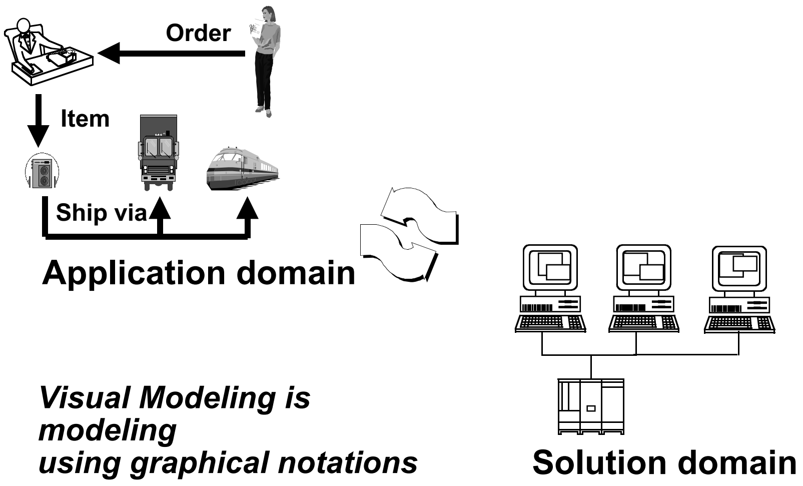
Overview

- What is modeling?
- What is UML?
- Use case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams
- Summary

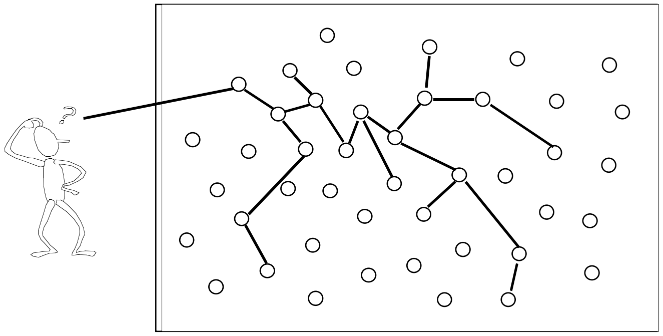
What is Modeling?



Visual Modeling is modeling using graphical notations

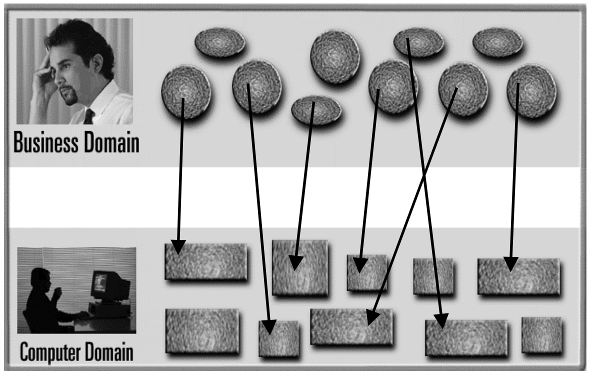
Visual Modeling Captures Application domain processes and objects

Use Case Analysis is a technique to capture application process from user s perspective

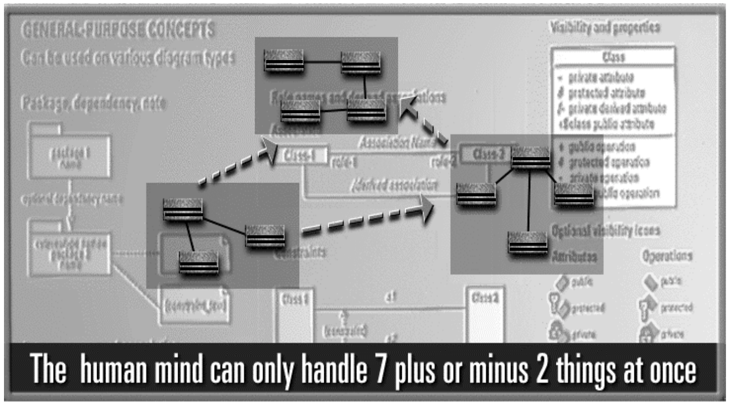


Modeling is a Communication Tool

Use modeling to capture application objects and logic

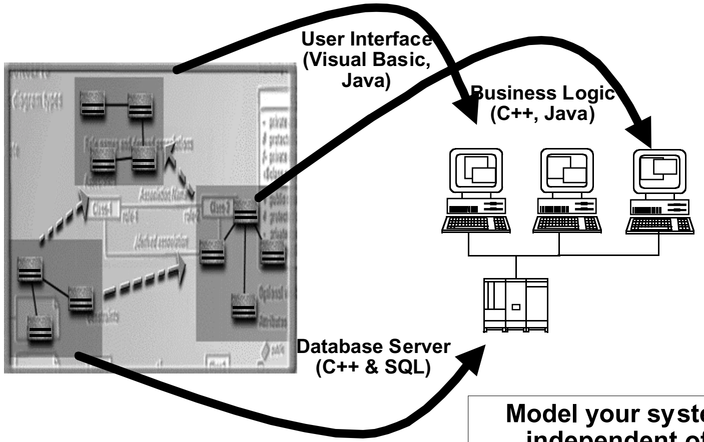


Modeling Manages Complexity


GENERAL-PURPOSE CONCEPTS
Can be used on various diagram types
Package, dependency, note
Visibility and properties
Class
Optional visibility icons
Attributes
Operations

The human mind can only handle 7 plus or minus 2 things at once

Visual Modeling can use and define Software Architecture and Design Patterns

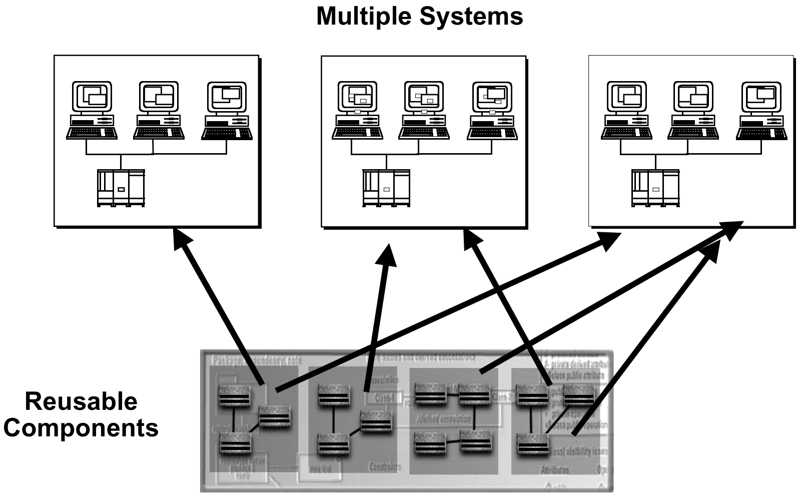

User Interface (Visual Basic, Java)

Business Logic (C++, Java)

Database Server (C++ & SQL)

Model your system independent of implementation language

Modeling Promotes Reuse

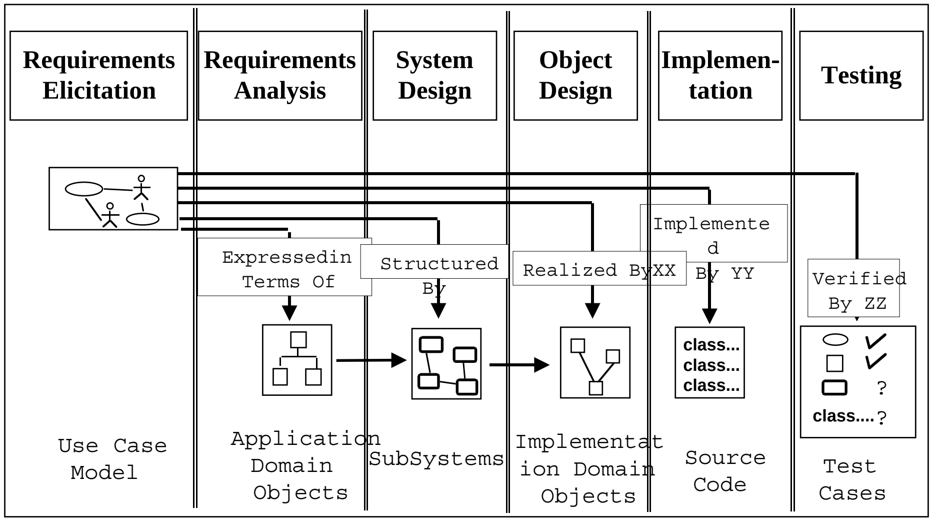

Multiple Systems

Reusable Components

What is the UML?

- UML stands for Unified Modeling Language
- The UML combines the best of the best from
  - **Data Modeling concepts (Entity Relationship Diagrams)**
  - **Application Modeling (work flow)**
  - **Object Modeling**
  - **Component Modeling**
- The UML is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software system
- Works best for OOD, OOP
- It can be used with all processes, throughout the development life cycle, and across different implementation technologies

Software Lifecycle Activities



Requirements Elicitation | Requirements Analysis | System Design | Object Design | Implemen-tation | Testing

Expressedin Terms Of

Structured By

Realized ByXX

Implemented By YY

Verified By ZZ

class... class... class...

class....?

Use Case Model

Application Domain Objects

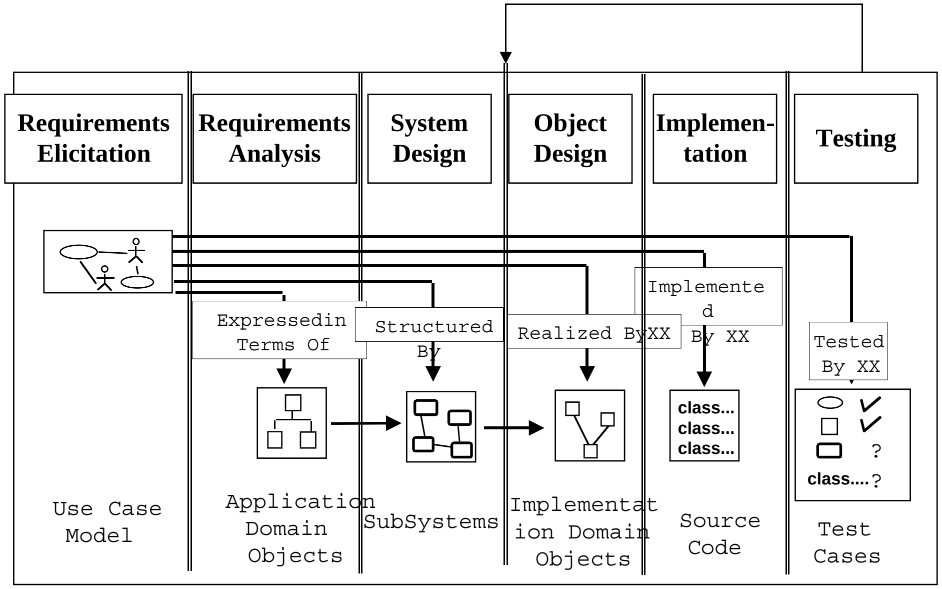
SubSystems

Implementation Domain Objects

Source Code

Test Cases

XProgramming Lifecycle Activities



Requirements Elicitation | Requirements Analysis | System Design | Object Design | Implemen-tation | Testing

Expressedin Terms Of

Structured By

Realized ByXX

Implemented By XX

Tested By XX

class... class... class...

class....?

Use Case Model

Application Domain Objects

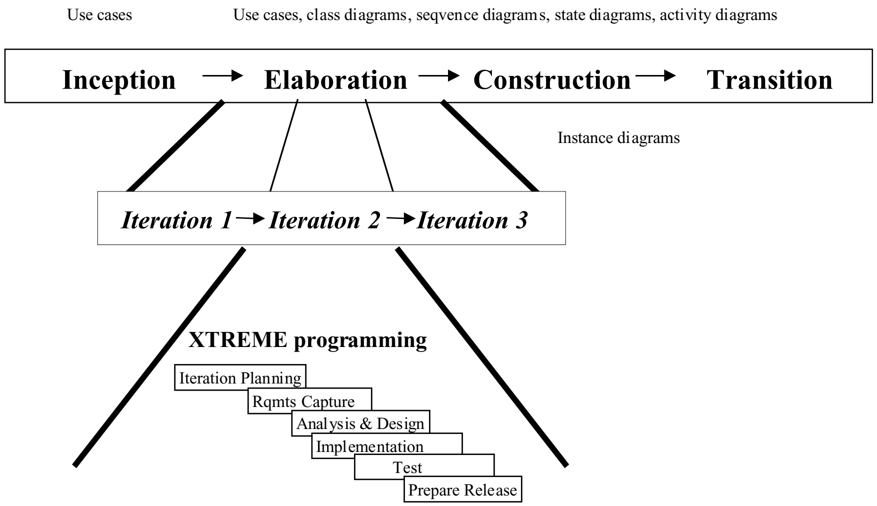
SubSystems

Implementation Domain Objects

Source Code

Test Cases

Rational Unified Software Life Cycle



Use cases

Use cases, class diagrams, seqvence diagrams, state diagrams, activity diagrams

Inception → Elaboration → Construction → Transition

Instance diagrams

Iteration 1 → Iteration 2 → Iteration 3

XTREME programming

Iteration Planning

Rqmts Capture

Analysis & Design

Implementation

Test

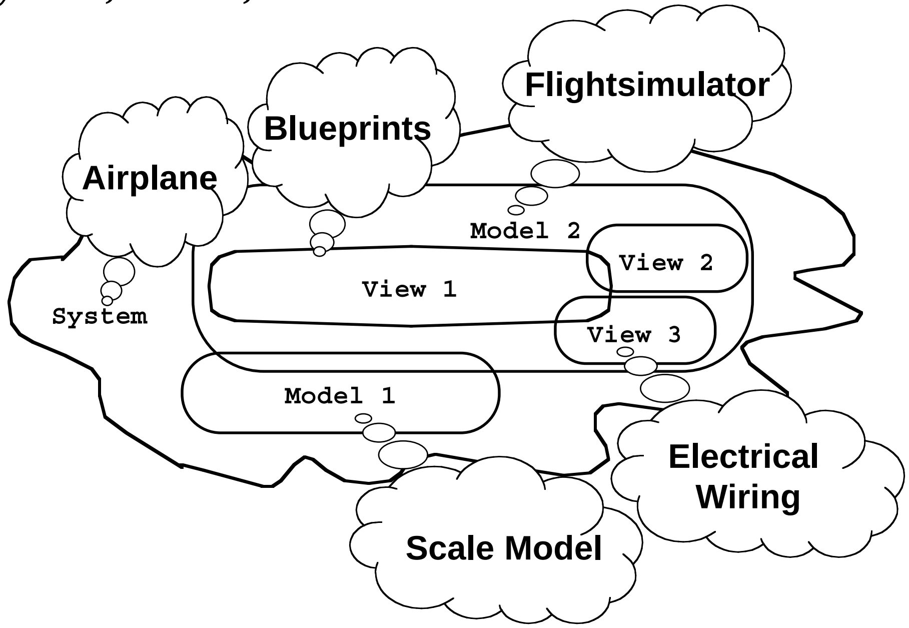
Prepare Release

Systems, Models, and Views

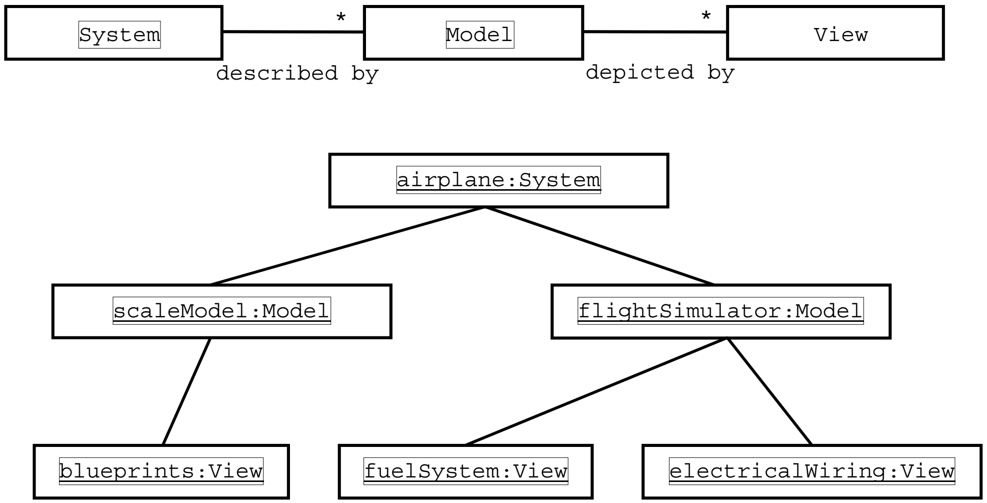
- A *model* is an abstraction describing system or a subset of a system
- A *view* depicts selected aspects of a model
- A *notation* is a set of graphical or textual rules for representing views

- Views and models of a single system may overlap each other

Systems, Models, and Views



Models, Views, and Systems (UML)


System * Model * View

described by depicted by

airplane:System

scaleModel:Model

flightSimulator:Model

blueprints:View

fuelSystem:View

electricalWiring:View

Concepts and Phenomena

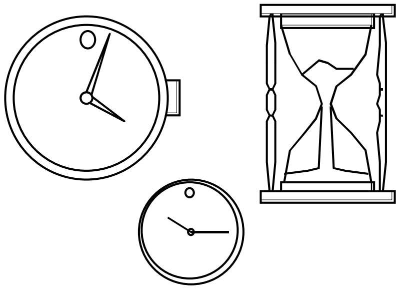
- *Phenomenon*: An object in the world of a domain as you perceive it, for example:
  - **The lecture you are attending**
  - **My black watch**
- *Concept*: Describes the properties of phenomena that are common, for example:
  - **Lectures on software engineering**
  - **Black watches**
- A concept is a 3-tuple:
  - **Its *Name* distinguishes it from other concepts.**
  - **Its *Purpose* are the properties that determine if a phenomenon is a member of a concept.**
  - **Its *Members* are the phenomena which are part of the concept.**

Concepts and Phenomena

Name **Purpose** **Members**

Clock

A device that measures time.



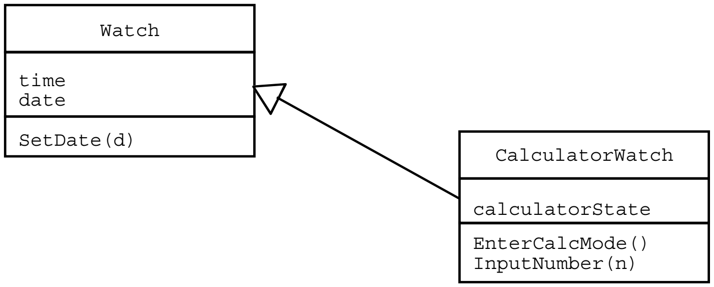
- Abstraction: Classification of phenomena into concepts
- Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Concepts In Software: Type and Instance

- Type:
  - **An abstraction in the context of programming languages**
  - **Name: int, Purpose: integral number, Members:** `0, -1, 1, 2, -2, . . .`
- Instance:
  - **Member of a specific type**
- The type of a variable represents all possible instances the variable can take.
- The relationship between “type” and “instance” is similar to that of “concept” and “phenomenon.”

Class

- Class:
  - **An abstraction in the context of object-oriented languages**
- Like an abstract data type, a class encapsulates both state (variables) and behavior (methods)
- Unlike abstract data types, classes can be defined in terms of other classes using inheritance



Watch

time
date

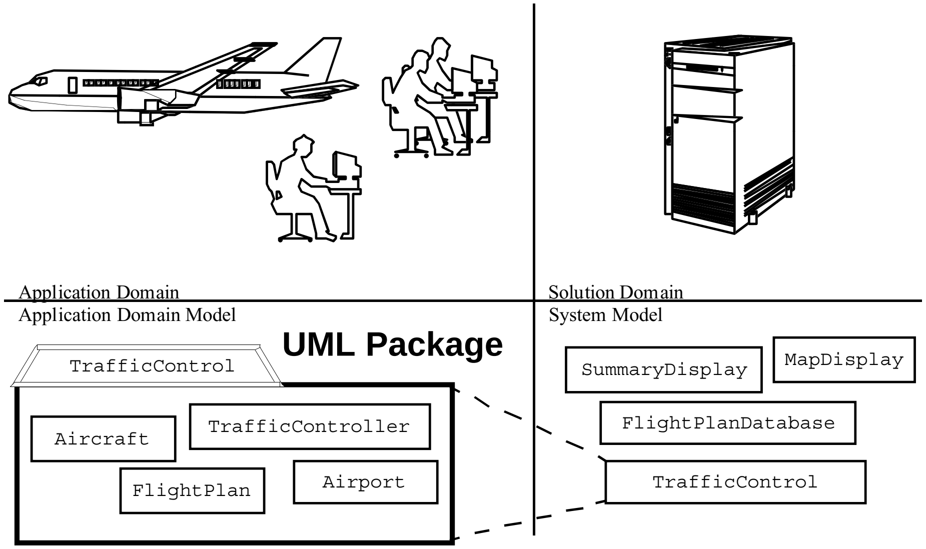
SetDate(d)

CalculatorWatch

calculatorState

EnterCalcMode()
InputNumber(n)

Object-Oriented Modeling



Application Domain

Application Domain Model

UML Package

TrafficControl

Aircraft

TrafficController

FlightPlan

Airport

Solution Domain

System Model

SummaryDisplay

MapDisplay

FlightPlanDatabase

TrafficControl

Application and Solution Domain

- Application Domain (Requirements Analysis):
  - **The environment in which the system is operating**
- Solution Domain (System Design, Object Design):
  - **The available technologies to build the system**

What is UML?

- ♦ UML (Unified Modeling Language)
  - w **A standard for modeling object-oriented software.**
  - w **Resulted from the convergence of notations from three leading object-oriented methods:**
    - t **OMT (James Rumbaugh)**
    - t **OOSE (Ivar Jacobson)**
    - t **Booch (Grady Booch)**
- ♦ Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- ♦ Supported by several CASE tools
  - w **Rational ROSE**
  - w **Together/J**
  - w **...**

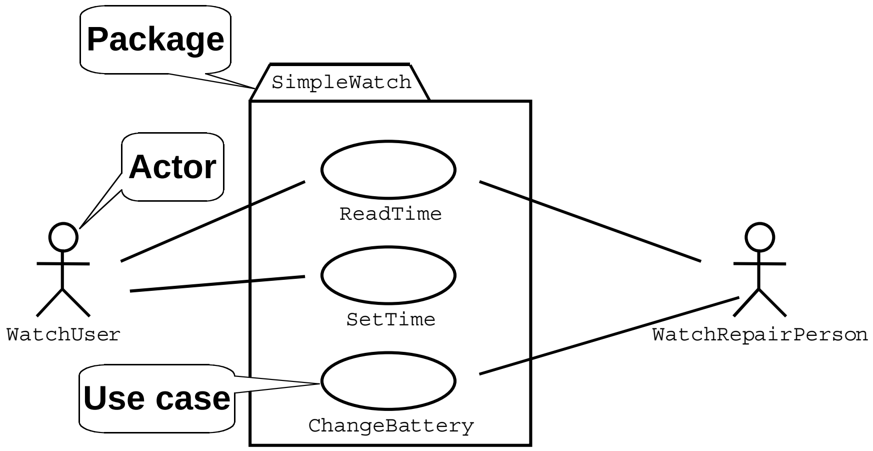
UML and This Course

- ♦ You can model 80% of most problems by using about 20% UML
- ♦ In this course, we teach you those 20%

UML First Pass

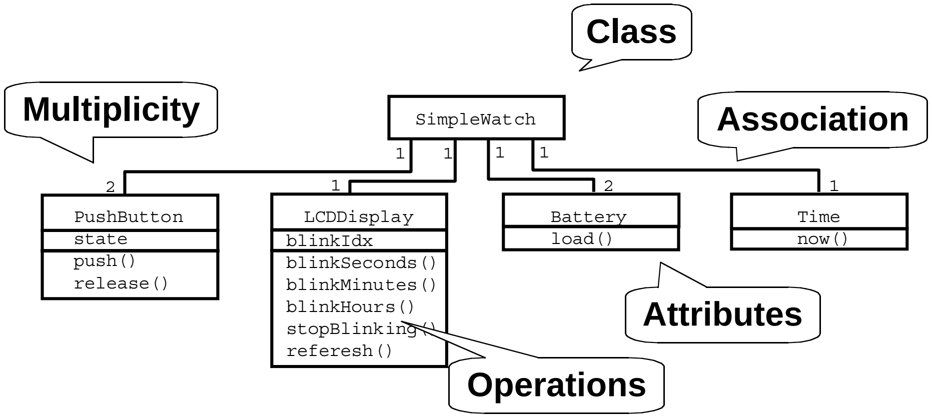
- ♦ Use case diagrams
  - w **Describe the functional behavior of the system as seen by the user.**
- ♦ Class diagrams
  - w **Describe the static structure of the system: Objects, Attributes, and Associations.**
- ♦ Sequence diagrams
  - w **Describe the dynamic behavior between actors and the system and between objects of the system.**
- ♦ Statechart diagrams
  - w **Describe the dynamic behavior of an individual object as a finite state machine.**
- ♦ Activity diagrams
  - w **Model the dynamic behavior of a system, in particular the workflow, i.e. a flowchart.**

UML First Pass: Use Case Diagrams



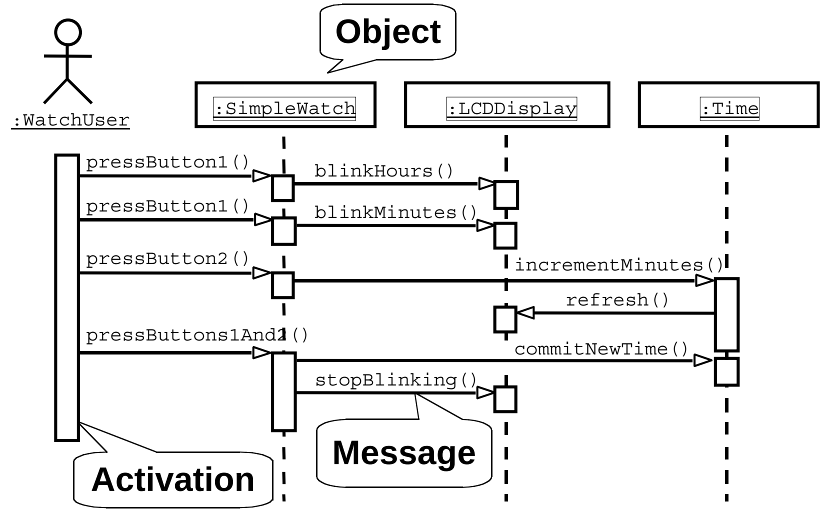
Use case diagrams represent the functionality of the system from user’s point of view

UML First Pass: Class Diagrams



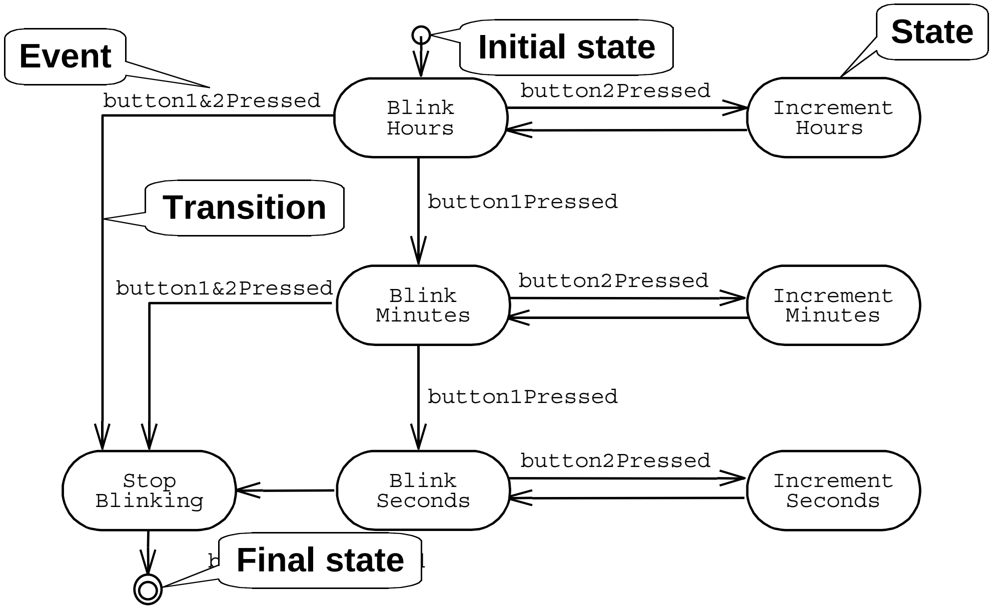
Class diagrams represent the structure of the system

UML First Pass: Sequence Diagram



Sequence diagrams represent the behavior as interactions

UML First Pass: Statechart Diagrams



Other UML Notations

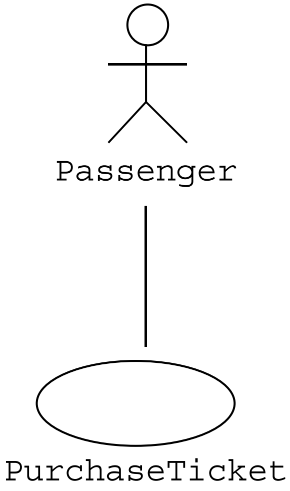
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - **Component diagrams**
  - **Deployment diagrams**
  - **Introduced in lecture on System Design**
- Object Constraint Language (OCL)
  - **Introduced in lecture on Object Design**

UML Core Conventions

- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - w `myWatch:SimpleWatch`
  - w `Joe:Firefighter`
- Types are denoted with nonunderlined names
  - w `SimpleWatch`
  - w `Firefighter`
- Diagrams are graphs
  - w **Nodes are entities**
  - w **Arcs are relationships between entities**

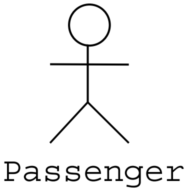
UML Second Pass: Use Case Diagrams



Used during requirements elicitation to represent external behavior

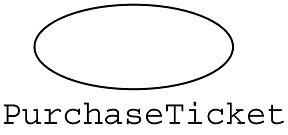
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



- An actor models an external entity which communicates with the system:
  - w **User**
  - w **External system**
  - w **Physical environment**
- An actor has a unique name and an optional description.
- Examples:
  - w **Passenger: A person in the train**
  - w **GPS satellite: Provides the system with GPS coordinates**

Use Case



A use case represents a class of functionality provided by the system as an event flow.

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Example

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

Exit condition:

- `Passenger` has ticket.

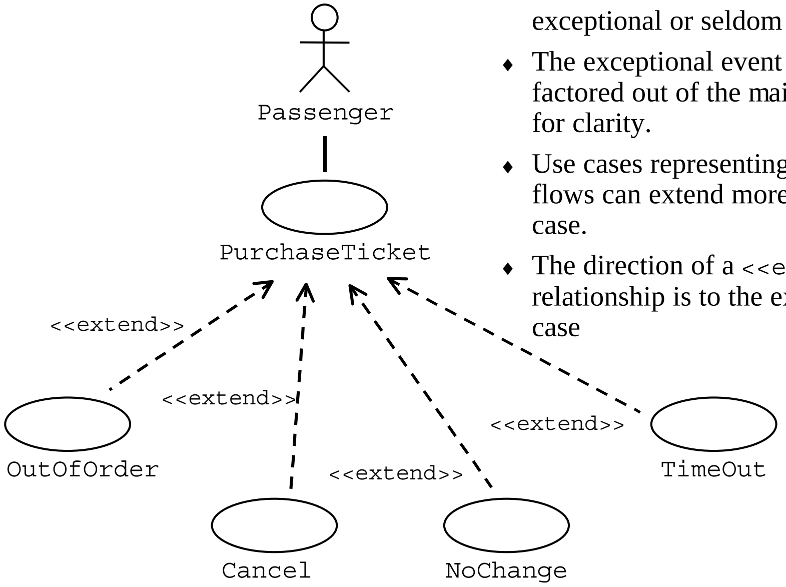
Event flow:

1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

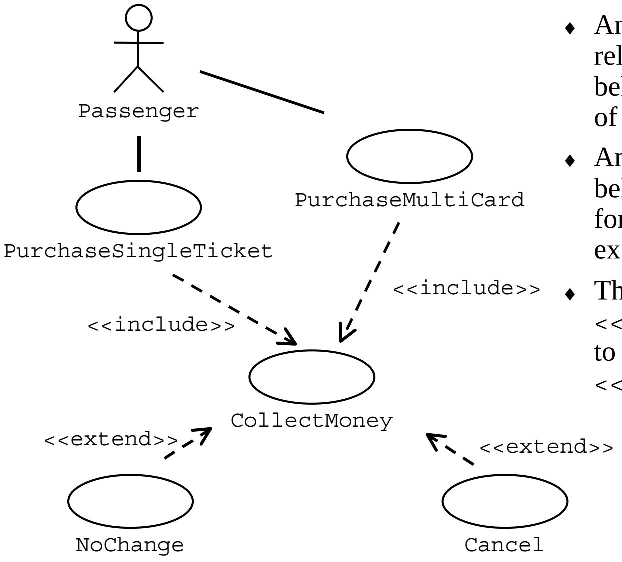
Exceptional cases!

The `<<extend>>` *Relationship*



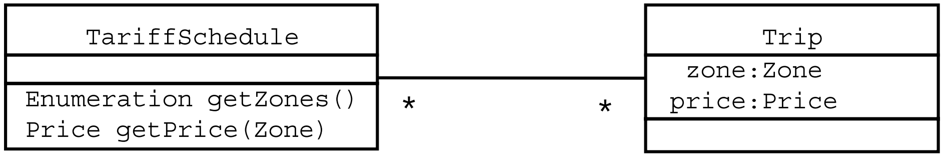
- `<<extend>>` relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a `<<extend>>` relationship is to the extended use case

The <<include>> Relationship



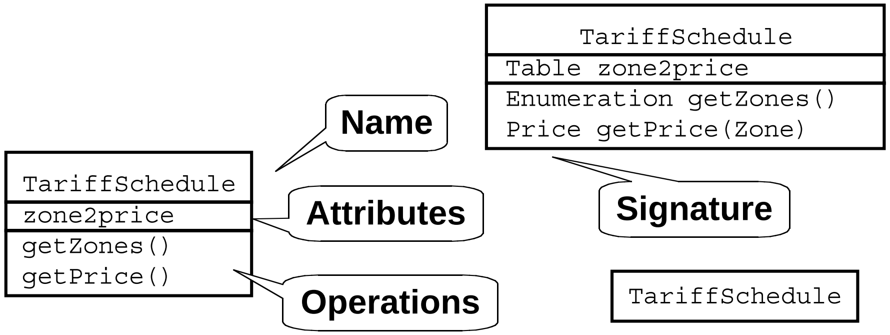
- An `<<include>>` relationship represents behavior that is factored out of the use case.
- An `<<include>>` represents behavior that is factored out for reuse, not because it is an exception.
- The direction of a `<<include>>` relationship is to the using use case (unlike `<<extend>>` relationships).

Class Diagrams



- Class diagrams represent the structure of the system.
- Class diagrams are used
  - **during requirements analysis to model problem domain concepts**
  - **during system design to model subsystems and interfaces**
  - **during object design to model classes.**

Classes



TariffSchedule

Table zone2price

Enumeration getZones()

Price getPrice(Zone)

Name

TariffSchedule

zone2price

getZones()

getPrice()

Attributes

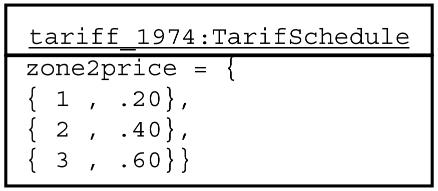
Signature

Operations

TariffSchedule

- A *class* represent a concept.
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

Instances



tariff_1974:TarifSchedule

zone2price = {

{ 1 , .20},

{ 2 , .40},

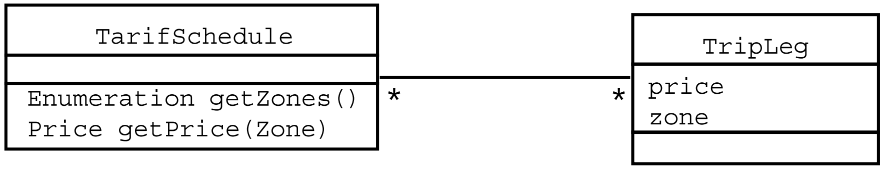
{ 3 , .60}}

- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Actor vs. Instances

- What is the difference between an actor and a class and an instance?
- Actor:
  - **An entity outside the system to be modeled, interacting with the system ("Pilot")**
- Class:
  - **An abstraction modeling an entity in the problem domain, inside the system to be modeled ("Cockpit")**
- Object:
  - **A specific instance of a class ("Joe, the inspector").**

Associations



TarifSchedule

Enumeration getZones()

Price getPrice(Zone)

*

*

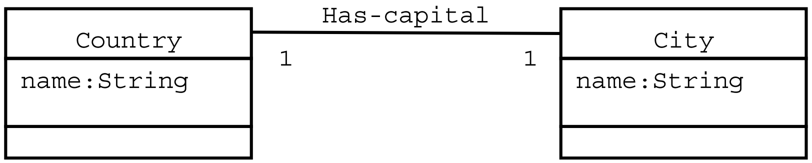
TripLeg

price

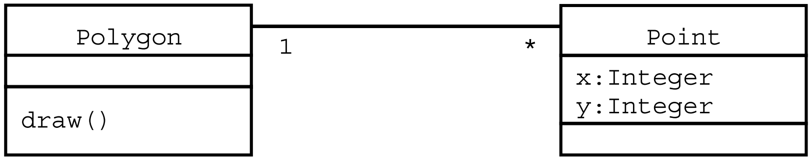
zone

- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

1-to-1 and 1-to-Many Associations



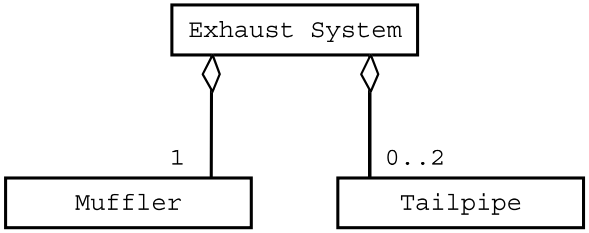
1-to-1 association



1-to-many association

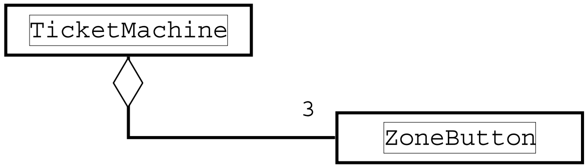
Aggregation

- An ***aggregation*** is a special case of association denoting a "consists of" hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.

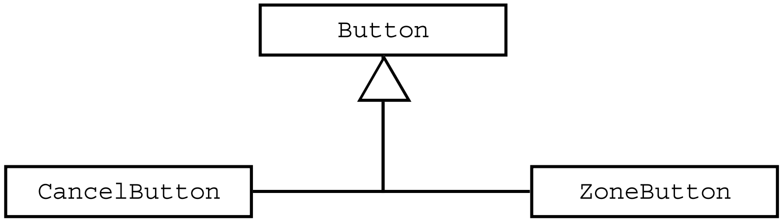


Composition

- A solid diamond denote ***composition***, a strong form of aggregation where components cannot exist without the aggregate.



Generalization



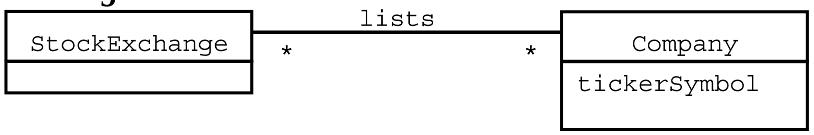
- Generalization relationships denote inheritance between classes.
- The children classes inherit the attributes and operations of the parent class.
- Generalization simplifies the model by eliminating redundancy.

From Problem Statement to Code

Problem Statement

A stock exchange lists many companies. Each company is identified by a ticker symbol

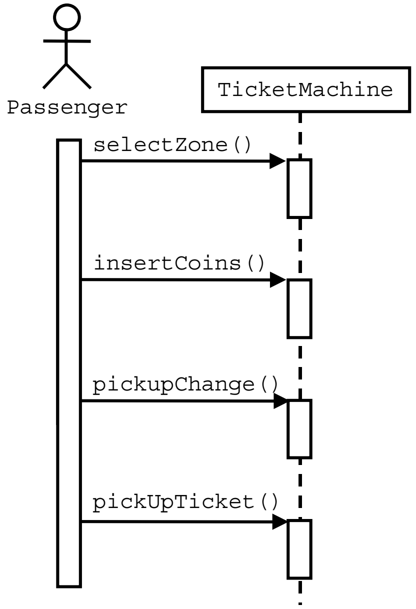
Class Diagram



Java Code

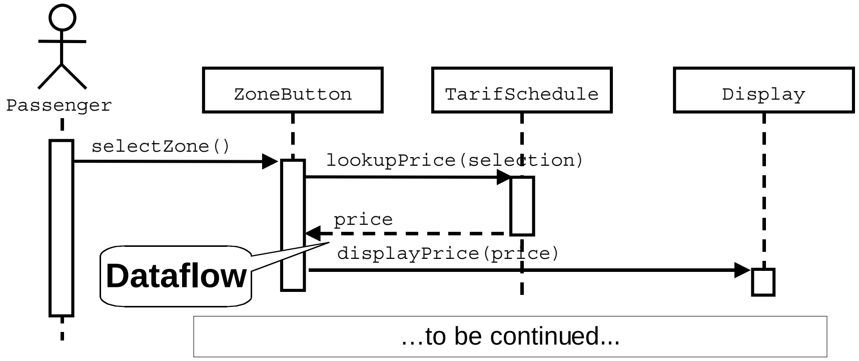
```
public class StockExchange {
    public Vector m_Company = new Vector();
};
public class Company {
    public int m_tickerSymbol;
    public Vector m_StockExchange = new Vector();
};
```

UML Sequence Diagrams



- Used during requirements analysis
  - **To refine use case descriptions**
  - **to find additional objects ("participating objects")**
- Used during system design
  - **to refine subsystem interfaces**
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

UML Sequence Diagrams: Nested Messages



…to be continued...

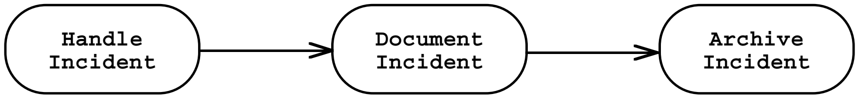
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

Sequence Diagram Observations

- UML sequence diagram represent behavior in terms of interactions.
- Complement the class diagrams which represent structure.
- Useful to find participating objects.
- Time consuming to build but worth the investment.

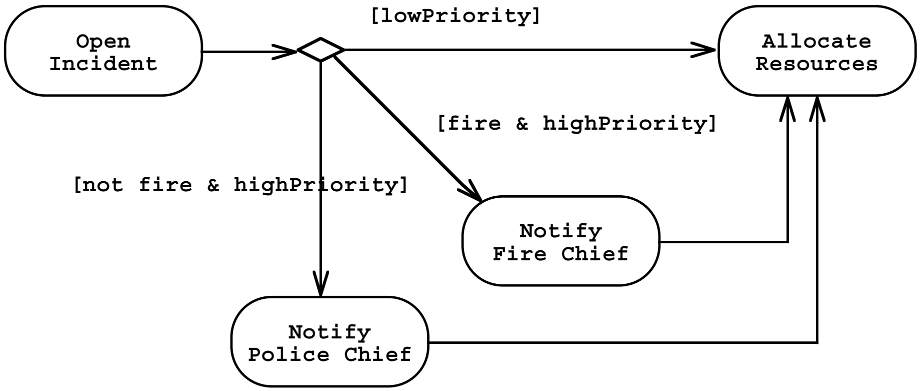
Activity Diagrams

- An activity diagram shows flow control within a system



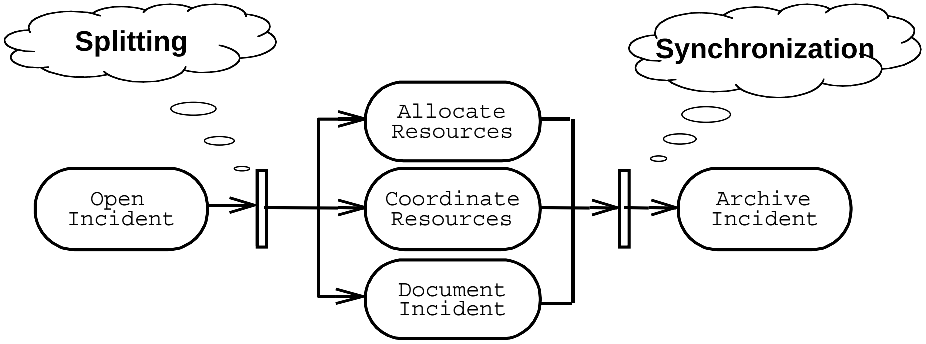
- An activity diagram is a special case of a state chart diagram in which states are activities (“functions”)
- Two types of states:
  - ***Action state:***
    - **Cannot be decomposed any further**
    - **Happens “instantaneously” with respect to the level of abstraction used in the model**
  - ***Activity state:***
    - **Can be decomposed further**
    - **The activity is modeled by another activity diagram**

Activity Diagram: Modeling Decisions



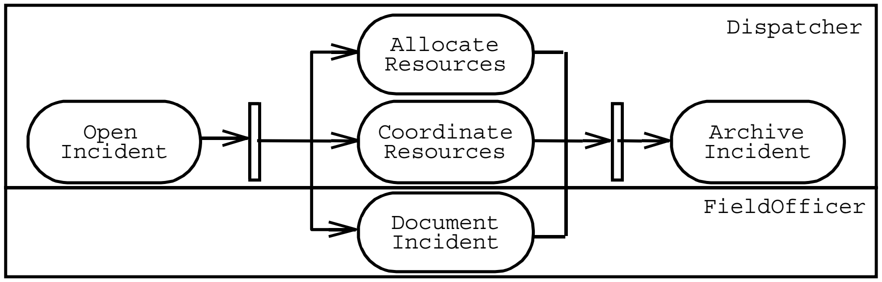
Activity Diagrams: Modeling Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



Activity Diagrams: Swimlanes

- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.



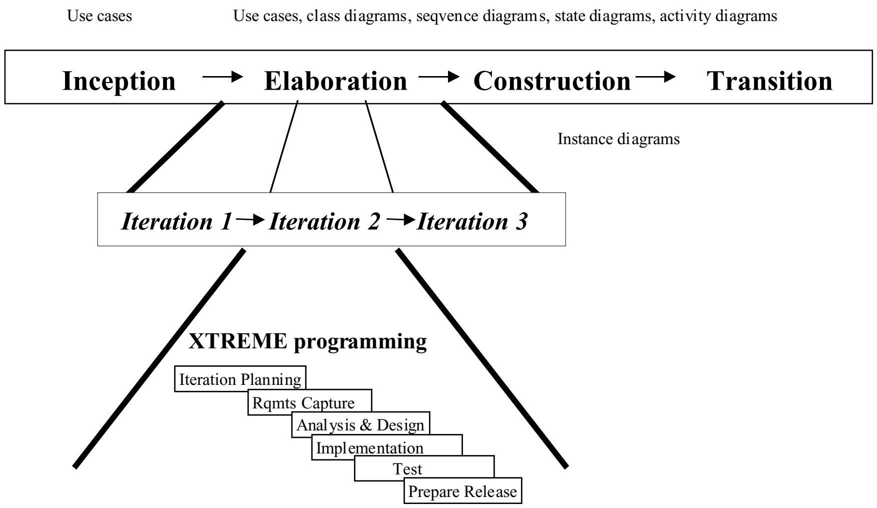
Extending the UML

- Stereotypes can be used to extend the UML notational elements
- Stereotypes may be used to classify and extend associations, inheritance relationships, classes, and components
- Examples:
  - **Class stereotypes: boundary, control, entity, utility, exception**
  - **Inheritance stereotypes: uses and extends**
  - **Component stereotypes: subsystem**

Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - **Powerful, but complex language**
  - **Can be misused to generate unreadable models**
  - **Can be misunderstood when using too many exotic features**
- We concentrate only on a few notations:
  - **Functional model: use case diagram**
  - **Object model: class diagram**
  - **Dynamic model: sequence diagrams, statechart and activity diagrams**

Rational Unified Software Life Cycle



Use cases

Use cases, class diagrams, seqvence diagrams, state diagrams, activity diagrams

Inception → Elaboration → Construction → Transition

Instance diagrams

Iteration 1 → Iteration 2 → Iteration 3

XTREME programming

Iteration Planning

Rqmts Capture

Analysis & Design

Implementation

Test

Prepare Release

What the Xprogramming Iterative Life Cycle Is Not

- It is not hacking
- It is not a playpen for developers
- It is not unpredictable
- It is not redesigning the same thing over and over until it is perfect
- It is not an excuse for not planning and managing a project

What the Xprogramming Iterative Life Cycle Is

- It is planned and managed
- It is predictable
- It accommodates changes to requirements with less disruption
- It is based on evolving executable prototypes, not documentation
- It involves the user/customer throughout the process
- It is risk driven

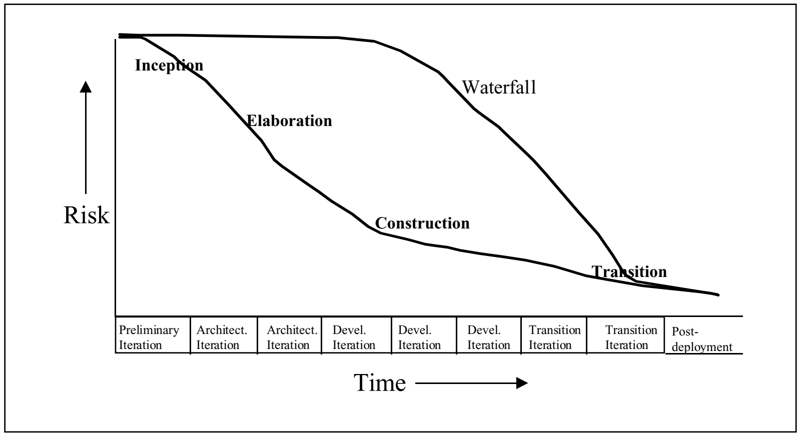
Three Important Features of the Iterative Approach

- Continuous integration
  - **Not done in one lump near the delivery date**
- Frequent, executable releases
  - **Some internal; some delivered**
- Attack risks through demonstrable progress
  - **Progress measured in products, not documentation or engineering estimates**

Resulting Benefits

- Releases are a forcing function that drives the development team to closure at regular intervals
  - **Cannot have the “90% done with 90% remaining” phenomenon**
- Can incorporate problems/issues/changes into future iterations rather than disrupting ongoing production
- The project’s supporting elements (testers/ writers, toolsmiths, QA, etc.) can better schedule their work

Risk Profile of an Iterative Development



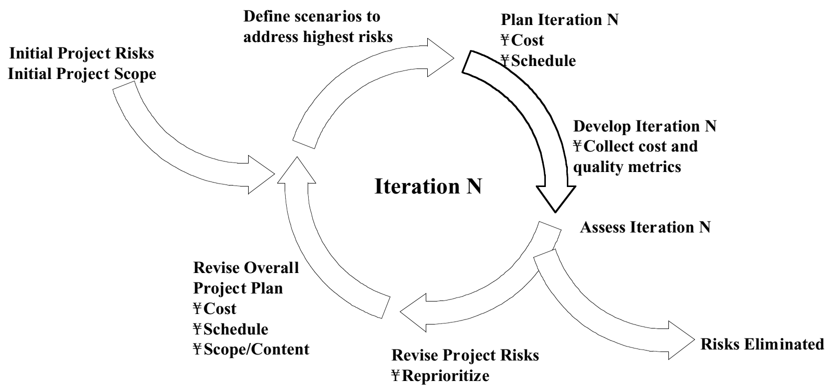
Risk Management Phase-by-Phase

- ♦ Inception
  - w **Bracket the project’s risks by building a proof of concept**
- ♦ Elaboration
  - w **Develop a common understanding of the system’s scope and desired behavior by exploring scenarios with end users and domain experts**
  - w **Establish the system’s architecture**
  - w **Design common mechanisms to address system-wide issues**

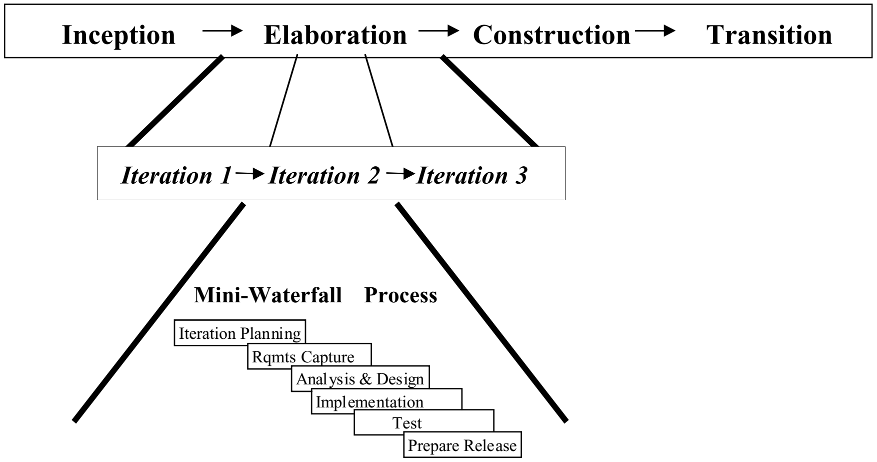
Risk Management Phase-by-Phase (cont.)

- ♦ Construction
  - w **Refine the design pattern**
  - w **Risk-driven iterations**
  - w **Continuous integration**
- ♦ Transition
  - w **Facilitate user acceptance**
  - w **Measure user satisfaction**
- ♦ Post-deployment cycles
  - w **Continue evolutionary approach**
  - w **Preserve architectural integrity**

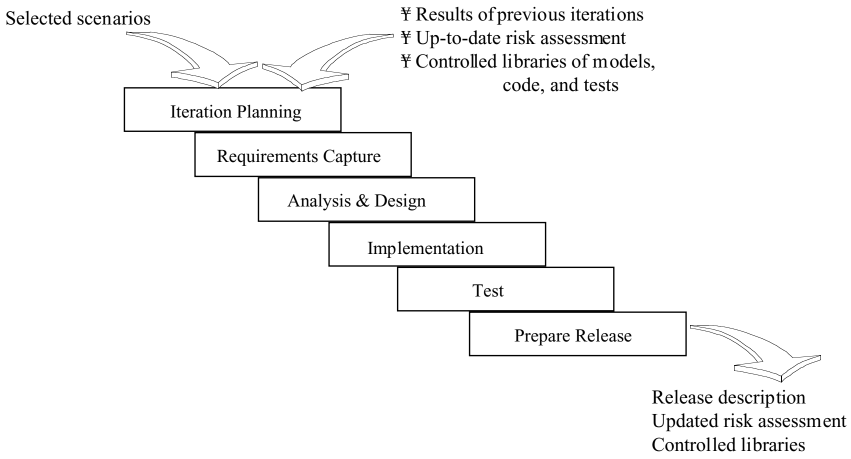
Risk Reduction Drives Iterations



Use Cases Drive the Iteration Process



The Iteration Life Cycle: A Mini-Waterfall



Selected scenarios

¥ Results of previous iterations
¥ Up-to-date risk assessment
¥ Controlled libraries of models, code, and tests

Iteration Planning

Requirements Capture

Analysis & Design

Implementation

Test

Prepare Release

Release description
Updated risk assessment
Controlled libraries

Detailed Iteration Life Cycle Activities

♦ Iteration planning

w **Before the iteration begins, the general objectives of the iteration should be established based on**

t **Results of previous iterations (if any)**

t **Up-to-date risk assessment for the project**

w **Determine the evaluation criteria for this iteration**

w **Prepare detailed iteration plan for inclusion in the development plan**

t **Include intermediate milestones to monitor progress**

t **Include walkthroughs and reviews**

Detailed Iteration Life Cycle Activities (cont.)

♦ Requirements Capture

w **Select/define the use cases to be implemented in this iteration**

w **Update the object model to reflect additional domain classes and associations discovered**

w **Develop a test plan for the iteration**

Detailed Iteration Life Cycle Activities (cont.)

♦ Analysis & Design

w **Determine the classes to be developed or updated in this iteration**

w **Update the object model to reflect additional design classes and associations discovered**

w **Update the architecture document if needed**

w **Begin development of test procedures**

♦ Implementation

w **Automatically generate code from the design pattern**

w **Manually generate code for operations**

w **Complete test procedures**

w **Conduct unit and integration tests**

Detailed Iteration Life Cycle Activities (cont.)

- Test
  - **Integrate and test the developed code with the rest of the system (previous releases)**
  - **Capture and review test results**
  - **Evaluate test results relative to the evaluation criteria**
  - **Conduct an iteration assessment**
- Prepare the release description
  - **Synchronize code and design patterns**
  - **Place products of the iteration in controlled libraries**

Work Allocation Within an Iteration

- Work to be accomplished within an iteration is determined by
  - **The (new) use cases to be implemented**
  - **The refactoring to be done**
- Packages make convenient work packages for developers
  - **High-level packages can be assigned to teams**
  - **Lower-level packages can be assigned to xprogrammeng pair developers**
- Use Cases make convenient work packages for development and test teams
- Packages are also useful in determining the granularity at which configuration management will be applied
  - **For example, check-in and check-out of individual packages**

Iteration Assessment

- Assess iteration results relative to the evaluation criteria established during iteration planning:
  - **Test Functionality**
  - **Test Performance**
  - **Test Capacity**
  - **Test Quality measures**
- Consider external changes that have occurred during this iteration
  - **For example, changes to requirements, user needs, competitor’s plans**
- Determine what refactoring, if any, is required and assign it to the remaining iterations

Selecting Iterations

- How many iterations do I need?
  - **On projects taking 6 months or less, 2 to 4 iterations are typical**
- Are all iterations on a project the same length?
  - **Usually**
  - **Iteration length may vary by phase. For example, elaboration iterations may be shorter than construction iterations**

The First Iteration

- The first iteration is usually the hardest
  - **Requires the entire development environment and most of the development team to be in place**
  - **Many tool integration issues, team-building issues, staffing issues, etc. must be resolved**
- Teams new to an iterative approach are usually overly-optimistic
- Be modest regarding the amount of functionality that can be achieved in the first iteration
  - **Otherwise, completion of the first iteration will be delayed,**
  - **The total number of iterations reduced, and**
  - **The benefits of an iterative approach reduced**

There Is No Silver Bullet

- Remember the main reason for using the iterative life cycle:
  - **You do not have all the information you need up front**
  - **Things will change during the development period**
- You must expect that
  - **Some risks will not be eliminated as planned**
  - **You will discover new risks along the way**
  - **Some rework will be required; some lines of code developed for an iteration will be thrown away**
  - **Requirements will change along the way**