OBJECT-ORIENTED SOFTWARE ENGINEERING

UNIT III

Design and Construction

Learning Objectives

- **Construction**: Introduction, the design model, block design, working with construction. Use case realization: the design discipline within UP iterations.
- **Designing the Subsystem**: Mapping design to code, Designing the data access layer, UI interfaces and system interfaces.
- **Reusable Design Patterns**: Importance of design patterns, Basic design patterns –Singleton, Multiton, Iterator, Adapter, Observer.
- **UML**: Communication Diagrams, Design Class Diagram, State Transition Diagram, Package Diagram, Component Diagram and Deployment Diagram

CONSTRUCTION

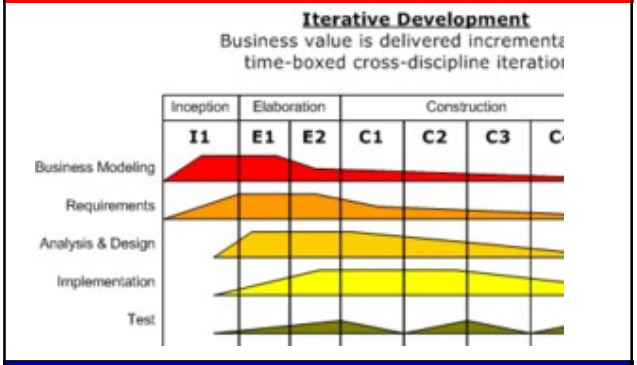
Learning Objectives

- What is Construction Phase
- Why Construction
- Add a Dimension
- Artifacts for Construction
- Design (What, Purpose, Goals, Levels)
- Implementation Environemnt
- Traceability
- Interaction Diagram
- Block design
- Block Behavior
- Implementation

What is Construction Phase?

- All about "**BUILDING**" the **system** from model of analysis & requirement phase.
- Consists of **Design** and **Implementation**.
- Start from **elaboration** & continues to **construction**.

What is Construction Phase?



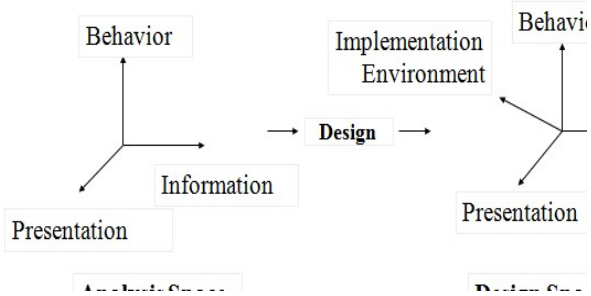
Construction Goals

- The primary goal of the **Construction phase** is to build a system capable of **operating successfully** in beta customer environments.
- During Construction, the **project team** performs tasks that involve building the system **iteratively** and **incrementally** making sure that the viability of the system is always evident in **executable form.**
- The major milestone associated with the Construction phase is called **Initial Operational Capability.**

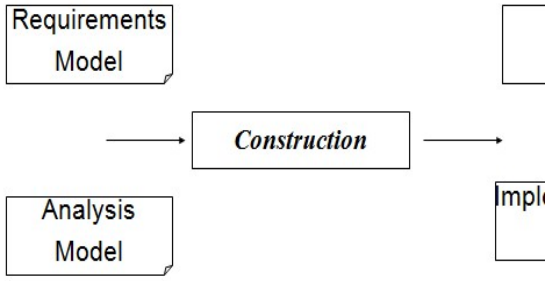
Why Construct?

- For **seamless transition** to **source code**; analysis model is not **sufficient**.
- The actual system must be **adapted** to the **implementation environment.**
- Must explore into more **dimensions.**
- To **validate** the **analysis result**.

Adding A Dimension: Analysis To Design Space



Artifacts of Construction



Design

"*There are two ways of **constructing** a software design*:

- make it so **simple** that there are obviously no deficiencies.
- make it so **complicated** that there are no obvious deficiencies."

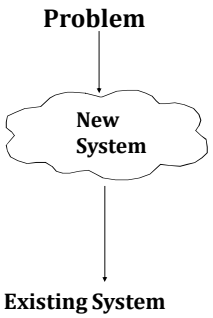
- C.A.R. Hoare

What is Design?

- **Specification** Is about **What**, and **Design** is the start of the **How**?
- **Inputs** to the ***design process***
  - Specification document, including models etc.
- **Outputs** of the ***design process***
  - A design document that describes how the code will be written.
    - What subsystems, modules or components are used
    - How these integrate (i.e. work together)
  - Information allowing **testing** of the system.

Purpose of System Design

- **Bridging** the **gap** between **desired** and **existing** system in a manageable way.
- Use *Divide and Conquer*
- We model the new system to be developed as a **set of subsystems.**



Why is Designing so difficult?

Analysis: Focuses on the application domain

Design: Focuses on the solution domain

- **Design knowledge** is a moving target
- The reasons for design decisions are changing very rapidly
  - ✓Half-time knowledge in software engineering
  - ✓Things will be out of date in 3 years
  - ✓Cost of hardware rapidly sinking

Design Goals

Qualities of a Good Design:

- Correct
- Complete
- Changeable
- Efficient
- Simple

Correctness:

- It Should Lead To A **Correct Implementation**

Completeness:

- It Should Do Everything. Everything? It should follow the **specifications**.

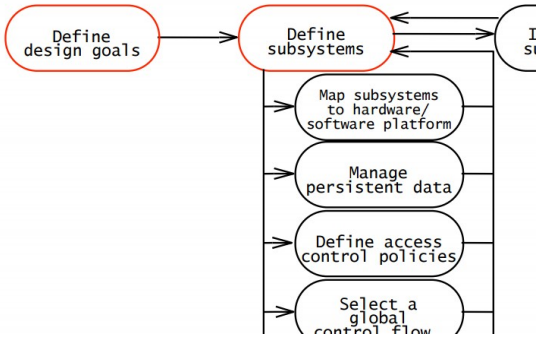
Design Goals

Changeable:
It Should **Facilitate Change**—Change Is Inevitable

Efficiency
- It Should **Not Waste Resources**.
- Better is a **Working Slow Design** Than a Fast Design That Does Not Work.

Simplicity
- It Should Be As **Understandable** As Possible.
- Designs are **blue-prints** for code construction.

Design Goals to Sub-systems



Levels of Design

Three possible levels:
- **System Design**,
  - Part of Systems Engineering.
- **High-level Software Design**
  - Architecture, architectural design.
- **Low-level Software Design**
  - Detailed Design, Module Design.

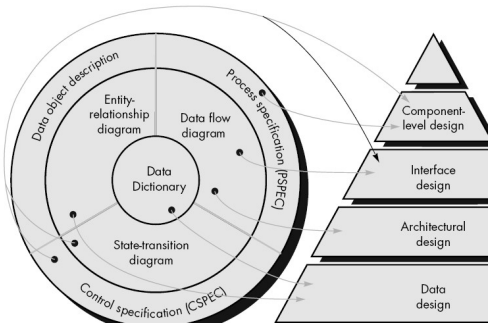
Develop the Design Model

- Create detailed "**plans**" (like **blueprints**) for **implementation.**
- Identify the "*Implementation Environment*" & draw **conclusions**.
- Incorporate the conclusions & develop a "*First approach to a design model*" from requirement models.
  - Use **analysis model** as **base** & translate analysis objects to design objects in design model fit for current implementation
  - Why can't this be **incorporated** in analysis model?

Develop the Design Model

- Describe how the "*Object Interact*" in each specific use case & how **stimuli** between objects is **exchanged**.
- Create **design models** before **coding** so that we can:
  - Compare different possible **design solutions**
  - Evaluate **efficiency**, **ease of modification**, **maintainability**, etc.

Analysis to Design Model

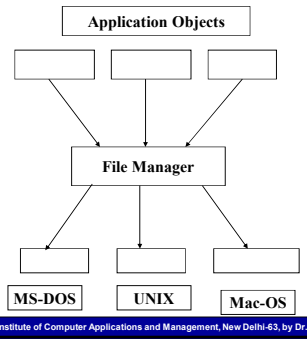


Implementation Environment

- Identify the actual technical constraints under which the system should be built like
  - The **target environment**
  - **Programming language**
  - **Existing products** that should be used (DBMSs, etc)

- **Strategies**:
  - *As few objects as possible* should be aware of the **constraints** of the **actual implementation environment.**

Implementation Env. : Target Env.



Several implementations of the file manager block

Implementation Environment

Target environment

- Create a new blocks that represent occurre changed parts in the target environment

Strategies:

- Specified an abstract class
  - ✓polymorphism
- The object can check the platform at run-time
  - ✓CASE statement in the source code
- Decide this when the system us delivered
  - ✓Provide several different modul which will be ch

- **Investigate whether the target environment will**

Implementation Environment

Programming language

- Affect the design in translating the concepts us
- The basic properties of the language and its are fundamental for the design
  - ✓Inheritance and Multiple inheritance
  - ✓Typing
  - ✓Standard
  - ✓Portability
  - ✓Strategies for handling errors during run-tim
    - **Exception (Ada)**
    - **Assertions (Eiffel)**
  - ✓Memory management

Implementation Environment

Using existing products

- DBMS
- UIMS (User Interface Management System)
- Network facilities
- Internally or externally developed applications tha incorporated
- Products used during development
  - ✓**Compilers**
  - ✓**Debuggers**
  - ✓**Preprocessor**

Other considerations

Implementation Environment

- Other considerations
  - Strategies:
    - ✓To postpone optimizations until they are neede absolutly sure that they will be needed
      - the real bottlenecks are often missed ar optimizations are necessary
  - Use simulation or prototyping to investiga optimization problem early
    - ✓Extensive experiences may help to jugde at an
- If you're not sure of the correctnessof a

Implementation Environment

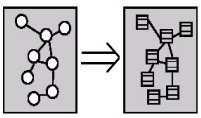
- The people and organization involvec development could also afect the design
  - The principal strategy:
    - ✓such factors should not affect the systen
    - ✓The reason: the circumtances (org staffing, competence areas) that are in c

Traceability

- **Refines** the **analysis model** in light of actual *implementation environment*.
- **Explicit definition** of interfaces of objects, semantics of operation. Additionally, different issues like DBMS, programming language etc. can be considered.
- The model is composed of "**BLOCKS** " which are the **design objects.**
- *One block is implemented as one class.*

Traceability

- The **blocks** abstract the **actual implementation**.
- **Traceability** is extremely important aspect of the system.
  - **Changes** made will be only local to a module.
  - Provides high **functiona localization** (high cohesion).



Analysis Model: *logical, conceptual, frozen*

Design Model: *a practical abstraction*

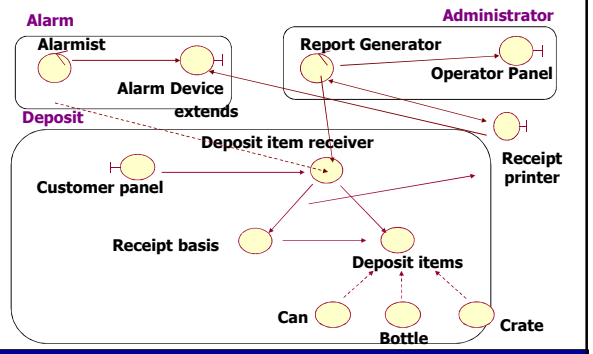
Traceability Matrix

- A **traceability matrix** is a **document**, usually in the form of a **table**, used to assist in **determining** the **completeness of a relationship** by **correlating** any **two baseline documents** using a many-to-many relationship comparison.
- It is often used with **high-level requirements** (these often consist of marketing requirements) and **detailed requirements** of the product to the matching parts of h**igh level design, detailed design, test plan**, and **test cases.**
- A requirements traceability matrix may be used to check to see if the current project requirements are being met, and to help in the creation of a request for proposal, software requirement specification various deliverable documents, and project plan tasks.

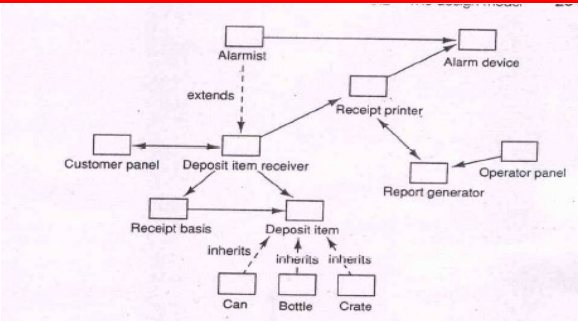
Traceability Matrix

Requirement Identifiers	Reqs Tested	REQ1 UC 1.1	REQ1 UC 1.2	REQ1 UC 1.3	REQ1 UC 2.1	REQ1 UC 2.2	REQ1 UC 2.3.1	REQ1 UC 2.3.2	REQ1 UC 2.3.3	REQ1 UC 2.4	REQ1 UC 3.1	REQ1 UC 3.2	R T
Test Cases	321	3	2	3	1	1	1	1	1	1	2	3	
Tested Implicitly	77												
1.1.1	1	x											
1.1.2	2		x	x									
1.1.3	2	x											
1.1.4	1			x									
1.1.5	2	x											
1.1.6	1		x										
1.1.7	1			x									
1.2.1	2				x		x						
1.2.2	2					x		x					
1.2.3	2								x	x			
1.3.1	1										x		
1.3.2	1										x		
1.3.3	1											x	

Recycling Machine Analysis Model



Recycling Machine Design Model



Working with Design Model

- **Changes** can and should occur, but all changes should be **justified** and **documented** (for *robustness reason*).
- We may have to change the **design model** in various way:
  - To **introduce new blocks** which don't have any representation in the analysis model.
  - To **delete blocks** from the design model.
  - To **change blocks** in the design model (splitting and joining existing blocks).
- To **change** the **associations** between the **blocks** in the design model.

Working with Design Model

- Changes can and should occur, but all changes should be justified and documented (for **robustness** reason).
- We may have to change the design model in various way:
  - To introduce **new blocks** which don't have any representation in the analysis model.
  - To **delete blocks** from the design model.
  - To **change blocks** in the design model (splitting and joining existing blocks).

Change in Environment

- Changing the associations between the blocks in the design model.
  - extensions to stimuli.
  - inheritance to delegation.

Interaction Diagram

- The interaction diagram describes how each **use case** is offered by **communicating objects**
  - ✓The diagram shows how the ***participating objects realize*** the **use case** through their **interaction**
  - ✓The **blocks** send **stimuli** between **one another**
  - ✓All stimuli are **defined** including their **parameters**
- For each **concrete use case**, we draw an **interaction diagram**

Interaction Diagram

- An interaction diagram shows an interaction,
- consisting of a set of objects and their relations
- include the messages that may be exchange them
- Model the dynamic aspect of the system
- Contain two sort of diagrams:
  - **Sequence diagrams**,
    - ✓**show the messages objects send to each timely manner**
  - **Collaboration diagrams**,

Interaction Diagram

- Using interaction diagrams, we can **clarify** the **sequence of operation calls** among **objects** used to complete a single use case
- Collaborations have the added advantage of **interfaces** and **freedom of layout**, but can be difficult to follow, understand and create.
- Interaction diagrams are used to diagram a **single use case**.
- When you want to examine the **behaviour** of a **single instance** over time use a **state diagram**, and if you want to look at the **behaviour** of the system over time use an activity diagram.

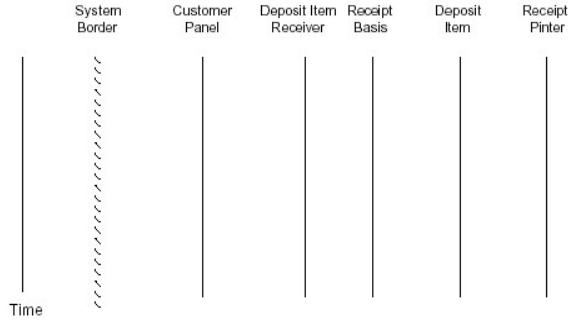
Building an Interaction Diagram

- **Identify blocks**
- Draw **skeleton**, consist of:
  - **System border**
  - **Bars** for each block that participates
- Describes the **sequences**
  - **Structured text** or **pseudo-code**
- Mark the bar to which **operations** belongs with a **rectangle representing operation**
- Define a **stimulus**

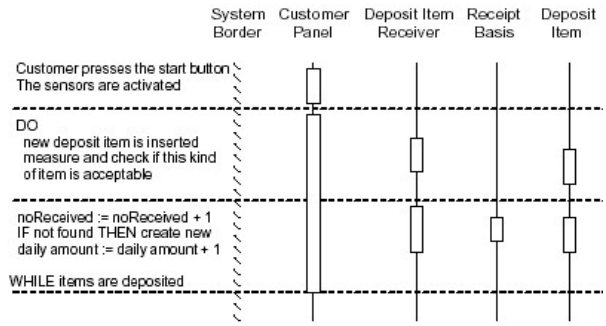
Building an Interaction Diagram

- Draw a **stimulus** as a **horizontal arrow**
  - **Start**: bar of the sending block
  - **End**: bar of the receiving block
- **Structure** the interaction diagram
  - **Fork diagram**
  - **Stair diagram**

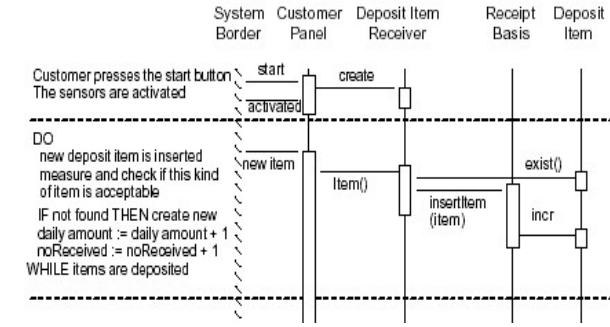
Skeleton of Interaction Diagram



Interaction Diagram for returning item use case



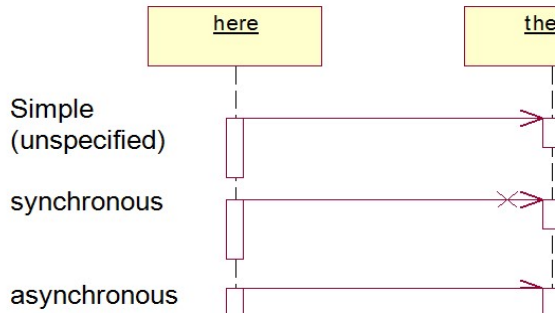
Interaction Diagram for returning item use case



Advanced Interaction Diagram

- A **synchronous message/signal i**s a control which has to *wait for an answer before continuing.*
  - The sender passes the control to the receiver and cannot do anything until the receiver sends the control back.
- An **asynchronous message** is a control which *does not need to wait before continuing.*
  - The sender actually does not pass the control to the receiver.
  - The sender and the receiver carry on their work **concurrently**.

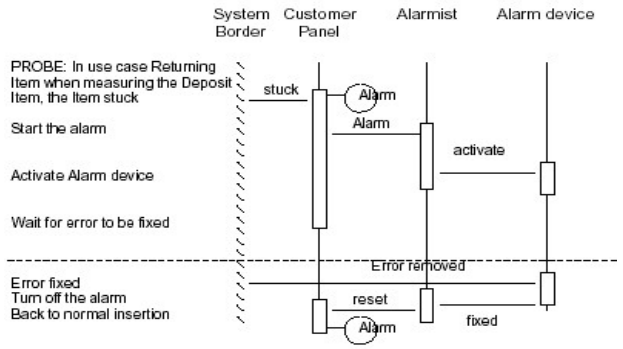
Synchronous and Asynchronous



Probe Condition

- Use case with extension is described by a probe position in the interaction diagram
- The probe position *indicates a position in the use case to be extended*
  - Often accompanied by a **condition** which indicates under what **circumstances** the extension should take place

Example of a Probe Condition



Homogenization

- In **parallel design process**, several **stimuli** with the **same purpose** or meaning are defined by several designers.
- These stimuli should be **consolidated** to obtain as few stimuli as possible.
  - Called *homogenization*.

Example-Homogenization

What_is_your_phone_number?
Where_do_you_live?
Get_address
Get_address_and_phone_number

Homogenized into:

- Get_address
- Get_phone_number

Sequence Diagram

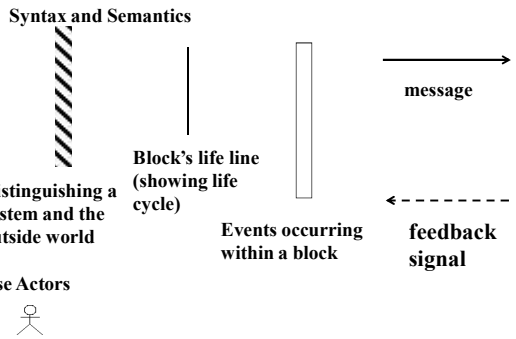
- The sequence diagram describes the flow of messages being passed from object to object.

The **purposes** of **interaction diagram** can be describes as:
- To capture **dynamic behavior** of a system.
- To describe the **message flow** in the system.
- To describe **structural organization** of the objects.
- To describe **interaction** among objects.

Sequence Diagram Elements

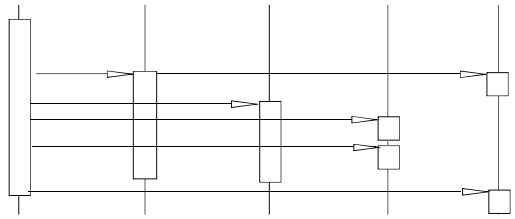
- **Class roles**, which represent **roles** that **objects** may play within the **interaction**.
- **Lifelines**, which represent the **existence** of an **object** over a period of time.
- **Activations**, which represent the **time** during which an object is performing an **operation**.
- The white rectangles on a **lifeline** are called **activations** and indicate that an object is ***responding to a message***. It starts when the message is received and ends when the object is done handling the message.
- **Messages**, which represent **communication** between **objects**.

Sequence Diagram



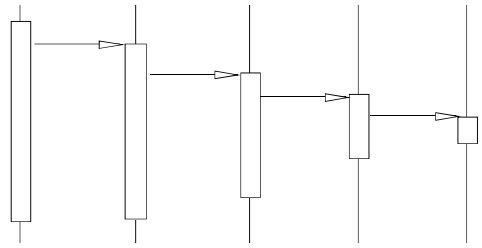
Sequence Diagram- Fork Structure

- **Centralised structure -- Fork: Everything is handled and controlled by the left-most block.**



Sequence Diagram- *Structure*

- **Decentralised structure -- Stair: There is no central control block.**

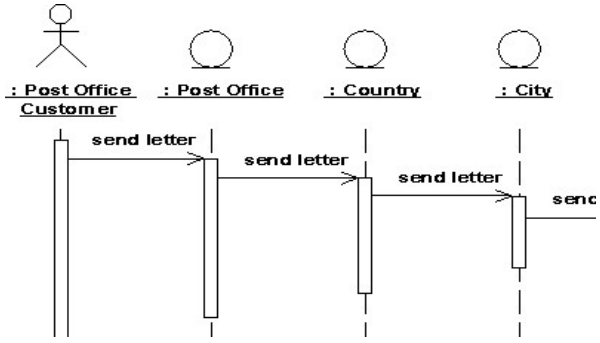


Structure of Sequence Diagram

Decentralized structure is appropriate:

- If the **sub-event phases** are tightly coupled. This will be the case if the participating objects:
  - Form **a part-of** or **consists-of** hierarchy, such as Country - State - City;
  - Form an **information hierarchy**, such as CEO - Division Manager - Section Manager;
  - Represent a **fixed chronological progression** (the sequence of sub-event phases will always be performed in the same order), such as Advertisement - Order - Invoice -Delivery - Payment; or
  - Form a **conceptual inheritance hierarchy**, such as Animal - Mammal - Cat.

Structure of Sequence Diagram

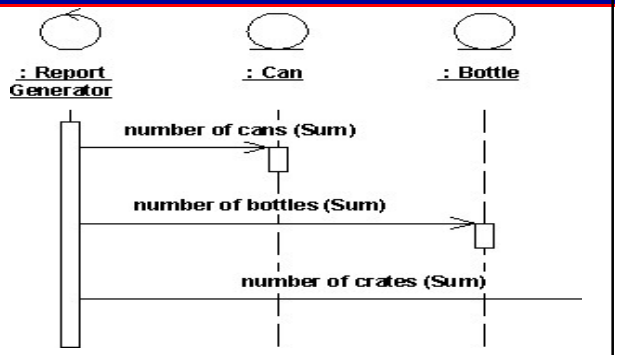


Structure of Sequence Diagram

A centralized structure is appropriate:

- If the order in which the **sub-event phases** will be performed is likely to change.
- If you expect to **insert** new **sub-event phases**.
- If you want to keep parts of the functionality **reusable** as separate piece

Structure of Sequence Diagram



Structure of Sequence Diagram

Fork

- Indicates a **centralized structure** and is characterized by the fact that it is an object controls the other objects interacted with it.
- This structure is appropriate when:
  - The operations can change order
  - New operations could be inserted

Structure of Sequence Diagram

Stair

- Indicates **decentralized structure** and is characterized by **delegated responsibility.**
- Each object only knows a few of the other objects and knows which objects can help with a **specific behavior**.
- This structure is appropriate when:
- The operation have a **strong connection**. Strong connection exists if the objects:
  - form a *'consist-of'* **hierarchy**
  - form an information hierarchy
  - form a fixed temporal relationship
  - form a (conceptual) inheritance relationship
- The operation will always be performed in the same order

Structure Control in Sequence Diagram

- **Optional Execution**
- **Conditional Execution**
- **Parallel Execution**
- **Loop Execution**
- **Nested**

Sequence Diagram

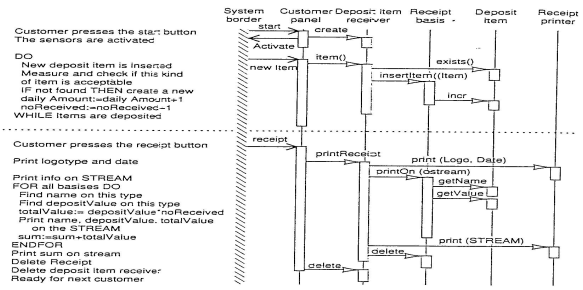


Figure 8.11 Interaction diagram for the use case *Returning Item*.

Block Design

- Block design can start when all the **block** have been identified.
- For block designing it is important to **identify** the **interface and operation** of **each block**.
- The **implementation** (code) for the block can start when the interfaces are **stable** and are **frozen**.
- When the implementation of the block starts, normally **ancestor block** should be implemented prior to **descendent blocks.**

Ex : the deposit item will design prior to can & bottle.

Block Design

- By taking **INTERACTION diagrams** where a block participates & extracting all the operation defined on that block.
- Using this diagram we are clear about the **interface** of the each block..
- The **interface** for **Deposit Item**: exists, incr, getName, getValue

Block Design Comments

- The description of the operation is extracte text to the left of the diagram.
- Can work in parallel once interfaces are fro the open closed principle).

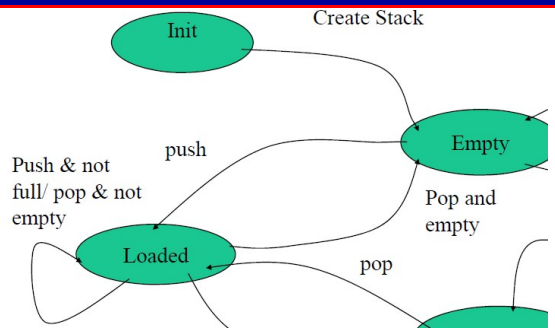
Object Behavior

- An **intermediate level** of **object internal behavior** may be described using a **state machine**.
- To provide a **simplified description** that increases understanding of the block without having to go down to source code.
- State represents **modes of operations** on object.
- **Less dependant** on programming language.
- This is particularly important in **reactive systems**.

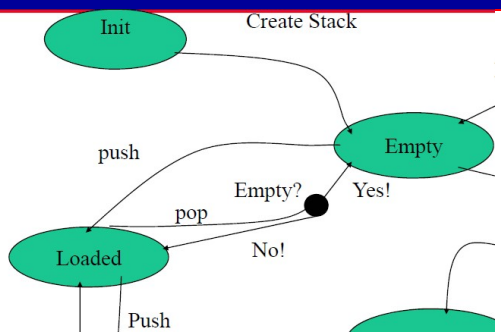
Object Behavior

```
Machine stack
   State init
   input createinstance
   nextstate empty
   otherwise error;
State empty
   input push
   do store on top
   nextstate loaded
   otherwise error;
   ..
endmachine
```

State transition diagram of a stack



State transition diagram of a stack



Stimulus Control Object

Stimulus Control Object

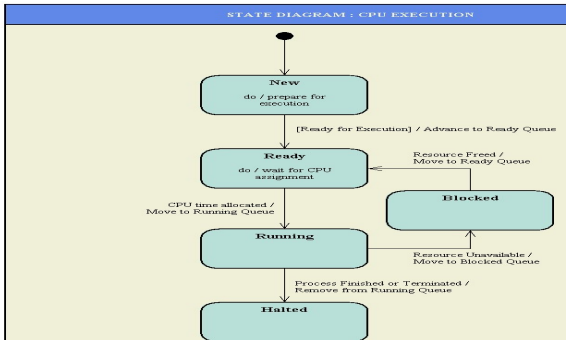
- An object that perform the same operation independent of state when a certain stimulus is received.
- Entity objects

State Controlled Object

State Control Object

- Objects that select operations not only from the stimulus received, but also from the current state
- Control object.

State Controlled Object



Internal Block Structure

- In case of OOPL object-module becomes classes otherwise module unit
- Generally more classes than object
- split class when required
- 5-10 times longer to design a component class than an ordinary class

Implementation

- Now, need to write code for each block.
- Implementation strategy depends on the programming language.
- In an **OOP** language, the implementation of a block starts with one class.
- Sometimes there is a need for additional classes, that are not seen by other blocks.

Mapping

Analysis	Design	Source code C++
Analysis objects	Block	1..N classes
Behavior in objects	Operations	Member functions
Attributes(class)	Attributes(class)	Static variables
Attributes(instance)	Attributes(instance)	Instance variables
Interaction between objects	Stimulus	Call to a function
Use case	Designed use case	Sequence of calls

Implementation Environment

Everything that does not come from analysis phase, including performance requirements.

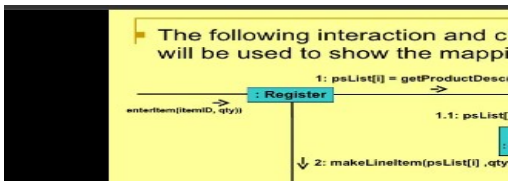
- Design must be adapted to implementation environment.
- Use of existing products must be decided. Includes previous version of the system.
- To use an existing product we must adapt our design.
- Tradeoff - less development vs. more complex architecture.
- Also consider testing costs.

Other Considerations in Construction

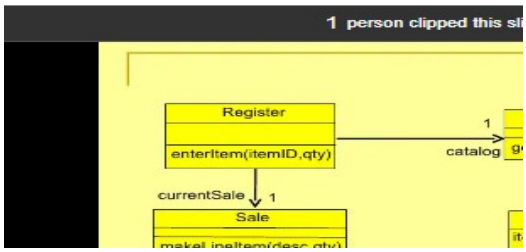
- Subsystems defined in analysis phase are used to guide the construction phase.
- Developed separately as much as possible.
- Incremental development - start construction phase in parallel with analysis phase - to identify implementation environment.
- How much refinement to do in analysis phase? (How early/late to move from analysis to design) - decided in each project

Designing the Subsystem

- The interaction diagrams and the design class diagrams created during design provide some of the necessary input for generating code.
- We now see how to map those artifacts to code in an object-oriented language. The following interaction and class diagram will be used to show the mapping process.



Design Class Diagram



Designing the Data Access Layer

- Design access layer
  - Create mirror classes: For every busine identified and created, create one acces Eg , if there are 3 business classes (clas class2 and class3), create 3 access laye classes (class1DB, class2DB and class3
  - Identify access layer class relationships
  - Simplify classes and their relationships - eliminate redundant classes and structu
    - Redundant classes: Do not keep 2 classes perform similar translate request and transla

Table-Class Mapping

- A tool to map relational data with objects sho following mapping capabilities: (all are two m
- Table-class mapping
- Table-multiple classes mapping
- Table-inherited classes mapping
- Tables-inherited classes mapping
- The tool must describe both how the foreign used to navigate among classes and instanc

Table-Class Mapping

- It is a simple one-to-one mapping of a class and the mapping of columns in a properties in a class. Here we map all to properties. But it is more efficient to those columns for which an object mod required by the application(s). Here ea the table represents an object instance column in the table corresponds to an

Table-Multiple Classes Ma

- Here a single table maps to multiple noninhe classes. Two or more distinct, noninheriting c have properties that are mapped to columns i table. At run time, mapped table row is acces instance of one of the classes, based on a co in the table.
- Table-Inherited Classes Mapping
- Here, a single table maps to many classes th

Tables-Inherited Classes M

- This mapping allows the translation of is-a r that exist among tables in the relational sch class inheritance relationships in the object relational database, an is-a relationship ofte by a primary key that acts as a foreign key t table. In the object-model, is-a is another te inheritance relationship.
- Keys for Instance Navigation
- In mapping columns to properties, the simpl is to translate a column's value into the corr class property value. Here either the colum value or it defines a navigable relationship b

REUSABLE DESIGN PATTERNS

The Beginning of Patterns

- Christopher Alexander, architect
  - A Pattern Language--Towns, Buildings, Construction
  - Timeless Way of Building (1979)
  - "Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- Other patterns: novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)

"Gang of Four" (GoF) Book

- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994
- Written by this "gang of four"
  - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
  - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
  - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
  - Dr. John Vlissides, then a researcher at IBM
    - ✓ Thomas J. Watson Research Center
    - ✓ See John's WikiWiki tribute page http://c2.com/cgi/wiki?JohnVliss

Object-Oriented Design Patterns

- This book defined 23 patterns in three categories
  - *Creational patterns* deal with the process of object creation
  - *Structural patterns*, deal primarily with the static composition and structure of classes and objects
  - *Behavioral patterns*, which deal primarily with dynamic interaction among classes and objects

Documenting Discovered Patterns

- Many other patterns have been introduced documented
  - For example, the book **Data Access Patterns** by Clifton Nock introduces 4 decoupling patterns, 5 resource patterns, 5 I/O patterns, 7 cache patterns, and 4 concurrency patterns.
  - Other pattern languages include telecommunications patterns, pedagogical patterns, analysis patterns

GoF Patterns

- *Creational Patterns*
  - ✓ Abstract Factory
  - ✓ Builder
  - ✓ Factory Method
  - ✓ Prototype
  - ✓ Singleton
- *Structural Patterns*
  - ✓ Adapter
  - ✓ Bridge
  - ✓ Composite
  - ✓ Decorator
  - ✓ Façade
  - ✓ Flyweight
  - ✓ Proxy
- *Behavioral Patterns*
  - ✓ Chain of Responsibility
  - ✓ Command
  - ✓ Interpreter
  - ✓ **Iterator**
  - ✓ Mediator
  - ✓ Memento
  - ✓ Observer
  - ✓ State
  - ✓ **Strategy**
  - ✓ Template Method
  - ✓ Visitor

Why Study Patterns?

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, "We could use Strategy here."
- Provide a higher level prospective
  - Frees us from dealing with the details too early

Other advantages

- Most design patterns make software more modifiable, less brittle
  - we are using time tested solutions
- Using design patterns makes software systems easier to change—more maintainable
- Helps increase the understanding of basic object-oriented design principles
  - encapsulation, inheritance, interfaces, polymorphism

Style for Describing Patterns

- We will use this structure:
  - *Pattern name*
  - *Recurring problem:* what problem the pattern addresses
  - *Solution:* the general approach of the pattern
  - *UML for the pattern*
    - ✓*Participants:* a description as a class diagram
  - *Use Example(s):* examples of this pattern, in Java

A few OO Design Patterns

- **Coming up:**
- **Singleton**
- **Multiton**
  - **Iterator**
    - ✓access the elements of an aggregate object sequentially without exposing its underlying representation
  - **Adaptor**
    - ✓A means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable
  - **Observer**
    - ✓One object stores a list of observers that are updated when the state of the object is changed

Singleton Pattern

6 people clipped this slide

Definition

- Ensures that a class has only one instance, and pr to access this instance.
- In other words, a class must ensure that only singl created and single object can be used by all other

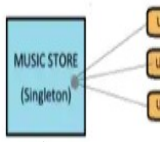
3 people clipped this slide

Advantages and Usage

- Advantages
  - Saves memory because object is not created at each r instance is reused again and again.
  - In cases when object creation is very costly (time takin create new object each time we need it. We just acce

Usage Example

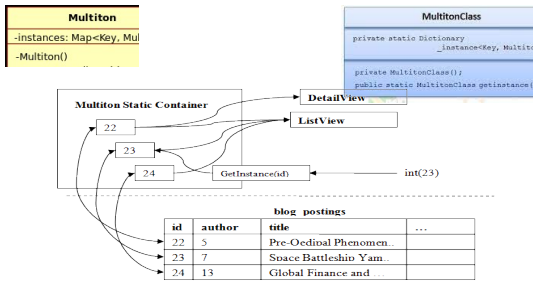
Usage Example



Multiton Pattern

- The **multiton pattern** is a design pattern which generalizes the singleton pattern. Whereas the singleton allows only one instance of a class to be created, the multiton pattern allows for the controlled creation of multiple instances, which it manages through the use of a map.
- Rather than having a single instance *per application* (e.g. the java.lang.Runtime object in the Java programming language) the multiton pattern instead ensures a single instance *per key*.
- *Drawback:* This pattern, like the Singleton pattern, makes unit testing far more difficult, as it introduces global state into an application.
- With garbage collected languages it may become a source of memory leaks as it introduces global strong references to the objects.

Example



Pattern: *Iterator*

- Name: Iterator (a.k.a Enumeration)
- Recurring Problem: How can you loop over all objects in any collection. You don't want to change client code when the collection changes. Want the same methods
- Solution: 1) Have each class implement an interface, and 2) Have an interface that works with all collections
- Consequences: Can change collection class details without changing code to traverse the collection

GoF Version of Iterator page 257

ListIterator
First()
Next()
IsDone()
CurrentItem()

```
// A C++ Implementation
ListIterator<Employee> itr = list.iterator();
for(itr.First(); !itr.IsDone(); itr.Next()) {
   cout << itr.CurrentItem().toString();
```

Java version of Iterator

interface Iterator

boolean hasNext()

Returns true if the iteration has more elements.

Object next()

Returns the next element in the iteration and updates the iteration to refer to the next (or have hasNext() return false)

void remove()

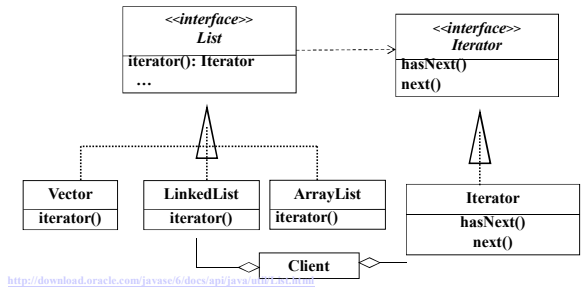
Removes the most recently visited element

Java's Iterator interface

```
// The Client code
List<BankAccount> bank =
                    new ArrayList<BankAccount>();
bank.add(new BankAccount("One", 0.01) );
// ...
bank.add(new BankAccount("Nine thousand", 9000.00));

String ID = "Two";
Iterator<BankAccount> itr = bank.iterator();
while(itr.hasNext())  {
   if(itr.next().getID().equals(searchAcct.getID()))
     System.out.println("Found " + ref.getID());
}
```


UML Diagram of Java's Iterator with a few Collections



http://download.oracle.com/javase/6/docs/api/java/util/Iterator.html

The Observer Design Pattern

- Name: Observer
- Problem: Need to notify a changing number of objects that something has changed
- Solution: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Examples

- From Heads-First: Send a newspaper to all who subscribe
  - People add and drop subscriptions, when a new version comes out, it goes to all currently described
- Spreadsheet
  - Demo: Draw two charts—two views--with some changing numbers--the model

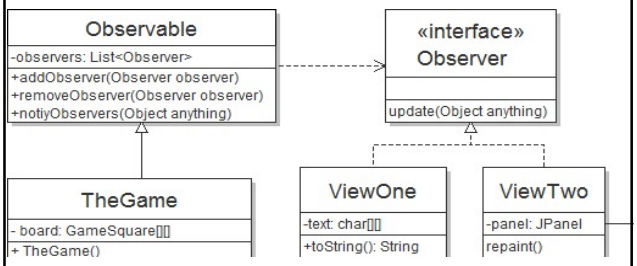
16-107

Examples

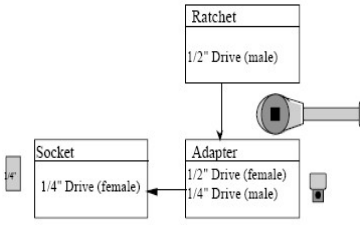
- File Explorer (or Finders) are registered observers (the view) of the file system (the model).
- Demo: Open several finders to view file system and delete a file
- Later in Java: We'll have two views of the same model that get an update message whenever the state of the model has changed

16-108

Observer Example



Adapter (Non software example)



Convert the interface of a class into another Interface clients expect.

Structural

Adapter (Software counterpart)

3 basic building blocks of UML - Diagrams

Graphical representation of a set of elements.

Represented by a connected graph: Vertices are things; Arcs are relationships/behaviors.

5 most common views built from UML 2.0: 12 diagram types

Structural Diagrams

Represent the static aspects of a system.

- Class; Object
- Component
- Deployment

Structural Diagrams

- Class; Object
- Component
- Deployment
- Composite Structure
- Package

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence; Collaboration
- Statechart
- Activity

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence; *Communication*

Class Diagrams

Structural Diagrams

- **Class;** Object
- Component
- Deployment
- Composite Structure
- Package

Class Diagram

The basis for all object modeling
All things lead to this

- **Most common diagram.**
- **Shows a set of classes, interfaces, and collaborations and their relationships (dependency, generalization, association and realization); notes too.**
- **Represents the static view of a system (With active classes, static process view)**

Three modeling perspectives for Class Diagram

- ***Conceptual*: the diagram reflects the domain**
- ***Specification*: focus on interfaces of the software (Java supports interfaces)**
- ***Implementation*: class (logical database schema) definition to be implemented in code and database.**

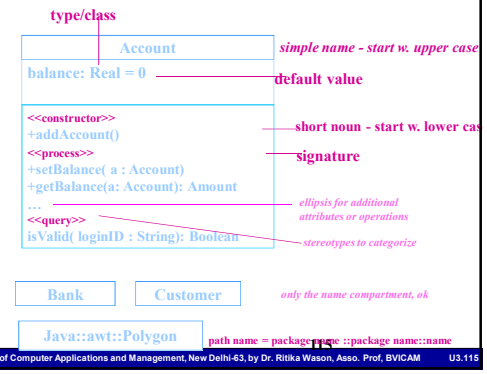
Most users of OO methods take an implementation perspective, which is a shame because the other perspectives are often more useful. -- Martin Fowler

Classes

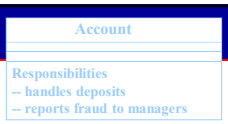
Names

Attributes

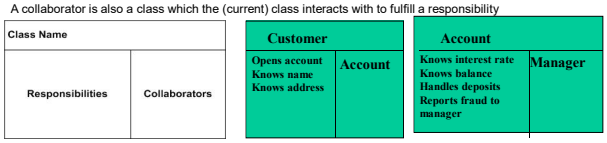
Operations
may cause object to change state



Responsibilities

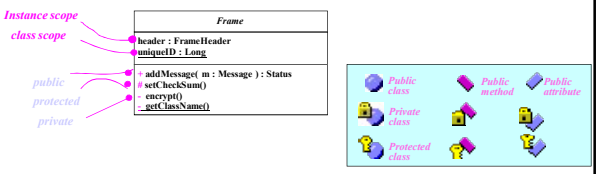


- anything that a class knows or does (Contract or obligation)
- An optional 4th item carried out by attributes and operations.
- Free-form text; one phrase per responsibility.
- Technique - CRC cards (Class-Responsibility-Collaborator); Kent Beck and Ward Cunningham'89
- A collaborator is also a class which the (current) class interacts with to fulfill a responsibility



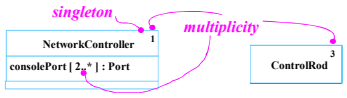
Scope & Visibility

- *Instance* Scope — each instance of the classifier holds its own value.
- *Class* Scope — one value is held for all instances of the classifier (underlined).

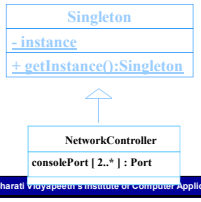


- Public - access allowed for any outside classifier (+).
- Protected - access allowed for any descendant of the classifier (#).
- Private - access restricted to the classifier itself (-).
- (using adornments in JBuilder)

Multiplicity



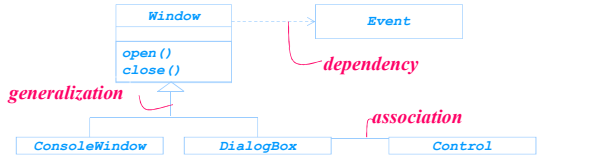
Using Design Pattern

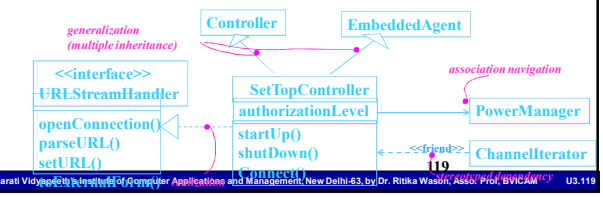



```
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {}
  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```


Relationships

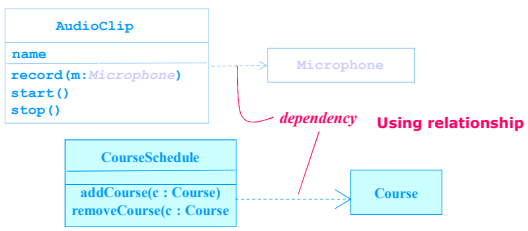




119

Dependency

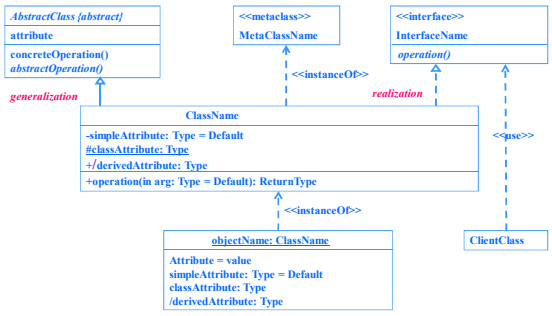
- A change in one thing may affect another.
  - **The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.**



Usually initial class diagrams will not have any significant number of dependencies in the beginning of analysis but will as more details are identified.

120

Dependency – Among Classes

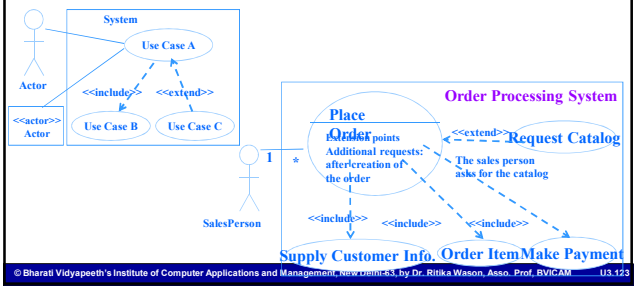


Dependency –Among Classes

- Eight Stereotypes of Dependency Among Classes
  - *bind*: the source instantiates the target template using the given actual parameters
  - *derive*: the source may be computed from the target
  - *friend*: the source is given special visibility into the target
  - ***instanceOf***: the source object is an instance of the target classifier
  - *instantiate*: the source creates instances of the target
  - *powertype*: the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
  - *refine*: the source is at a finer degree of abstraction than the target
  - *use*: the semantics of the source element depends on the semantics of the public part of the target

Dependency –Among Use Cases

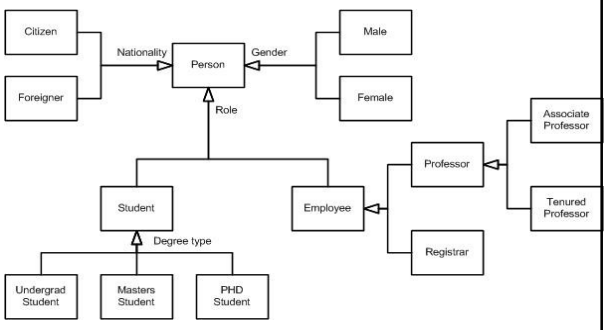
- Two Stereotypes of Dependency Among Use Cases:
  - *extend*: the target use case extends the behavior of the source
  - *include*: the source use case explicitly incorporates the behavior of another use case *at a location* specified by the source



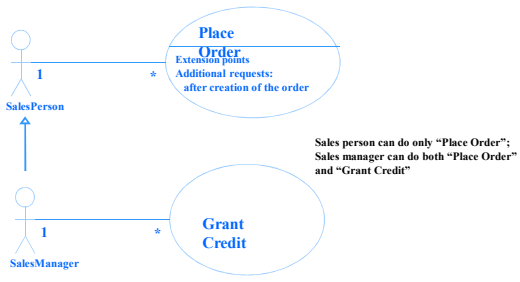
Generalization

- Four Standard Constraints
  - *complete*: all children in the generalization have been specified; no more children are permitted
  - *incomplete*: not all children have been specified; additional children are permitted
  - *disjoint*: objects of the parent have no more than one of the children as a type
  - *overlapping*: objects of the parent may have more than one of the children as a type
- One Stereotype
  - *implementation*: the child inherits the implementation of the parent but does not make public nor support its interfaces

Generalization – Along Roles

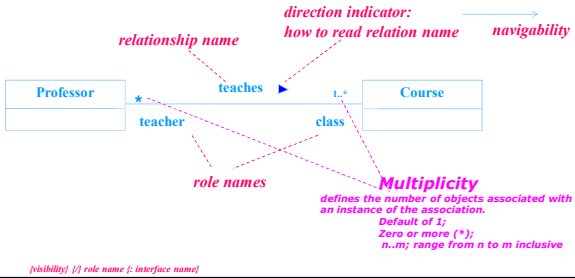


Generalization –Among Actors



Associations

- **Represent conceptual relationships between classes**
  (cf. dependency with no communication/message passing)

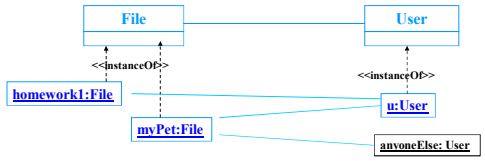


Associations – A Question

- How would you model the following situation?

"You have two files, say homework1 and myPet, where homework1 is read-accessible only by you, but myPet is write-accessible by anybody."

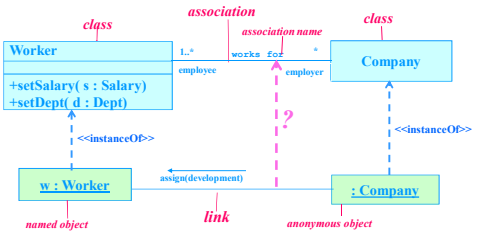
You could create two classes, File and User.
Homework1 and MyPet are files, and you are a user.



Approach 1: Now, would you associate the file access right with File?
Approach 2: Or, would you associate the file access right with User?

Associations – Links

- link is a semantic connection among objects.
- A link is an instance of an association.

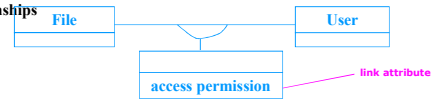


Association generalization is not automatic, but should be explicit in UML

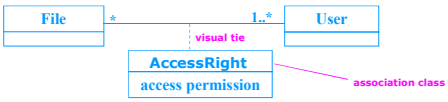
Associations – Link Attributes

- **Link Attributes**

The most compelling reason for having link attributes is for-many-to-many relationships



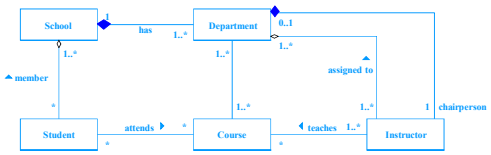
- **Association Class**



- **With a refactoring**

Modeling Structural Relationships

- Considering a bunch of classes and their association relationships



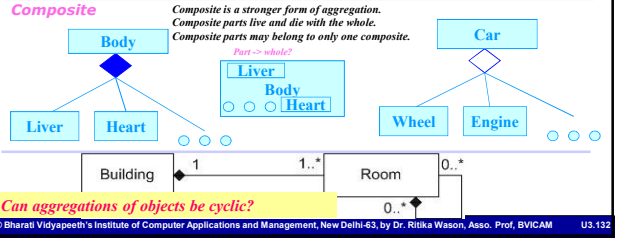
The model above is from Rational Rose. How did the composite symbol ( )get loaded versus the aggregation? **Use the Role Detail and select aggregation and then the "by value" radio button.**

Modeling Structural Relationships

Aggregation - structural association representing "whole/part" relationship.
- "has-a" relationship.

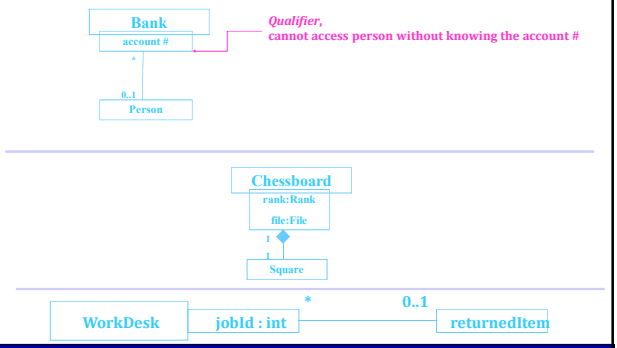
Composite

Composite is a stronger form of aggregation.
Composite parts live and die with the whole.
Composite parts may belong to only one composite.

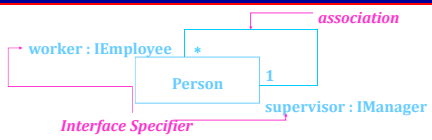


Can aggregations of objects be cyclic?

Association – Qualification



Association – Interface Specifier

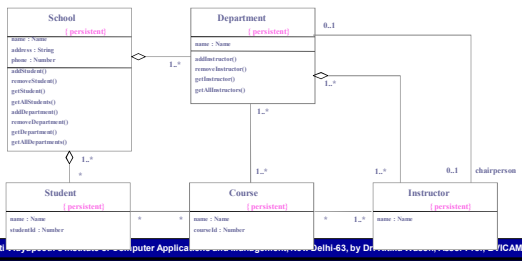


Realization

- **A semantic relationship between classifiers in which one classifier specifies a contract that another guarantees to carry out.**
- **In the context of interfaces and collaborations**
- **An interface can be realized by many classes/components**
- **A class/component may realize many interfaces**

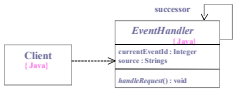
Modeling a Logical Database

- Class diagrams to provide more semantics
- From a general class diagram, first identify classes whose state must be persistent (e.g. after you turn off the computer, the data survives, although the program doesn't).
- Create a class diagram using standard tagged value, (e.g. {persistent}).
- Include attributes and associations.
- Use tools, if available, to transform logical design (e.g., tables and attributes) into physical design (e.g., layout of data on disk and indexing mechanisms for fast access to the data).



Forward/ Reverse Engineering

- translate a collaboration into a logical database schema/operations
- transform a model into code through a mapping to an implementation language.
- Steps
  - Selectively use UML to match language semantics (e.g. mapping multiple inheritance in a collaboration diagram into a programming language with only single inheritance mechanism).
  - Use tagged values to identify language..




```
public abstract class EventHandler
{
    private EventHandler successor;
    private Integer currentEventId;
    private String source;

    EventHandler() {}
    public void handleRequest() {}
}
```

- **translate a logical database schema/operations into a collaboration**
- **transform code into a model through mapping from a specific implementation language.**

Object Diagrams

Structural Diagrams

- Class;
- **Object**
- Component
- Deployment
- Composite Structure
- Package

Instances & Object Diagrams

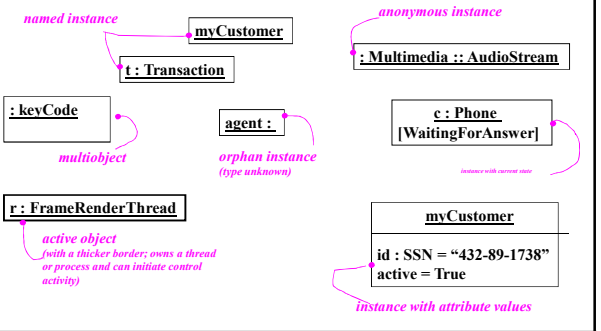
- "instance" and "object" are largely synonymous; used interchangeably.
- difference:
  - instances of a class are called objects or instances; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only instances.

What is an instance of an association called?

Object Diagrams

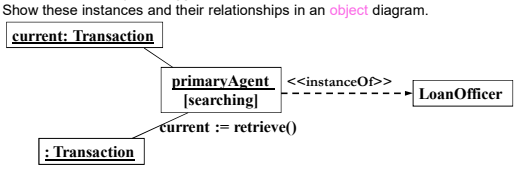
- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of objects that collaborate in that scenario (e.g., from use case flows).
  - Expose these object's states, attribute values and links among these objects.

Instances & Objects - Visual Representation



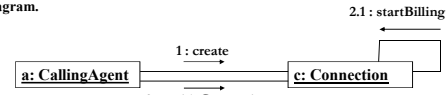
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

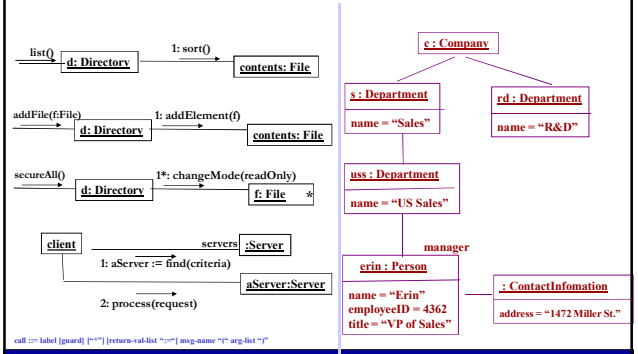


Instances & Objects - Modeling Prototypical Instances

- Show these instances and their relationships in an interaction diagram or an activity diagram.



Instances & Objects - More Examples



Component Diagrams

Structural Diagrams

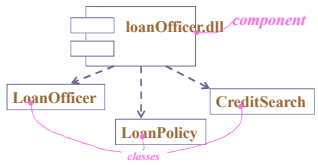
- Class;
  Object
- **Component**
- Deployment
- Composite Structure
- Package

Component Diagram

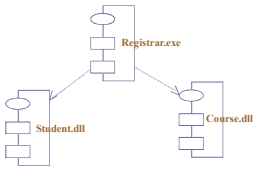
UML1.x – implementation view

- Shows a set of components and their relationships.
- Represents the static implementation view of a system.
- Components map to one or more classes, interfaces, or collaborations.

Mapping of Components into Classes



Components and their Relationships

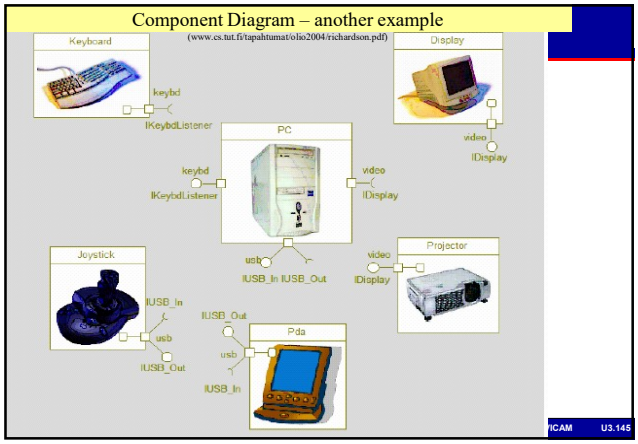


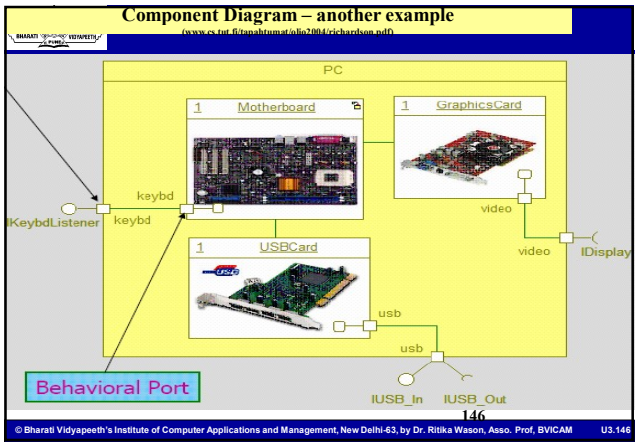
Component Diagram

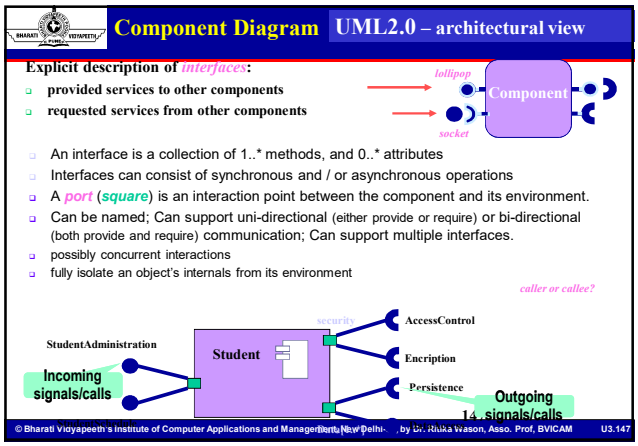
UML2.0 – architectural view

- ***Big demand, hmm…*** ind architecture
- Architecture still an emerging discipline
- Challenges, a bumpy road ahead

- UML and architecture evolving in parallel
- Component diagram in need of better formalization and experimentation

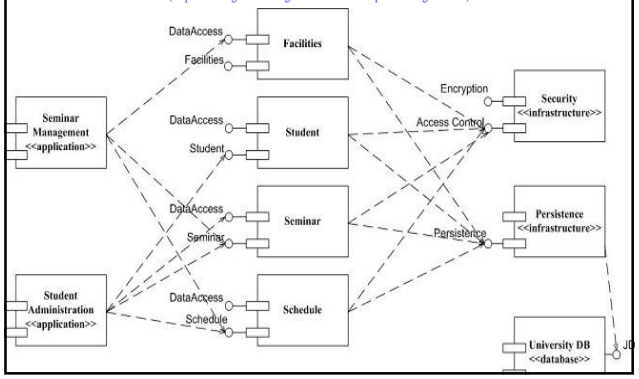






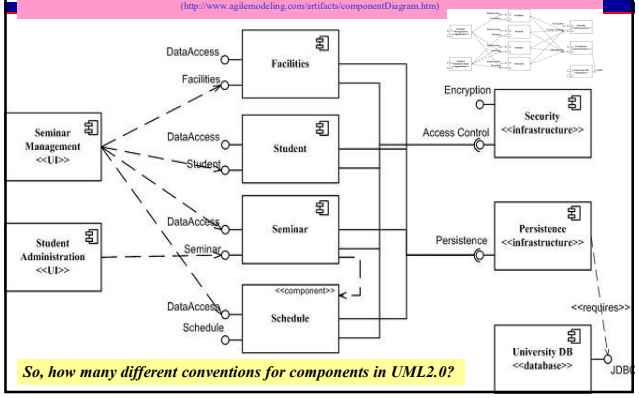
Component Diagram: UML 1.x and UML 2.0

(http://www.agilemodeling.com/artifacts/componentDiagram.htm)



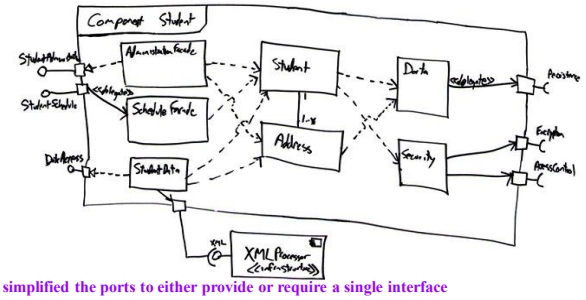
Component Diagram: UML 1.x and UML 2.0

(http://www.agilemodeling.com/artifacts/componentDiagram.htm)



So, how many different conventions for components in UML2.0?

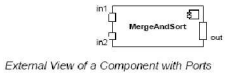
Building a Component



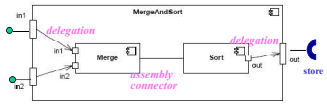
- simplified the ports to either provide or require a single interface
- relationships between ports and internal classes in three different ways:
  i) as *stereotyped delegates* (flow), as *delegates*, and as *realizes* (logical->physical) relationships
- Cohesive reuse and change of classes; acyclic component dependency ???

Component Diagram – Connector & Another Example

- a connector: just a link between two or more connectable elements (e.g., ports or interfaces)
- 2 kinds of connectors: *assembly* and *delegation*. For *"wiring"*
  - An *assembly* connector: a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another; *simple line/ball-and-socket/lollipop-socket notation*
  - A *delegation* connector binds a component's external behavior (as specified at a port) to an internal realization of that behavior by one of its parts *(provide-provide, request-request).*



External View of a Component with Ports



Internal View of a Component with Ports

Left delegation: direction of arrowhead indicates "provides"

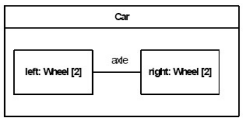
Right delegation: direction of arrowhead indicates "requires"

So, what levels of abstractions for connections?

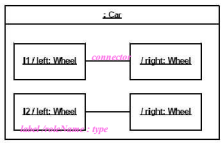
Structured Class

- A *structured class(ifier)* is defined, in whole or in part, in terms of a number of *parts* - contained instances owned or referenced by the structured class(ifier).
- With a similar meaning to a composition relation
- A structured classifier's parts are created within the containing classifier (either when the structured classifier is created or later) and are destroyed when the containing classifier is destroyed.
- Like classes and components, combine the descriptive capabilities of structured classifiers with *ports and interfaces*

Any difference?



component or class?

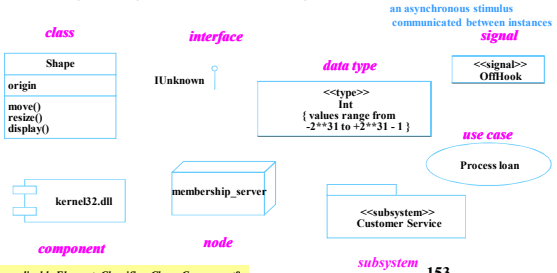


***Components extend classes* with additional features such as**

- **the ability to *own more types of elements* than classes can; e.g., packages, constraints, use cases, and artifacts**
- **deployment specifications that define the execution parameters of a component deployed to a node**

Classifiers

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
- *cf. Packages and generalization relationships do not have instances.*



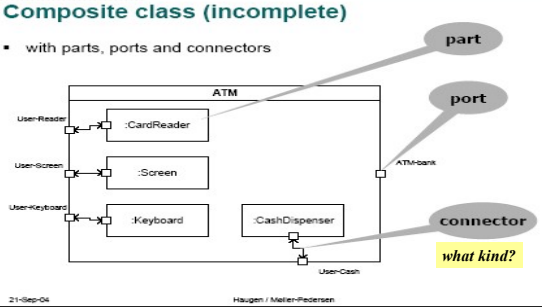
Generalizable Element, Classifier, Class, Component?

Structured Class – Another Example

UNIVERSITY OF OSLO

Composite class (incomplete)

- with parts, ports and connectors



OMG seminar 2004

21-Sep-04 Haugen / Moller-Pedersen 13

Deployment Diagrams

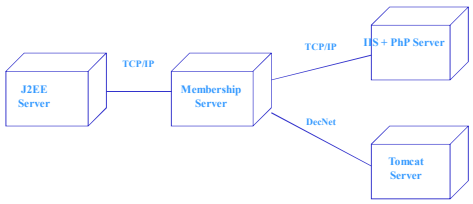
Structural Diagrams

- Class;
  Object
- Component
- **Deployment**
- Composite Structure
- Package

155

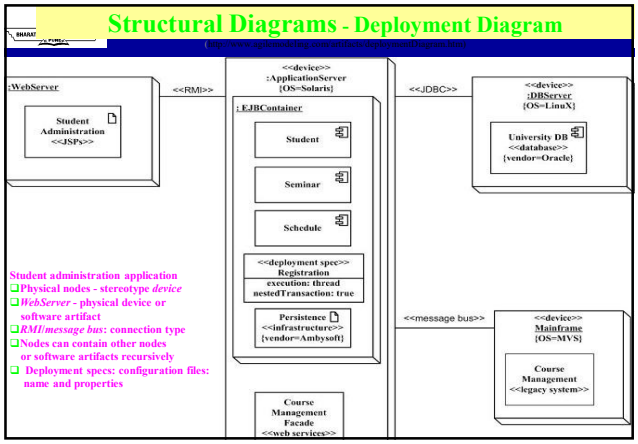
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more components.



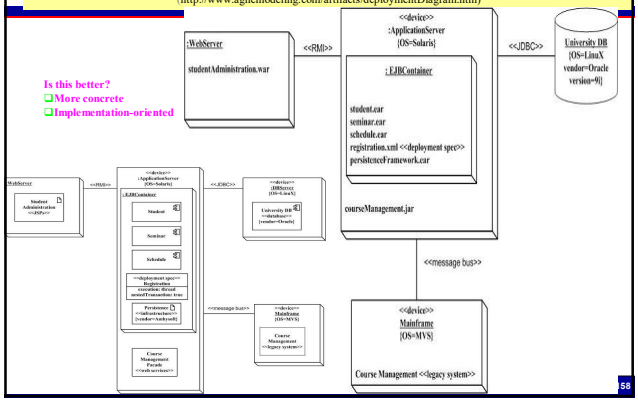
156

Structural Diagrams - Deployment Diagram



Structural Diagrams - Deployment Diagram

(http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)



Composite Structure Diagrams

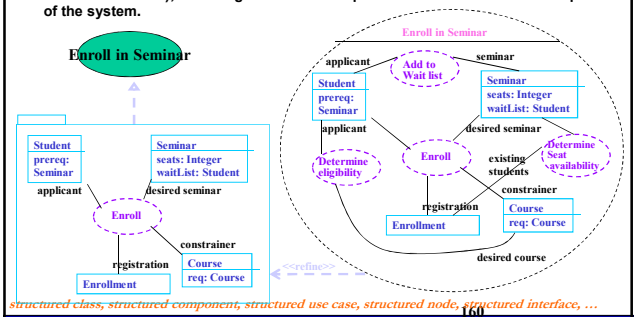
Structural Diagrams

- Class; Object
- Component
- Deployment
- **Composite Structure**
- Package

Composite Structure Diagrams

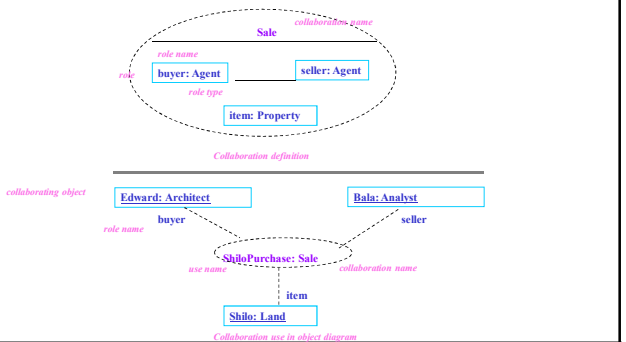
(http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm)

- Depicts the internal structure of a classifier (such as a class, component, or collaboration), including the interaction points of the classifier to other parts of the system.



structured class, structured component, structured use case, structured node, structured interface, …

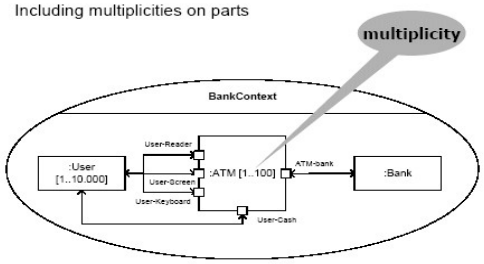
Variations [Rumbaugh – UML 2.0 Reference: p234]



UNIVERSITY OF OSLO

Context Model in UML2.0 - II

- Including multiplicities on parts



OMS seminar 2004

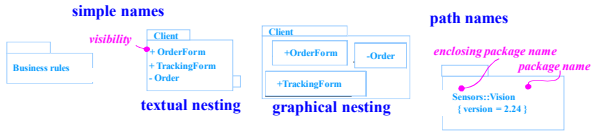
21-Sep-04 Haugen / Møller-Pedersen 15

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- Package

Packages

- Package — general-purpose mechanism for organizing elements into groups.
- **Nested Elements: Composite relationship (When the whole dies, its parts die as well, but not necessarily vice versa)**
- **(C++ namespace; specialization means "derived")**



Visibility

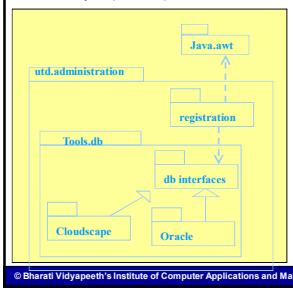
- **Packages that are friends to another may see all the elements of that package, no matter what their visibility.**
- **If an element is visible within a package, it is visible within all packages nested inside the package.**

Dependency –Among Packages

- Two Stereotypes of Dependency Among Packages:
  - *access*: the source package is granted the right to reference the elements of the target package *(:: convention)*
  - *import*: a kind of access; the ***public*** contents of the target package enter the flat namespace of the source as if they had been declared in the source

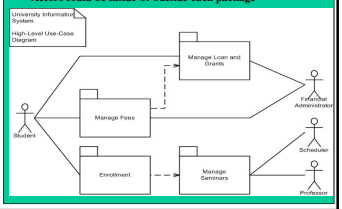
Modeling Groups of Elements

- Look for "clumps" of elements that are semantically close to one another.
- Surround "clumps" with a package.
- Identify public elements of each package.
- Identify import dependencies.



Use Case package Diagram

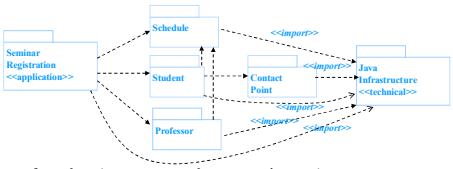
- Included and extending use cases belong in the same package as the parent/base use case
- Cohesive, and goal-oriented packaging
- Actors could be inside or outside each package



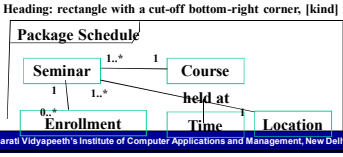
Class Package Diagrams

(http://www.agilemodeling.com/artifacts/packageDiagram.htm)

- Classes related through inheritance, composition or communication often belong in the same package



- A *frame* depicts the contents of a package (or components, classes, operations, etc.)
- Heading: rectangle with a cut-off bottom-right corner, [kind] name [parameter]



A frame encapsulates a collection of collaborating instances or refers to another representation of such

Common Mechanisms

- **Adornments**
  - **Notes & Compartments**
- **Extensibility Mechanisms**
  - **Stereotypes** - Extension of the UML metaclasses.
  - **Tagged Values** - Extension of the properties of a UML element.
  - **Constraints** - Extension of the semantics of a UML element.

168

Adornments

- Textual or graphical items added to an element's basic notation.
- Notes - Graphical symbol for rendering constraints or comments attached to an element or collection of elements; No Semantic Impact

Rendered as a rectangle with a dog-eared corner.

See smartCard.doc for details about this routine.

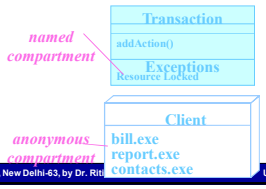
May contain combination of text and graphics.

See http://www.rational.com for related info.

May contain URLs linking to external documents.

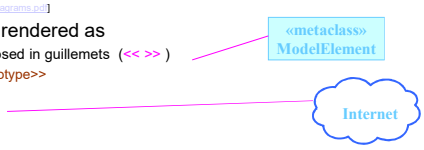
Additional Adornments

- Placed near the element as
  - Text
  - Graphic
- Special compartments for adornments in
  - Classes
  - Components
  - Nodes



Stereotypes

- **Mechanisms for extending the UML vocabulary.**
- **Allows for new modeling building blocks or parts.**
- Allow controlled extension of metamodel classes.
  [UML11_Metamodel_Diagrams.pdf]
- Graphically rendered as
  - Name enclosed in guillemets (<< >>)
    - ✓ <<stereotype>>
  - New icon



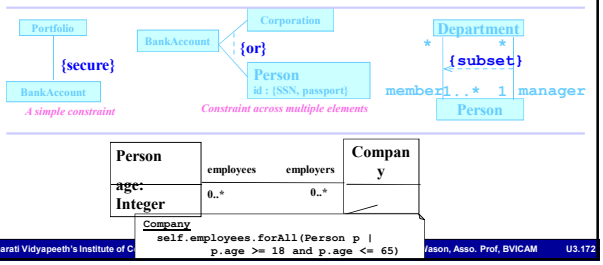
- **The new building block can have**
  - **its own special properties through a set of tagged values**
  - **its own semantics through constraints**

Tagged Values

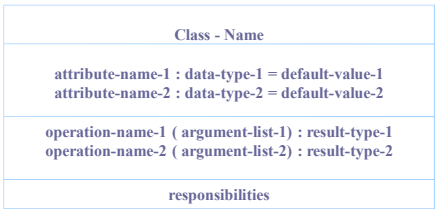
- **a (name, value) pair describes a property of a model element.**
- **Properties allow the extension of "*metamodel*" element attributes.**
- **modifies the semantics of the element to which it relates.**
- **Rendered as a text string enclosed in braces { }**
- **Placed below the name of another element.**

Constraints

- Extension of the semantics of a UML element.
- Allows new or modified rules
- Rendered in braces {}.
  - Informally as free-form text, or
  - Formally in UML's Object Constraint Language (OCL):
    E.g., {self.wife.gender = female and self.husband.gender = male}



Classes: Notation and Semantics



To model the <<semantics>> (meaning) of a class:

- Specify the body of each method (pre-/post-conditions and invariants)
- Specify the state machine for the class
- Specify the collaboration for the class
- Specify the responsibilities (contract)

Attributes

- Syntax
  [visibility] name [multiplicity] [: type] [= initial-value] [{property-string }]
- Visibility
  + public; - private; # protected; {default = +}
- type
  - There are several defined in Rational Rose.
  - You can define your own.

  Or you can define your own: e.g. {leaf}
- property-string
  Built-in property-strings:
  - *changeable*—no restrictions (default)
  - *addOnly*—values may not be removed or altered, but may be added
  - *frozen*—may not be changed after initialization

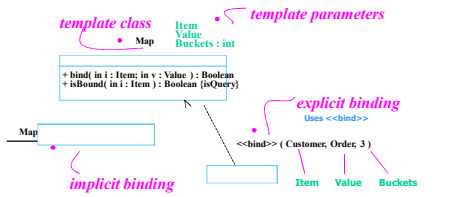
origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = { 0, 0 }	Name, type, and initial value
id : Integer { frozen }	Name and property

Operations

- Syntax
  [visibility] name [(parameter-list)] [: return-type] [(property-string)]
- Visibility
  + public; - private; # protected; {default = +}
- parameter-list syntax
  [direction] name : type [= default-value]
- direction
  - *in*—input parameter; may not be modified
  - *out*—output parameter; may be modified
  - *inout*—input parameter; may be modified
- **property-string**
  - ***leaf***
  - ***isQuery*—state is not affected**
  - ***sequential*—not thread safe**
  - ***guarded*—thread safe (Java synchronized)**
  - ***concurrent*—typically atomic; safe for multiple flows of control**

Template Classes; Primitive Types

- A template class is a parameterized element and defines a family of classes
- In order to use a template class, it has to be instantiated
- Instantiation involves binding formal template parameters to actual ones, resulting in a concrete class



Primitive Types using a class notation

Interface: A Java Example

```
public interface SoundFromSpaceListener extends EventListener {
    void handleSoundFromSpace(SoundFromSpaceEventObject sfseo);
}

public class SpaceObservatory implements SoundFromSpaceListener
    public void handleSoundFromSpace(SoundFromSpaceEventObject sfseo) {
            soundDetected = true;
            callForPressConference();
    }
```

Can you draw a UML diagram corresponding to this?

Package Diagrams: Standard Elements

- *Façade* — only a view on some other package.
- *Framework* — package consisting mainly of patterns.
- *Stub* — a package that serves as a proxy for the public contents of another package.
- *Subsystem* — a package representing an independent part of the system being modeled.
- *System* — a package representing the entire system being modeled.

Is <<import>> transitive?
Is visibility transitive?
Does <<friend>> apply to all types of visibility: +, -, #?

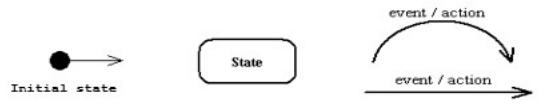
Dependency –Among Objects

- 3 Stereotypes of Dependency in Interactions among Objects:
  - *become*: the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
  - *call*: the source operation invokes the target operation
  - *copy*: the target object is an exact, but independent, copy of the source

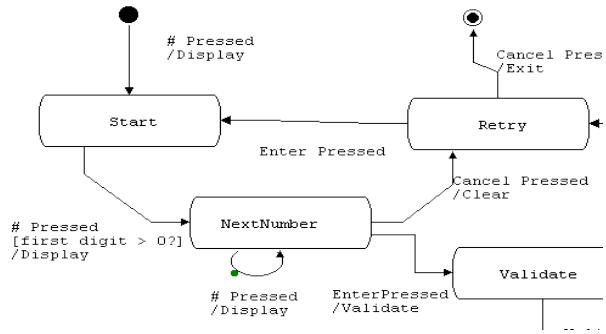
State Machine Diagram

- UML state machine diagrams depict the **various states** that an **object** displays.
- And the **transitions** between those states
- State diagrams show the ***change of an object over time***
- Very useful for **concurrent** and **real-time systems**

State Machine Diagram Notations



State Machine Diagram (*Login*)



State Machine

- "The state machine view describes the **dynamic behavior** of objects over time by **modeling** the **lifecycles** of **objects** of each **class**.
- Each **object** is treated as an **isolated entity** that communicates with the rest of the world by **detecting events** and responding to them.
- Events represent the **kinds of changes** that objects can detect... Anything that can affect an object can be characterized as an event."

  - *The UML Reference Manual, [Rumbaugh,99]*

State Chart Diagram

- It shows a **machine** consisting of **states**, **transitions**, **events** and **activities**.
- It addresses the **dynamic view** of the **system**.
- It is **depicted** as follows in rational rose.

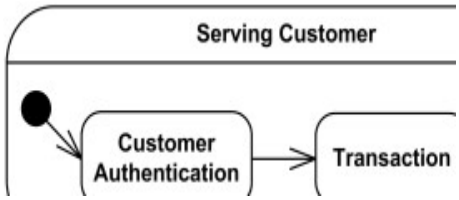
State Diagram Features

- **Event –** something that happens at a **specific point**
  - Alarm goes off
- **Condition –** something that has a **duration**
  - Alarm is on
  - Fuel level is low
- **State** – an abstraction of the **attributes** and relationships of an **object** (or system)
  - The fuel tank is in a **too low level** when the fuel level is **below level *x*** for *n* seconds

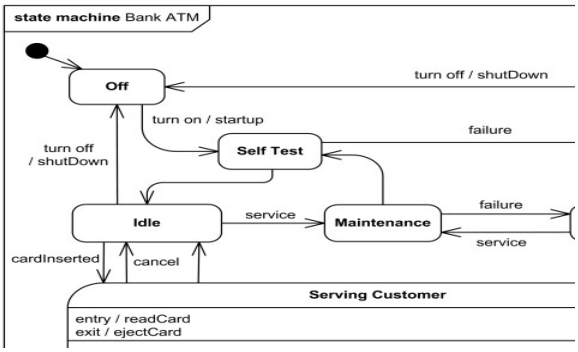
State Diagram Features

- ***State***. A **condition during the life** of an object in which it satisfies some condition, performs some action, or waits for some event.
- ***Event***. An occurrence that may **trigger a state transition.** Event types include an explicit **signal** from outside the system, an invocation from inside the system, the passage of a designated period of time, or a designated **condition becoming true.**
- ***Guard***. A boolean expression which, if true, **enables an event to cause a transition.**
- ***Transition***. A transition represents the change from one state to another:
- ***Action***. One or more actions taken by an object in response to a state change.

Composite States



State Chart Diagram



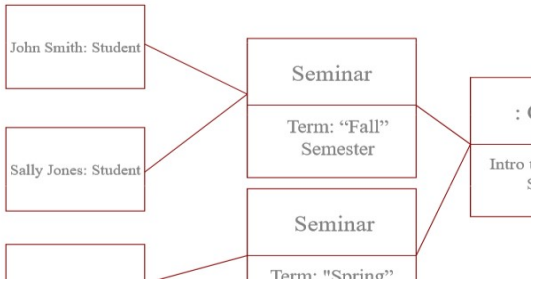
State Diagram- *Case Study*

- A simple digital watch has a display and two buttons A button and the B button. The watch has tw operation, display time and set time.
- In the display time mode, the watch displays hours separated by a flashing colon.
- The set time mode has two submodes, set hours and The A button selects modes. Each time it is presse advances in the sequence: display, set hours, display, etc.
- Within the submodes, the B button advances t minutes once each time it is pressed. Buttons must

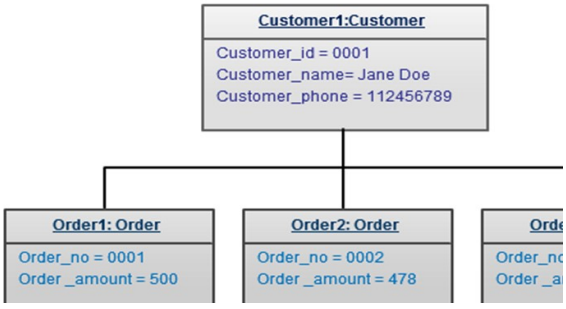
Object Diagram

- Object diagram is a **pictorial representation** of the **relationships** between the **instantiated classes** at any point in time.
- It's depicted as follows in Rational Rose-

Object Diagram Example



Object Diagram Example



Activity Diagram

- Activity diagrams are typically used for **business process modeling**
- For modeling the **detailed logic** of a **business rule**
- Model the **internal logic** of a **complex operation**
- An activity diagram is a special case of a **state chart diagram** in which states are **activities** ("*functions*")
- Activity diagrams are the **object-oriented equivalent** of **flow charts** and data flow diagrams (**DFDs**)

Activity Diagram Structuring

These diagrams are similar to **state chart diagrams** and use similar conventions, but activity diagrams describe the *behavior of a class* in response to **internal processing.**

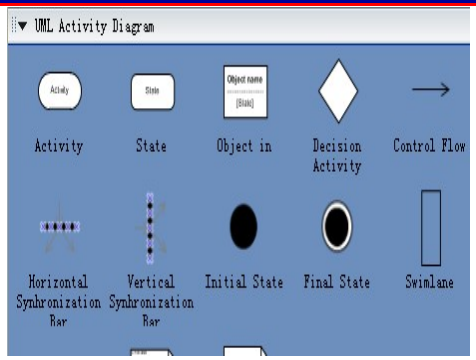
Swimlanes, which represent **responsibilities** of one or more objects for actions within an overall activity; that is, they divide the activity states into groups and assign these groups to objects that must perform the activities.

- **Action States**, which represent **atomic**, or **non-interruptible**, **actions** of entities or steps in the execution of an algorithm.
- **Action flows**, which represent **relationships** between the different action states of an entity.
- **Object flows**, which represent the **utilization of objects** by action states and the influence of action states on objects.

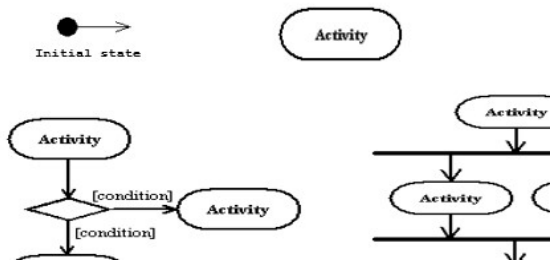
Activity Diagram Notations

- **Rounded rectangles** represent **activities**;
- **Diamonds** represent **decisions**;
- **Bars** represent the start (split) or end (join) **of concurrent activities**;
- **A black circle** represents the start (**initial state**) of the workflow;
- An **encircled black circle** represents the end (**final state**).

Activity Diagram Notations

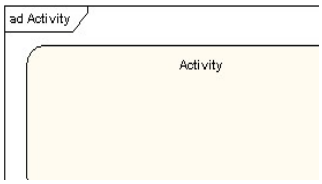


Activity Diagram Notations



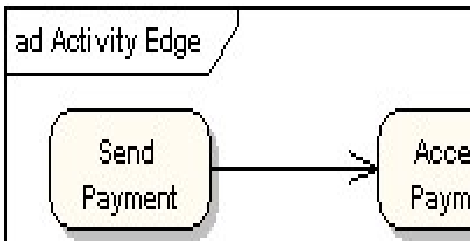
Activities

An activity is the specification of a **parameterized sequence** of **behaviour**. An activity is shown as a **round-cornered rectangle** enclosing all the actions, control flows and other elements that make up the activity.



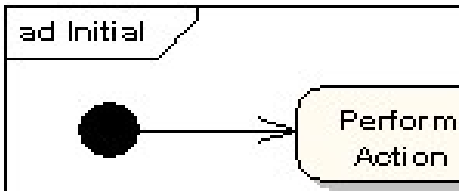
Control Flow

A control flow shows the **flow of control** from one action to the next. Its notation is a **line** with an **arrowhead**.



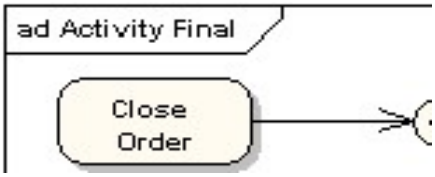
Initial Node

An initial or start node is depicted by a **large black spot**, as shown below.



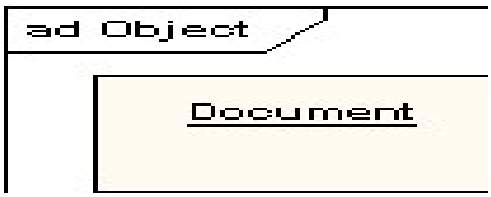
Final Node

The activity final node is depicted as a **circle** with a **dot** inside.



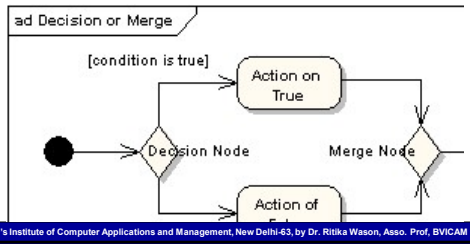
Object and Object Flows

An object flow is a **path** along which objects or data can pass. An object is shown as a **rectangle**.



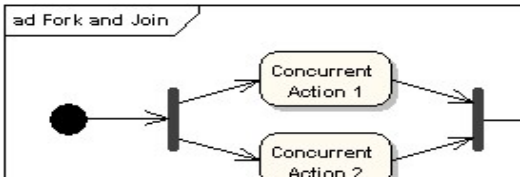
Decision and Merge Nodes

Decision nodes and merge nodes have the same notation: a **diamond shape**. They can both be named. The control flows coming away from a decision node will have **guard conditions** which will allow control to flow if the guard condition is met.



Fork and Join Nodes

Forks and joins have the **same notation**: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of **concurrent threads of control**.



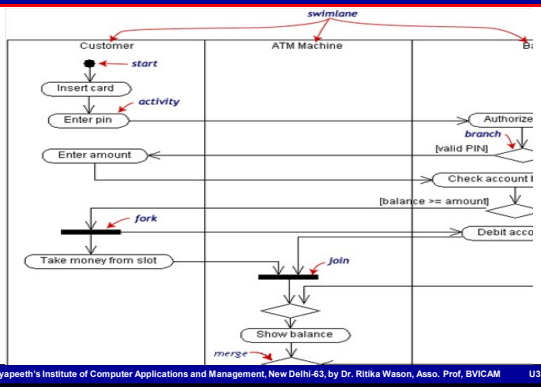
Steps to create an AD

1. Identify activities (steps) of a process
2. Identify who/what performs activities steps)
3. Draw swimlines
4. Identify decision points (if-then)
5. Determine if step is in loop (*For each.* based loop)
6. Determine if step is parallel

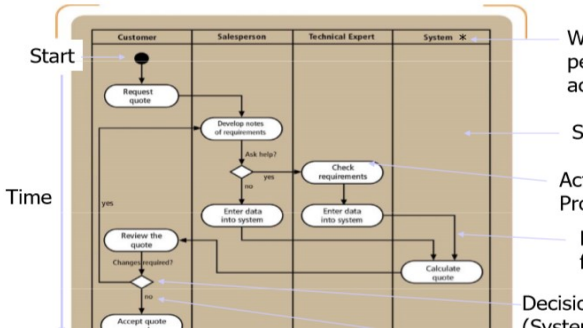
Steps to create an AD

8. Draw the start point of the process in the swir first activity (step)
9. Draw the oval of the first activity (step)
10. Draw an arrow to the location of the second a
11. Draw subsequent activities, while insertin points and synchronization/loop bars where a

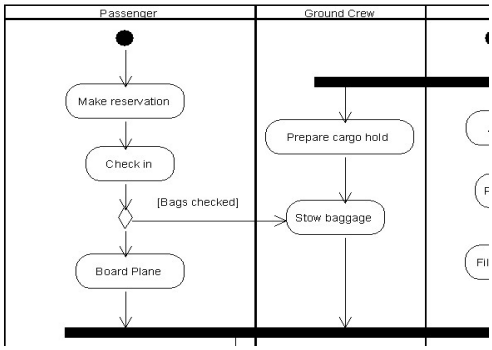
Activity Diagram- ATM Machine



Activity Diagram- Quote System



Activity Diagram- Flight Mgt. System



Dynamic Diagram Types

- Interaction Diagrams - Set of objects or role messages that can be passed among them.
  - Sequence Diagrams - emphasize time order
  - Communication Diagrams - emphasize struc ordering
- State Diagrams
  - State machine consisting of states, transitio and activities of an object
- Activity & Swimlane Diagrams

Interaction Diagram

- An interaction diagram shows an interaction,
- consisting of a set of objects and their relations
- include the messages that may be exchange them
- Model the dynamic aspect of the system
- Contain two sort of diagrams:
  - **Sequence diagrams**,
    - ✓**show the messages objects send to eacl timely manner**
  - **Collaboration diagrams**,

Interaction Diagram Details

- Using interaction diagrams, we can **clarify** the **sequence of operation calls** among **objects** used to complete a single use case
- Collaborations have the added advantage of **interfaces** and **freedom of layout**, but can be difficult to follow, understand and create.
- Interaction diagrams are used to diagram a **single use case**.
- When you want to examine the **behaviour** of a **single instance** over time use a **state diagram**, and if you want to look at the **behaviour** of the system over time use an activity diagram.

Sequence Diagram

- The sequence diagram describes the flow of messages being passed from object to object.

The **purposes** of **interaction diagram** can be describes as:
- To capture **dynamic behavior** of a system.
- To describe the **message flow** in the system.
- To describe **structural organization** of the objects.
- To describe **interaction** among objects.

Sequence Diagram Elements

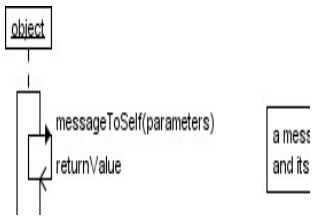
- **Class roles**, which represent **roles** that **objects** may play within the **interaction**.
- **Lifelines**, which represent the **existence** of an **object** over a period of time.
- **Activations**, which represent the **time** during which an object is performing an **operation**.
- The white rectangles on a **lifeline** are called **activations** and indicate that an object is ***responding to a message***. It starts when the message is received and ends when the object is done handling the message.
- **Messages**, which represent **communication** between **objects**.

Messages

- An **interaction** between **two objects** is performed as a **message** sent from **one object** to **another** (simple operation call, Signaling, RPC)
- If object obj_1 sends a message to another object obj_2 some link must exist between those two objects .
- A message is represented by an **arrow** between the life lines of two objects.
  - ***Self calls*** are also allowed
  - The time required by the receiver object to process the message is denoted by an ***activation-box***.
- A message is labeled at minimum with the **message name**.
  - **Arguments** and **control information** (conditions, iteration) may be included.

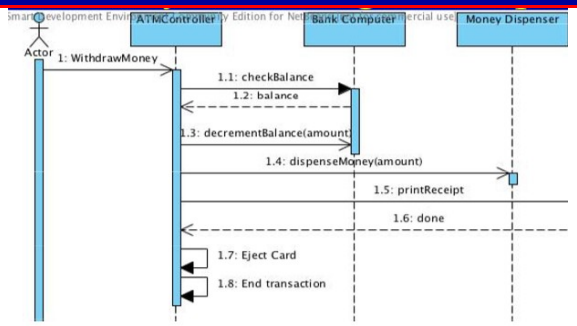
Message to Self

- A message that an object sends itself can be shown as follows:



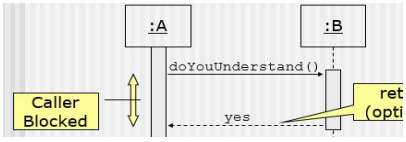
- Keep in mind that the purpose of a sequence diagram is to show the **interaction between objects**, so think twice about every self message you put on a diagram.

Asynchronous Messages



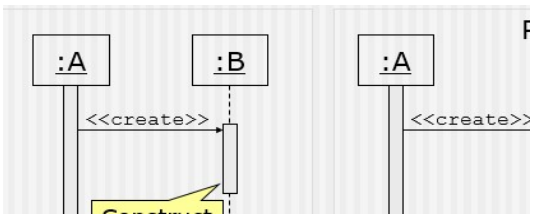
Return Values

- Optionally indicated using a **dashed arrow** with a label indicating the **return value**.
  - Don't model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to *refer to it elsewhere*, e.g. as a **parameter passed** in another message.



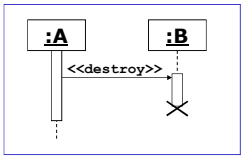
Object Creation

An object may create another object via a <<create>> message.

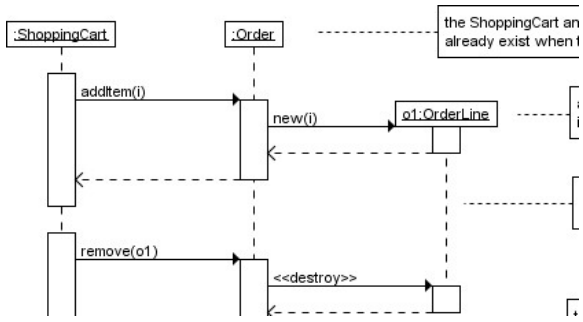


Object Destruction

- An object may destroy another object via a <<destroy>> message.
  - An object may destroy **itself**.
  - Avoid modeling object destruction unless **memory management** is critical.

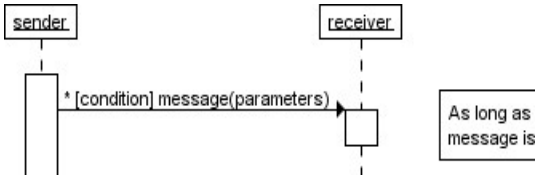


Creation and Destruction

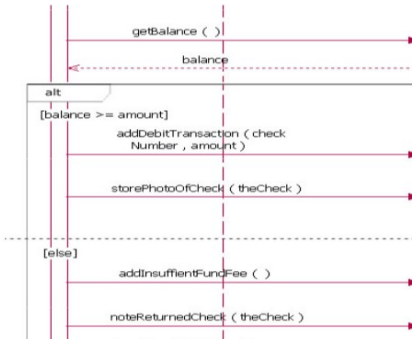


Repeated Interaction

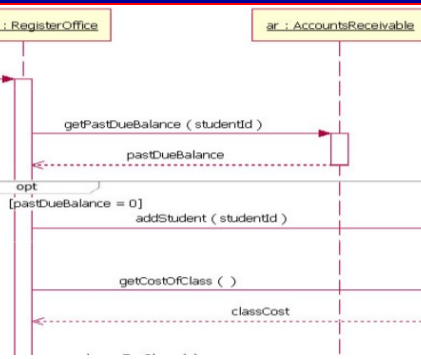
- When a message is prefixed with an **asterisk** (the '*'-symbol), it means that the message is *sent repeatedly*.
- A **guard** indicates the **condition** that determines whether or not the message should be sent (again). As long as the condition holds, the message is **repeated**.



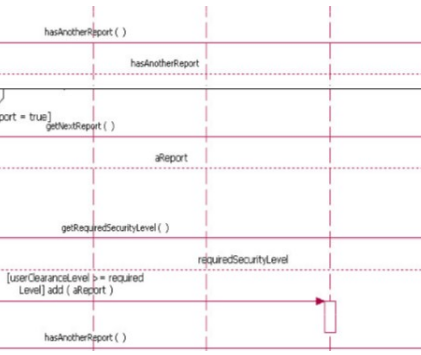
Components: alt/else



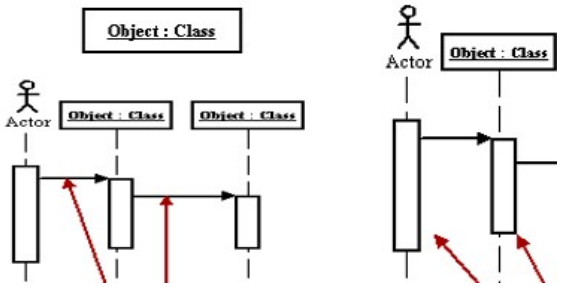
Components: option



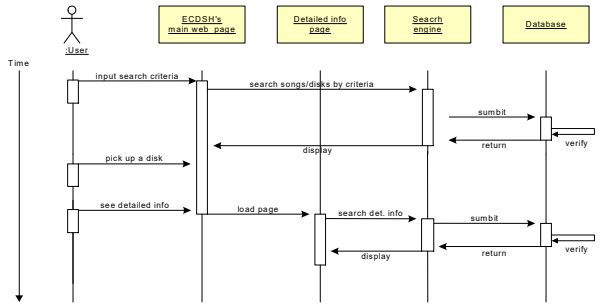
Components: loop



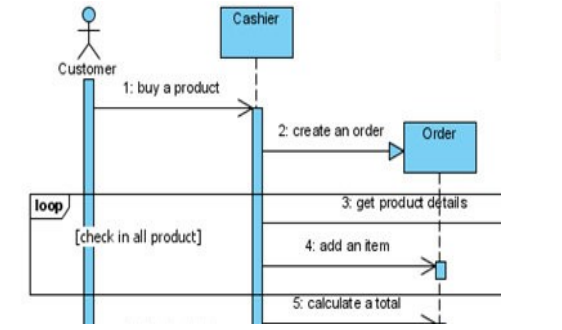
Sequence Diagram Notations



Sequence Diagram- *Search Engine*

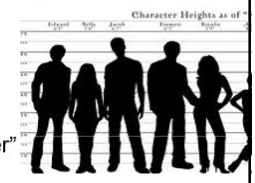


Sequence Diagram- *OMS*



Sequence Diagram- *Summary*

- Sequence diagrams model object interactions w
  emphasis on time ordering
- Method call lines
  - Must be horizontal!
  - Vertical height matters!
    "Lower equals Later"
  - Label the lines
- Lifeline – dotted vertical line
- Execution bar – bar around lifeline when code is
- Arrows
  - Synchronous call (you're waiting for a return value) –



Collaboration/ Communication Diagram

- Collaboration diagrams model the **interactions** between **objects**.
- This type of diagram is a **cross** between an **object diagram** and a **sequence diagram**.
- Unlike the Sequence diagram, which models the interaction in a column and row type format, the Collaboration diagram uses the **free-form arrangement** of **objects** as found in an Object diagram.
- This makes it easier to see all interations involving a particular object.
- Here in collaboration diagram the **method call sequence** is indicated by some **numbering technique** as shown below.
- The number indicates how the **methods** are called one after another.

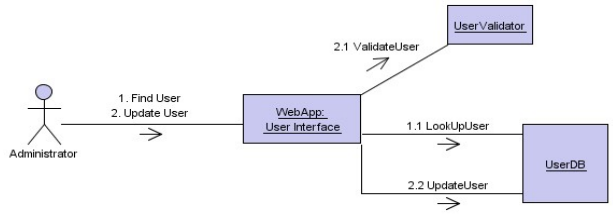
Communication Diagram

- The **method calls** are similar to that of a sequence diagram. But the difference is that the *sequence diagram does not describe the object organization* where as the **collaboration diagram** shows the **object organization.**
- If the **time sequence** is **important** then sequence diagram is used and if organization is required then collaboration diagram is used.

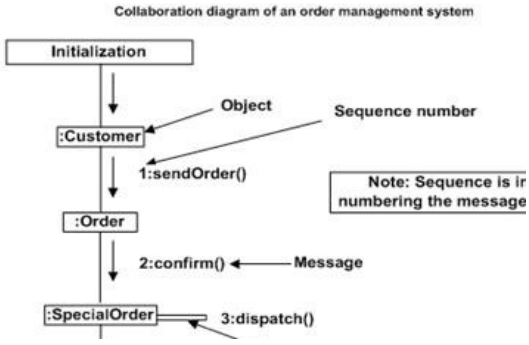
Communication Diagram Elements

- Collaboration Diagrams describe **interactions among classes** and **associations**. These interactions are modeled as *exchanges of messages between classes* through their **associations**. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements.
- **Class roles**, which represent **roles that objects may play** within the interaction.
- **Association roles**, which represent **roles that links may play** within the interaction.
- **Message flows**, which represent **messages** sent between objects via links. Links transport or implement the delivery of the message.

Communication Diagram Example



Communication Diagram- *OMS*



Sequence vs. Communication

- Semantically both are the same
- Express different sides of the model
- Sequence diagram expresses time ordering
- Collaboration diagram is used to define class behavior