypes, such as «server», «OS»,

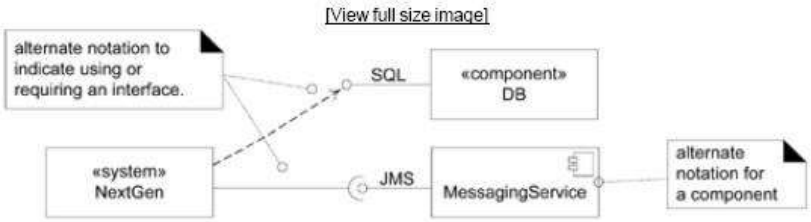
«database», or «browser», but these are not official predefined UML stereotypes.
Note that a device node or EEN may contain another EEN. For example, a virtual machine within an OS within a computer. A particular EEN can be implied, or not shown, or indicated informally with a UML property string; for example, {OS=Linux}. For example, there may not be value in showing the OS EEN as an explicit node. Figure 37.1 shows alternate styles, using the OS as an example.
The normal connection between nodes is a **communication path**, which may be labeled with the protocol.
These usually indicate the network connections.
A node may contain and show an **artifact**a concrete physical element, usually a file. This includes executables such as JARs, assemblies, .exe files, and scripts. It also includes data files such as XML, HTML, and so forth.
A deployment diagram usually shows an example set of *instances* (rather than classes). For example, an instance of a server computer running an instance of the Linux OS. Generally in the UML, concrete **instances** are shown with an underline under their name, and the absence of an underline signifies a class rather than an instance. Note that a major exception to this rule is instances in interaction diagramsthere, the names of things signifying instances in lifeline boxes are not underlined. In any event, in deployment diagrams, you will usually see the objects with their name underlined, to indicate instances. However, the UML specification states that for deployment diagrams, the underlining may be omitted and assumed. Therefore, you can see examples in both styles.

24) Define a component.

A **component** *represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces*

25) Explain UML components with a diagram.

Figure 37.2. UML components.



CS8592 Object Oriented Analysis and Design

V-Sem-CSE 2017-Regulations

UNIT IV DESIGN PATTERNS 9

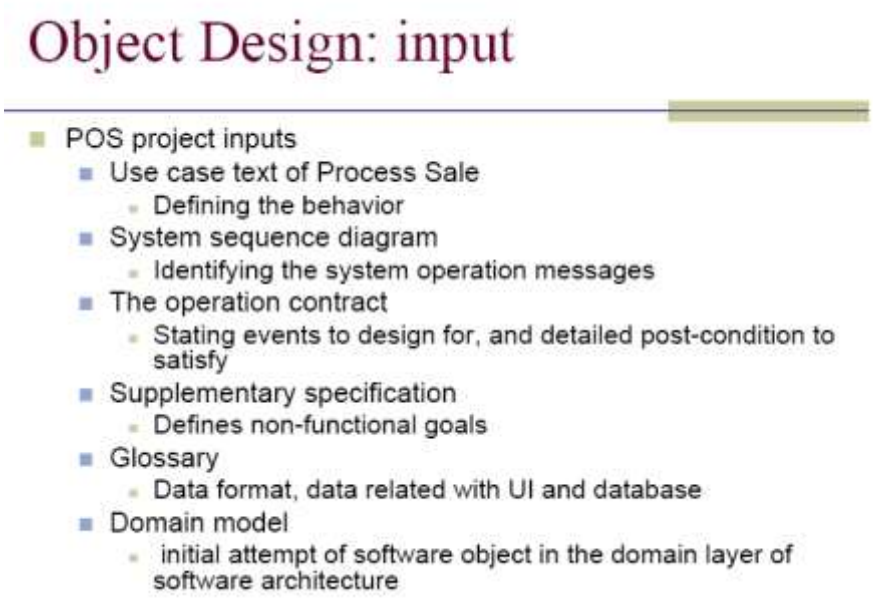
GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion – Controller Design Patterns – creational – factory method – structural – Bridge – Adapter – behavioural – Strategy – observer –Applying GoF design patterns – Mapping design to code

---

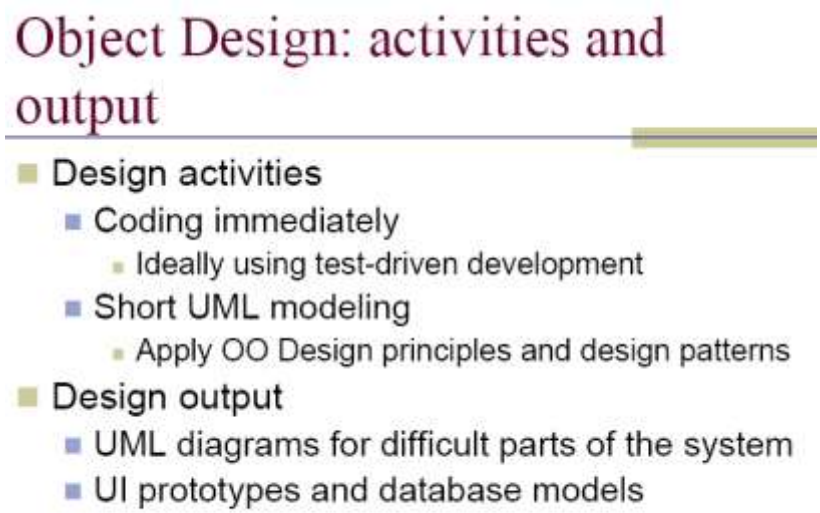
Question Bank

UNIT-IV

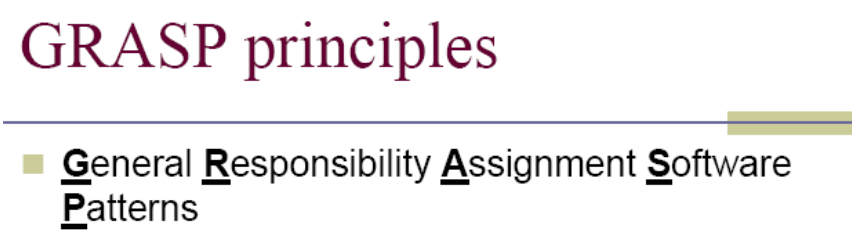
1) What are the artifact inputs to Object Design?



2) What are the activities of Object Design?



3) **What is GRASP?**



- GRASP is an acronym for '**G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns'.
- GRASP provides guidance for assigning responsibilities to classes and, to a limited extent, determining the classes that will be in a design in an object-oriented system. In short GRASP stands for designing objects with responsibilities

4) What are GRASP principles?

- The GRASP principles or patterns are a learning aid to help you understand essential object design in a methodical,rational,explainable way. This approach is based on patterns of assigning responsibilities.
- GRASP defines nine basic Object Oriented Design principles or basic building blocks in design.
    - Information Expert.
    - Creator
    - Controller
    - Low coupling
    - High Cohesion
    - Polymorphism
    - Pure fabrication
    - Indirection

- Protected variations

GRASP: Examples

Larman presents GRASP as “named problem/solution pairs”, for example:

Name	Problem
1) Expert 2) Creator 3) Low coupling 4) High Cohesion 5) Controller	What is the most basic principle by which responsibilities are assigned? Who should be responsible for creating a new instance of a class? How to support low dependency and increased reuse? How to keep complexity manageable? Who should be responsible for handling a system event?

5) What are patterns?

In Object Oriented Design,a pattern is a named description of a Problem and Solutions that can be applied to new contexts. Many patterns,given a category of problem,guide the assignment of responsibilities to objects.

6) What are Design Patterns?

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations

7) What is GOF?

GOF refers to the four authors(Gamma,Helm,Johnson,and Vlissides) known as the Gang-of-four who have written a book “design Patterns” which is considered as Bible of design pattern books. The book describes 23 patterns for Object Oriented Design..

8) **What is Responsibility Driven Design(RDD) approach?**

In RDD,we think about the design of Software Objects as having responsibilities – an abstraction of what they do. The UML defines a responsibility as “a Contract or Obligation of a Classifier

Responsibility Driven Design

- Responsibility is a contract or obligation of a class
- What must a class "know"? [knowing responsibility]
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate
- What must a class "do"? [doing responsibility]
  - Take action (create an object, do a calculation)
  - Initiate action in other objects
  - Control/coordinate actions in other objects
- Responsibilities are assigned to classes of objects during object design

9) **Explain Responsibility Driven Design using examples.**

Examples of responsibilities

"A Sale is responsible for creating SalesLineItems" (doing)

"A Sale is responsible for knowing its total" (knowing)

Knowing responsibilities are related to

attributes, associations in the domain model

Doing responsibilities can be expressed at

different granularities.

Doing responsibilities are implemented by

means of methods

10) Explain Creator Pattern with an example.

Creator

- Problem: Who should be responsible for creating a new instance of some class ?
- The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for assignment of creation responsibilities. Assigned well, the design can support low coupling increased clarity, encapsulation, and reusability.

Solution

- 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
  - B is a creator of A objects.

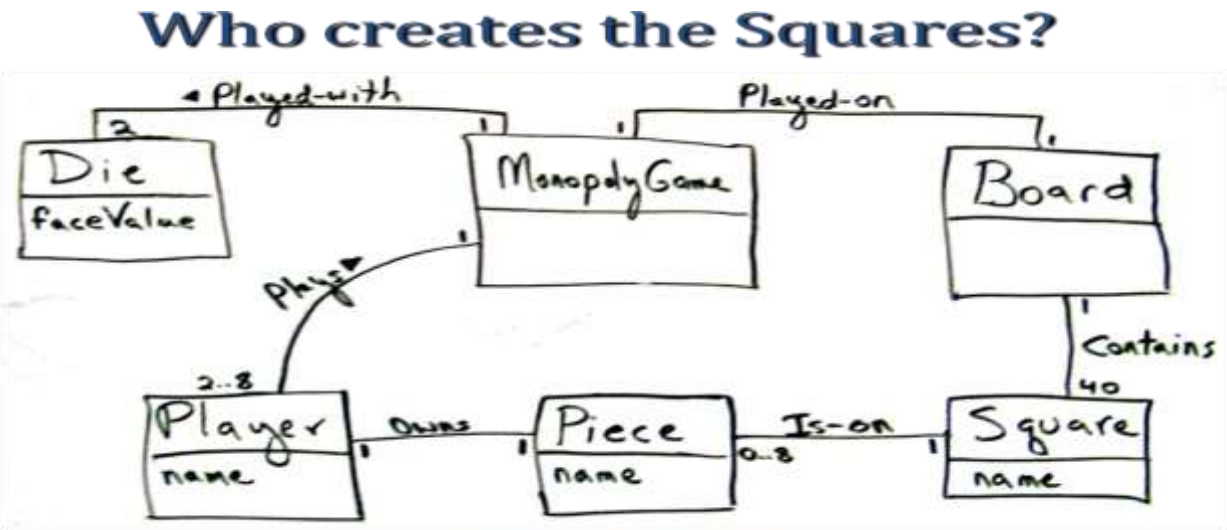


Figure 17.3, page 283

How does Create pattern develop this Design Class Diagram (DCD)?

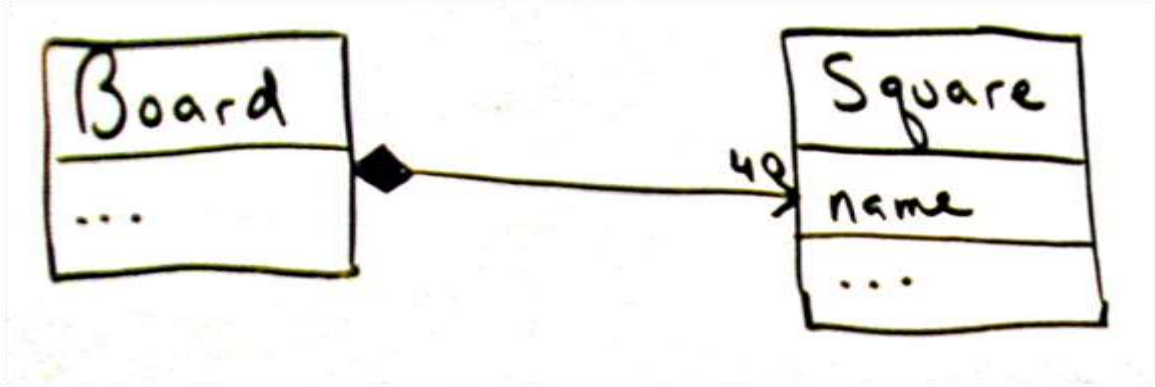


Figure 17.5 , page 283

Board has a composite aggregation relationship with *Square*
- I.e., Board contains a collection of Squares

Discussion of Creator pattern

- Responsibilities for object creation are common
- Connect an object to its creator when:
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records
  - Initializing data passed in during creation

11) Explain 'Information Expert' with an example.

Information Expert

- By Information expert we should look for the class of objects that has the information needed to determine the total.
- Problem : What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert – a class that has the information necessary to fulfill the responsibility.

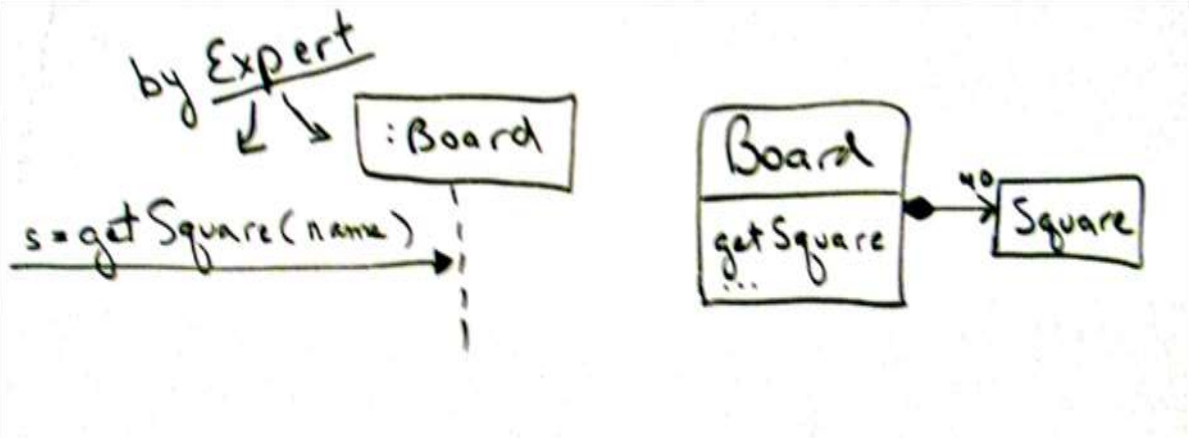
Information Expert pattern or principle

Name: Information Expert

Problem: How to assign responsibilities to objects?

Solution: Assign responsibility to the class that has the information needed to fulfill it?

- E.g., Board information needed to get a Square



12) Explain 'Low Coupling' with an example.

Low Coupling

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other element**.**
- **Low Coupling Pattern**
- Problem:
- How to support a low dependency,low change impact ,and increased reuse?
- Solution:
- Assign a responsibility so that coupling remains low.

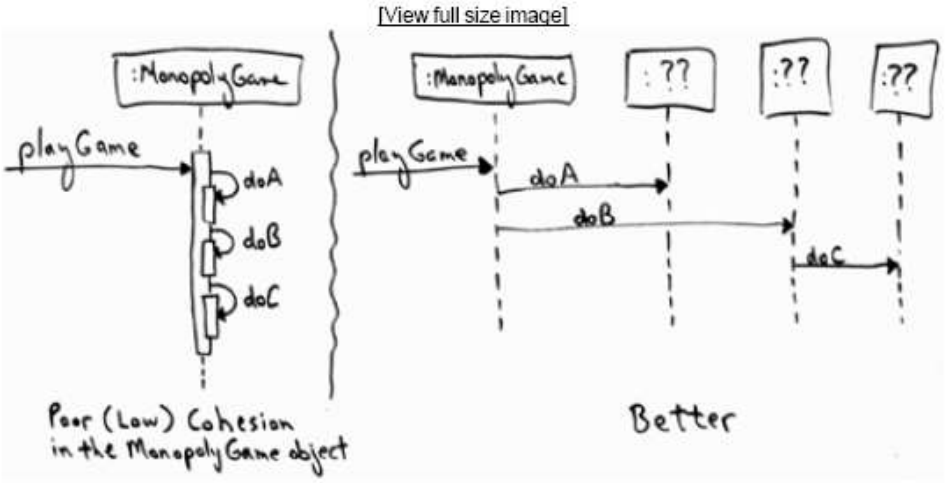
13) Define pattern 'Controller'

Controller pattern

- Name: **Controller**
  (see Model-View-Controller architecture)
- Problem: Who should be responsible for UI events?
- Solution: Assign responsibility for receiving or handling a system event in one of two ways:
  - Represent the overall system (*façade* pattern)
  - Represent a use case scenario within which the system event occurs (a *session* controller)

14) Define pattern 'High Cohesion'.

Figure 17.11. Contrasting the level of cohesion in different designs.



In software design a basic quality known as **cohesion** informally measures how functionally related the operations of a software element are, and also measures how much work a software element is doing. As a simple contrasting example, an object *Big* with 100 methods and 2,000 source lines of code (SLOC) is doing a lot more than an object *Small* with 10 methods and 200 source lines. And if the 100 methods of *Big* are covering many different areas of responsibility (such as database access *and* random number generation), then *Big* has less focus or functional cohesion than *Small*. In summary, both the amount of code and the relatedness of the code are an indicator of an object's cohesion.
To be clear, bad cohesion (low cohesion) doesn't just imply an object does work only by itself; indeed, a low cohesion object with 2,000 SLOC probably collaborates with many

other objects. Now, here's a *key point*: All that interaction tends to *also* create bad (high) coupling. Bad cohesion and bad coupling often go hand-in-hand.
In terms of the contrasting designs in Figure 17.11, the left-hand version of *MonopolyGame* has worse cohesion than the right-hand version, since the left-hand version is making the *MonopolyGame* object itself do all the work, rather than delegating and distributing work among objects. This leads to the principle of High Cohesion, which is used to evaluate different design choices. All other things being equal, prefer a design with higher
cohesion.

Name: **High Cohesion** Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling? Solution: (advice) Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives. We can say that the right-hand design better supports High Cohesion than the left-hand version.

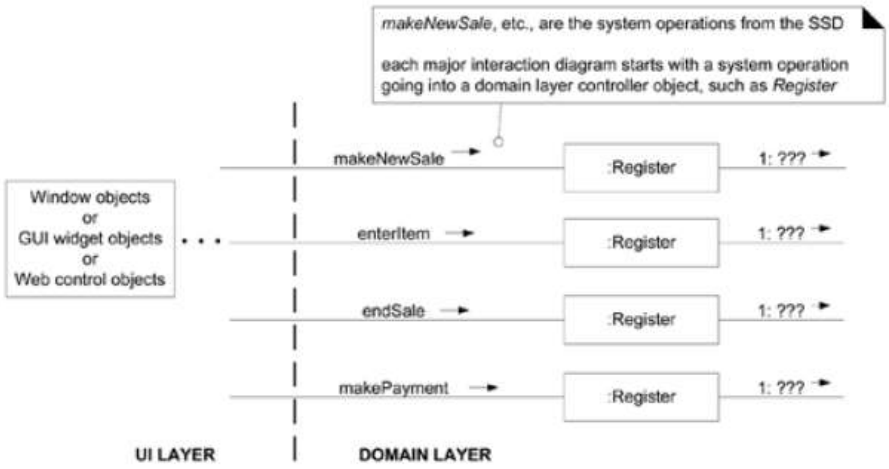
15) What is use case realization?

"A **use-case realization** describes how a particular use case is realized within the Design Model, in terms of collaborating objects" [RUP]. 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 (though non-standard, perhaps better called a **scenario realization**). *Use case realization* is a UP term used to remind us of the connection between the requirements expressed as use cases and the object design that satisfies the requirements.
UML diagrams are a common language to illustrate use case realizations. And as we explored in the prior chapter, we can apply principles and patterns of object design, such as Information Expert and Low Coupling, during this use case realization design work.
To review, Figure 18.1 illustrates the relationship between some UP artifacts, emphasizing the Use Case Model and the Design Modeluse case realizations.

Some relevant artifact-influence points include the following:
The use case suggests the system operations that are shown in SSDs.
The system operations become the starting messages entering the Controllers for domain layer interaction diagrams. See Figure 18.2.

This is a ***key point*** often missed by those new to OOA/D modeling.

Figure 18.2. Communication diagrams and system operation handling.



Domain layer interaction diagrams illustrate how objects interact to fulfill the required tasks the use case realization.

16) What is Visibility?

Visibility is the ability of one object to see or have reference to another.

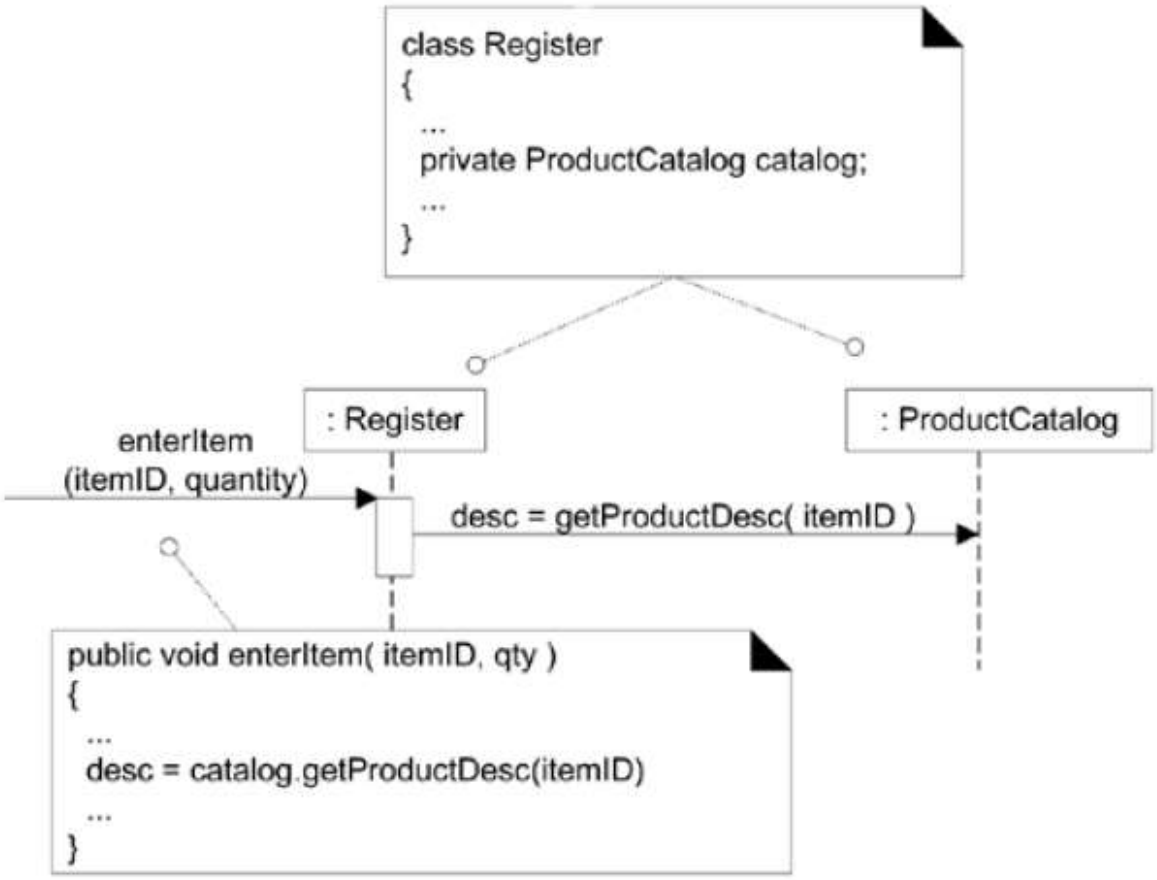
In common usage, **visibility** is the ability of an object to "see" or have a reference to another object. More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another?

Visibility Between Objects

The designs created for the system operations (*enterItem*, and so on) illustrate messages between objects. For a sender object to send a message to a receiver object, the sender must be *visible* to the receiver,the sender must have some kind of reference or pointer to the receiver object.

For example, the *getProductDescription* message sent from a *Register* to a *ProductCatalog* implies that the *ProductCatalog* instance is visible to the *Register* instance, as shown in Figure 19.1.

Figure 19.1. Visibility from the Register to ProductCatalog is required.



When creating a design of interacting objects, it is necessary to ensure that the necessary visibility is present to support message interaction.

17) Write short notes on a)Attribute visibility b)Parameter Visibility c)Local visibility d)Global visibility

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.
The motivation to consider visibility is this:
For an object A to send a message to an object B, B must be visible to A.
For example, to create an interaction diagram in which a message is sent from a *Register* instance to a *ProductCatalog* instance, the *Register* must have visibility to the *ProductCatalog*. A typical visibility solution is that a reference to the *ProductCatalog* instance is maintained as an attribute of the *Register*.

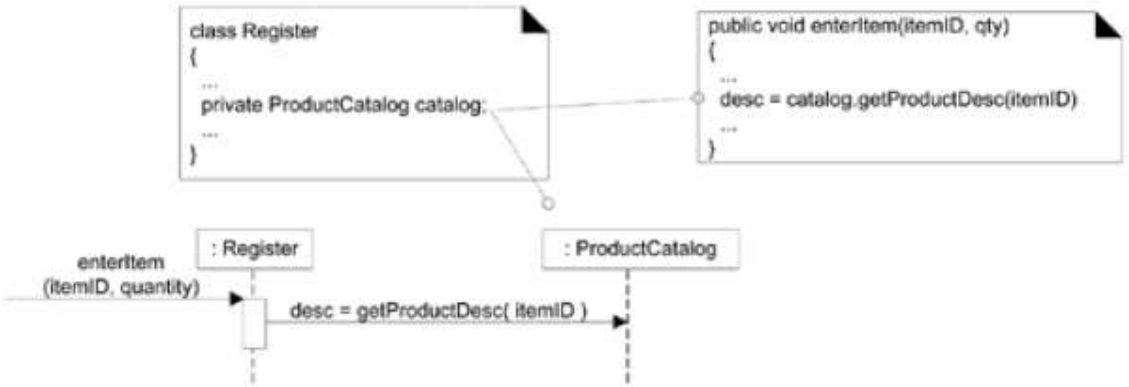
Attribute Visibility

Attribute visibility from A to B exists when B is an attribute of A. It is a relatively permanent visibility because it persists as long as A and B exist. This is a very common form of visibility in object-oriented systems.
To illustrate, in a Java class definition for *Register*, a *Register* instance may have attribute visibility to a *ProductCatalog*, since it is an attribute (Java instance variable) of the *Register*.

```
public class Register
{
...
private ProductCatalog catalog;
...
}
```

This visibility is required because in the *enterItem* diagram shown in Figure 19.2, a *Register* needs to send the *getProductDescription* message to a *ProductCatalog*:

Figure 19.2. Attribute visibility.



Parameter Visibility

Parameter visibility from A to B exists when B is passed as a parameter to a method of A. It is a relatively temporary visibility because it persists only within the scope of the method. After attribute visibility, it is the second most common form of visibility in object-oriented systems. To illustrate, when the *makeLineItem* message is sent to a *Sale* instance, a *ProductDescription* instance is passed as a parameter. Within the scope of the *makeLineItem* method, the *Sale* has parameter visibility to a *ProductDescription* (see Figure 19.3).

Figure 19.3. Parameter visibility.

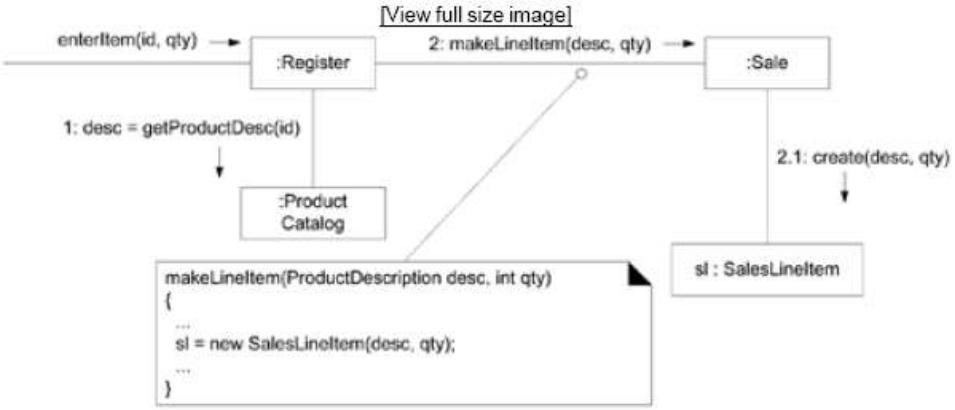
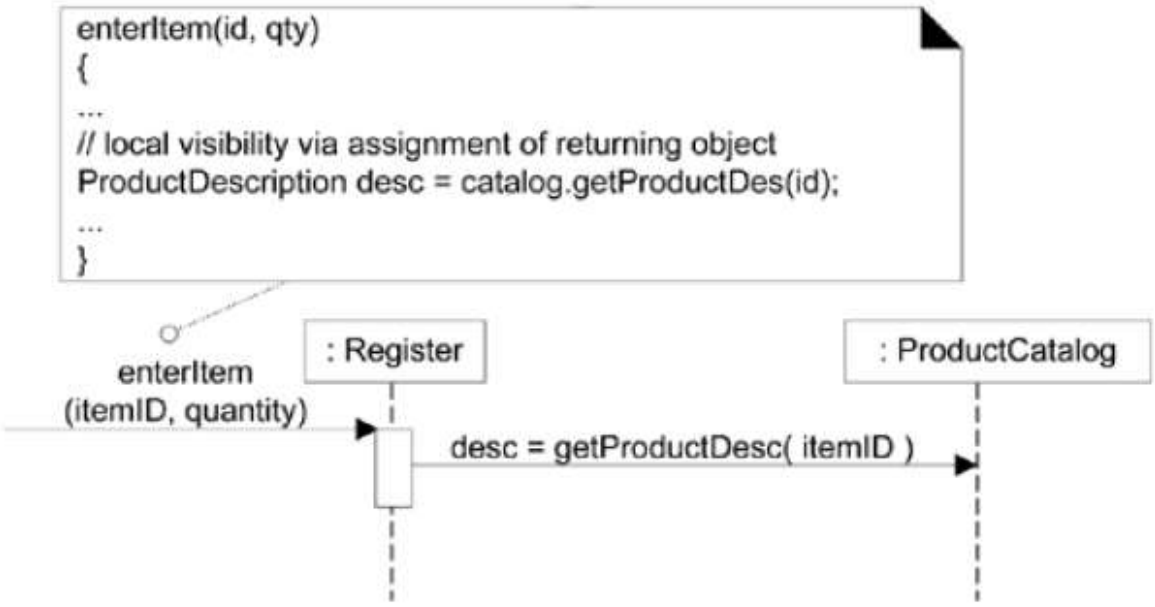


Figure 19.5. Local visibility.



Local Visibility

Local visibility from A to B exists when B is declared as a local object within a method of A. It is a relatively temporary visibility because it persists only within the scope of the method. After parameter visibility, it is the third most common form of visibility in object-oriented systems.

Two common means by which local visibility is achieved are:

Create a new local instance and assign it to a local variable.

Assign the returning object from a method invocation to a local variable.

As with parameter visibility, it is common to transform locally declared visibility into attribute visibility.

An example of the second variation (assigning the returning object to a local variable) can be found in the *enterItem* method of class *Register* (Figure 19.5).

Global Visibility

Global visibility from A to B exists when B is global to A. It is a relatively permanent visibility because it persists as long as A and B exist. It is the least common form of visibility in object-oriented systems.

One way to achieve global visibility is to assign an instance to a global variable, which is possible in some languages, such as C++, but not others, such as Java.

The preferred method to achieve global visibility is to use the **Singleton** pattern

18) Explain with a diagram Factory pattern.

Factory

This is also called **Simple Factory** or **Concrete Factory.** This pattern is not a GoF design pattern, but extremely widespread. It is also a simplification of the GoF Abstract Factory pattern (p. 597), and often described as a variation of Abstract Factory, although that's not strictly accurate. Nevertheless, because of its prevalence and association with GoF, it is presented now.

The adapter raises a new problem in the design: In the prior 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. Fundamentally, it is an application of the GRASP High Cohesion principle. 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.
A common alternative in this case is to apply the **Factory** pattern, in which a Pure Fabrication "factory" object is defined to create objects.

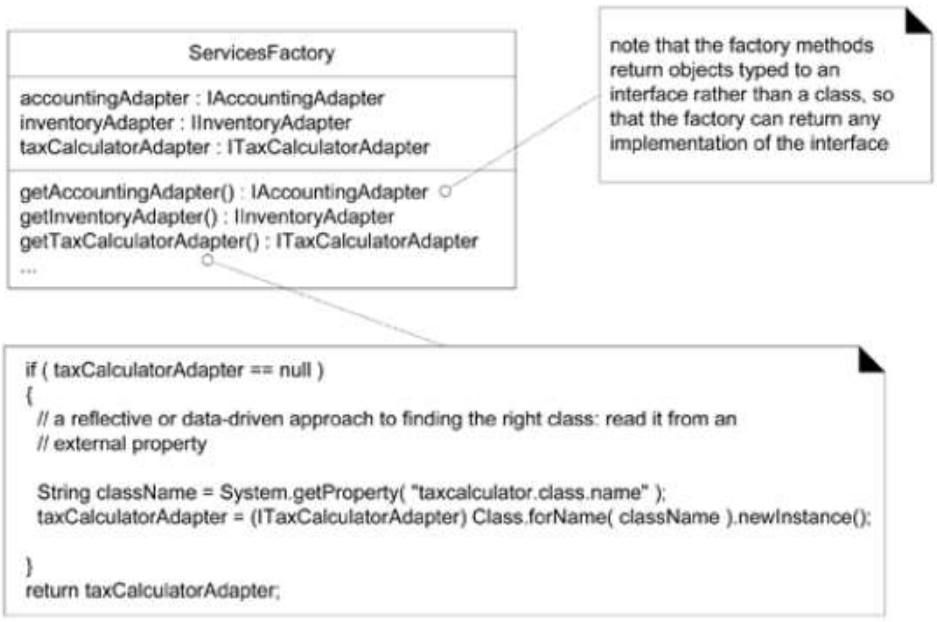
Factory objects have several advantages:
Separate the responsibility of complex creation into cohesive helper objects.
Hide potentially complex creation logic.
Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

Name: **Factory**

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: (advice) Create a Pure Fabrication object called a Factory that handles the creation.
A Factory solution is illustrated in Figure 26.5.

Figure 26.5. The Factory pattern.



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 that the new class is visible in the Java class path for loading.

Related Patterns

Factories are often accessed with the Singleton pattern.

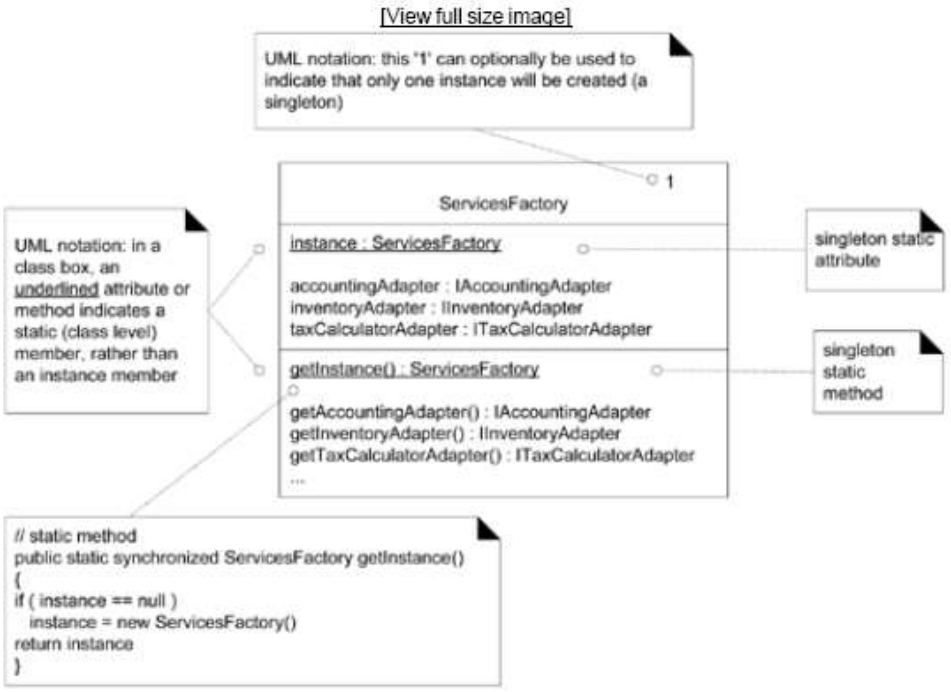
19) Explain Singleton pattern with a diagram.

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.

Name: **Singleton** Problem: Exactly one instance of a class is allowedit is a "singleton." Objects need a global and single point of access. Solution: (advice) Define a static method of the class that returns the singleton. For example, Figure 26.6 shows an implementation of the Singleton pattern.

Figure 26.6. The Singleton pattern in the *ServicesFactory* class.



Applying UML: Notice how a singleton is illustrated, with a '1' in the top right corner of the name compartment.

Thus, the key idea is that class X defines a static method *getInstance* that itself provides a single instance of X.
With this approach, a developer has global visibility to this single instance, via the static *getInstance* 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…
} // end of class
```

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.getInstance()
in order to obtain visibility to the singleton instance, and then send it a message, such as *SingletonClass.getInstance().doFoo()*. And it's hard to beat the feeling of being able to globally *doFoo*!

20) Explain Adapter pattern with a diagram

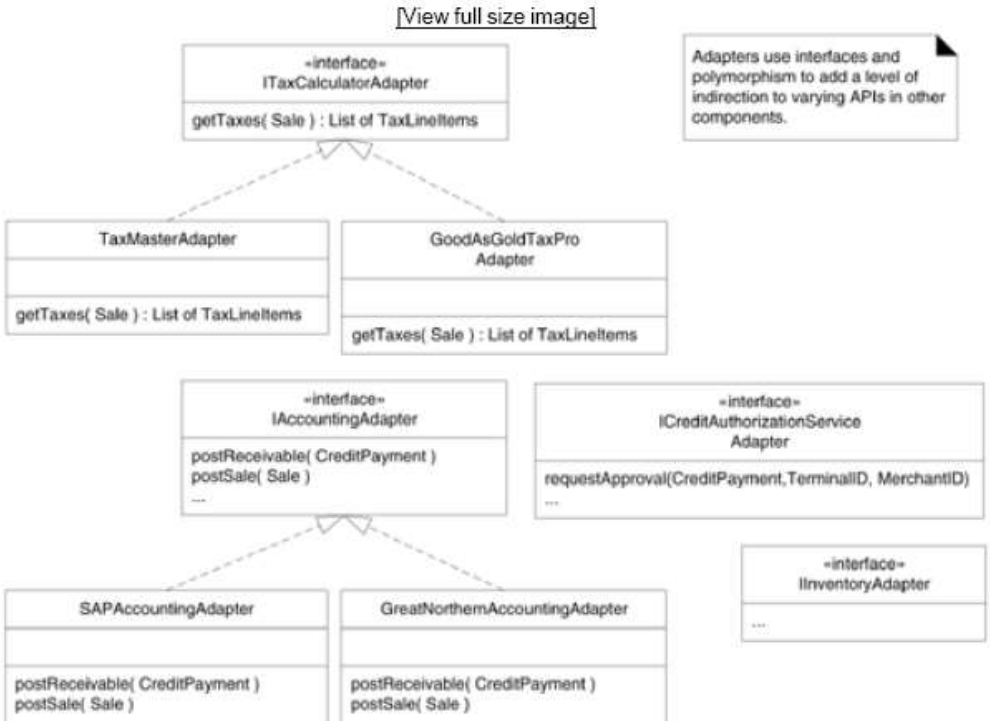
Adapter (GoF)
The NextGen problem explored on p. 414 to motivate the Polymorphism pattern and its solution is more specifically an example of the GoF **Adapter** pattern.

Name: **Adapter** Problem: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces? Solution: (advice) Convert the original interface of a component into another interface, through an intermediate adapter object.

To review: The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.

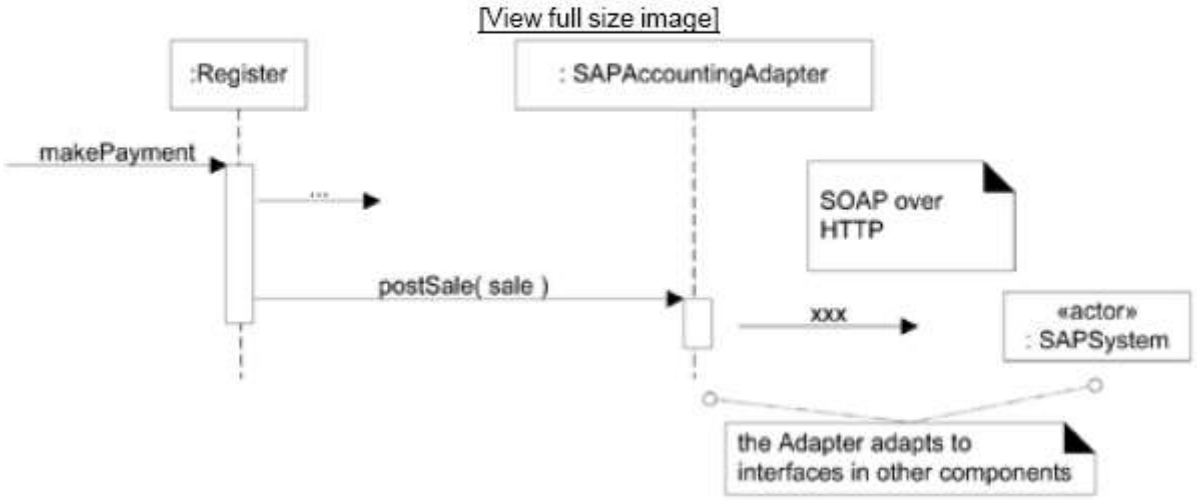
A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application. The solution is illustrated in Figure 26.1.

Figure 26.1. The Adapter pattern.



As illustrated in Figure 26.2, a particular adapter instance will be instantiated for the chosen external service[1] , such as SAP for accounting, and will adapt the *postSale* request to the external interface, such as a SOAP XML interface over HTTPS for an intranet Web service offered by SAP.

Figure 26.2. Using an Adapter.



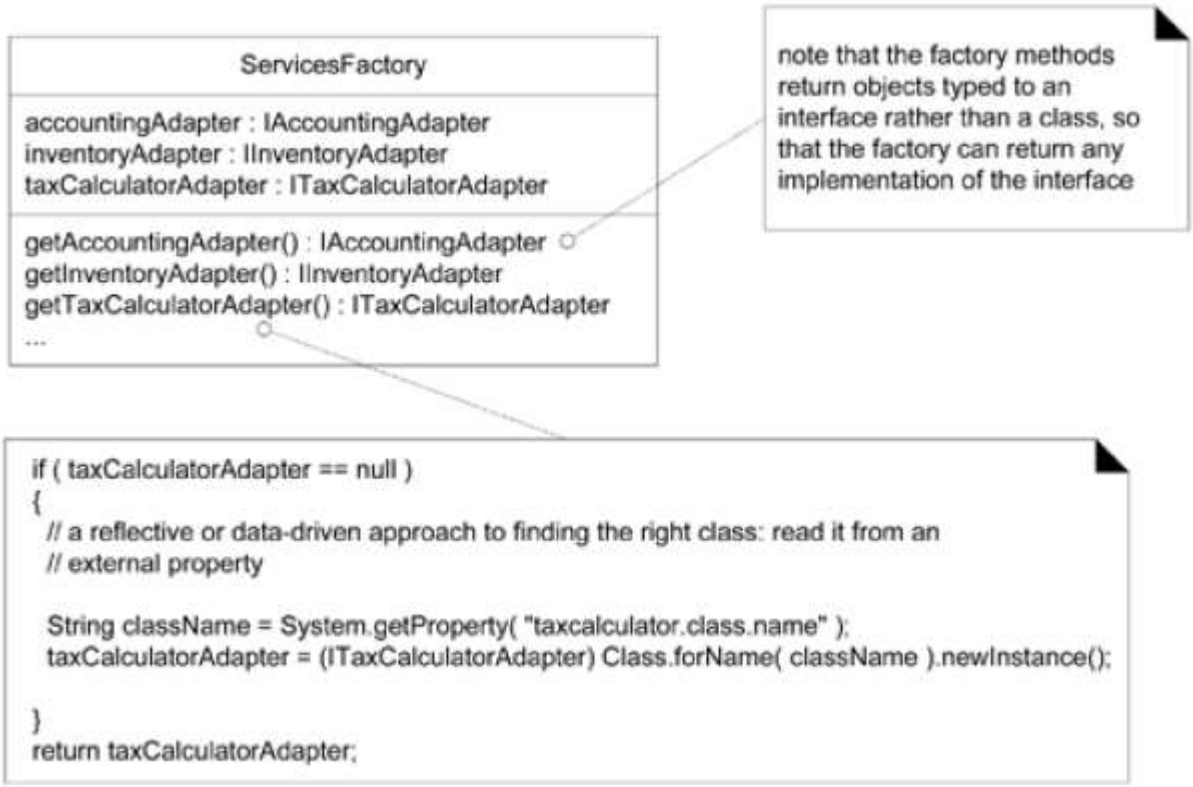
21) Explain Adapter,Factory,and Singleton patterns applied to the design with a diagram.

Factory

This is also called **Simple Factory** or **Concrete Factory.** This pattern is not a GoF design pattern, but extremely widespread. It is also a simplification of the GoF Abstract Factory pattern (p. 597), and often described as a variation of Abstract Factory, although that's not strictly accurate. Nevertheless, because of its prevalence and association with GoF, it is presented now.
The adapter raises a new problem in the design: In the prior 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. Fundamentally, it is an application of the GRASP High Cohesion principle. 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.
A common alternative in this case is to apply the **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.

Name: **Factory**
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: (advice) Create a Pure Fabrication object called a Factory that handles the creation.

A Factory solution is illustrated in Figure 26.5.
Figure 26.5. The Factory pattern.



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 that the new class is visible in the Java class path for loading.

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.

Name: **Singleton** Problem: Exactly one instance of a class is allowedit is a "singleton." Objects need a global and single point of access. Solution: (advice) Define a static method of the class that returns the singleton. For example, Figure 26.6 shows an implementation of the Singleton pattern.

Figure 26.6. The Singleton pattern in the *ServicesFactory* class.

22) Explain with a diagram composite pattern

Composite (GoF) and Other Design Principles

To raise yet another interesting requirements and design problem: How do we handle the case of multiple, conflicting pricing policies? For example, suppose a store has the following policies in effect today (Monday):
20% senior discount policy
preferred customer discount of 15% off sales over $400
on Monday, there is $50 off purchases over $500
buy 1 case of Darjeeling tea, get 15% discount off of everything

Suppose a senior who is also a preferred customer buys 1 case of Darjeeling tea, and $600 of veggieburgers
(clearly an enthusiastic vegetarian who loves chai). What pricing policy should be applied?

To clarify: There are now pricing strategies that attach to the sale by virtue of three factors:
1. time period (Monday)
2. customer type (senior)
3. a particular line item product (Darjeeling tea)
Another point of clarification: Three of the four example policies are really just "percentage discount" strategies, which simplifies our view of the problem.
Part of the answer to this problem requires defining the store's **conflict resolution strategy**. Usually, a store applies the "best for the customer" (lowest price) conflict resolution strategy, but this is not required, and it could change. For example, during a difficult financial period, the store may have to use a "highest price" conflict resolution strategy.
The first point to note is that there can exist multiple co-existing strategies, that is, one sale may have several pricing strategies. Another point to note is that a pricing strategy can be related to the type of customer (for example, a senior). This has creation design implications: The customer type must be known by the *StrategyFactory* at the time of creation of a pricing strategy for the customer.
Similarly, a pricing strategy can be related to the type of product being bought (for example, Darjeeling tea).
This likewise has creation design implications: The *ProductDescription* must be known by the *StrategyFactory* at the time of creation of a pricing strategy influenced by the product.
Is there a way to change the design so that the *Sale* object does not know if it is dealing with one or many pricing strategies, and also offer a design for the conflict resolution? Yes, with the Composite pattern.

Name: **Composite**

Problem: How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

Solution: (advice) Define classes for composite and atomic objects so that they implement the same interface.

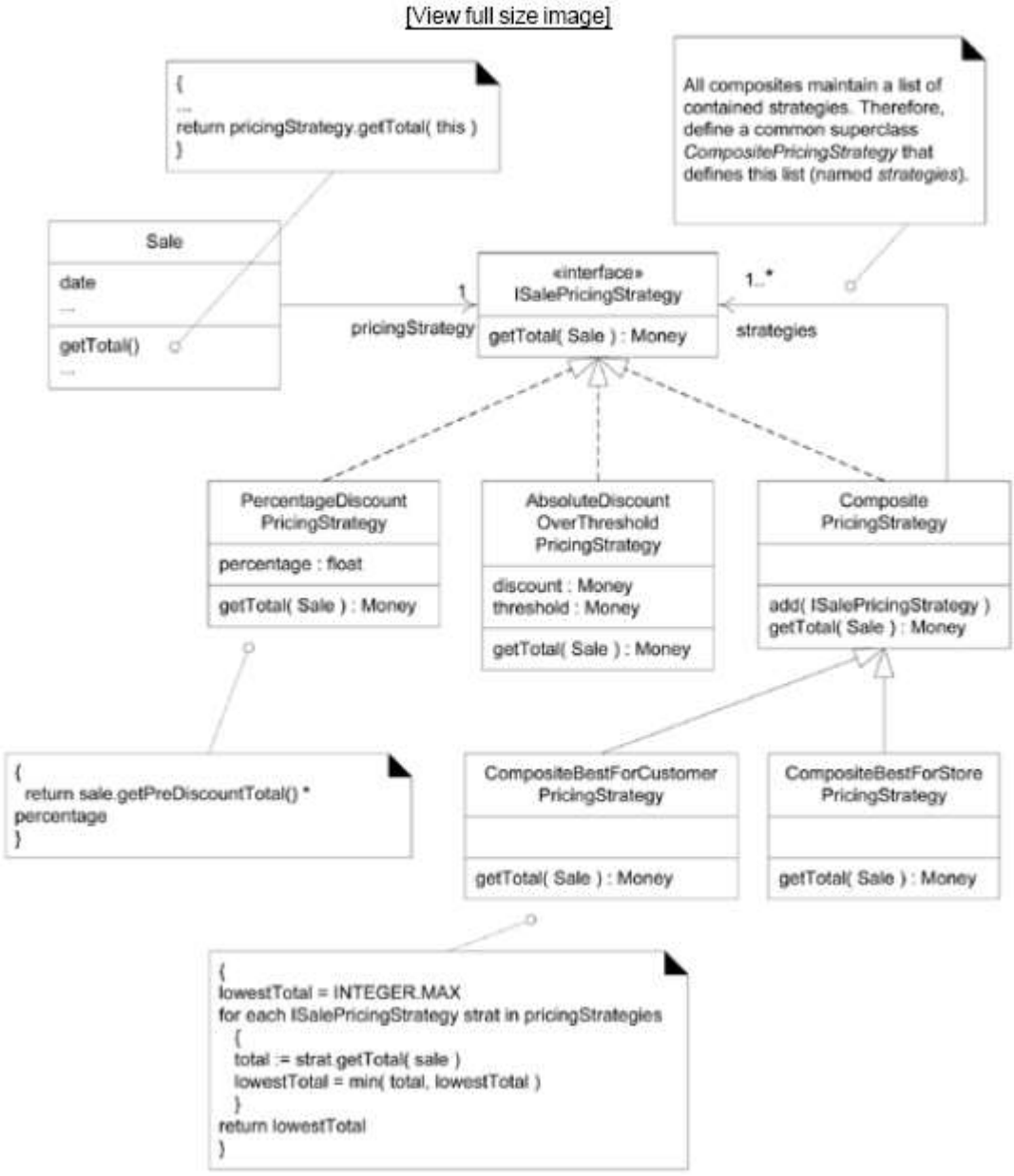
For example, a new class called *CompositeBestForCustomerPricingStrategy* (well, at least it's descriptive) can implement the *ISalesPricingStrategy* and itself contain other *ISalesPricingStrategy* objects. Figure 26.14 explains the design idea in detail.

Observe that in this design, the composite classes such as *CompositeBestForCustomerPricingStrategy* inherit an attribute *pricingStrategies* that contains a list of more *ISalePricingStrategy* objects. This is a signature feature of a composite object: The outer composite object contains a list of inner objects, and

both the outer and inner objects implement the same interface. That is, the composite class itself implements the *ISalePricingStrategy* interface.

Figure 26.14. The Composite pattern.

[View full size image]



23) What is the use of Façade pattern(GOF)

Facade (GoF)

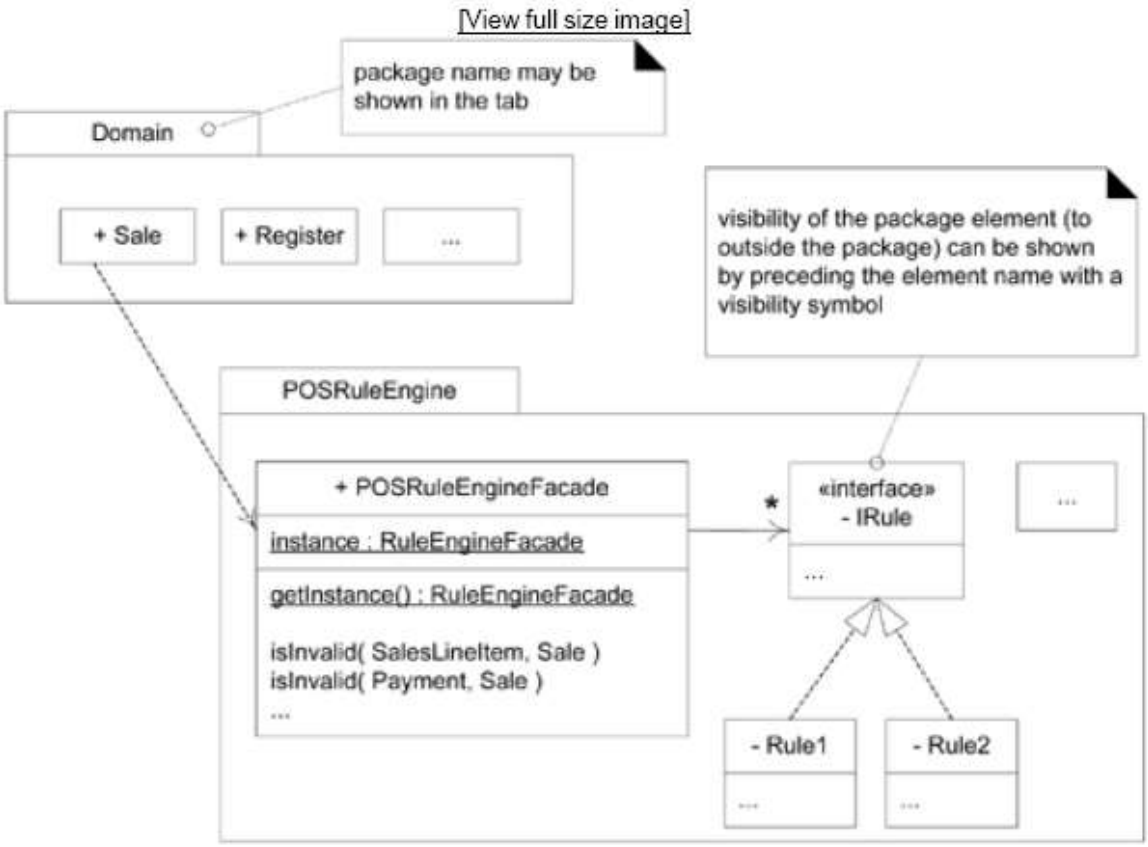
Another requirement chosen for this iteration is *pluggable business rules*. That is, at predictable points in the scenarios, such as when *makeNewSale* or *enterItem* occurs in the *Process Sale* use case, or when a cashier starts cashing in, different customers who wish to purchase the NextGen POS would like to customize its behavior slightly.
To be more precise, assume that rules are desired that can invalidate an action. For example:

Suppose when a new sale is created, it is possible to identify that it will be paid by a gift certificate (this is possible and common). Then, a store may have a rule to only allow one item to be purchased if a gift certificate is used. Consequently, subsequent *enterItem* operations, after the first, should be invalidated.
If the sale is paid by a gift certificate, invalidate all payment types of change due back to the customer except for another gift certificate. For example, if the cashier requested change in the form of cash, or as a credit to the customer's store account, invalidate those requests.
Suppose when a new sale is created, it is possible to identify that it is for a charitable donation (from the store to the charity). A store may also have a rule to only allow item entries less than $250 each, and also to only add items to the sale if the currently logged in "cashier" is a manager. In terms of requirements analysis, the specific scenario points across all use cases (*enterItem, chooseCashChange*, ...) must be identified. For this exploration, only the *enterItem* point will be considered, but the same solution applies equally to all points.
Suppose that the software architect wants a design that has low impact on the existing software components.
That is, she or he wants to design for a separation of concerns, and factor out this rule handling into a separate concern. Furthermore, suppose that the architect is unsure of the best implementation for this pluggable rule handling, and may want to experiment with different solutions for representing, loading, and evaluating the rules. For example, rules can be implemented with the Strategy pattern, or with free open-source rule interpreters that read and interpret a set of IF-THEN rules, or with commercial, purchased rule interpreters, among other solutions.
To solve this design problem, the Facade pattern can be used.

Name: **Façade** Problem: A common, unified interface to a disparate set of implementations or interfacessuch as within a subsystemis required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do? Solution: (advice) Define a single point of contact to the subsystema facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

A Facade is a "front-end" object that is the single point of entry for the services of a subsystem[2] ; the implementation and other components of the subsystem are private and can't be seen by external components.
Facade provides Protected Variations from changes in the implementation of a subsystem.
[2] "Subsystem" is here used in an informal sense to indicate a separate grouping of related components, not exactly as defined in the UML.
For example, we will define a "rule engine" subsystem, whose specific implementation is not yet known.[3] It will be responsible for evaluating a set of rules against an operation (by some hidden implementation), and then indicating if any of the rules invalidated the operation.

Figure 26.20. UML package diagram with a Facade.



Note the use of the Singleton pattern. Facades are often accessed via Singleton.
With this design, the complexity and implementation of how rules will be represented and evaluated are hidden in the "rules engine" subsystem, accessed via the *POSRuleEngineFacade* facade. Observe that the subsystem hidden by the facade object could contain dozens or hundreds of classes of objects, or even a non-objectoriented solution, yet as a client to the subsystem, we see only its one public access point.
And a separation of concerns has been achieved to some degreeall the rule-handling concerns have been delegated to another subsystem.

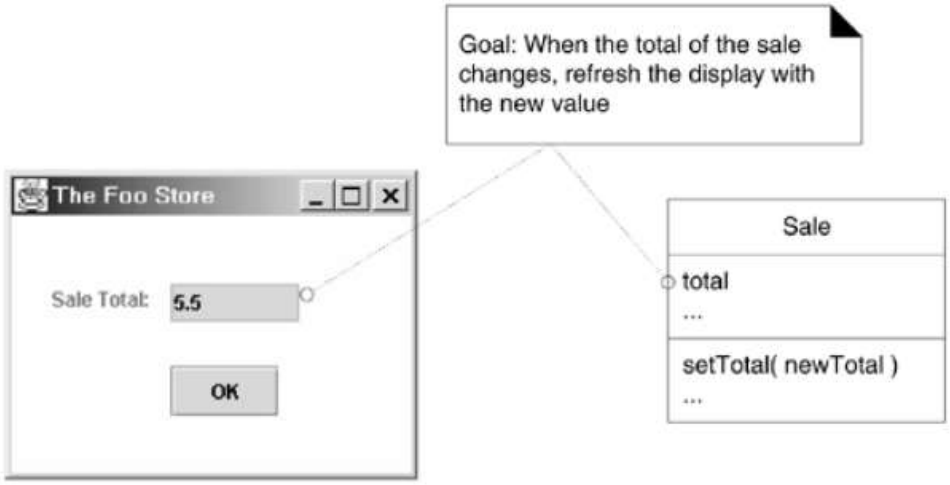
24) Explain with an example Observer pattern.

Observer/Publish-Subscribe/Delegation Event Model

(GoF)

Another requirement for the iteration is adding the ability for a GUI window to refresh its display of the sale total when the total changes (see Figure 26.21). The idea is to solve the problem for this one case, and then in later iterations, extend the solution to refreshing the GUI display for other changing data as well.

Figure 26.21. Updating the interface when the sale total changes.



Why not do the following as a solution? When the *Sale* changes its total, the *Sale* object sends a message to a window, asking it to refresh its display.
To review, the Model-View Separation principle discourages such solutions. It states that "model" objects (non- UI objects such as a *Sale*) should not know about view or presentation objects such as a window. It promotes Low Coupling from other layers to the presentation (UI) layer of objects.
A consequence of supporting this low coupling is that it allows the replacement of the view or presentation layer by a new one, or of particular windows by new windows, without impacting the non-UI objects. If model objects do not know about Java Swing objects (for example), then it is possible to unplug a Swing interface, or unplug a particular window, and plug in something else. Thus, Model-View Separation supports Protected Variations with respect to a changing user interface.
To solve this design problem, the Observer pattern can be used.

Name: **Observer (Publish-Subscribe)**

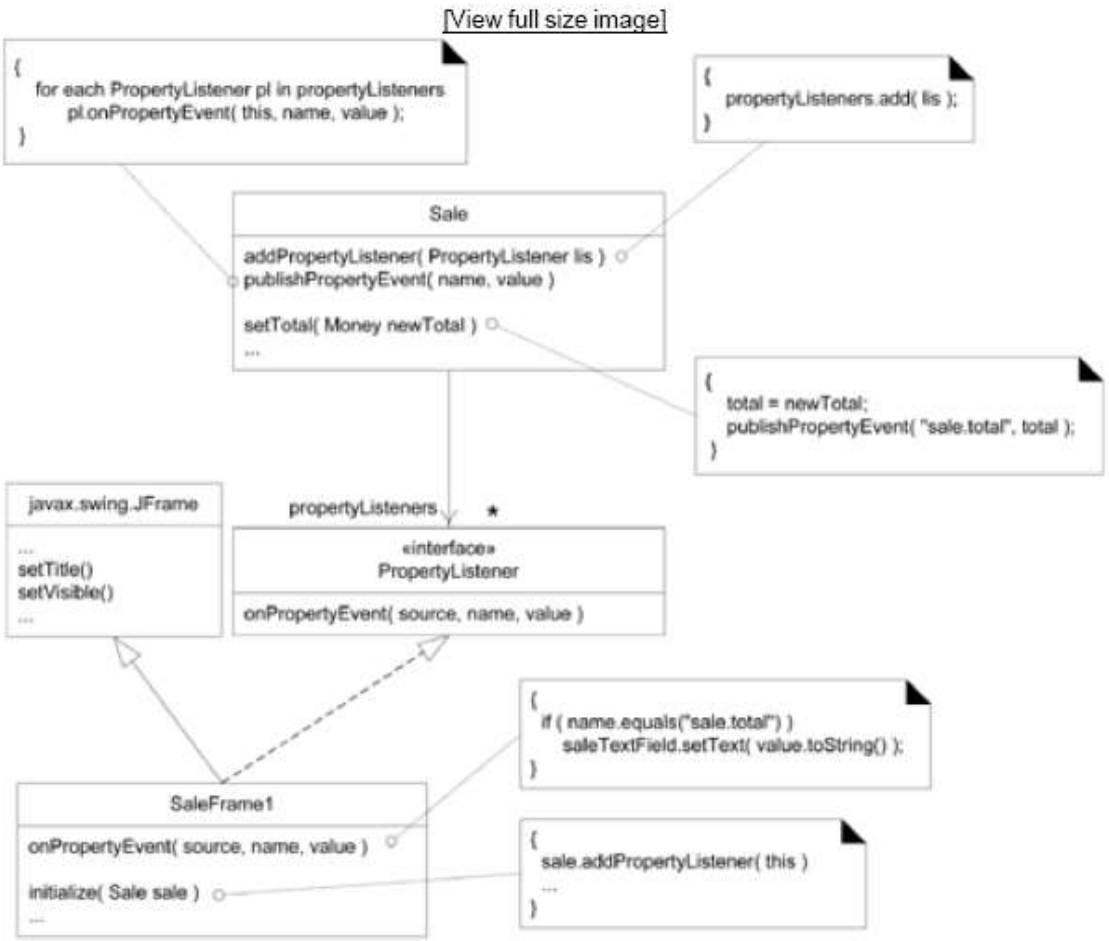
Problem: Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers.
What to do?

Solution: (advice) Define a "subscriber" or "listener" interface. Subscribers implement this interface. The
publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

An example solution is described in detail in Figure 26.22.

Figure 26.22. The Observer pattern.

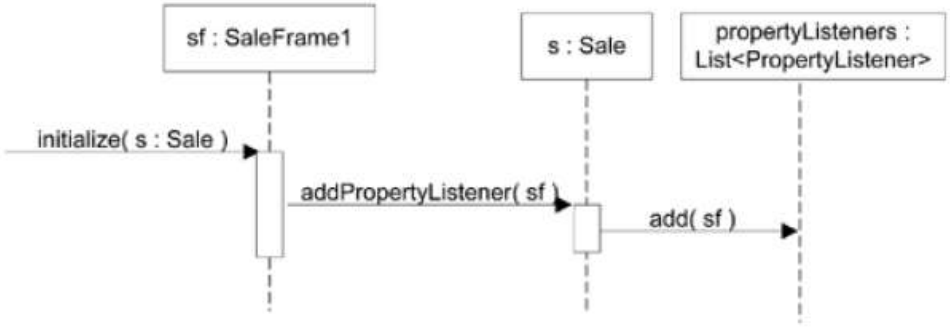
[View full size image]



The major ideas and steps in this example:
1. An interface is defined; in this case, *PropertyListener* with the operation *onPropertyEvent*.

2. Define the window to implement the interface.
SaleFrame1 will implement the method *onPropertyEvent*.
3. When the *SaleFrame1* window is initialized, pass it the *Sale* instance from which it is displaying the total. The *SaleFrame1* window registers or *subscribes* to the *Sale* instance for notification of "property events," via the *addPropertyListener* message. That is, when a property (such as total) changes, the window wants to be notified.
4. Note that the *Sale* does not know about *SaleFrame1* objects; rather, it only knows about objects that implement the *PropertyListener* interface. This lowers the coupling of the *Sale* to the windowthe coupling is only to an interface, not to a GUI class.
5. The *Sale* instance is thus a *publisher* of "property events." When the total changes, it iterates across all subscribing *PropertyListeners*, notifying each.
6. The *SaleFrame1* object is the observer/subscriber/listener. In Figure 26.23, it *subscribes* to interest in property events of the *Sale*, which is a *publisher* of property events. The *Sale* adds the object to its list of *PropertyListener* subscribers. Note that the *Sale* does not know about the *SaleFrame1* as a *SaleFrame1* object,
but only as a *PropertyListener* object; this lowers the coupling from the model up to the view layer.

Figure 26.23. The observer *SaleFrame1* subscribes to the publisher *Sale*.



As illustrated in Figure 26.24, when the Sale total changes, it iterates across all its registered subscribers, and "publishes an event" by sending the *onPropertyEvent* message to each.

Figure 26.24. The Sale publishes a property event to all its subscribers.

CS8592 Object Oriented Analysis and Design

V-Sem-CSE - 2017-Regulations

UNIT V TESTING **9**

Object Oriented Methodologies – Software Quality Assurance – Impact of object orientation on Testing – Develop Test Cases and Test Plans

---

Question Bank

UNIT-V

1) What are different Object Oriented Methodologies?

A methodology is explained as the science of methods.
• A method is a set of procedures in which a specific goal is approached step by step.

Too Many Methodologies

• 1986: Booch came up with the object-oriented design concept, the Booch method.
• 1987: Sally Shlaer and Steve Mellor came up with the concept of the recursive design approach.
1989: Beck and Cunningham came up with class-responsibilitycollaboration (CRC) cards.
• 1990: Wirfs-Brock, Wilkerson, and Wiener came up with responsibilitydriven design.
• 1991: Peter Coad and Ed Yourdon developed the Coad lightweight and prototype-oriented approach.
• 1991: Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT).
• 1994: Ivar Jacobson introduced the concept of the use case.

Survey of Some of the ObjectOriented Methodologies

• Many methodologies are available to choose from for system development.
• Here, we look at the methodologies developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML) and the bases of the UA.

28) a) What is OMT? Explain with examples a) Object Model b) OMT Dynamic Model.

The object modeling technique (OMT) presented by Jim Rumbaugh and his co-workers describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. Details such as class attributes, method, inheritance, and association also can be expressed easily. The dynamic behavior of objects within a system can be described using the OMT dynamic model. This model lets you specify detailed state transitions and their descriptions within a system. Finally, a process description and consumer-producer relationships can be expressed using OMT's functional model. OMT consists of four phases, which can be performed iteratively:

4.3.1 The Object Model

The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations. The object model is represented graphically with an object diagram (see Figure 4–1). The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among the classes. Each association line represents a set of links from the objects of one class to the objects of another class.

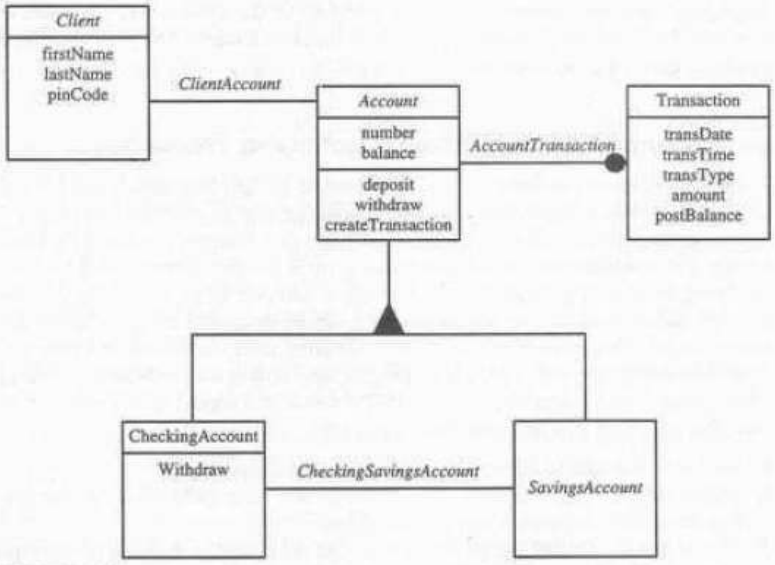


FIGURE 4–1
The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one too many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship betwen Client and Account classes is one to one: A client can have only one account and account can belong to only one person (in this model joint accounts are not allowed).

4.3.2 The OMT Dynamic Model

OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions. The OMT state transition diagram is a network of states and events (see Figure 4–2). Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.

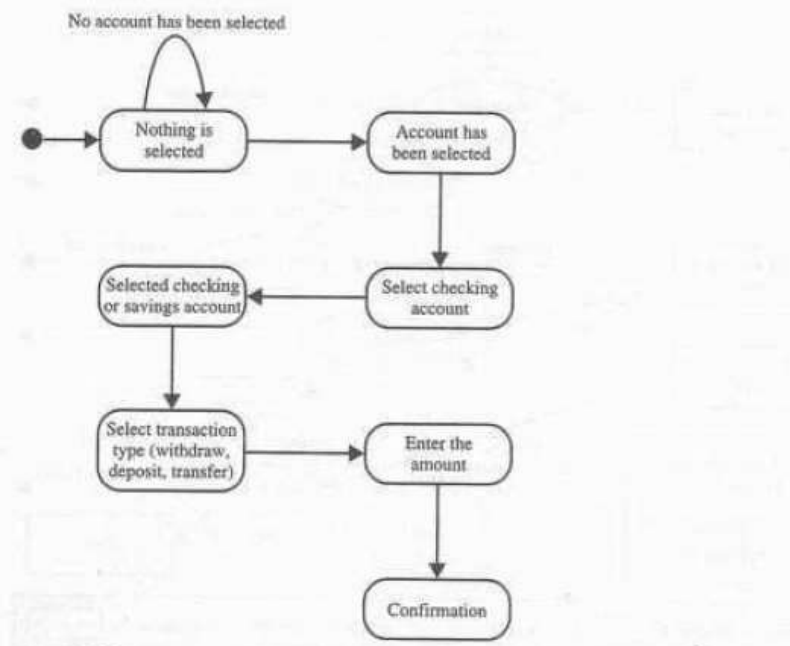


FIGURE 4–2
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

(OR)

b) Explain briefly the four phases of OMT model. Explain with example OMT Functional Model.

I. The object modeling technique (OMT) presented by Jim Rumbaugh and his coworkers describes a metod for analysis, design, implementation of using an Object Oriented Technique. OMT is fast intuitive approach for identifying and modeling all the objects making up the system. Details such as class, attributes, method, inheritance, and association also can be expressed easily.

OMT consists of four phases, which can be performed iteratively :

1) Analysis – The results are objects and dynamic and functional models.
2) System Design – the results are structure of the basic architecture of the system along with high-level strategy decisions.
3) Object Design – This phase produces a design document , consisting of detailed objects static, dynamic, and functional models.
4) Implementation – This activity produces reusable, extendible and robust code.

The OMT Functional Model :

The OMT Data flow Diagram (DFD) shows the flow of data between different processes in business. An OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems.

Data Flow Diagrams use four primary symbols :

1) The ***process*** is any function being performed ; for example , verify password or PINin the ATM system.

2) The ***data flow*** shows the direction of data element movement; for example PIN code.
3) The ***data store*** is a location where data are stored; for example , account is a data store in the ATM example.
4) An ***external entity*** is a source or destination of a data element; for example, the ATM card reader.

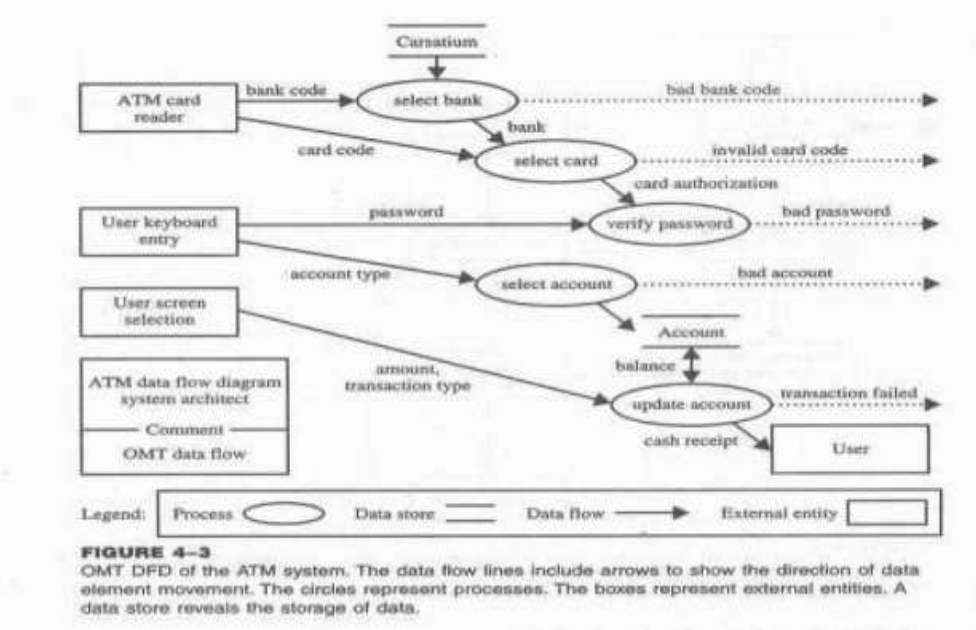


FIGURE 4–3
OMT DFD of the ATM system. The data flow lines include arrows to show the direction of data element movement. The circles represent processes. The boxes represent external entities. A data store reveals the storage of data.

10) Explain briefly units of OO Testing.

Individual methods are units

Levels of testing – 1

Four levels

- **Method**
- **Class**
- **Integration**
- **System**

At port level – same as traditional testing

Levels of testing – 2

Classes are units

Three levels

- **Class**
  Unit testing
- **Integration**
  Interclass testing
- **System**

- At port level

12) What are the implications of composition and Encapsulation?

Composition (as opposed to decomposition) is the central design strategy in O-O Software development. Together with the goal of reuse , composition creates the need for very stron unit testing. Because a unit(class) may be composed with previously unknown other units , the

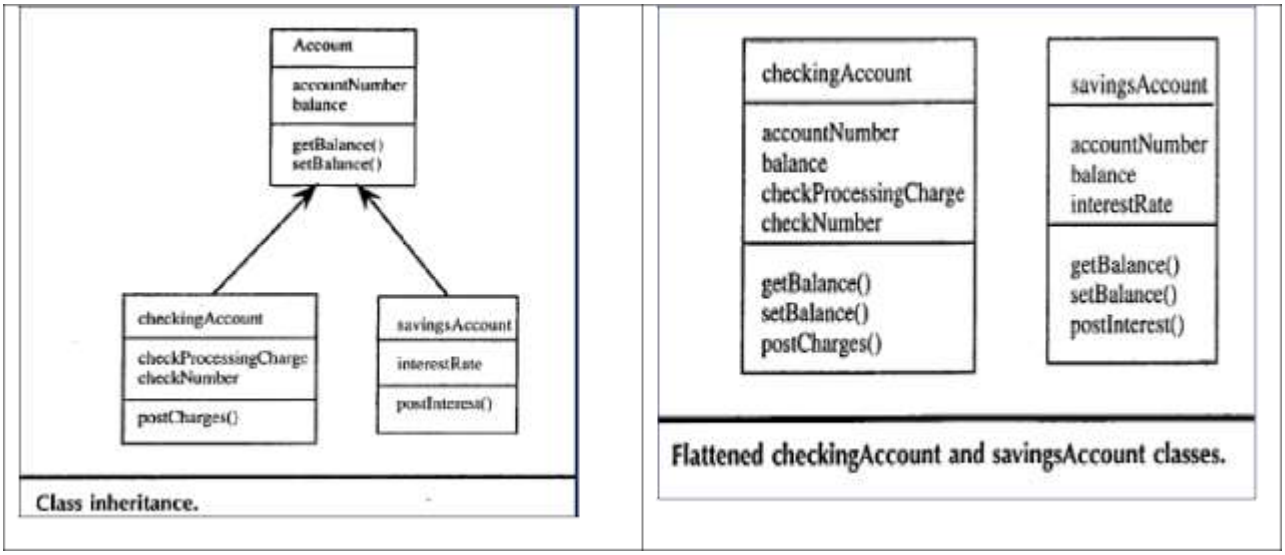
traditional notions of coupling and cohesion are applicable. Encapsulation has the potential to resolve this concern, but only if the units (classes) are highly cohesive and very loosely coupled. The main implication of composition is that , even presuming very good unit-level testing, the real burden is at the integration testing level.

13) What are the implications of Inheritance in OO-Testing?

Eventhough , a class is considered as unit of testing as a natural choice, the role of Inheritance complicates the matter. If a given class inherits attributes and/or operations from super classes , the stand
alone compilation criterion of a unit is sacrificed. However, the "flattened classes" concept alleviates this problem. A flattened class is an original class expanded to include all the attributes and operations it inherits.

Unit Testing on a flattened class solves the inheritance problem , but it raises another. A flattened class will not be part of a final system, so some uncertainty remains. Also the methods in a flattened class might not be sufficient to test the class. This leads to unending chain of methods testing other methods.

EXAMPLE



The LHS Fig. shows a UML inheritance diagram of a part of a simple automated teller machine. The fig. on the RHS shows the "flattened" checkingAccount and savingsAccount classes. Here we need to test the getbalance and setBalance operations twice resulting in losing some of the benefits of Object Orientation.

14) What are the implications of Polymorphism on OO-Testing?

*Testing with different objects introduces r***edundant tests on inherited methods**

- **Lose hoped for economies**
- Similarly testing polymorphism introduces redundant testing

Polymorphism Issues are summarized as below :

- Repeatedly testing same methods
- Time can then be wasted if not addressed
- Potentially can be avoided, and actually save time

15) Briefly explain class testing.

Class Testing

- Class test cases are created by examining the specification of the class.
  - This is more difficult if the class is a **subclass** and inherits **data and behavior** from a **super class**. A complicated class hierarchy can be pose significant testing problems.
- If you have a **state model** for the class, **test each transition** - devise a driver that sets an object of the class in the source state of the transition and generates the transition event.
- **Class Constraints/Invariants** should be incorporated into the **class test**.
- All **class methods** should be tested, but testing a method in isolation from the rest of the class is usually meaningless
  - **Method Testing**
- A **public method** in a class is typically tested using a **black-box** approach.
  - Start from the specification, no need to look at the method body.
  - Consider each **parameter** in the **method signature**, and identify its equivalence classes.
  - Incorporate pre-conditions and post-conditions in your test of a method.
  - Test exceptions.
- For complex logic, also use **white-box testing** or static testing.

16) What are the common guide lines for Units?

The common guidelines are that an Unit is :

(1) The smallest chunk that can be compiled by itself.
(2) A single procedure/function (Standalone)
(3) Something so small it would be developed by one person.

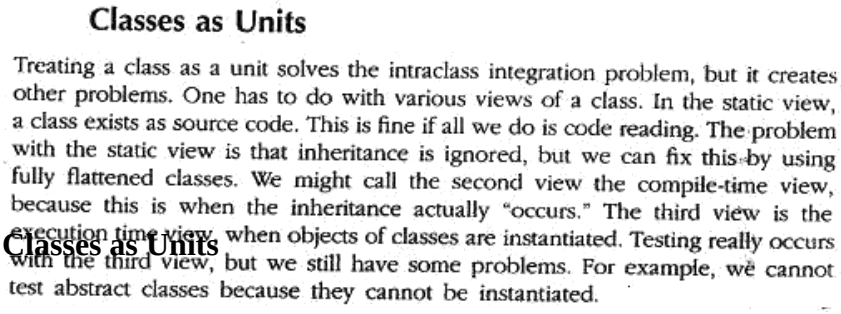
17) Discuss the merits and demerits of a) Methods as Units b) Classes as Units in OO unit testing?

(a) Methods as Units.

A method implements a single function, and it would not be assigned to more than one person, so methods might ideally be considered as units. The smallest compilation is problematic.

Methods as Units

Superficially, this choice reduces o-o unit testing to traditional (procedural) unit testing. A method is nearly equivalent to a procedure, so all the traditional functional and structural testing techniques should apply. Unit testing of procedural code requires stubs and a driver test program to supply test cases and record



Classes as Units

Treating a class as a unit solves the intraclass integration problem, but it creates other problems. One has to do with various views of a class. In the static view, a class exists as source code. This is fine if all we do is code reading. The problem with the static view is that inheritance is ignored, but we can fix this by using fully flattened classes. We might call the second view the compile-time view, because this is when the inheritance actually "occurs." The third view is the execution time view, when objects of classes are instantiated. Testing really occurs with the third view, but we still have some problems. For example, we cannot test abstract classes because they cannot be instantiated.

(b) Classes as Units

Also if we are using fully flattened classes , we will need to unflatten them to their original form when unit testing is complete.

Define Object Oriented Integration testing.

- **Interclass Testing**
- The first level of ***integration testing*** for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to "depends" relation
  - A depends on B: Build and test B, then A
- Start from use/include hierarchy
    - Implementation-level parallel to logical "depends" relation
    - Class A makes method calls on class B
    - Class A objects include references to class B methods
      - but only if reference means "is part of"

18) Compare unit testing and Integration testing in OO Testing.

OO definitions of unit and integration testing

- **Procedural software**
  - unit = *single program, function, or procedure*
    more often: a unit of work that may correspond to one or more intertwined functions or programs
- **Object oriented software**
  - unit = *class or (small) cluster of strongly related classes*
    (e.g., sets of Java classes that correspond to exceptions)
  - unit testing = ***intra-class testing***
  - integration testing = ***inter-class testing*** (cluster of classes)
  - dealing with single methods separately is usually too expensive (complex scaffolding), so methods are usually tested in the context of the class they belong to

19) Explain in detail about OO Integration testing.

Object-Oriented Integration Testing

Of the three main levels of software testing, integration testing is the least understood; this is true for both traditional and object-oriented software. As with traditional procedural software, object-oriented integration testing presumes complete unit-level testing. Both unit choices have implications for object-oriented integration testing. If the operation/method choice is taken, two levels of integration are required: one to integrate operations into a full class, and one to integrate the class with other classes. This should not be dismissed. The whole reason for the operation-as-unit choice is that the classes are very large, and several designers were involved.

Turning to the more common class-as-unit choice, once the unit testing is complete, two steps must occur: (1) if flattened classes were used, the original class hierarchy must be restored, and (2) if test methods were added, they must be removed.

Once we have our "integration test bed," we need to identify what needs to be tested. As we saw with traditional software integration, static and dynamic choices can be made.

UML Support for Integration Testing

In UML-defined, object-oriented software, collaboration and sequence diagrams are the basis for integration testing. Once this level is defined, integration-level details are added. A collaboration diagram shows (some of) the message traffic among classes. Figure 15.9 is a collaboration diagram for the ooCalendar application. A collaboration diagram is very analogous to the Call Graph

As such, a collaboration diagram supports both the pairwise and neighborhood approaches to integration testing.

With pairwise integration, a unit (class) is tested in terms of separate "adjacent" classes that either send messages to or receive messages from the class being integrated. To the extent that the class sends/receives messages from other classes, the other classes must be expressed as stubs. All this extra effort makes pairwise integration of classes as undesirable as we saw pairwise integration of procedural units to be. On the basis of the collaboration diagram in Figure 15.9, we would have the following pairs of classes to integrate:

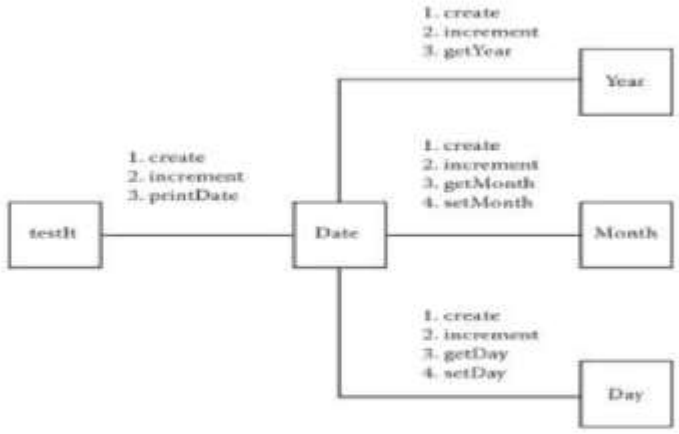


Figure 15.9 Collaboration diagram for ooCalendar.

testIt and Date, with stubs for Year, Month, and Day
Date and Year, with stubs for testIt, Month, and Day
Date and Month, with stubs for testIt, Year, and Day
Date and Day, with stubs for testIt, Month, and Year
Year and Month, with stubs for Date and Day
Month and Day, with stubs for Date and Year

A sequence diagram traces ana execution-time path through a collaboraton diagram. Thick vertical lines represent either a class or an instance of a class, and the arrows are labelled with the messsages sent by (instances of) classes in their time order. The portion of oocalendar application that prints out the new date is shown as sequence diagram as in Fig below :

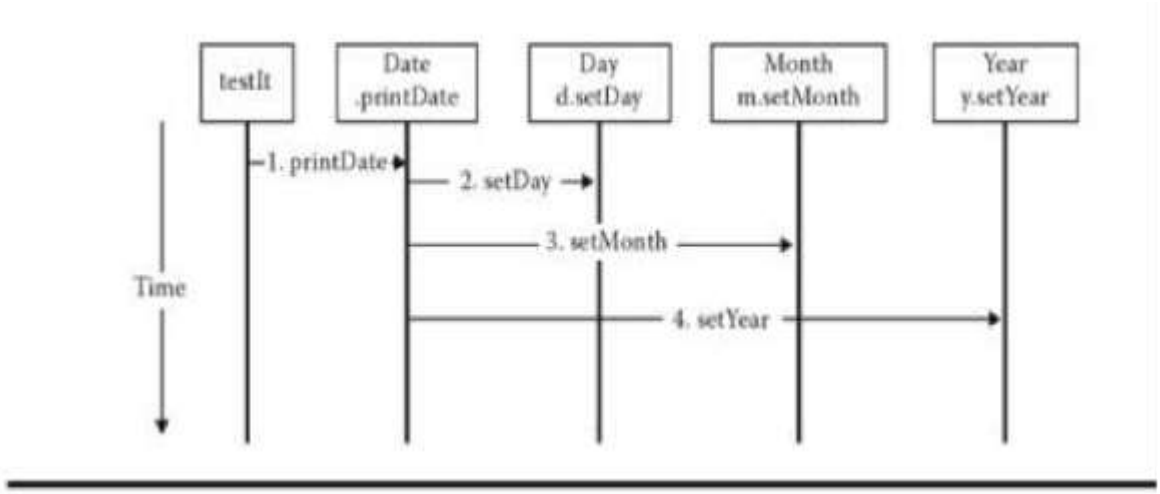


Figure 15.10 Sequence diagram for printDate.

To the extent that sequence diagrams are created, they are a reasonable basis for object-oriented integration testing. They are nearly equivalent to the object-oriented version of MM-paths (which we define in the next subsection). An actual test for this sequence diagram would have pseudocode similar to this:

```
1.     testDate
2.            d.setDay(27)
3.            m.setMonth(5)
4.            y.setYear(2013)
5.            Output ("Expected value is 5/27/2013")
6.            testIt.printDate
7.            Output ("Actual output is...")
8.     End testDate
```

Statements 2, 3, and 4 use the previously unit-tested methods to set the expected output in the classes to which messages are sent. As it stands, this test driver depends on a person to make a pass/fail judgment based on the printed output. We could put comparison logic into the testDriver class to make an internal comparison. This might be problematic in the sense that, if we made a mistake in the code tested, we might make the same mistake in the comparison logic.

20) What are MM-Paths for OO Software? Give an example.

An MM-Path in Object-Oriented Software is a sequence of method executions linked by messages.

An MM-Path starts with a method and ends when it reaches a method that does not issue any message of its own; this is the point of quiescence.

The figure below is an example of MM-Path for the collaboration diagram of OO Calendar example shown above. With this formulation we can view object oriented integration testing independently of whether the units were chosen as methods or classes.

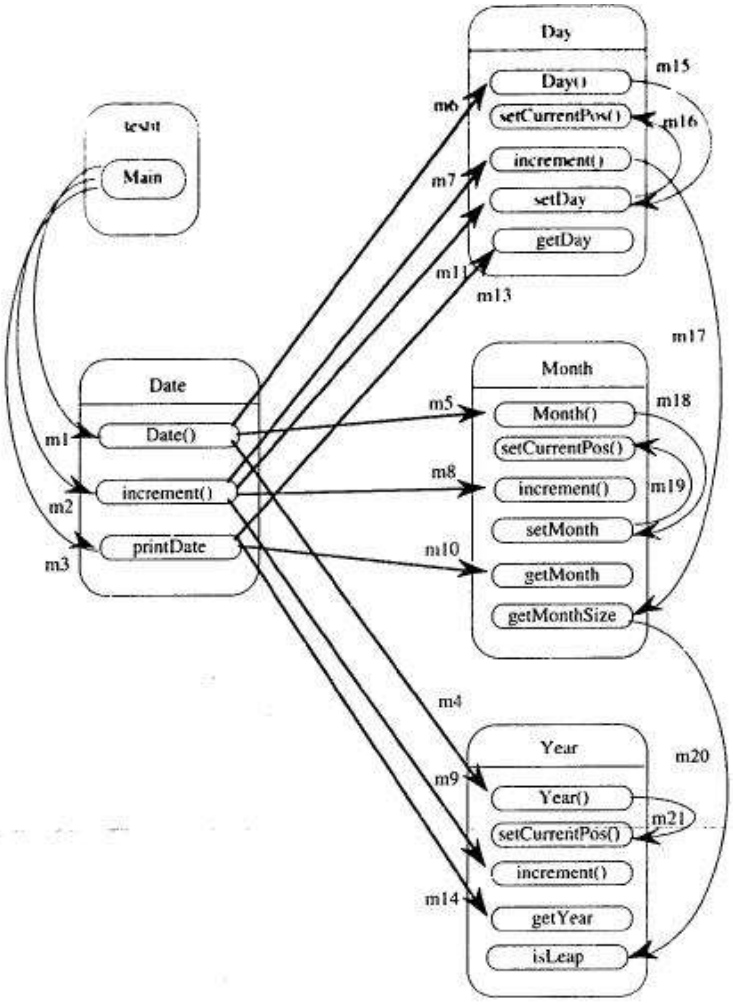


Figure 18.3 Messages in o-oCalendar.