



**BITS Pilani**  
Pilani Campus

# BITS Pilani presentation

Dr. Yashvardhan Sharma  
Computer Science and Information Systems





# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.1**

# What is Analysis and Design?

---

**Analysis** - investigation of the problem (what);

- What does the system do?
  - Investigation of the problem.
- 
- **Design** - conceptual solution to fulfill the requirements (how); how will the system do what it is intended to do.
  - What (conceptual) solution will full the requirements



# What is OO analysis and design?

---

**Essence of OO analysis** - consider a problem domain from the perspective of objects (real world things, concepts)

- **Essence of OO design** - define the solution as a collection of software objects (allocating responsibilities to objects)

OO Analysis - in the case of library information systems, one would *find* concepts like *book*, *library*, *patron*



# Examples

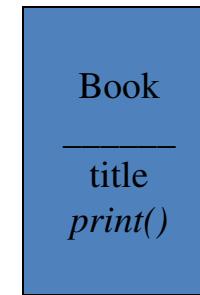
- OO Design - emphasis on defining the software objects; ultimately these objects are implemented in some programming language; *Book* may have a method named *print*.

# Example - contd.

Domain concept



Representation in  
analysis of concepts



**public class Book**  
{  
  **public void print();**  
  **private string title;**  
}

Representation in OO  
programming language

# OO Concepts-Objects

---

[https://docs.oracle.com/javase/tutorial/java/concepts/ /](https://docs.oracle.com/javase/tutorial/java/concepts/)

- **Objects:** Anything that has a state and exhibits behavior.
- **Real world objects:** Bicycle, student, course, dog, university,....
- **Software objects:** Model real-world or abstract objects (e.g. a list).
- **Methods:** Procedures through which objects communicate amongst themselves. Example: Bicycle: brake, park. Dog: bark, eat. Student: register, study.
- **Attributes:** Variables that hold state information. Bicycle: speed, color, owner. Dog:name, breed. Student: name, ID.

# OO Concepts-Class

**Class:** Prototype for all objects of a certain kind. Student, animal, university, shape, etc.

- **Objects:** Created from a class. For example: s1, s2 are objects from class Student.  
BITS and Purdue are objects from class University. myCircle and mySquare are objects from class Shape.
- **Inheritance:** A class inherits attributes and methods from its super class. This allows hierarchical organization of classes.
- **Interface:** A contract between a class and its users. A class implements an interface (methods and attributes).

# Object-oriented Design

---

*With traditional analysis methods, we model the world using functions or behaviors as our building blocks... With object-oriented analysis, we model reality with objects as our building blocks.”*

# Why we use objects ?

---

We deal with objects in everyday life – the world is full of objects

With object-oriented programming, data and the methods that act on the data are nicely packaged together (encapsulation)

Commonalities between objects can be captured with a common base class (inheritance), while their differences can be preserved (polymorphism)

# The problem: How do we find objects ?

---

This is a necessary step between requirements/specifications and the actual implementation of the solution.

# What are business processes?

---

*First step* - consider what the business must do; in the case of a library  
- lending books, keeping track of due dates, buying new books.

- *In OO terms* - requirements analysis; represent the business processes in textual narration (Use Cases).

# Roles in the organization

---

Identify the **roles** of people who will be involved in the business processes.

- In OO terms - this is known as **domain analysis**
- Examples - customer, library assistant, programmer, navigator, sensor, etc.

*Examples from class projects?*

# Who does what? Collaboration

Business processes and people identified; time to determine how to fulfill the processes and who executes these processes.

- In OO terms - object oriented design; assigning responsibilities to the various software objects.
- Often expressed in [class diagrams](#).

# In Summary...

## Business Analogy

What are the  
business processes?

What are  
employee roles?

Who is  
responsible for  
what?

## OO Analysis and Design

Requirements  
analysis

Domain analysis

Responsibility  
assignment;

## Associated Documents

Use cases

Conceptual  
model

Design class  
diagrams

# Simple example to see big picture

---

Define use cases

- Define conceptual model
- Define collaboration diagrams
- Define design class diagrams

Example: Dice game a player rolls two die. If the total is 7 they win; otherwise they lose

# Define use cases

---

**Use cases** - narrative descriptions of **domain processes** in a structured prose format

**Use case:** Play a game  
**Actors:** Player

**Description:** This use case begins when the player picks up and rolls the die....

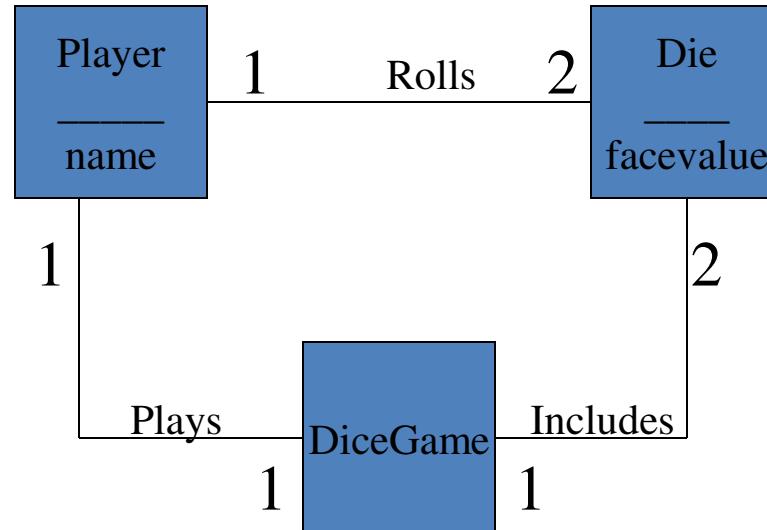
# Define domain model

---

OO Analysis concerns

- specification of the problem domain
  - identification of concepts (objects)
- 
- Decomposition of the problem domain includes
    - identification of objects, attributes, associations
  - Outcome of analysis expressed as a **domain model.**

# Domain model - game of dice



Conceptual model is not a description of the software components; it represents concepts in the real world problem domain

# Collaboration diagram

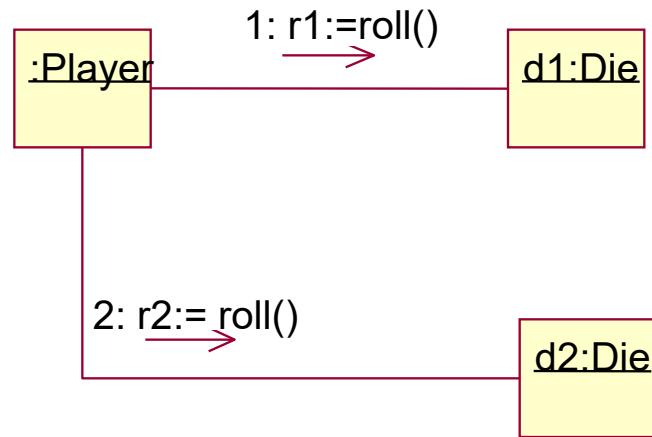
---

Essential step - allocating responsibility to objects and illustrating how they interact with other objects.

- OO Design is concerned with
  - defining logical software specification that fulfills the requirements
- Expressed as Collaboration diagrams

*Collaboration diagrams express the flow of messages between Objects.*

# Example - collaboration diagram



# Defining class diagrams

---

## Key questions to ask

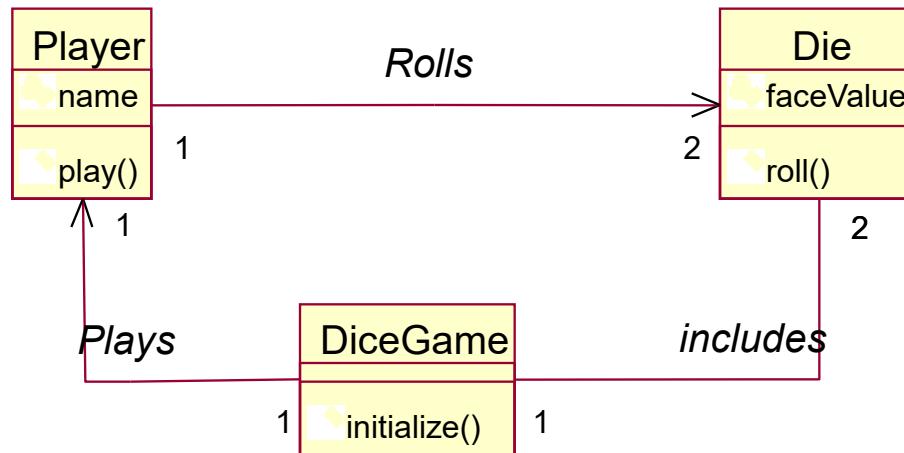
- How do objects connect to other objects?
  - What are the behaviors (methods) of these objects?
- 
- Collaboration diagrams suggests connections; to support these connections *methods* are needed
  - Expressed as class diagrams

# Class diagram

---

The class diagram is core to object-oriented design. It describes the types of objects in the system and the static relationships between them.

# Example - Class diagram



*A line with an arrow at the end may suggest an attribute.  
For example, **DiceGame** has an attribute that points to an instance of a **Player***

# Defining Models and Artifacts

---

## Objectives

- analysis and design models
  - familiarize UML notations and diagrams
- 
- Real world software systems are inherently complex
  - Models provide *a* mechanism for decomposition and expressing specifications

# Analysis and Design models

---

Analysis model - models related to an investigation of the domain and problem space (Use case model qualifies as an example)

- Design model - models related to the solution (class diagrams qualifies as an example)



**BITS** Pilani  
Pilani Campus

# BITS Pilani presentation

- Dr. Yashvardhan Sharma
- Computer Science and Information Systems



# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.2**

# Defining Models and Artifacts

---

## Objectives

- analysis and design models
  - familiarize UML notations and diagrams
- 
- Real world software systems are inherently complex
  - Models provide *a* mechanism for decomposition and expressing specifications

# Analysis and Design models

---

Analysis model - models related to an investigation of the domain and problem space (Use case model qualifies as an example)

- Design model - models related to the solution (class diagrams qualifies as an example)

# Unified Modeling Language

---

- “A language for specifying, visualizing and constructing the artifacts of software system” [Booch, Jacobson, Rumbaugh]
- It is a notational system aimed at modeling systems using OO concepts
  - ... not a methodology
  - ... not a process

# Software Development Process

Steps correspond to one or more tasks related to software development.

- Tasks:
  - Requirements gathering
  - Requirements analysis
  - Design
  - Coding
  - Integration
  - Test
  - Delivery
  - Maintenance
  - Training
- **Software life cycle:** Software Life Cycle consists of all phases from its inception until its retirement. These are (for **Unified Process**): Inception, elaboration, construction, transition.

# The software process

---

Software process: organizing a structured set of activities to develop software systems.

Many different software processes but all involve the following activities:

- Specification – defining what the system should do;
- Design and implementation – defining the organization of the system and implementing the system;
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

# Software Process Model descriptions

---

- A software process model is an abstract representation of a process. It presents a description of a process.
- Process descriptions may also include:
  - Products, which are the outcomes of a process activity;
  - Roles, which reflect the responsibilities of the people involved in the process;
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.
- Notation: activities, products

# The Software Development Process

- Why a Process?
  - Software projects are large, complex, sophisticated
  - time to market is key
  - many facets involved in getting to the end
- Common process should
  - integrate the many facets
  - provide guidance to the order of activities
  - specify what artifacts need to be developed
  - offer criteria for monitoring and measuring a project

# Software Development Process

---

*"It is better not to proceed at all, than to proceed without method." --Descartes*

The Software Development Process:

- The framework for the set of tasks that are required to develop a software system.
- Process defines *how* a software product is developed and maintained.
- A well-defined and rigorously enforced process forms the basis for high-quality software development.

***A good software process is repeatable, predictable, learnable, measurable and improvable.***

# A software life cycle is a process

- A process involves activities, constraints and resources that produce an intended output.
- Each process activity, e.g., design, must have entry and exit criteria—**why?**
- A process uses resources, subject to constraints (e.g., a schedule or a budget)
- A process is organized in some order or sequence, structuring activities as a whole
- A process has a set of guiding principles or criteria that explain the goals of each activity

# Software Life Cycle

---

Having a defined process is essential

- life cycle is the series of steps that software undergoes from concept exploration through retirement

Maturity of the process is some gauge of success of organization

# Software Life Cycle Model

---

## Definition

- Describes an abstract collection of software processes that share common characteristics such as timing between phases, entry and exit criteria for phases.

## The models specifies

- the various phases of the process
  - e.g., requirements, specification, design...
- the order in which they are carried out

# Importance of Lifecycle Models

---

Provide guidance for project management

- what major tasks should be tackled next? milestones!
- what kind of progress has been made?

The necessity of lifecycle models

- character of software development has changed
  - early days: programmers were the primary users
  - modest designs; potential of software unknown
- more complex systems attempted
  - more features, more sophistication → greater complexity, more chances for error
  - heterogeneous users

# Life Cycle Models: Summary [1]

---

- **Build and fix:** Acceptable for short programs that do not require maintenance.
- **Waterfall:** Disciplined approach, document driven; delivered product may not meet client needs.
- **Rapid prototyping:** Ensures that delivered product meets client needs; might become a build-and-fix model.
- **Incremental:** Maximizes early return on investment; requires open architecture; may degenerate into build-and-fix.

# Life Cycle Models: Summary [2]

---

- **Spiral:** Risk driven, incorporates features of the above models; useful for very large projects
- **UDP:** Iterative, supports OO analysis and design; may degenerate into code-a-bit-test-a-bit.

# Object-Oriented Life-Cycle Models

- Need for iteration within and between phases
  - Fountain model
  - Recursive/parallel life cycle
  - Unified software development process
- All incorporate some form of
  - Iteration
  - Parallelism
  - Incremental development
- Danger
  - CABTAB

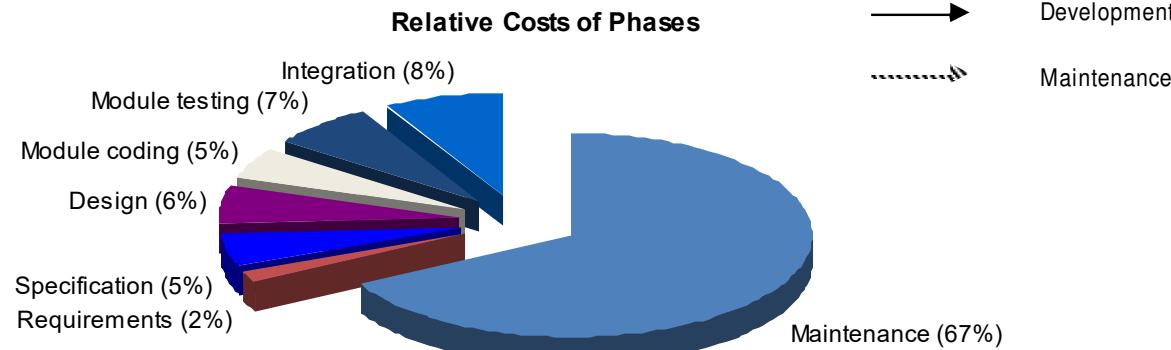
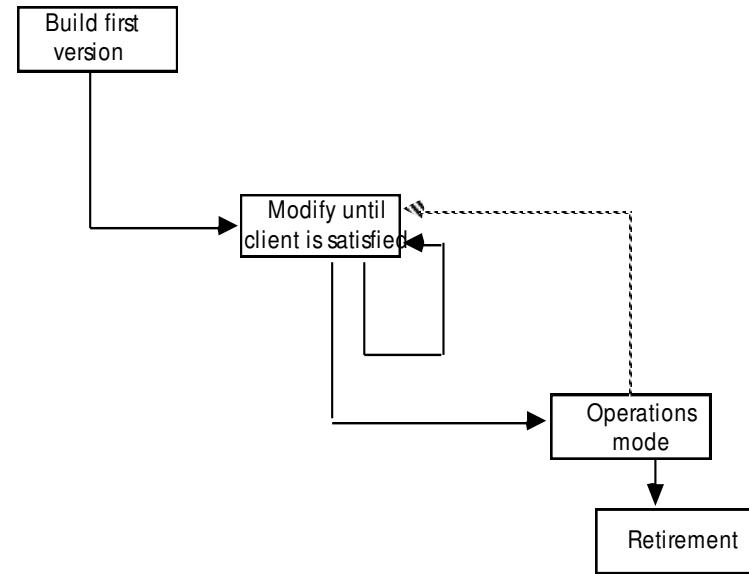
# Lifecycle Models

## Build-and-fix

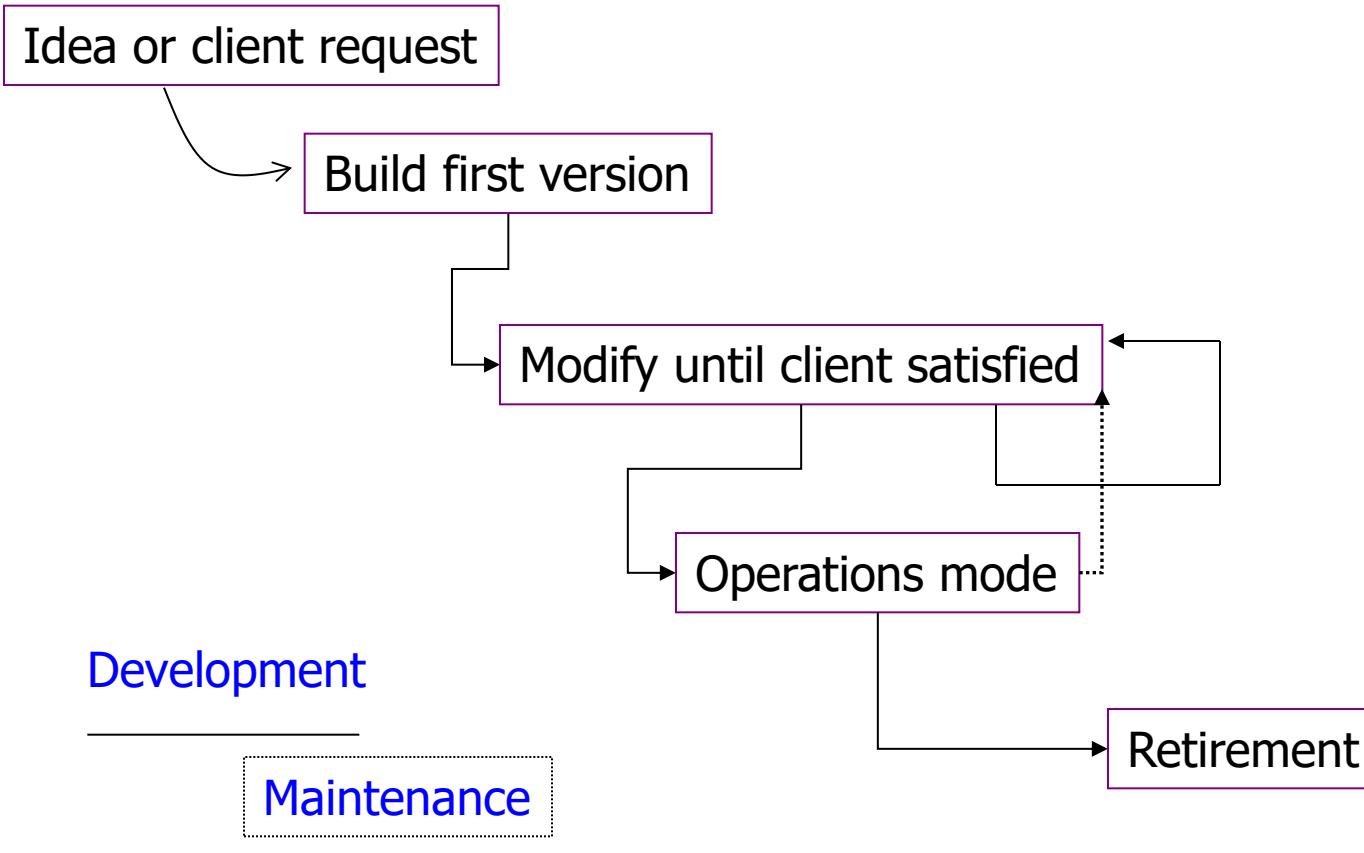
- develop system
  - without specs or design
  - modify until customer is satisfied

## Why doesn't build-and-fix scale?

- changes during maintenance
  - most expensive!



# Build and fix model [1]



# Build and fix model [2]

---

Product is constructed without specifications.

- There is no explicit design. However, a design will likely evolve in the mind of the developer.
- Modify until customer is satisfied.
- The approach might work for small programming projects [TA 162/252].

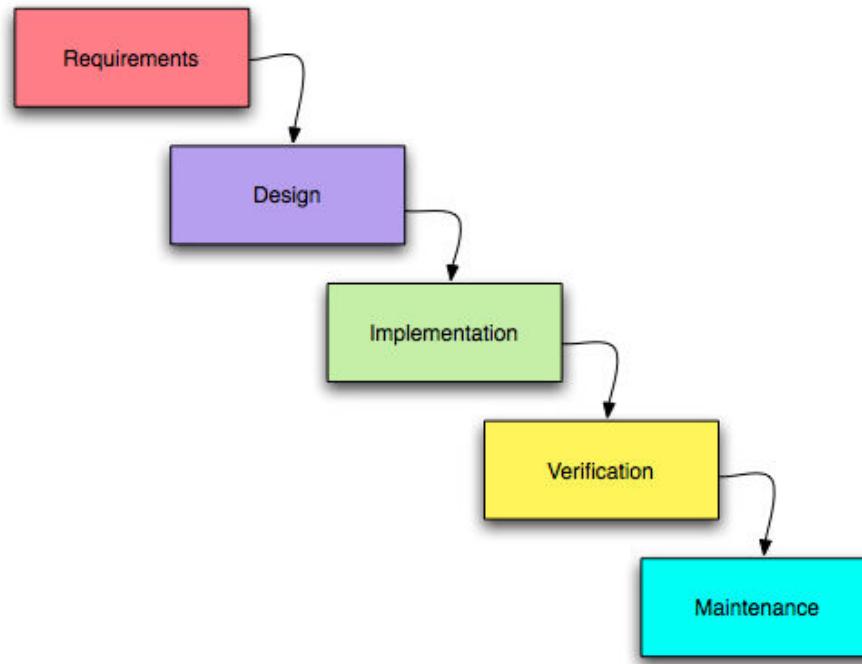
# Build and fix model [3]

---

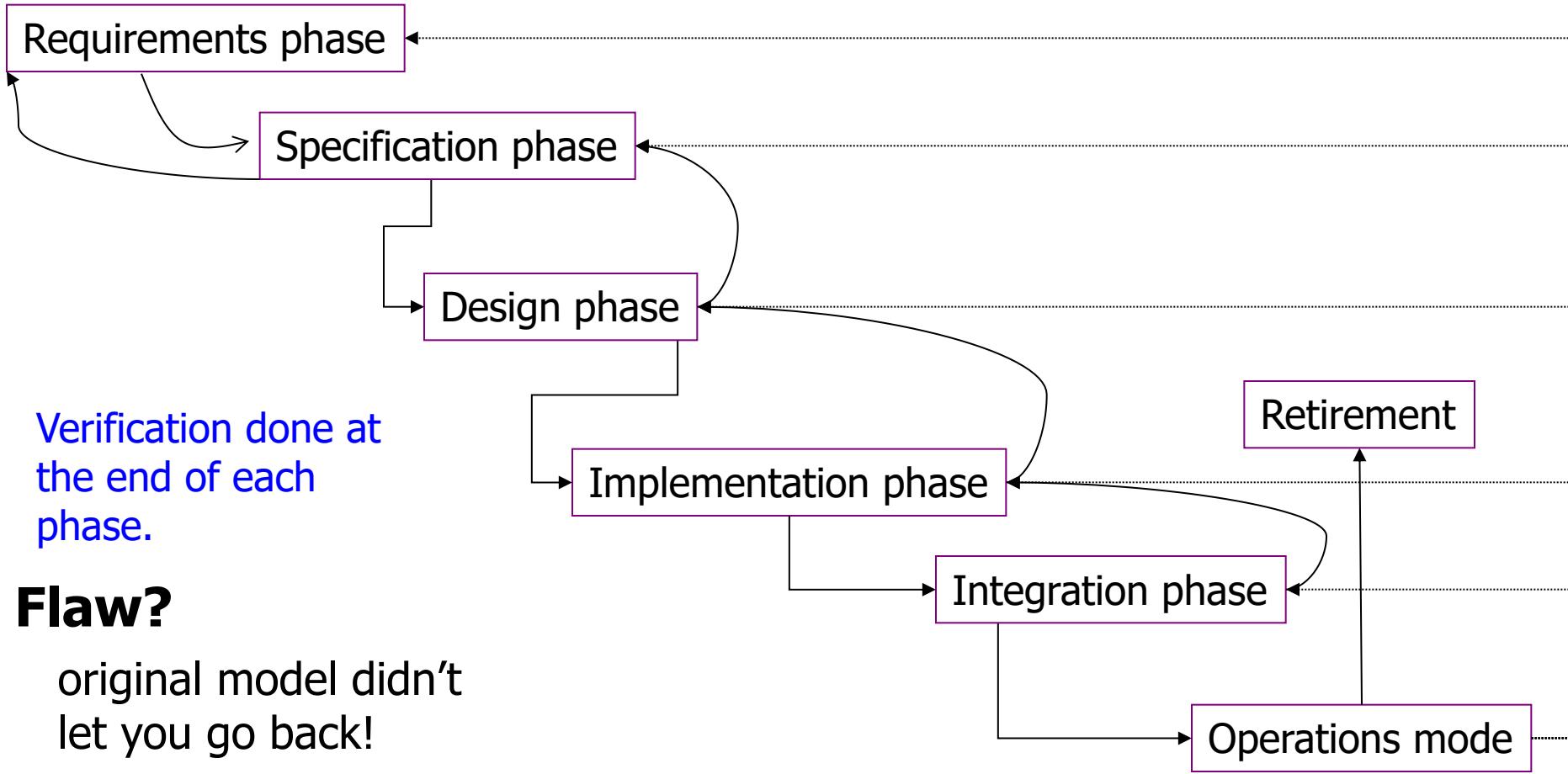
Cost of fixing an error increases as one moves away from the phase in which the error was injected.

- There is a good chance that many errors will be found in the operations phase thereby leading to high cost of maintenance.
  
- Rarely used in commercial projects.

# Waterfall Model



# Waterfall model [1]



# Waterfall model [2]

---

Popular in the 70's.

- Requirements are determined and verified with the client and members of the SQA group.
- Project management plan is drawn, cost and duration estimated, and checked with the client and the SQA group
- Then the design (i.e. "How is the product going to do what it is supposed to do.") begins and the project proceeds as in the figure.

# Waterfall model [3]

---

Each phase terminates only when the documents are complete and approved by the SQA group.

- Testing is inherent in every phase
  
- Maintenance begins when the client reports an error after having accepted the product. It could also begin due to a change in requirements after the client has accepted the product.

# Waterfall model: Advantages

## Disciplined approach

- Careful checking by the Software Quality Assurance Group at the end of each phase.
- Testing in each phase.
- Documentation available at the end of each phase.
- Concrete evidence of progress.
- Works best when you know what you're doing
  - when requirements are stable & problem is well-known

# Waterfall model: Disadvantages

Documents do not always convey the entire picture.  
Customers cannot understand these

- imagine an architect just showing you a textual spec!

first time client sees a working product is after it has been coded. Problem here?

- leads to products that **don't meet customers needs**

- Assumes feasibility before implementation
  - re-design is problematic
- Feedback from one phase to another might be too late and hence expensive.
- Linear nature leads to 'blocking states'
- Difficult to estimate time and cost for each stage of the development process.

# Unified Development Process [2]

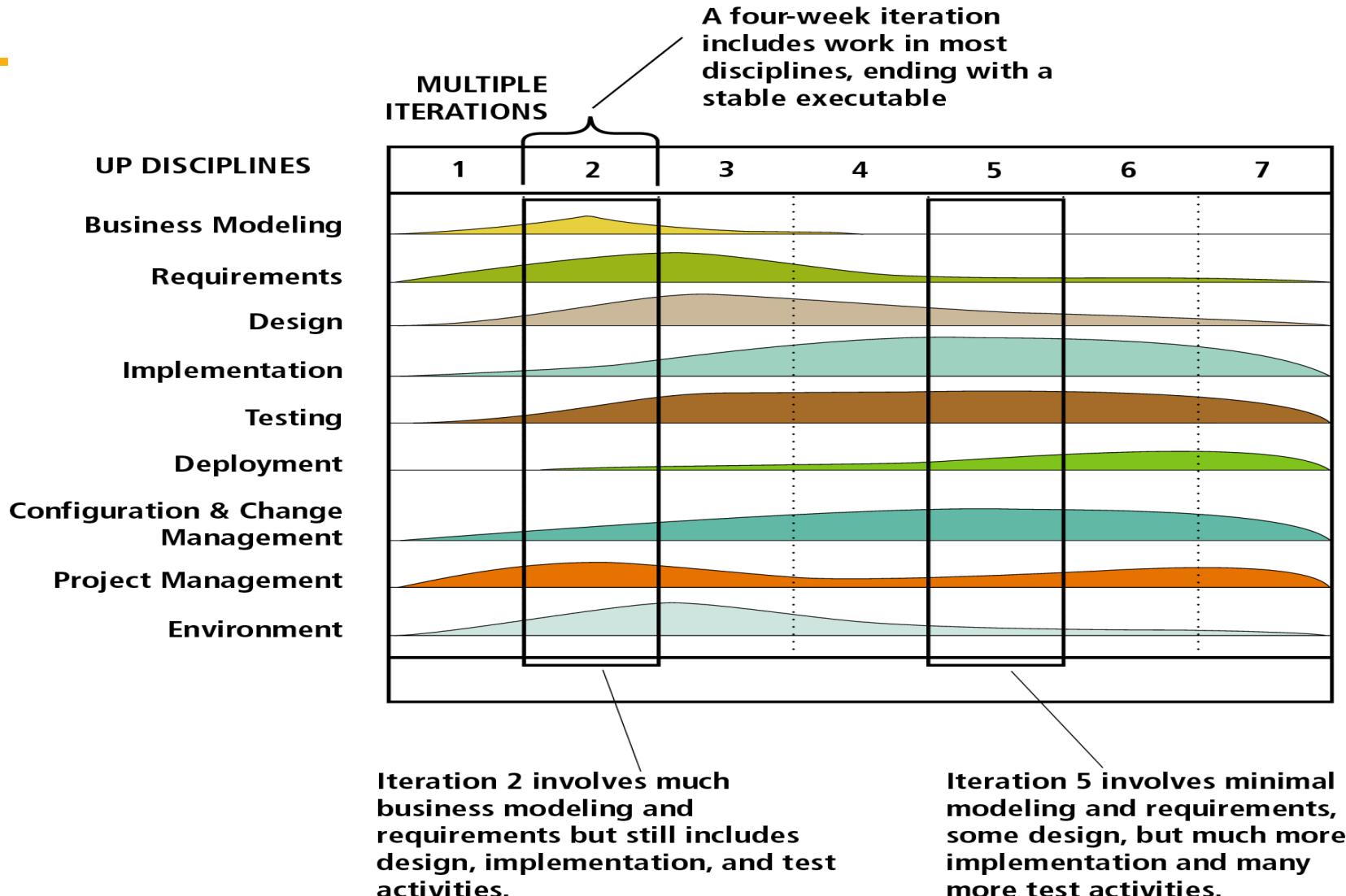
---

Architecture is built during early iterations.

- Early iterations seek feedback from the customer. Risk and value to customer is managed through early feedback.

Customer is engaged continuously in evaluation and requirements gathering.

# Unified Development Process [3]



# The Unified Process

Component based - meaning the software system is built as a set of software components interconnected via interfaces

Uses the Unified Modeling Language (UML)

**Use case driven**  
**Architecture-centric**  
**Iterative and incremental**

This is what makes  
the Unified process  
Unique

**Component:** A physical and replaceable part of a system that conforms to and provides realization of a set of interfaces.

**Interface:** A collection of operations that are used to specify a service of a class or a component

# The Unified Process



Based around the 4Ps - People, Project, Product, Process

# The Unified Process

- Use Case driven
  - A use case is a piece of functionality in the system that gives a user a result of value.
- Use cases capture functional requirements
- Use case answers the question: *What is the system supposed to do for the user?*

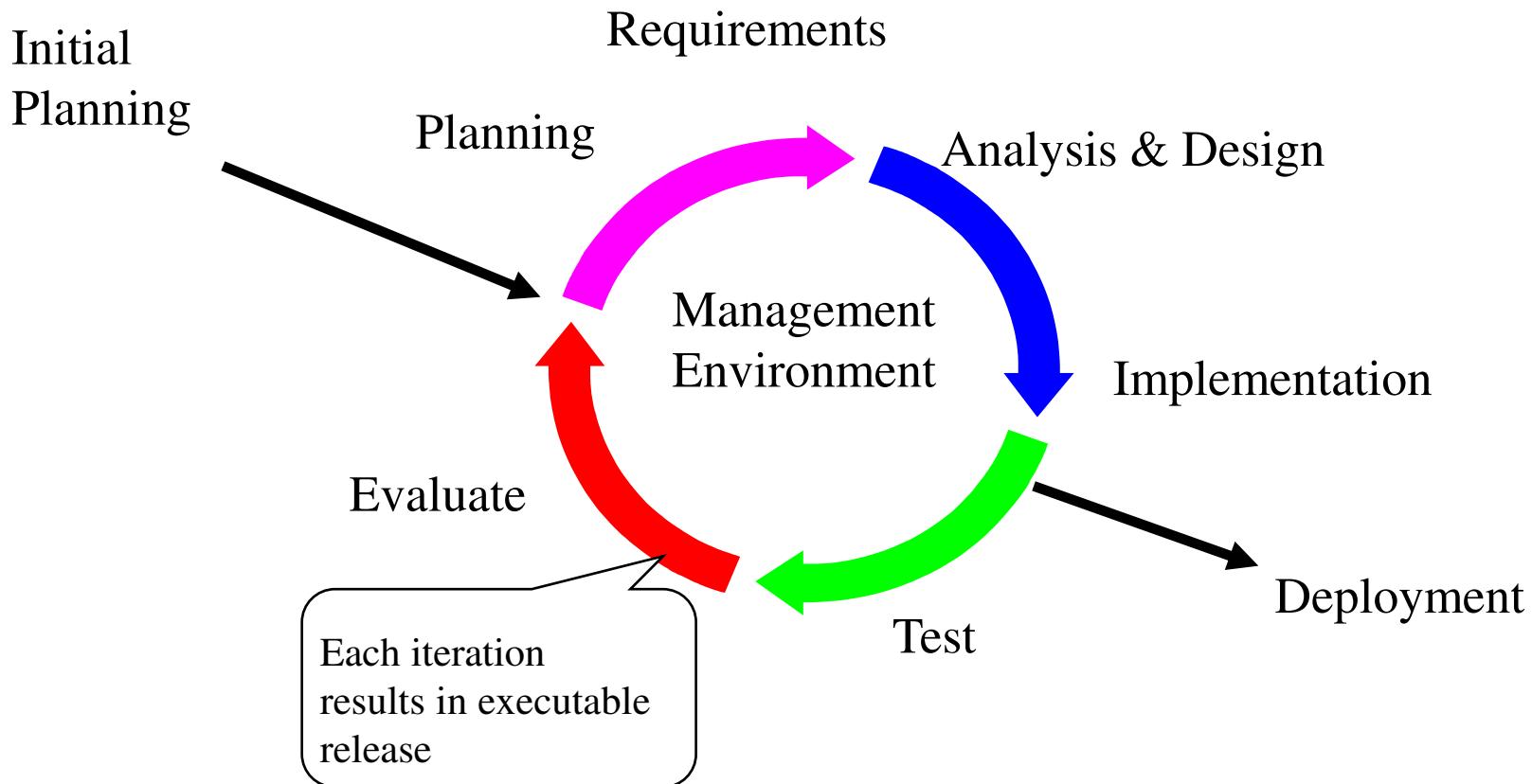
# The Unified Process

- Architecture centric
  - similar to architecture for building a house
  - Embodies the most significant static and dynamic aspects of the system
  - Influenced by platform, OS, DBMS etc.
  - Related as ***function*** (use case) and ***form*** (architecture)
  - Primarily serves the realization of use cases
  - The form must allow the system to evolve from initial development through future requirements (i.e. the design needs to be flexible)

# The Unified Process

- **Iterative and Incremental**
  - commercial projects continue many months and years
  - to be most effective - break the project into *iterations*
- Every iteration - identify use cases, create a design, implement the design
- Every iteration is a complete development process

# An iterative and incremental process



(Kruchten, 1999)

# Iterations

- Iterations must be selected & developed in a planned way i.e. in a logical order - early iterations must offer utility to the users
  - iteration based on a group of use cases extending the usability of the system developed so far
  - iterations deal with the most important risks first
  - not all iterations are additive - some replace earlier “superficial” developments with a more sophisticated and detailed one.

# Benefits of an iterative approach

- Risks are mitigated earlier
- Change is more manageable
- Higher level of reuse
- Project team can learn along the way
- Better overall quality

# Unified Software Development Process (UP)

Selects from best practices to

- Provide a generic process framework
  - instantiate/specialize for specific application areas, organizations, project sizes, etc.
- Define a set of activities (*workflows*)
  - transforms users' requirements into a software system
- Define a set of models
  - from abstract (user-level) to concrete (code)
- Allow component-based development
  - software components interconnected via well-defined interfaces
  - use-case (UML) and risk driven
  - architecture-centric
  - iterative and incremental

## Unified Process - Milestones

**Milestone:** a management decision point in a project that determines whether to authorize movement to the next iteration/phase

**Inception phase** - agreement among customers/developers on the system's life cycle objectives

**Elaboration phase** - agreement on the viability of the life cycle architecture, business case and project plan

**Construction phase** - agreement on the acceptability of the software product both operationally and in terms of cost

**Transition phase** - final agreement on the acceptability of the software product

# Lifecycle Phases

- Inception – “Daydream”
- Elaboration – “Design/Details”
- Construction – “Do it”
- Transition – “Deploy it”
- Phases are *not* the classical requirements/  
design/coding/implementation processes
- Phases iterate over many cycles

# Timeboxing

Management of a UP project.

Iterations are “timeboxed” or fixed in length.

Iteration lengths of between two to six weeks are recommended.

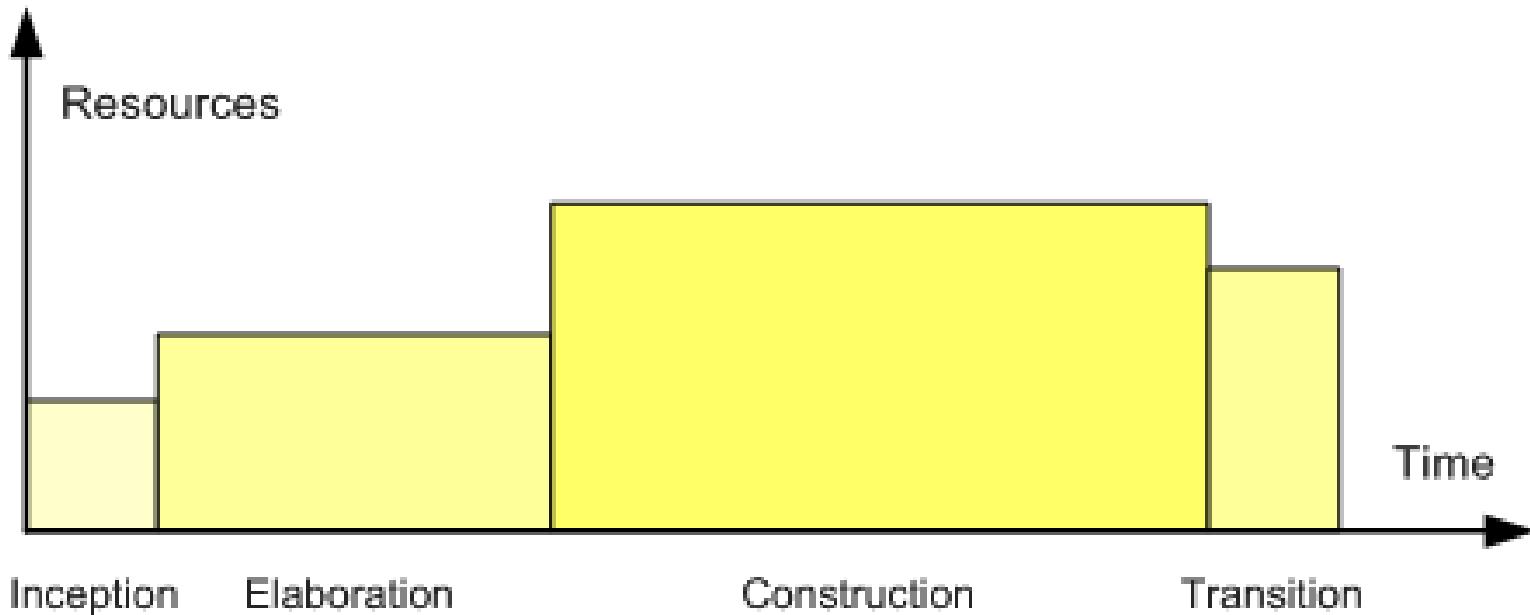
Each iteration period has its own development plan.

If all the planned activities cannot be completed during an iteration cycle, the completion date should not be extended, but rather tasks or requirements from the iteration should be removed and added to the next iteration cycle.

# The Unified Process

- The Unified Software Development Process is a definition of a complete set of activities to transform users' requirements through a consistent set of artifacts into a software product
- Look at the whole process
  - Life cycle
  - Artifacts
  - Workflows
  - Phases
  - Iterations
- A process is described in terms of ***workflows*** where a workflow is a set of activities with identified artifacts that will be created by those activities

# Phases in Unified Process



Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

## Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.

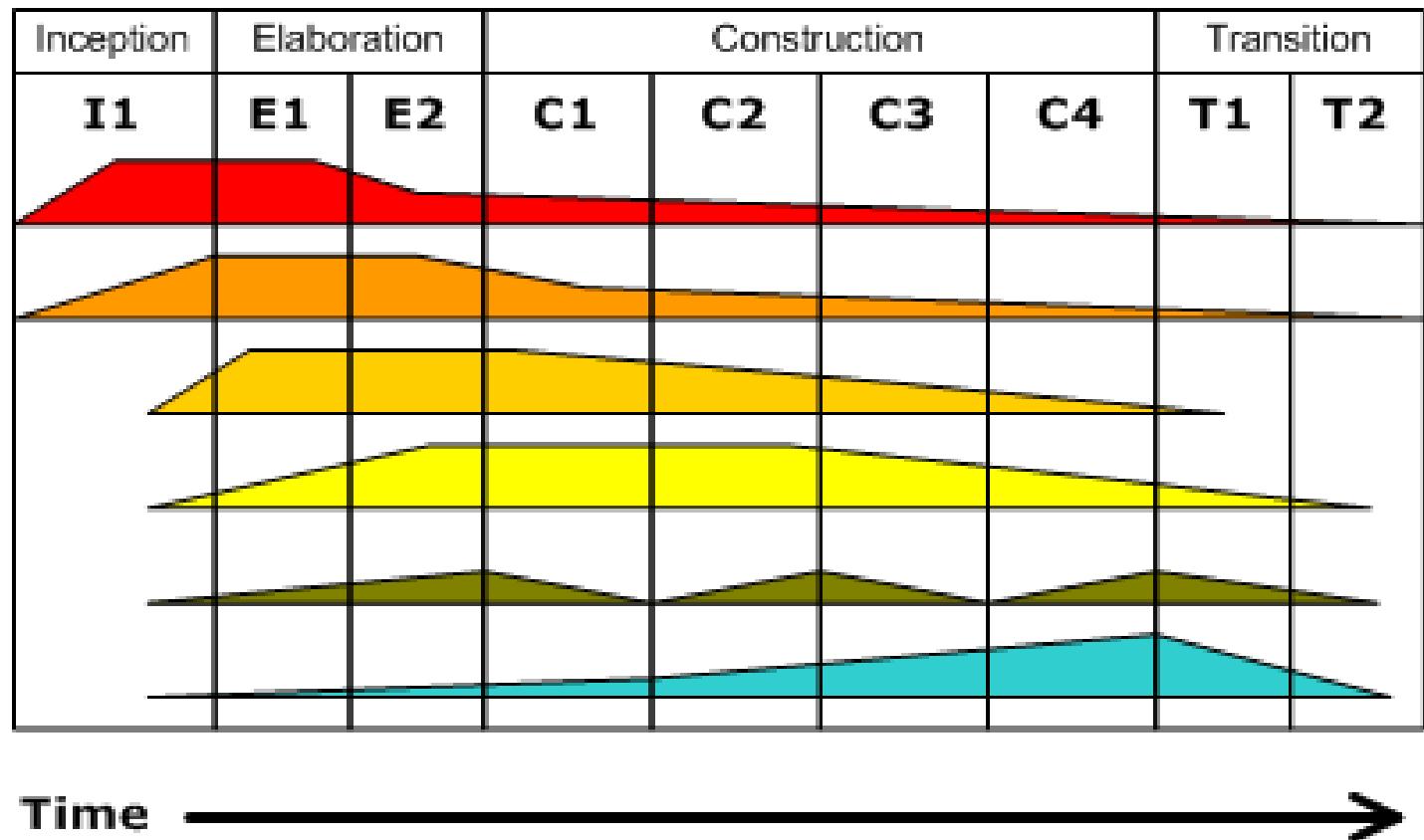
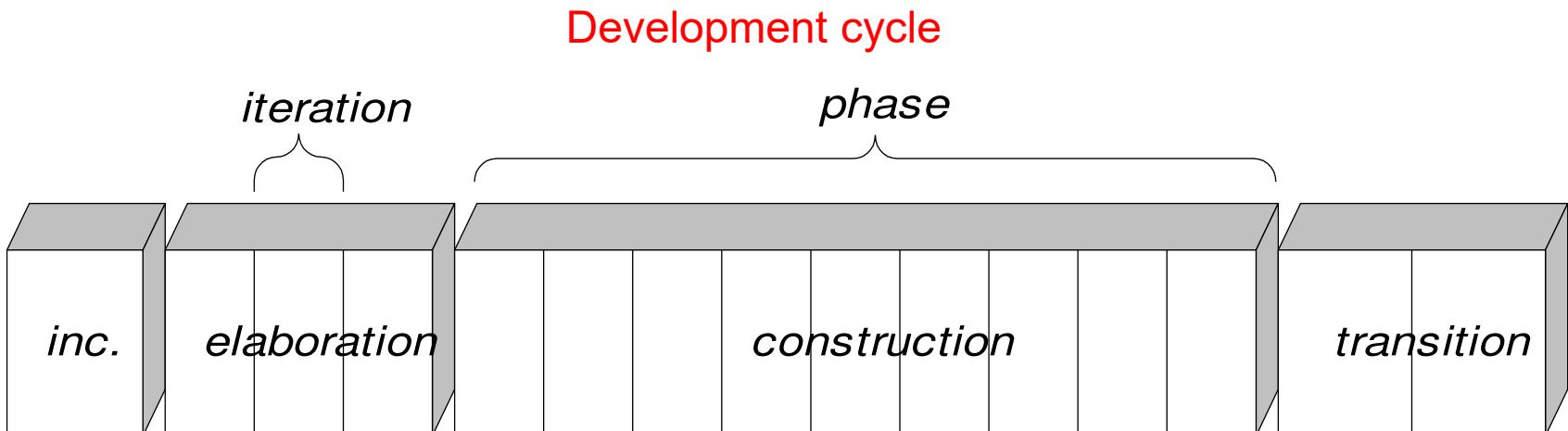


Diagram illustrating how the relative emphasis of different disciplines changes over the course of the project

# UP phases are iterative & incremental

- Inception
  - Feasibility phase and approximate vision
- Elaboration
  - Core architecture implementation, high risk resolution
- Construction
  - Implementation of remaining elements
- Transition
  - Beta tests, deployment



# Inception → Elaboration → ...

- During **inception**, establish business rationale and scope for project
  - Business case: how much it will cost and how much it will bring in?
  - Scope: try to get sense of size of the project and whether it's doable
  - Creates a *vision and scope document* at a high level of abstraction
- In **elaboration**, collect more detailed requirements and do high-level analysis and design
  - Inception gives you the go-ahead to start a project, elaboration determines the **risks**
    - Requirement risks: big danger is that you may build the wrong system
    - Technological risks: can the technology actually do the job? will the pieces fit together?
    - Skills risks: can you get the staff and expertise you need?
    - Political risks: can political forces get in the way?
  - Develop use cases, non-functional requirements & domain model

# ... → Construction → Transition

- **Construction** builds production-quality software in many increments, tested and integrated, each satisfying a subset of the requirements of the project
  - Delivery may be to external, early users, or purely internal
  - Each iteration contains usual life-cycle phases of analysis, design, implementation and testing
  - Planning is crucial: use cases and other UML documents
- **Transition** activities include beta testing, performance tuning (optimization) and user training
  - No new functionality unless it's small and essential
  - Bug fixes are OK

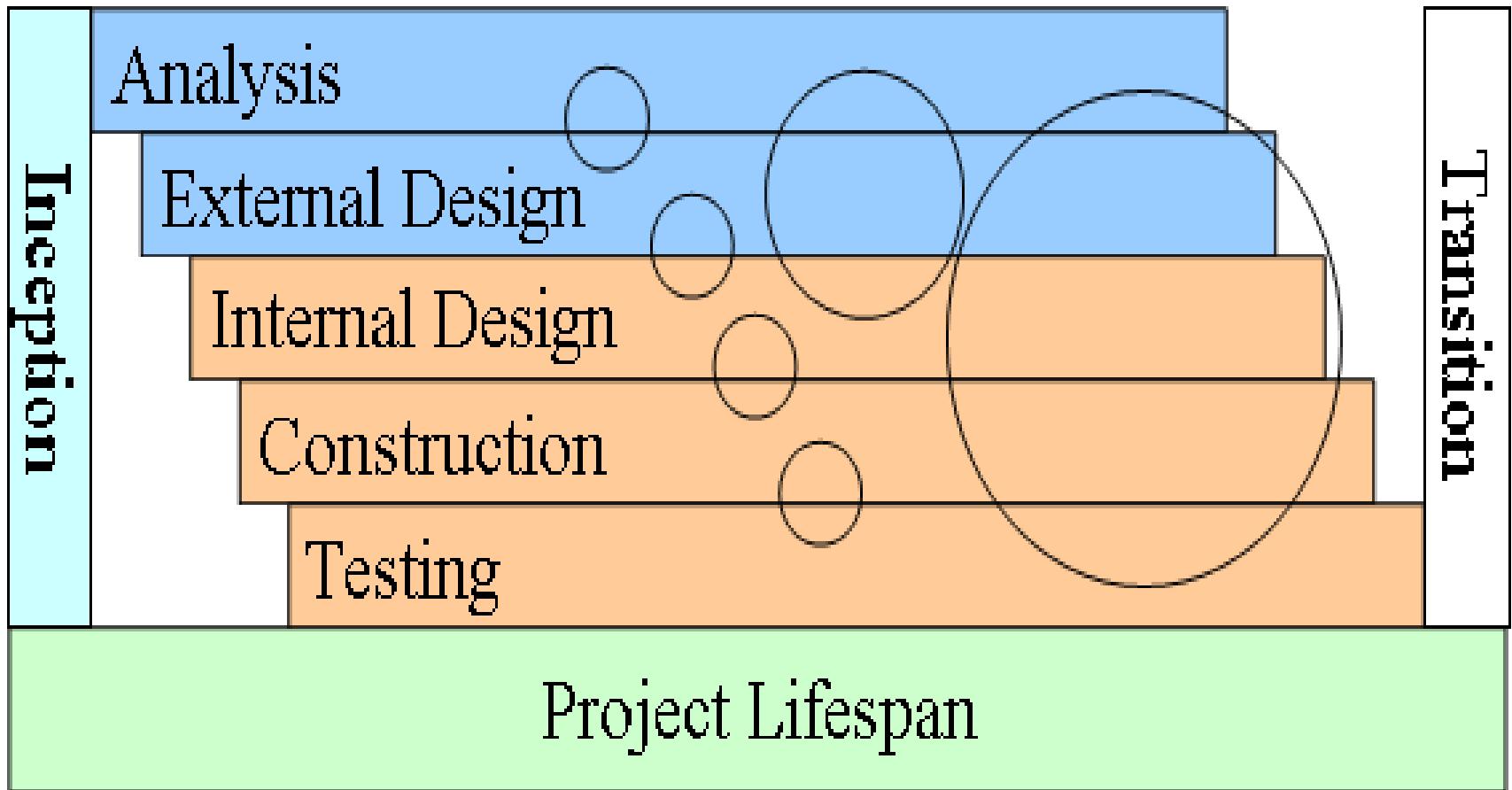
# UP artifacts

- The UP describes work activities,  
which result in *work products* called *artifacts*
- Examples of artifacts:
  - Vision, scope and business case descriptions
  - Use cases (describe scenarios for user-system interactions)
  - UML diagrams for domain modeling, system modeling
  - Source code (and source code documentation)
  - Web graphics
  - Database schema

# Milestone for first Elaboration

- At start of elaboration, identify part of the project to design & implement
  - A typical and crucial scenario (from a use case)
- After first elaboration, project is, say, 1/5<sup>th</sup> done
- Can then provide estimates for rest of project
  - Significant risks are identified and understood
- How is such a milestone different from a stage in the waterfall model?

# What does diagram imply about UP?



How can iterations reduce risk or reveal problems?

# Inception Phase

- Purpose
  - To establish the business case for a new system or for a major update of an existing system
  - To specify the project scope
- Outcome
  - A general vision of the project's requirements, i.e., the core requirements
    - Initial use-case model and domain model (10-20% complete)
    - An initial business case, including:
      - Success criteria (e.g., revenue projection)
      - An initial risk assessment
      - An estimate of resources required
- Milestone: Lifecycle Objectives

# Elaboration Phase

- Purpose
  - To analyze the problem domain
  - To establish a sound architectural foundation
  - To address the highest risk elements of the project
  - To develop a comprehensive plan showing how the project will be completed
- Outcome
  - Use-case and domain model 80% complete
  - An executable architecture and accompanying documentation
  - A revised business case, incl. revised risk assessment
  - A development plan for the overall project
- Milestone: Lifecycle Architecture

# Construction Phase

- Purpose
  - To incrementally develop a complete software product which is ready to transition into the user community
- Products
  - A complete use-case and design model
  - Executable releases of increasing functionality
  - User documentation
  - Deployment documentation
  - Evaluation criteria for each iteration
  - Release descriptions, including quality assurance results
  - Updated development plan
- Milestone: Initial Operational Capability

# Transition Phase

- Purpose
  - To transition the software product into the user community
- Products
  - Executable releases
  - Updated system models
  - Evaluation criteria for each iteration
  - Release descriptions, including quality assurance results
  - Updated user manuals
  - Updated deployment documentation
  - “Post-mortem” analysis of project performance
- Milestone: Product Release

# Best Practices and Key Concepts

- Tackle high-risk and high-value issues in early iterations.
- Continuously engage users for evaluation, feedback and requirements
- Build a cohesive, core architecture in early iterations
- Continuously verify quality; test early, often and realistically
- Apply use cases where appropriate
- Do some visual modeling (with the UML)
- Carefully manage requirements
- Practice change request and configuration management.

# Questions

- What are the four lifecycle phases of UP?
- What happens in each?
- What are the process disciplines?
- What are some major differences between distinguishes UP and the waterfall model?



**BITS Pilani**  
Pilani Campus

# BITS Pilani presentation

Dr. Yashvardhan Sharma  
Computer Science and Information Systems





# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.3**

# Unified Development Process [1]

- Key features: Iterative development; OO analysis and design.
- Development organized as a series of short iterations
- Each iteration produces a working, executable, product that *might not be a deliverable*.
- No rush to code. Also, not a long drawn design process.
- Lots of visual modeling aids. Unified Modeling Language (UML) used.

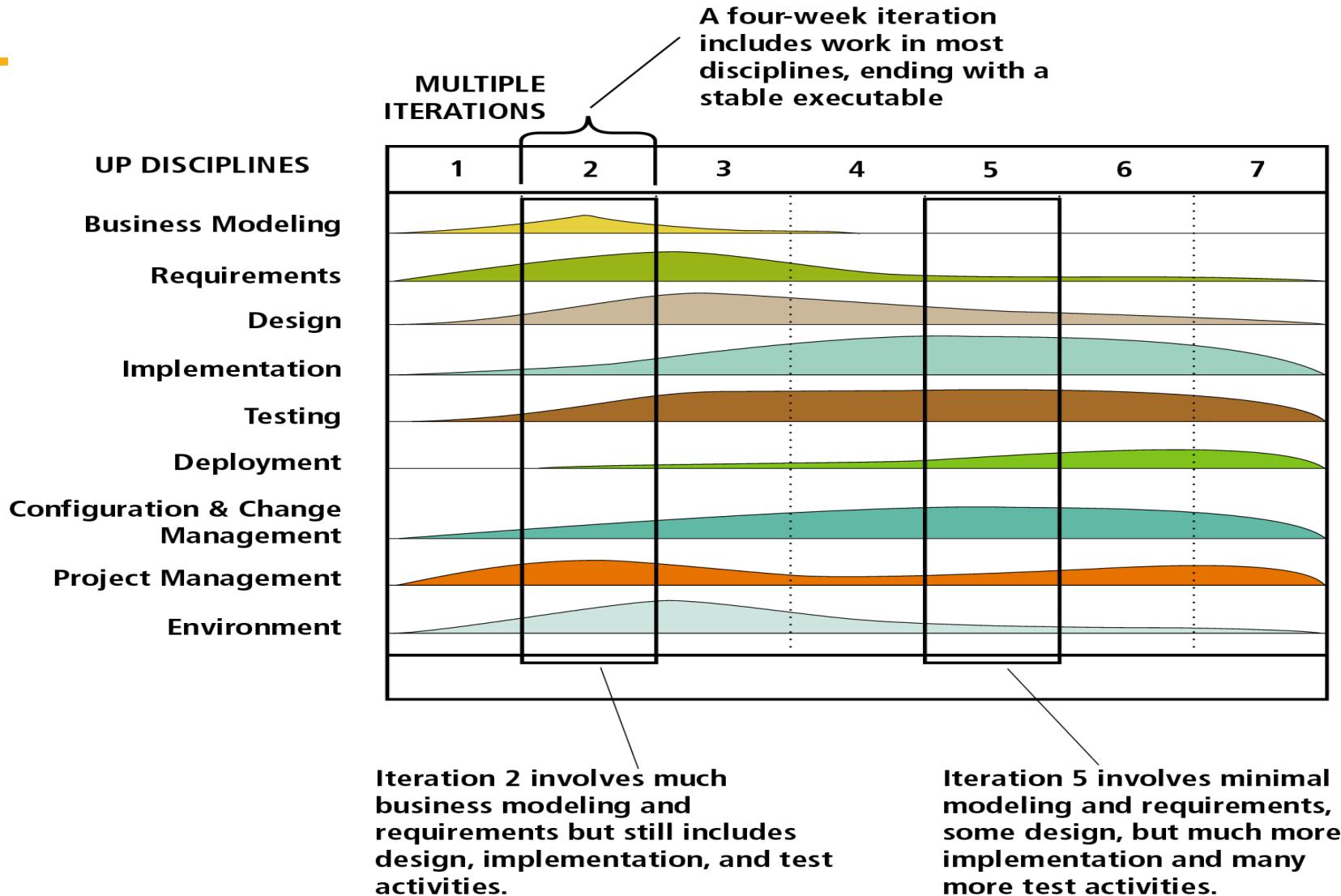
# **Unified Development Process [2]**

- Early iterations seek feedback from the customer. Risk and value to customer is managed through early feedback.

Customer is engaged continuously in evaluation and requirements gathering.

Architecture is built during early iterations.

# Unified Development Process [3]



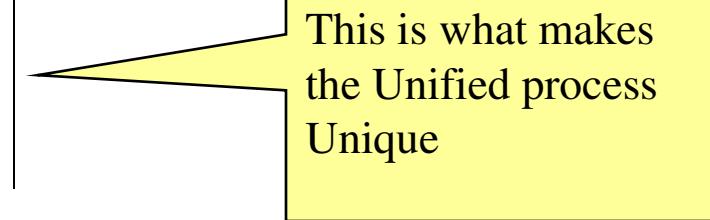
# The Unified Process

---

Component based - meaning the software system is built as a set of software components interconnected via interfaces

Uses the Unified Modeling Language (UML)

**Use case driven**  
**Architecture-centric**  
**Iterative and incremental**



This is what makes  
the Unified process  
Unique

**Component:** A physical and replaceable part of a system that conforms to and provides realization of a set of interfaces.

**Interface:** A collection of operations that are used to specify a service of a class or a component

# The Unified Process



Based around the 4Ps - People, Project, Product, Process

# The Unified Process

- Use Case driven
  - A use case is a piece of functionality in the system that gives a user a result of value.
- Use cases capture functional requirements
- Use case answers the question: *What is the system supposed to do for the user?*

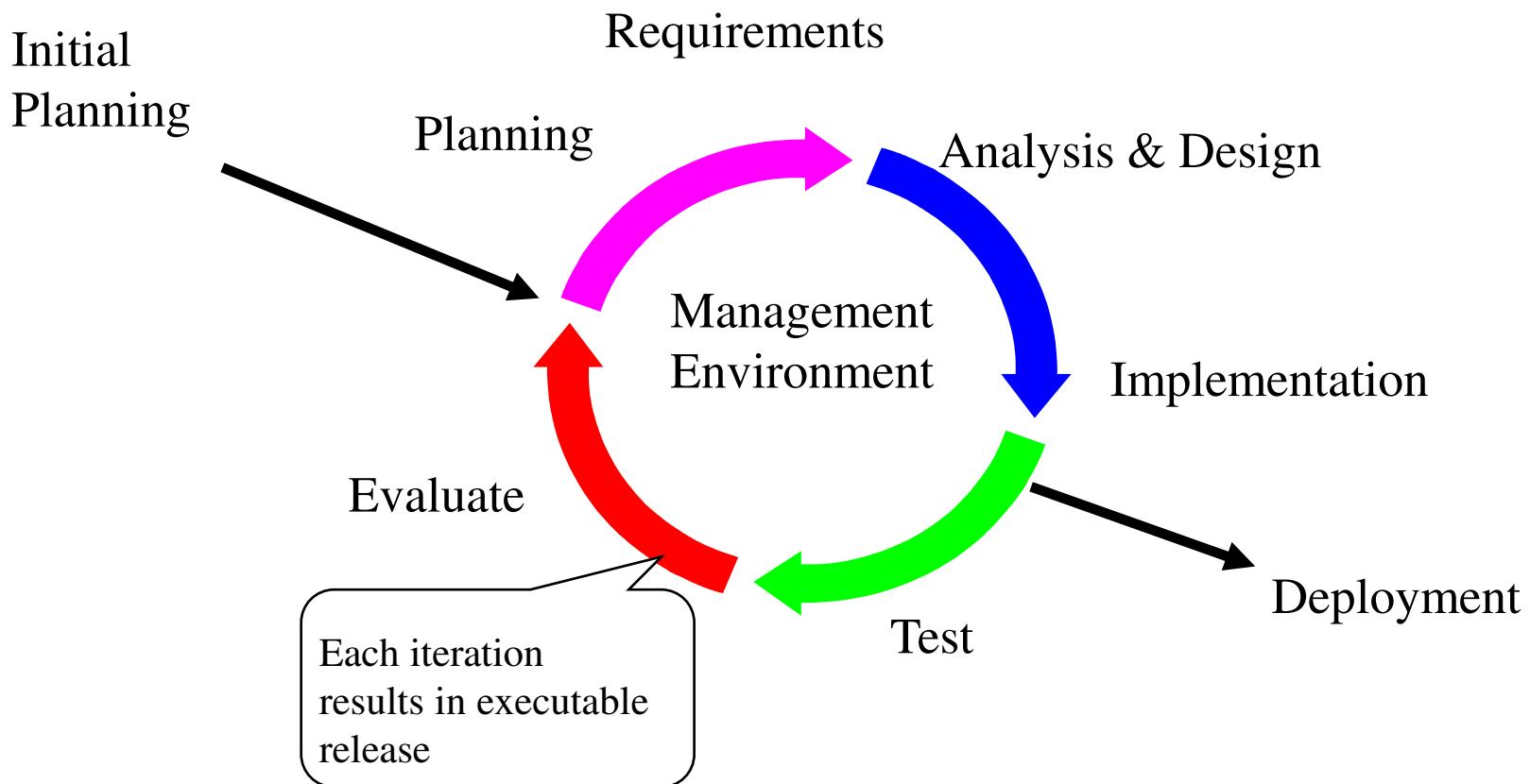
# The Unified Process

- Architecture centric
  - similar to architecture for building a house
  - Embodies the most significant static and dynamic aspects of the system
  - Influenced by platform, OS, DBMS etc.
  - Related as ***function*** (use case) and ***form*** (architecture)
  - Primarily serves the realization of use cases
  - The form must allow the system to evolve from initial development through future requirements (i.e. the design needs to be flexible)

# The Unified Process

- **Iterative and Incremental**
  - commercial projects continue many months and years
  - to be most effective - break the project into *iterations*
- Every iteration - identify use cases, create a design, implement the design
- Every iteration is a complete development process

# An iterative and incremental process



(Kruchten, 1999)

# Iterations

- Iterations must be selected & developed in a planned way i.e. in a logical order - early iterations must offer utility to the users
  - iteration based on a group of use cases extending the usability of the system developed so far
  - iterations deal with the most important risks first
  - not all iterations are additive - some replace earlier “superficial” developments with a more sophisticated and detailed one.

# Benefits of an iterative approach

- Risks are mitigated earlier
- Change is more manageable
- Higher level of reuse
- Project team can learn along the way
- Better overall quality

# Unified Software Development Process (UP)

Selects from best practices to

- Provide a generic process framework
  - instantiate/specialize for specific application areas, organizations, project sizes, etc.
- Define a set of activities (*workflows*)
  - transforms users' requirements into a software system
- Define a set of models
  - from abstract (user-level) to concrete (code)
- Allow component-based development
  - software components interconnected via well-defined interfaces
  - use-case (UML) and risk driven
  - architecture-centric
  - iterative and incremental

## Unified Process - Milestones

**Milestone:** a management decision point in a project that determines whether to authorize movement to the next iteration/phase

**Inception phase** - agreement among customers/developers on the system's life cycle objectives

**Elaboration phase** - agreement on the viability of the life cycle architecture, business case and project plan

**Construction phase** - agreement on the acceptability of the software product both operationally and in terms of cost

**Transition phase** - final agreement on the acceptability of the software product

# Lifecycle Phases

- Inception – “Daydream”
- Elaboration – “Design/Details”
- Construction – “Do it”
- Transition – “Deploy it”
- Phases are *not* the classical requirements/  
design/coding/implementation processes
- Phases iterate over many cycles

# Timeboxing

Management of a UP project.

Iterations are “timeboxed” or fixed in length.

Iteration lengths of between two to six weeks are recommended.

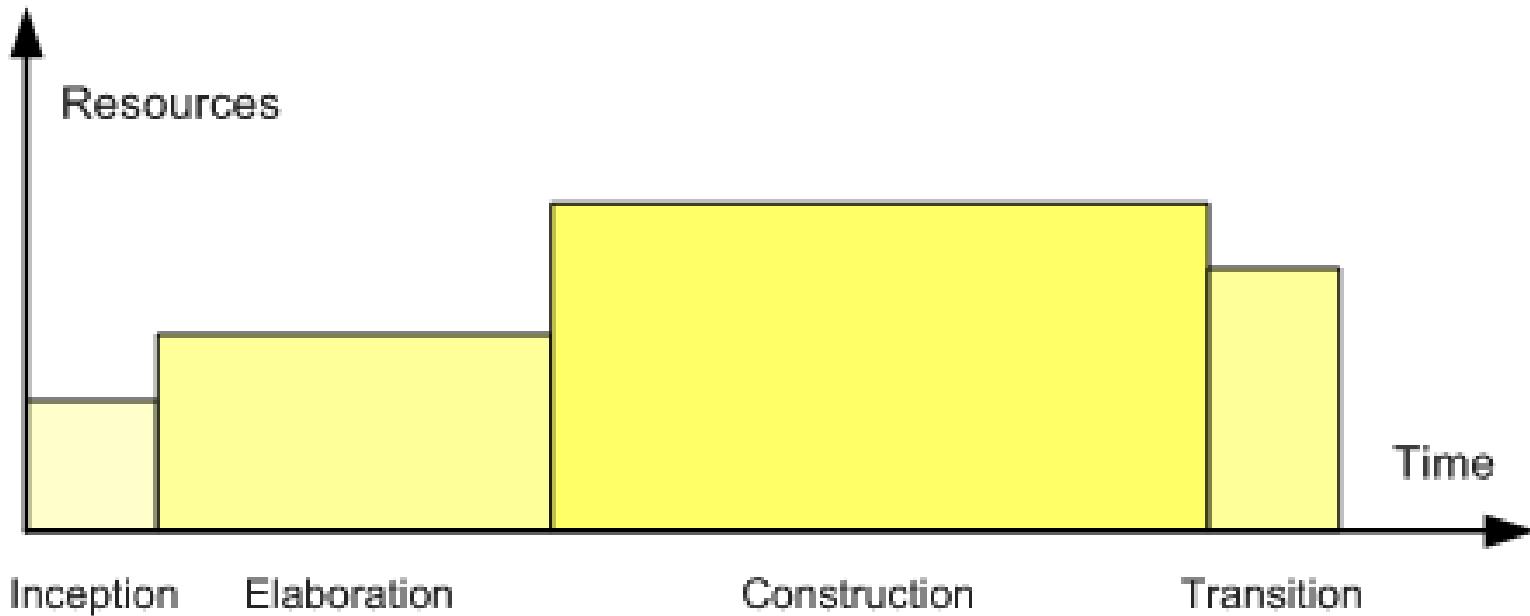
Each iteration period has its own development plan.

If all the planned activities cannot be completed during an iteration cycle, the completion date should not be extended, but rather tasks or requirements from the iteration should be removed and added to the next iteration cycle.

# The Unified Process

- The Unified Software Development Process is a definition of a complete set of activities to transform users' requirements through a consistent set of artifacts into a software product
- Look at the whole process
  - Life cycle
  - Artifacts
  - Workflows
  - Phases
  - Iterations
- A process is described in terms of **workflows** where a workflow is a set of activities with identified artifacts that will be created by those activities

# Phases in Unified Process



Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

## Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.

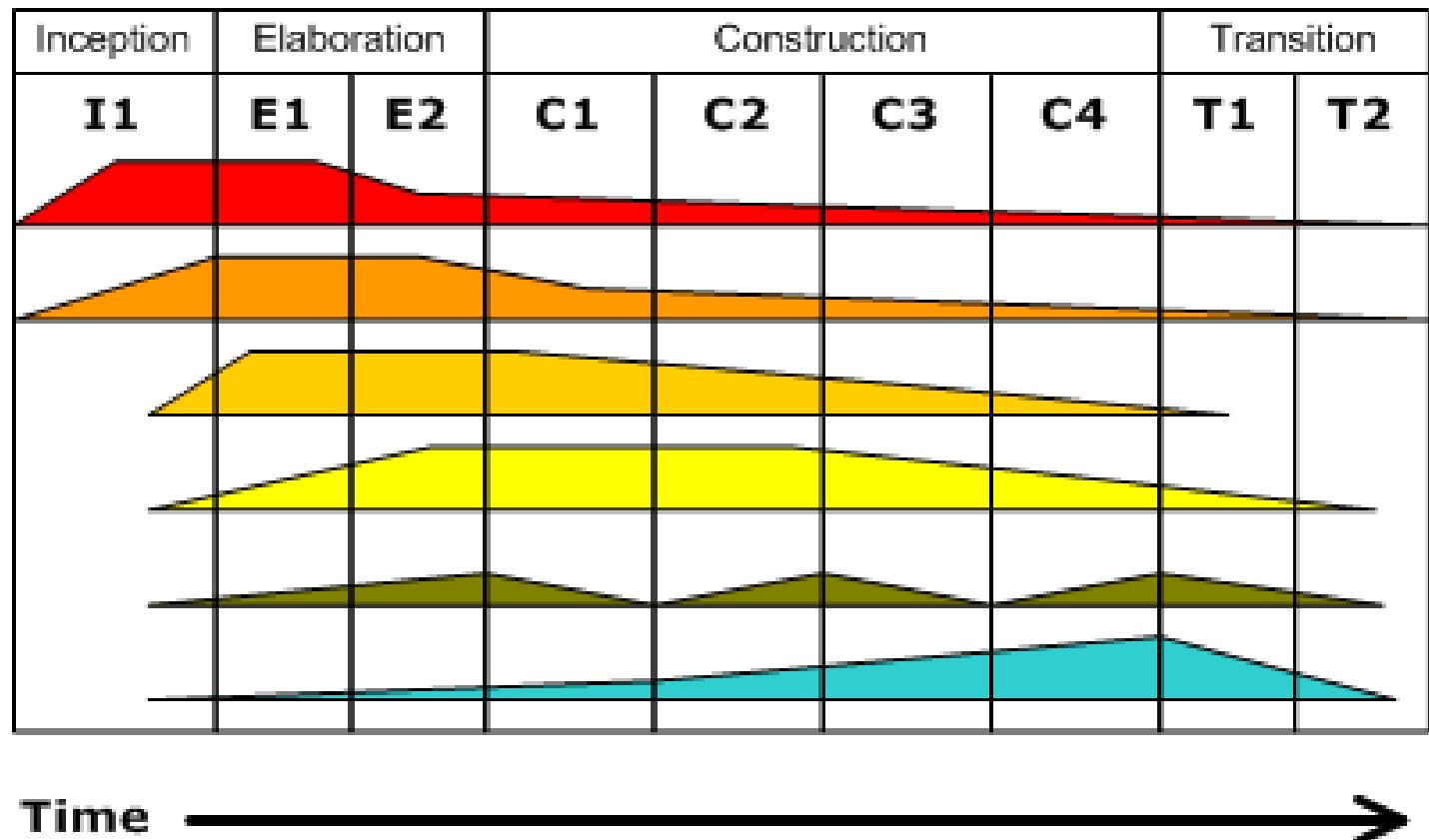
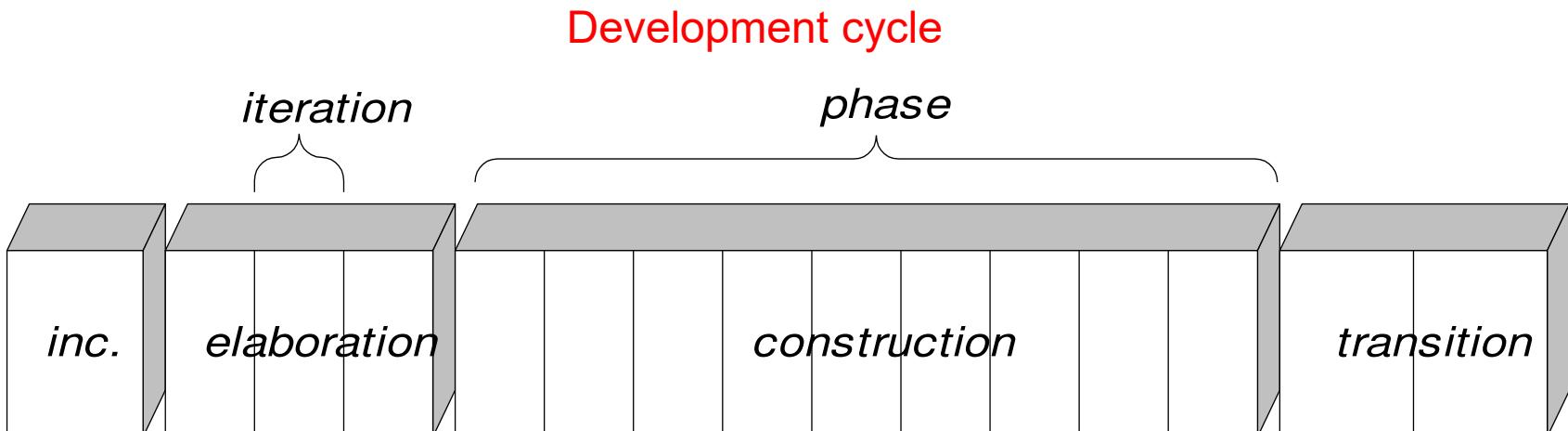


Diagram illustrating how the relative emphasis of different disciplines changes over the course of the project

# UP phases are iterative & incremental

- Inception
  - Feasibility phase and approximate vision
- Elaboration
  - Core architecture implementation, high risk resolution
- Construction
  - Implementation of remaining elements
- Transition
  - Beta tests, deployment



# Inception → Elaboration → ...

- During **inception**, establish business rationale and scope for project
  - Business case: how much it will cost and how much it will bring in?
  - Scope: try to get sense of size of the project and whether it's doable
  - Creates a *vision and scope document* at a high level of abstraction
- In **elaboration**, collect more detailed requirements and do high-level analysis and design
  - Inception gives you the go-ahead to start a project, elaboration determines the **risks**
    - Requirement risks: big danger is that you may build the wrong system
    - Technological risks: can the technology actually do the job? will the pieces fit together?
    - Skills risks: can you get the staff and expertise you need?
    - Political risks: can political forces get in the way?
  - Develop use cases, non-functional requirements & domain model

# ... → Construction → Transition

- **Construction** builds production-quality software in many increments, tested and integrated, each satisfying a subset of the requirements of the project
  - Delivery may be to external, early users, or purely internal
  - Each iteration contains usual life-cycle phases of analysis, design, implementation and testing
  - Planning is crucial: use cases and other UML documents
- **Transition** activities include beta testing, performance tuning (optimization) and user training
  - No new functionality unless it's small and essential
  - Bug fixes are OK

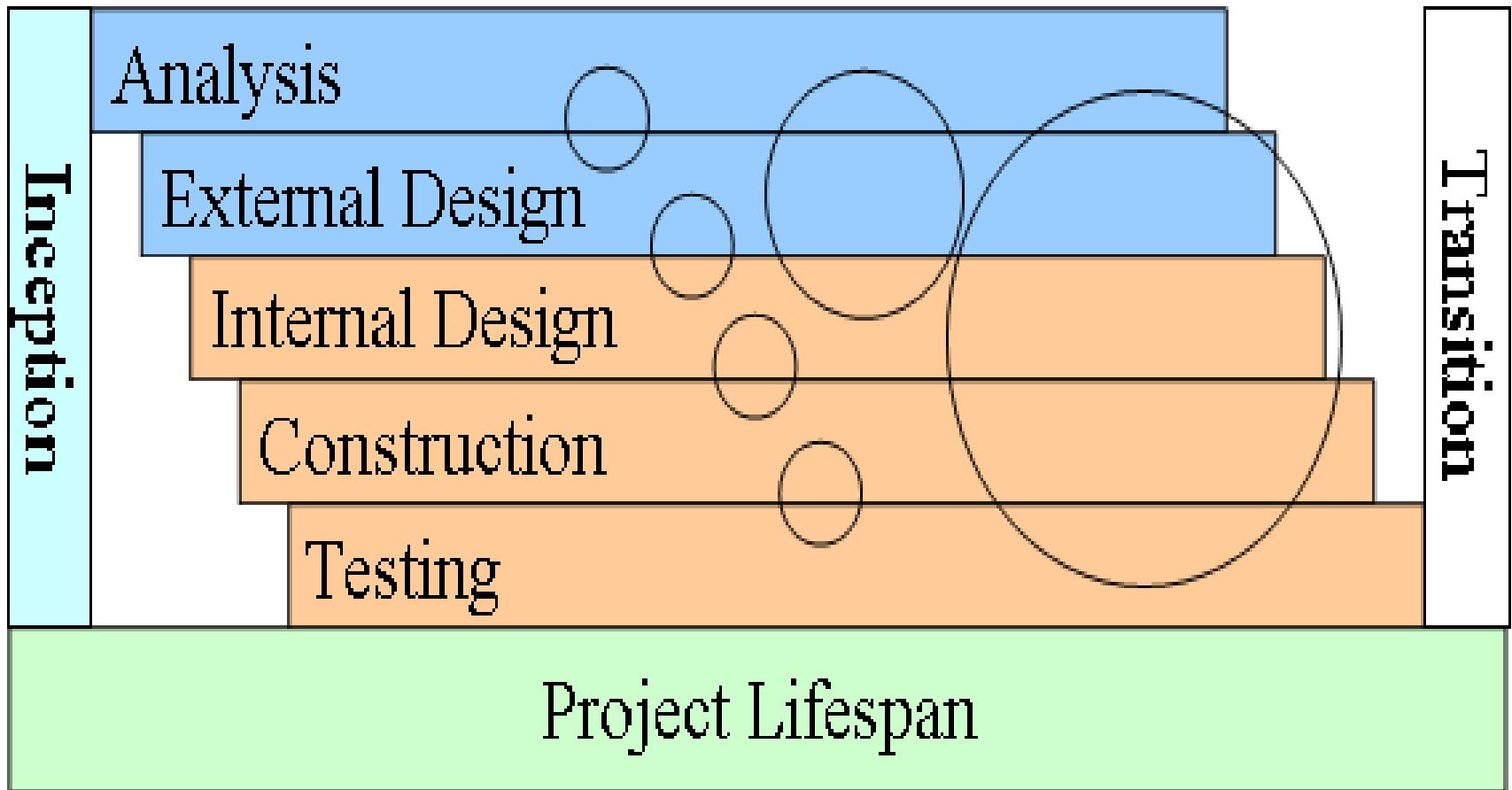
# UP artifacts

- The UP describes work activities,  
which result in *work products* called *artifacts*
- Examples of artifacts:
  - Vision, scope and business case descriptions
  - Use cases (describe scenarios for user-system interactions)
  - UML diagrams for domain modeling, system modeling
  - Source code (and source code documentation)
  - Web graphics
  - Database schema

# Milestone for first Elaboration

- At start of elaboration, identify part of the project to design & implement
  - A typical and crucial scenario (from a use case)
- After first elaboration, project is, say, 1/5<sup>th</sup> done
- Can then provide estimates for rest of project
  - Significant risks are identified and understood
- How is such a milestone different from a stage in the waterfall model?

# What does diagram imply about UP?



How can iterations reduce risk or reveal problems?

# Inception Phase

- Purpose
  - To establish the business case for a new system or for a major update of an existing system
  - To specify the project scope
- Outcome
  - A general vision of the project's requirements, i.e., the core requirements
    - Initial use-case model and domain model (10-20% complete)
    - An initial business case, including:
      - Success criteria (e.g., revenue projection)
      - An initial risk assessment
      - An estimate of resources required
- Milestone: Lifecycle Objectives

# Elaboration Phase

- Purpose
  - To analyze the problem domain
  - To establish a sound architectural foundation
  - To address the highest risk elements of the project
  - To develop a comprehensive plan showing how the project will be completed
- Outcome
  - Use-case and domain model 80% complete
  - An executable architecture and accompanying documentation
  - A revised business case, incl. revised risk assessment
  - A development plan for the overall project
- Milestone: Lifecycle Architecture

# Construction Phase

- Purpose
  - To incrementally develop a complete software product which is ready to transition into the user community
- Products
  - A complete use-case and design model
  - Executable releases of increasing functionality
  - User documentation
  - Deployment documentation
  - Evaluation criteria for each iteration
  - Release descriptions, including quality assurance results
  - Updated development plan
- Milestone: Initial Operational Capability

# Transition Phase

- Purpose
  - To transition the software product into the user community
- Products
  - Executable releases
  - Updated system models
  - Evaluation criteria for each iteration
  - Release descriptions, including quality assurance results
  - Updated user manuals
  - Updated deployment documentation
  - “Post-mortem” analysis of project performance
- Milestone: Product Release

# Best Practices and Key Concepts

- Tackle high-risk and high-value issues in early iterations.
- Continuously engage users for evaluation, feedback and requirements
- Build a cohesive, core architecture in early iterations
- Continuously verify quality; test early, often and realistically
- Apply use cases where appropriate
- Do some visual modeling (with the UML)
- Carefully manage requirements
- Practice change request and configuration management.

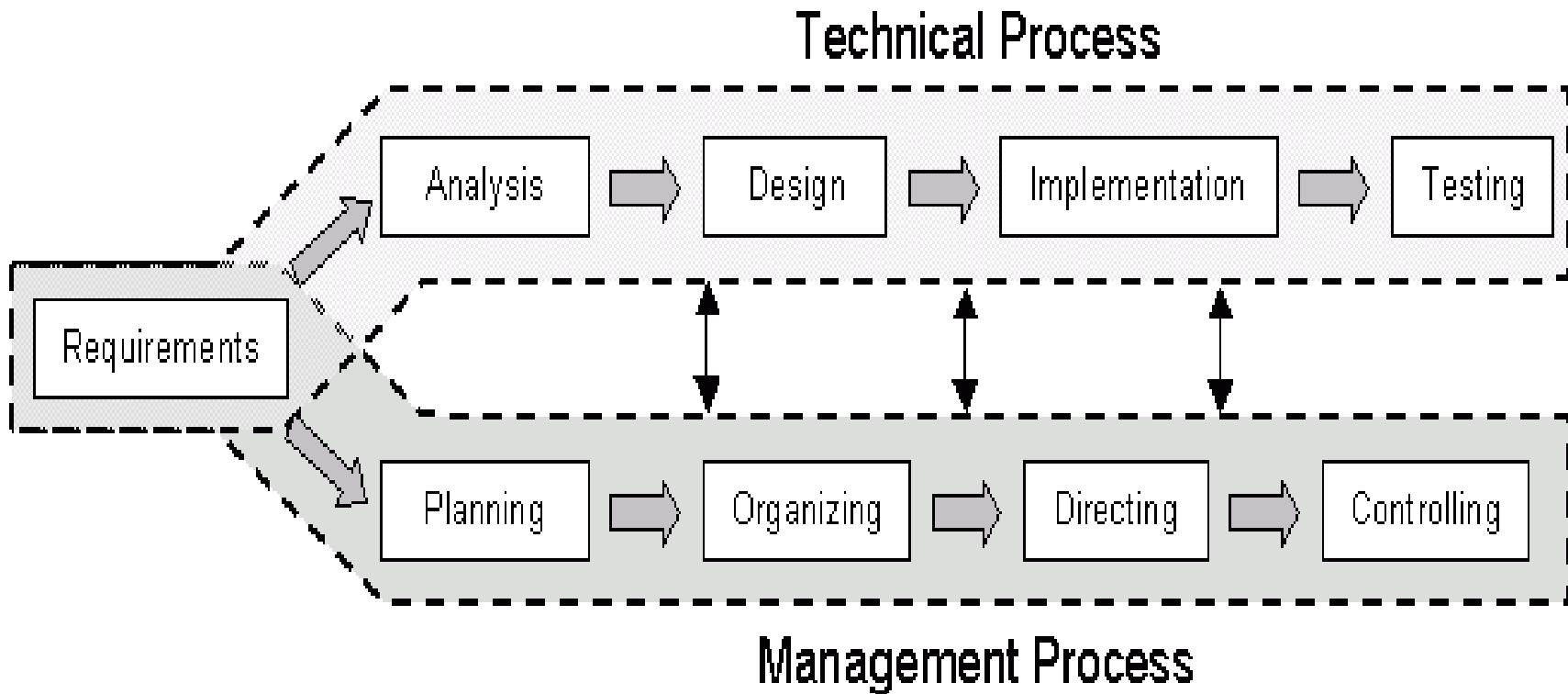
# Questions

- What are the four lifecycle phases of UP?
- What happens in each?
- What are the process disciplines?
- What are some major differences between distinguishes UP and the waterfall model?

# Requirements Engineering

- "The hardest single part of building a software system is deciding what to build. ..No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

# Requirements drive the technical and management processes



# What is a Software Requirement

- [IEEE 90] defines a requirement as:
  - (1) A condition or capability needed by a user to solve a problem or achieve an objective.
  - (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
  - (3) A documented representation of a condition or capability as in (1) or (2).

# What is this Phase For?

- Major misconception
  - determining what client wants

*“I know you believe you understood what you think I said,  
but I am not sure you realize that what you heard is not  
what I meant!”*

- Must determine client's & **user's** needs
  - chances for success slim if you don't figure this out!

# Requirements--What are They?

- A requirement is a description of a system feature, capability, or constraint.
- Classes of Requirements:
  - functional
  - nonfunctional (constraints)
- Requirements Priority
  - essential (“shalls”)
  - highly desirable (“shoulds”)
  - desirable but low priority

# Types of Requirements

- Physical environment
- Interfaces
- Users and human factors
- functionality
- performance
- documentation/training
- data
- resources
- security
- reliability
- portability
- maintenance
- etc.

# Levels of Requirements

Business Requirements – Vision and scope



User Requirements – Use cases



System Specification – Behavior of the system

# Levels of Requirements

- ***Business requirements*** - objectives of the customer or user of the system
- ***User requirements*** - tasks the user needs to accomplish with the system.
- ***System specification*** (functional requirements) - system behavior that will allow users to perform their tasks.

# Levels of Requirements-Examples

**Business requirement** = Hospital needs to get drug information in the hands of pharmacists and doctors so they can make better decisions about which drugs to prescribe for patients.

**User Requirement** = The pharmacist should be able to enter a drug name and get back information about side affects, dosage and drug interactions.

**System specification** = The system will keep a database of drug names including scientific name, generic name and brand name. The system will return all drug names that match a full or partial name. If more than one drug name matches the user will have the option of selecting one of the drugs in the list.

# Requirements Analysis [1]

- What is it?
  - The process by which customer needs are understood and documented.
  - Expresses “what” is to be built and NOT “how” it is to be built.
- Example 1:
  - The system shall allow users to withdraw cash. [*What?*]
- Example 2:
  - A sale item’s name and other attributes will be stored in a hash table and updated each time any attribute changes. [*How?*]

# Requirements Analysis [2]

- C- and D-Requirements
  - C-: Customer wants and needs; expressed in language understood by the customer.
  - D-: For the developers; may be more formal.

# **Requirements Analysis [2]**

## **Why document requirements?**

Serves as a contract between the customer and the developer.

Serves as a source of test plans.

Serves to specify project goals and plan development cycles and increments.

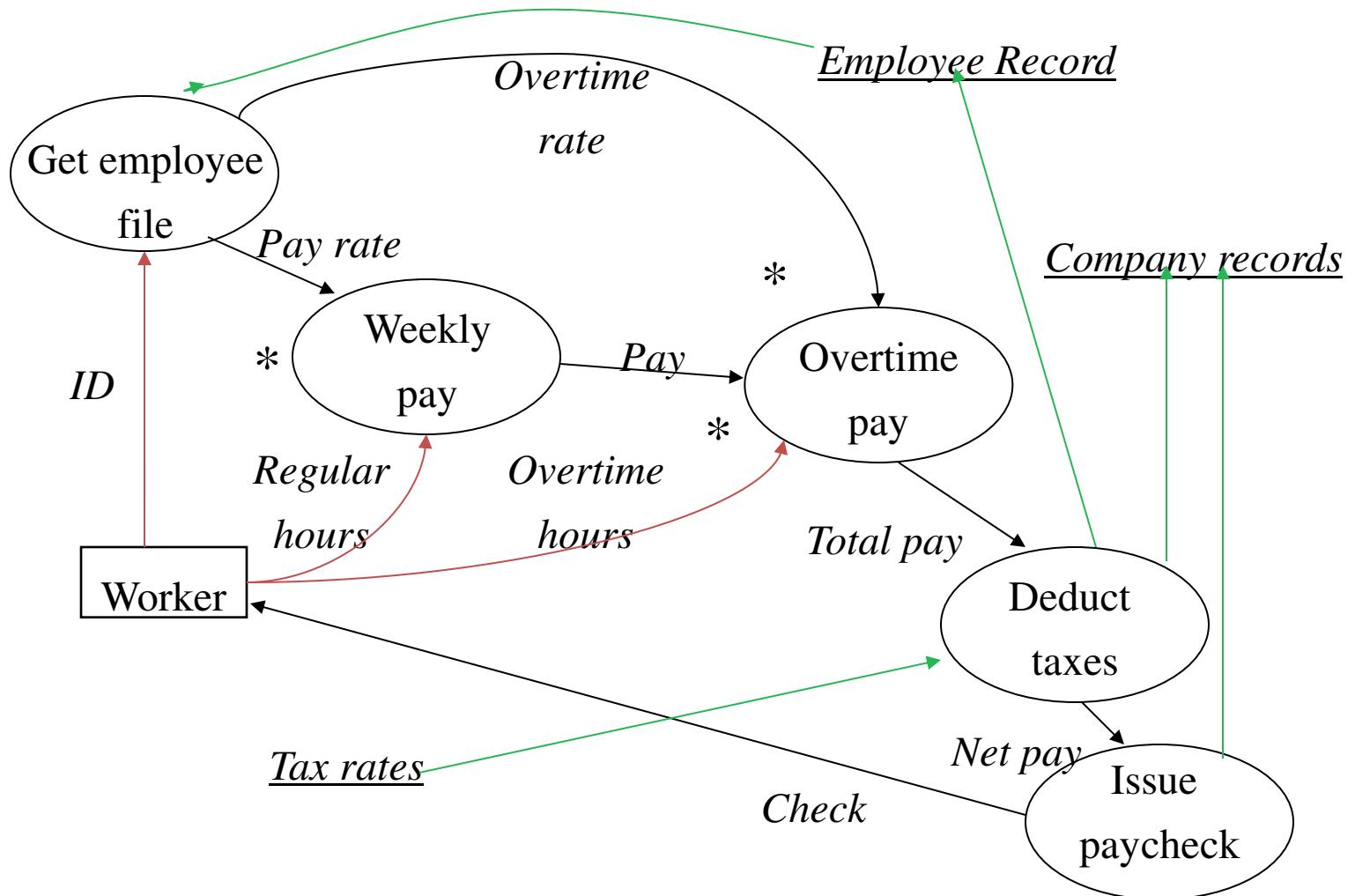
# Requirements Analysis [3]

- Roadmap:
  - Identify the customer.
  - Interview customer representatives.
  - Write C-requirements, review with customer, and update when necessary.
  - Write D-requirements; check to make sure that there is no inconsistency between the C- and the D-requirements.

# Requirements Analysis [4]

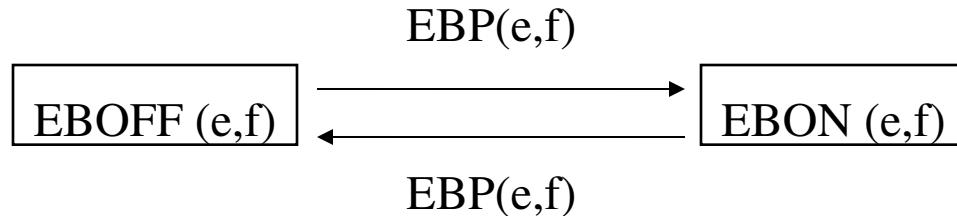
- C-requirements:
  - Use cases expressed individually and with a use case diagram. A use case specifies a collection of scenarios.
    - Sample use case: *Process sale*.
  - *Data flow diagram*:
    - Explains the flow of data items across various functions. Useful for explaining system functions. [Example on the next slide.]
  - *State transition diagram*:
    - Explains the change of system state in response to one or more operations. [Example two slides later.]
  - *User interface*: Generally not a part of requirements analysis though may be included.

# Data Flow Diagram



# State Transition Diagram (STD)

Elevator example (partial):



EBOFF (e,f): Elevator  $e$  button OFF at floor  $f$ .

EBON (e,f): Elevator  $e$  button ON at floor  $f$ .

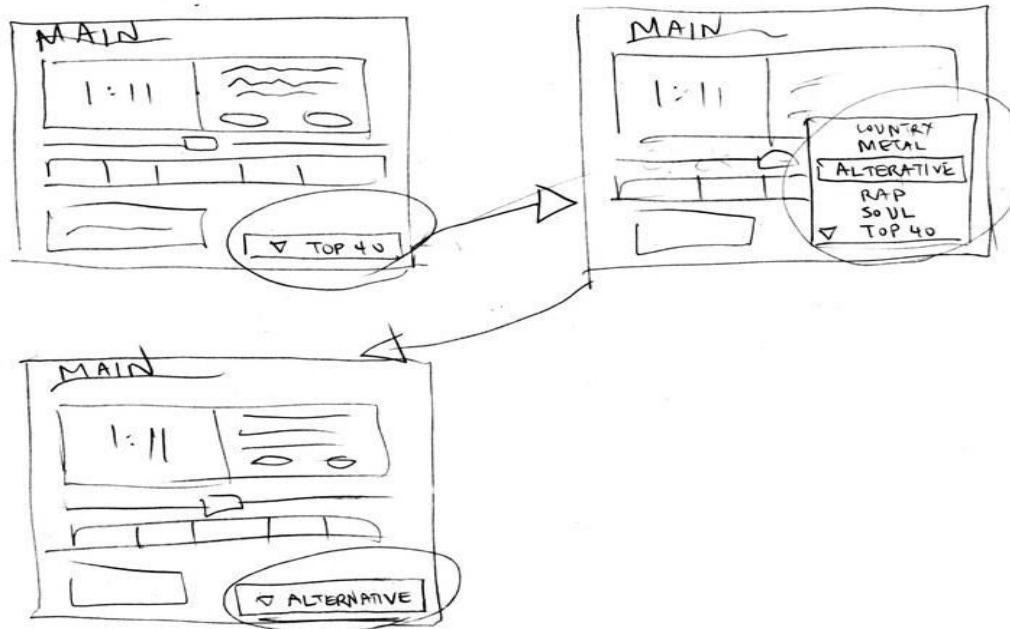
EBP(e,f): Elevator  $e$  button  $f$  is pressed.

EAF(e,f): Elevator  $e$  arrives at floor  $f$ .

# Scenarios

- Illustrate the major events/actions to users
  - often use storyboards & paper prototypes

SCENARIO 1    "I want to listen to alternative music"



# Rapid Prototyping

- Allow users to “see” & use proposed solutions
- Develop specification from the prototype/requirements
- Prototype must be constructed & changed quickly
  - key functionality
    - omits things like scalability, error checking, saving, etc.
  - do not spend a lot of time perfecting the code/structure
    - it is OK if it hardly works or crashes every few minutes
    - plan to throw it away! because you will!
  - put in front of customer ASAP & modify in response

# Requirements Analysis [5]

- D-requirements:
  1. Organize the C-requirements.
  2. Create System sequence diagrams for use cases:
    - *Obtain D-requirements from C-requirements and customer.*
    - *Outline test plans*
    - *Inspect*
  3. Validate with customer.
  4. Release:

# Requirements Analysis [6]

- Organize the D-requirements.

- (a) Functional requirements

*The blood pressure monitor will measure the blood pressure and display it on the in-built screen*

- (b) Non-functional requirements

- (i) Performance

*The blood pressure monitor will complete a reading within 10 seconds.*

- (i) Reliability

*The blood pressure monitor must have a failure probability of less than 0.01 during the first 500 readings.*

# Requirements Analysis [7]

- (c) Interface requirements: interaction with the users and other applications

*The blood pressure monitor will have a display screen and push buttons. The display screen will....*
- (d) Constraints: timing, accuracy

*The blood pressure monitor will take readings with an error less than 2%.*

# Requirements Analysis [7]

Properties of D-requirements:

1. Traceability: Functional requirements

D-requirement → inspection report → design segment →  
code segment → code inspection report → test plan →  
test report

2. Traceability: Non-Functional requirements

- (a) Isolate each non-functional requirement.
- (b) Document each class/function with the related non-functional requirement.

# Requirements Analysis [8]

Properties of D-requirements:

3. Testability

*It must be possible to test each requirement. Example ?*

4. Categorization and prioritization

# Categorizing Requirements

- How to categorize system functions?

<u>Function Category</u>	<u>Meaning</u>
Evident	Should perform, user is aware
Hidden	Should perform but not visible to users
Frill	Optional; Nice to have

# Prioritizing (Ranking) Use Cases

- Strategy :
  - pick the use cases that significantly influence the core architecture
  - pick the use cases that are critical to the success of the business
  - a useful rule of thumb - pick the use cases that are the highest risk!!!

# Requirements Analysis [9]

Properties of D-requirements:

- 5. Completeness

*Self contained, no omissions.*

- 6. Error conditions

*State what happens in case of an error. How should the implementation react in case of an error condition?*

# Requirements Analysis [10]

Properties of D-requirements:

## 7. Consistency

*Different requirements must be consistent.*

*Example:*

*R1.2: The speed of the vehicle will never exceed 250 mph.*

*R5.4: When the vehicle is cruising at a speed greater than 300 mph, a special “watchdog” safety mechanism will be automatically activated.*



**BITS Pilani**  
Pilani Campus

# BITS Pilani presentation

Dr. Yashvardhan Sharma  
Computer Science and Information Systems





# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.4**

# Today's Agenda

---

- Use Case Modeling

# Use Case Model

# The Use Case Model

---

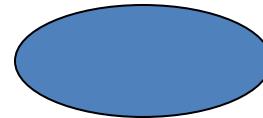
- Writing Requirements in Context
- Identify user goals, corresponding system functions / business processes
- Brief, casual, and “fully-dressed” formats
- Iterative refinement of use cases

# Use Cases

---

Use case - narration of the sequence of events of an actor using a system

- UML icon for use case

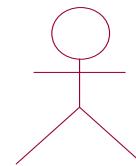


# Actors [1]

---

**Actor** - an entity external to the system that in some way participates in the use case

- An actor typically stimulates the system with input events or receives outputs from the system or does both.
- UML notation for actor:



Customer

# Actors [2]

---

*Primary Actor* - an entity external to the system that uses system services in a direct manner.

- *Supporting Actor*- an actor that provides services to the system being built.
  
- Hardware, software applications, individual processes, can all be actors.

# Use case and Scenario

---

**Scenario:** specific sequence of actions and interactions between actor(s) and system  
(= one story; success or failure)

**Use Case:** collection of related success and failure scenarios describing the actors attempts to support a specific goal

# Use-Case Modeling

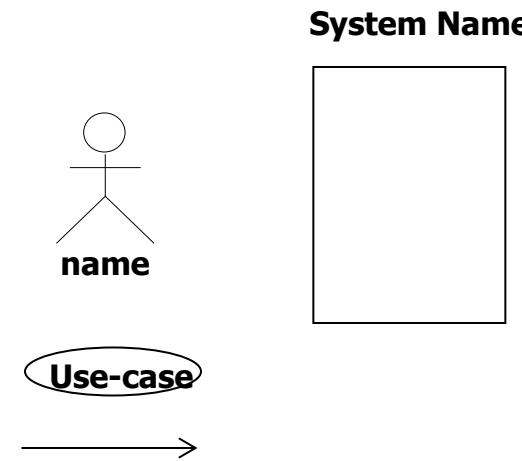
---

- In use-case modeling, the system is looked upon as a black box whose boundaries are defined by its functionality to external stimuli.
- The actual description of the use-case is usually given in plain text. A popular notation promoted by UML is the stick figure notation.
- Both visual and text representation are needed for a complete view.
- A use-case model represents the use-case view of the system. A use-case view of a system may consist of many Use-case diagrams.
- An use-case diagram shows (the system), the actors, the use-cases and the relationship among them.

# Components of Use-case Model

The components of a Use-case model are:

- System Modeled
- Actors
- Use-cases
- Stimulus



# System

---

As a part of the use-case modeling, the boundaries of the system are developed.

System in the use-case diagram is a box with the name appearing on the top.

Define the scope of the system that you are going to design with your (software scope)

Software Appln.



# Actors

---

An actor is something or someone that interacts with the system.

Actor communicates with the system by sending and receiving messages.

An actor provides the stimulus to activate an Use-case.

Message sent by an actor may result in more messages to actors and to Use-cases.

Actors can be ranked: primary and secondary; passive and active.

Actor is a role not an individual instance.

# Finding Actors

---

The actors of a system can be identified by answering a number of questions:

- Who will use the functionality of the system?
- Who will maintain the system?
- What devices does the system need to handle?
- What other system does this system need to interact?
- Who or what has interest in the results of this system?

# Use-cases

---

A Use-case in UML is defined as a set of sequences of actions a system performs that yield an observable result of value to a particular actor.

Actions can involve communicating with number of actors as well as performing calculations and work inside the system.

A Use-case

- is always initiated by an actor.
- provides a value to an actor.
- must always be connected to at least one actor.
- must be a complete description.

# Finding Use-cases

---

For each actor ask these questions:

- Which functions does the actor require from the system?
- What does the actor need to do?
- Could the actor's work be simplified or made efficient by new functions in the system?
- What events are needed in the system?
- What are the problems with the existing systems?
- What are the inputs and outputs of the system?

# Describing Use-cases

---

Use-case Name:

Use-case Number: system#.diagram#.Use-case#

Authors:

Event(Stimulus):

Actors:

Overview: brief statement

Related Use-cases:

Typical Process description: Algorithm

Exceptions and how to handle exceptions:

# Stories

---

Using a system to meet a goal; e.g. (brief format) --

**Process Sale:** A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running Total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a Receipt from the system and then leaves with the items.

Usually, writing the stories is more important than diagramming a use case model in UML

# Example (casual format)

---

## Handle Returns

*Main Success Scenario:* A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item...

*Alternate Scenarios:*

If they paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the cashier and suggest manual entry of the identifier code

If the system detects failure to communicate with the external accounting system... (etc.)

# The Primary Course-Success Scenario

---

## *Check Out Item:*

1. *Cashier swipes product over scanner, scanner reads UPC code.*
2. *Price and description of item, as well as current subtotal appear on the display facing the customer. The price and description also appear on the cashier's screen.*
3. *Price and description are printed on receipt.*
4. *System emits an audible “acknowledgement” tone to tell the cashier that the UPC code was correctly read.*

# Alternate Courses -Failure Scenario

---

## UPC Code Not Read:

- *If the scanner fails to capture the UPC code, the system should emit the “reswipe” tone telling the cashier to try again. If after three tries the scanner still does not capture the UPC code, the cashier should enter it manually.*

## No UPC Code:

- *If the item does not have a UPC code, the cashier should enter the price manually.*

# Observable Result of Value

---

“A key attitude in use case work is to focus on the question ‘How can using the system provide observable value to the user, or fulfill their goals?’, rather than merely thinking of requirements in terms of a list of features or functions”

Focus on business process, not technology features

# Black Box Use Cases

---

- Focus on “what”, not “how”
- Good: *The system records the sale*
- Bad: *The system writes the sale to a database*
- Worse: *The system generates a SQL INSERT statement...*

Premature design decisions!

**Analysis vs. Design = “What” vs. “How”**

# Degrees of Use Case Formality

---

Brief: one-paragraph summary, main success scenario

Casual: multiple, informal paragraphs covering many scenarios

Fully-Dressed: all steps and variations written in detail with supporting sections

# NextGen POS Example

---

Starts on page 68 in the book (Larman)  
A fully-dressed example

# Fully-Dressed Format

---

- Primary Actor
- Stakeholders & Interests
- Preconditions
- Success Guarantee  
(Postconditions)
- Main Success Scenario  
(Basic Flow)
- Extensions  
(Alternative Flows)
- Special Requirements
- Technology and Data Variations
- Frequency of Occurrence
- Open Issues

# Details...

---

## Stakeholders and Interests

- Important: defines what the use case covers (“all and only that which satisfies the stakeholders’ interests”)

## Preconditions

- What must always be true before beginning a scenario in the use case  
(e.g., “user is already logged in”)

# Details...[2]

---

## Success Guarantees (Postconditions)

- What must be true on successful completion of the use case; should meet needs of all stakeholders

## Basic Flow

- Records steps: interactions between actors; a validation (by system); a state change (by system)

# Details...[3]

---

## Extensions (Alternative Flows)

- Scenario branches (success/failure)
- Longer/more complex than basic flow
- Branches indicated by letter following basic flow step number, e.g. “3a”
- Two parts: *condition, handling*

## Special Requirements

- Non-functional considerations (e.g., performance expectations)

# Details...[4]

---

## Technology & Data Variations

- Non-functional constraints expressed by the stakeholders  
(e.g., “must support a card reader”)

# Identify Use Cases

---

Capture the specific ways of using the system as dialogues between an actor and the system.

Use cases are used to

- Capture system requirements
- Communicate with end users and Subject Matter Experts
- Test the system

# Specifying Use Cases

---

Create a written document for each Use Case

- Clearly define intent of the Use Case
- Define Main Success Scenario (Happy Path)
- Define any alternate action paths
- Use format of Stimulus: Response
- Each specification must be testable
- Write from actor's perspective, in actor's vocabulary

# Identification of Use Cases [1]

---

- Method 1 - Actor based
  - Identify the actors related to the system
  - Identify the scenarios these actors initiate or participate in.
- Method 2 - Event based
  - Identify the external events that a system must respond to
  - Relate the events to actors and use cases
- Method 3 – Goal based
  - [Actors have goals.]
  - Find user goals. [Prepare actor-goal list.]
  - Define a use case for each goal.

# Identification of Use Cases[2]

---

To identify use cases, focus on *elementary business processes (EBP)*.

- An **EBP** is a task performed by one person in one place at one time, in response to a business event. This task adds measurable business value and leaves data in a consistent state.

# Use Cases and Goals

---

Q1: “What do you do?”

Q2: “What are your goals?”

Which is a better question to ask a stakeholder?

Answer: Q2. Answers to Q1 will describe current solutions and procedures; answers to Q2 support discussion of improved solutions

# Finding Actors, Goals & Use Cases

---

Choose the system boundary

- Software, hardware, person, organization

Identify the primary actors

- Goals fulfilled by using the system

Identify goals for each actor

- Use highest level that satisfies EBP
- Exception: CRUD (next slide)

Define use cases that satisfy goals

# Create, Retrieve, Update, Delete (CRUD)

---

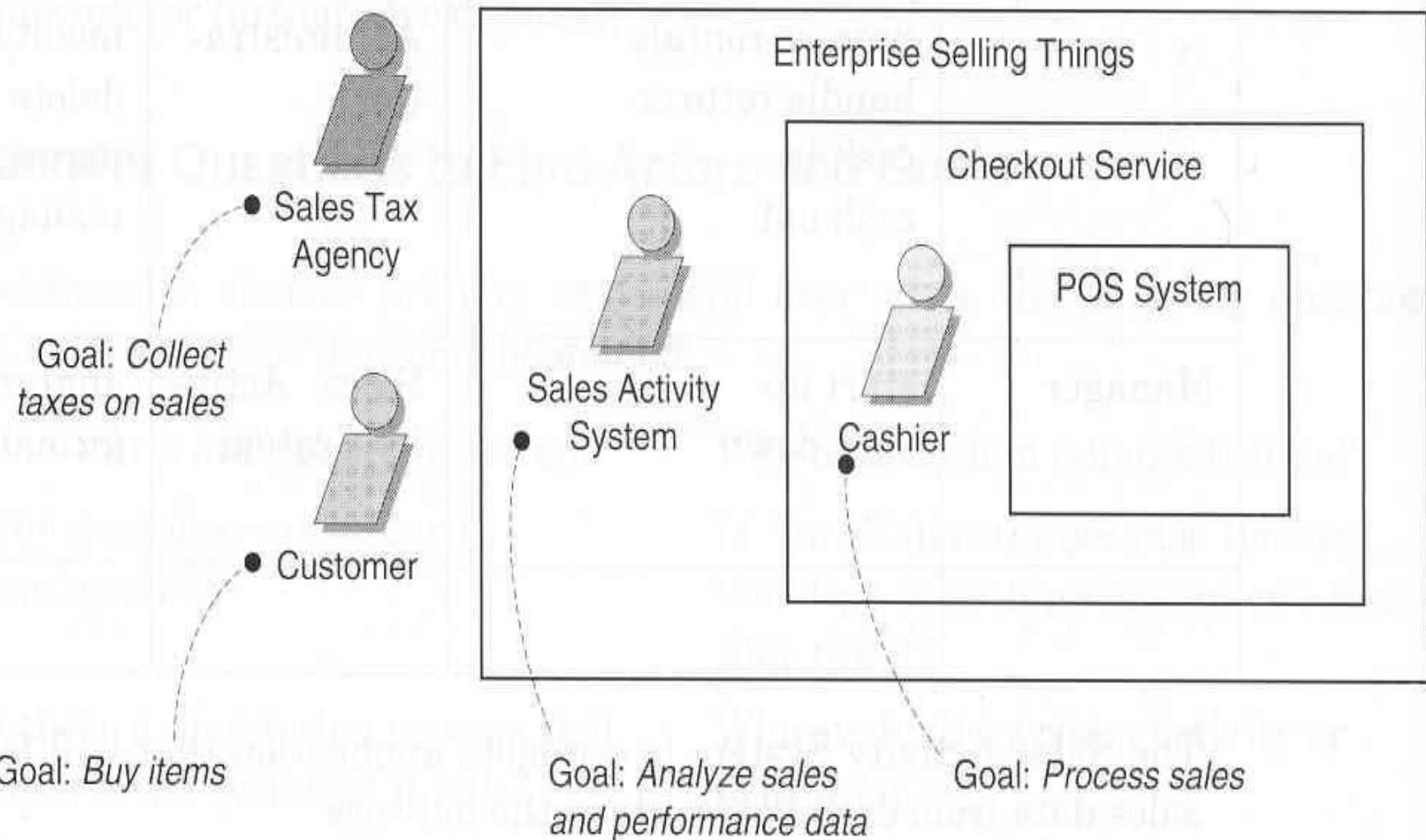
E.g., “edit user”, “delete user”, ...

Collapse into a single use case  
(e.g., *Manage Users*)

Exception to EBP

- Don’t take place at the same place/time
- But since it’s so common, it would inflate the number of use cases to model each action as a separate use case

# Actors, Goals & Boundaries



# Back to POST - Actors

---

Actors:

- Cashier
- Customer
- Supervisor
- Choosing actors:
  - Identify **system boundary**
  - Identify **entities**, human or otherwise, that will interact with the system, from outside the boundary.
- Example: A *temperature sensing device* is an actor for a temperature monitoring application.

# POST - Use Cases: First Try

---

## Cashier

- Log In
- Cash out

- Customer
  - Buy items
  - Return items

# Common mistake

---

Representing individual steps as use cases.

- Example: printing a receipt (Why is this being done ?)

# High level vs. Low Level Use cases[1]

Consider the following use cases:

- Log out
- Handle payment
- Negotiate contract with a supplier

- These use cases are at different levels. Are they all valid?  
To check, use the EBP definition.

Use Cases can be defined at  
different levels of granularity

# High level vs. Low Level Use cases [2]

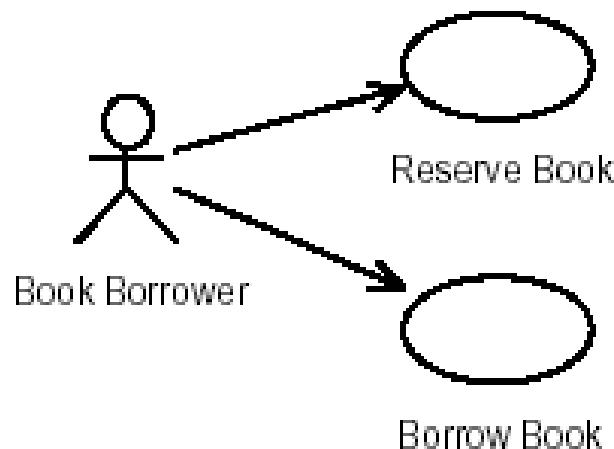
---

**Log out:** a secondary goal; it is necessary to do something but not useful in itself.

- **Handle payment:** A **necessary EBP**. Hence a primary goal.
- **Negotiate contract:** Most likely this is **too high a level**. It is composed of several EBPs and hence must be broken down further.

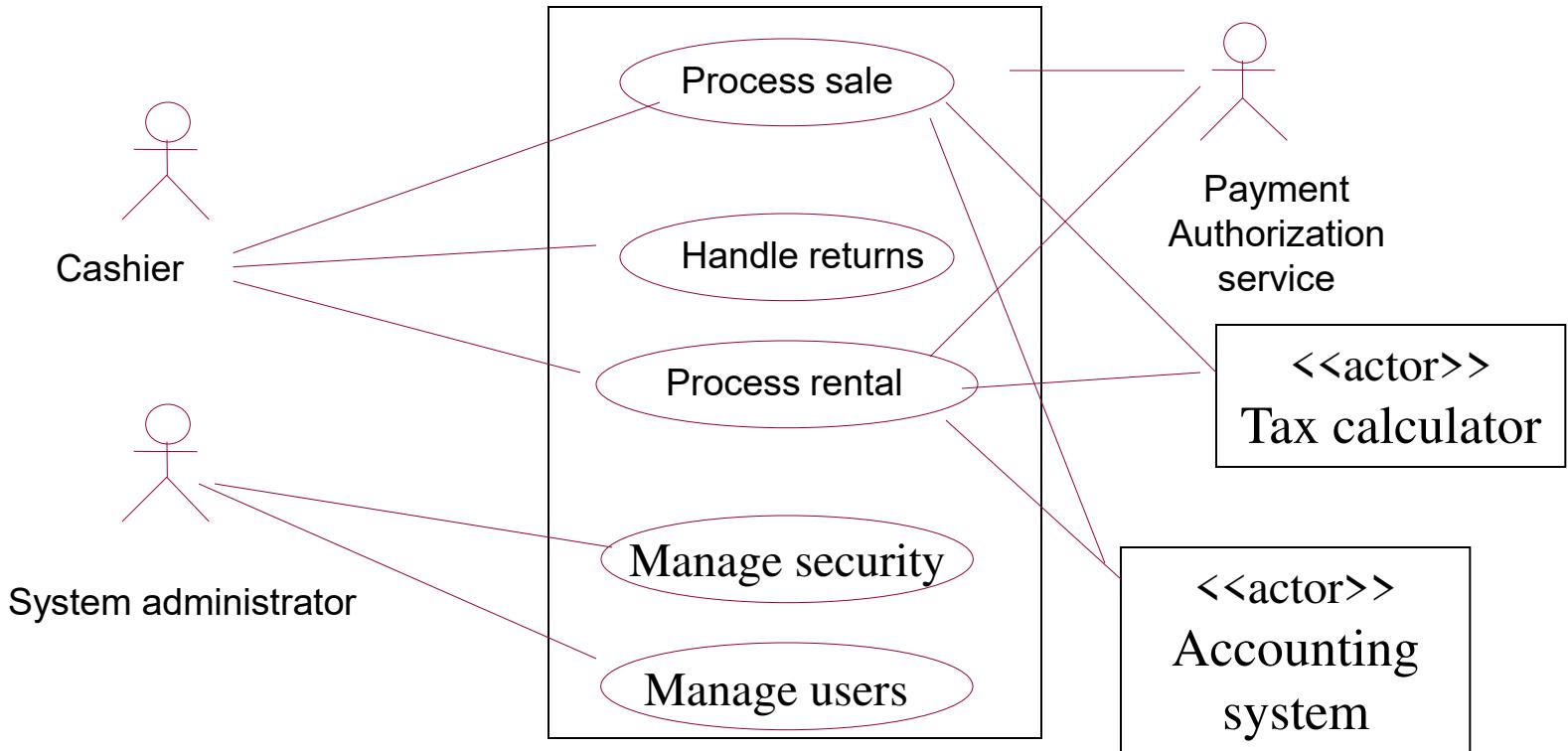
- *Use case diagrams* in *UML* are very easy to understand without knowing *UML* in detail!
  - *Actors* (external users of the system - also includes other systems)
  - *Use cases* (how the system can be used by the user)

### Example



Book Borrower  
can Reserve  
Book and Borrow  
Book – the only  
two operations  
defined

# Use Case Diagram - Example



**Use Case Diagram:** illustrates a set of use cases for a system.

# More on Use Cases

---

Narrate use cases **independent** of implementation

- State **success** scenarios (how do you determine the success of a use case).
- A use case corresponds to one or more **scenarios**.
- Agree on a **format** for use case description
- Name a use case starting with a verb in order to emphasize that it is a process (**Buy** Items, **Enter** an order, **Reduce** inventory)

# Exception handling

---

Document exception handling or branching

- What is expected of the system when a “Buy Item” fails ?
  
- What is expected of the system when a “credit card” approval fails ?

# A sample Use Case

---

**Use case:**

Buy Items

**Actors:**

Customer, Cashier

**Description:**

A customer arrives at a checkout with items to purchase. The cashier records the purchase items and collects payment.

# Ranking Use Cases

---

Use some ordering that is customary to your environment

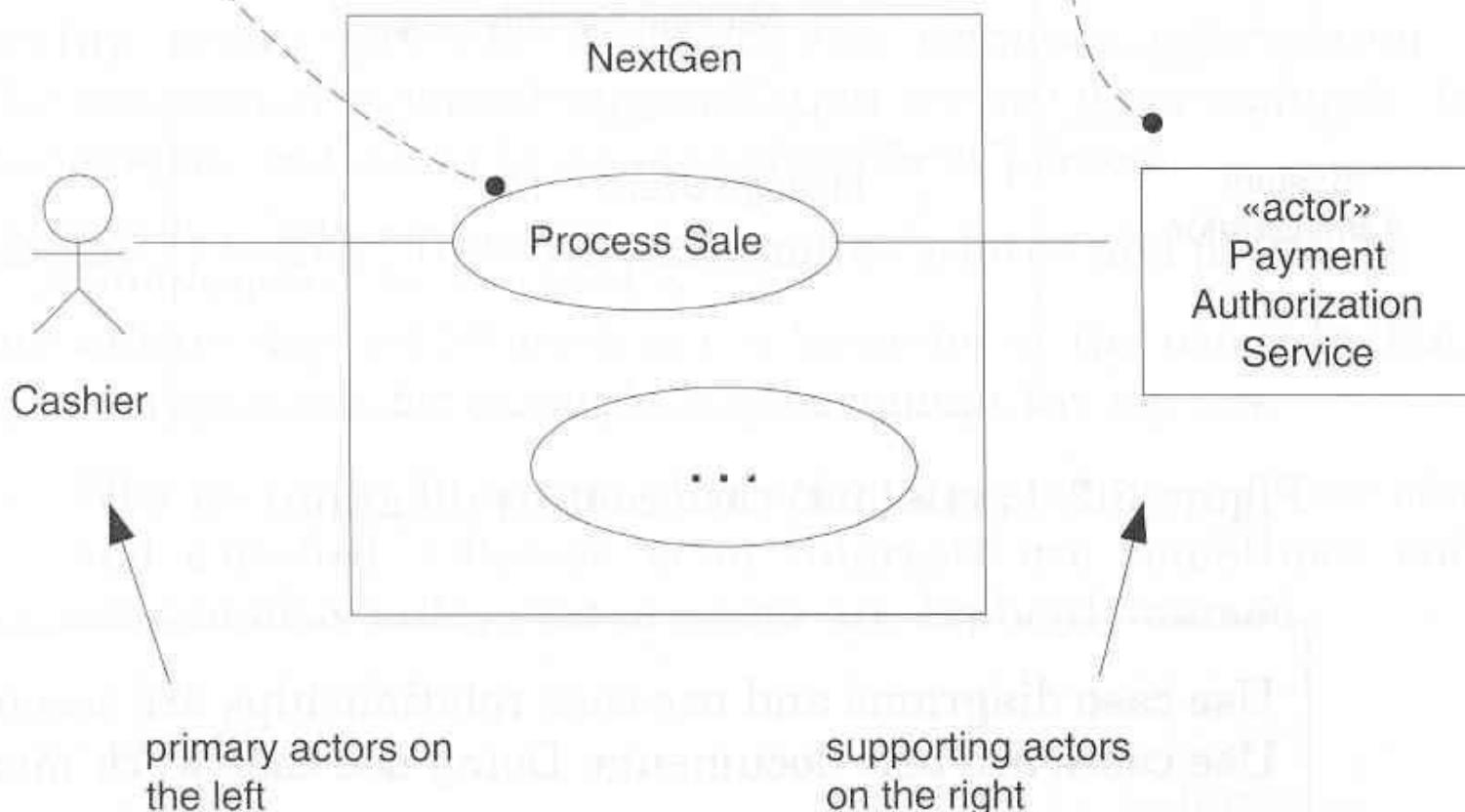
- Example: High, Medium, Low
- Example: Must have, Essential, Nice to have

- Useful when deciding what goes into an increment
- POST example:
  - Buy Items - High
  - Refund Items - Medium (Why?)
  - Shut Down POST terminal - Low

# Notation Guidelines

For a use case context diagram, limit the use cases to user-goal level use cases.

Show computer system actors with an alternate notation to human actors.



# Use Case Tips

---

Keep them Simple.

Don't worry about capturing all the use cases.

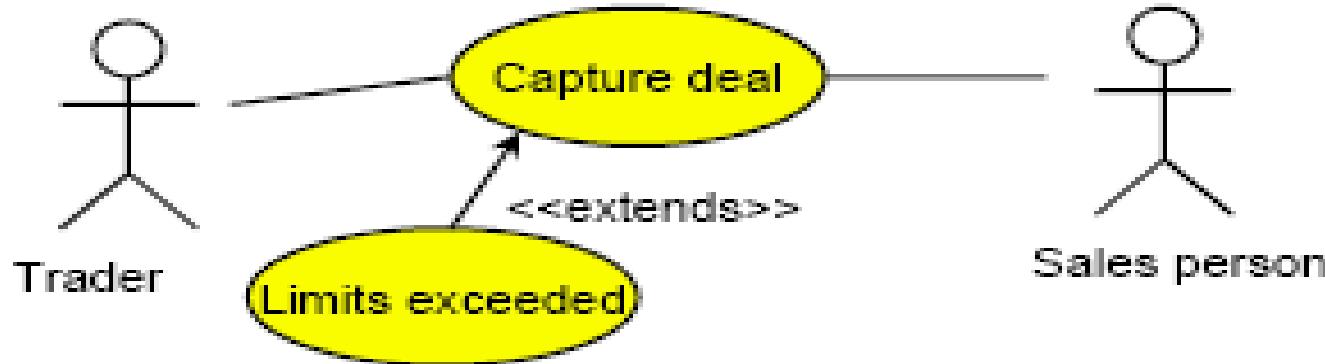
Don't worry about finding all the details

They are going to change tomorrow.

They are *Just in Time Requirement*.

# Extend

Use the **extend** relationship when you have a use case that is **similar** to another use case but does a bit more.



- In this example the basic use case is Capture deal.
- In case certain limits for a deal are exceeded the additional use case Limits exceeded is performed.

# Extend

---

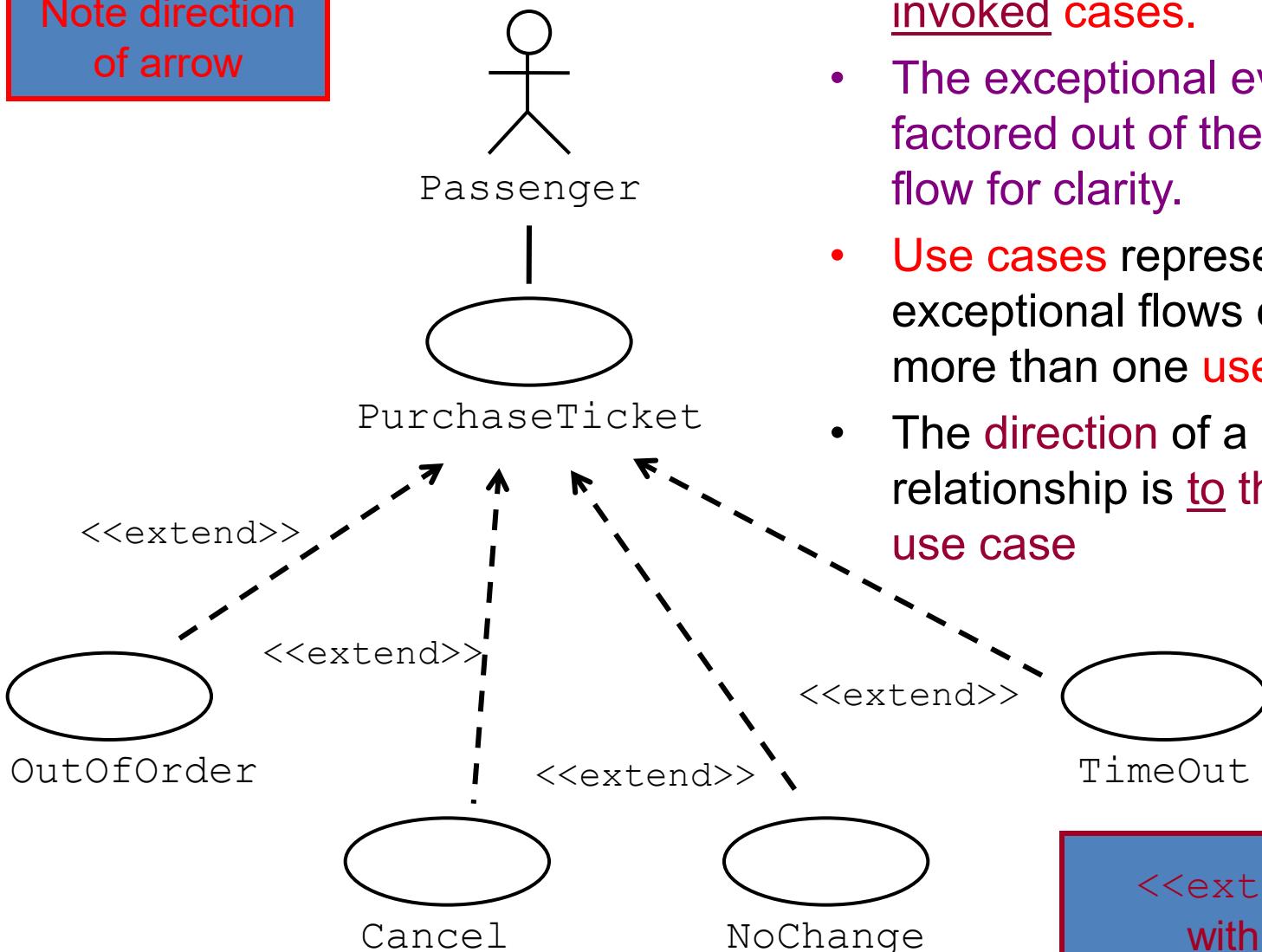
Extensions are used instead of modeling every **extra** case by a single **use case**.

Extensions are used instead of creating a **complex use case** that covers all variations.

How do you address case variation?

- Capture the simple, **normal** use case **first**.
- For **every step** in that use case ask: “What could go wrong here?”
- Plot all **variations** as extensions of the given use case.

Note direction  
of arrow

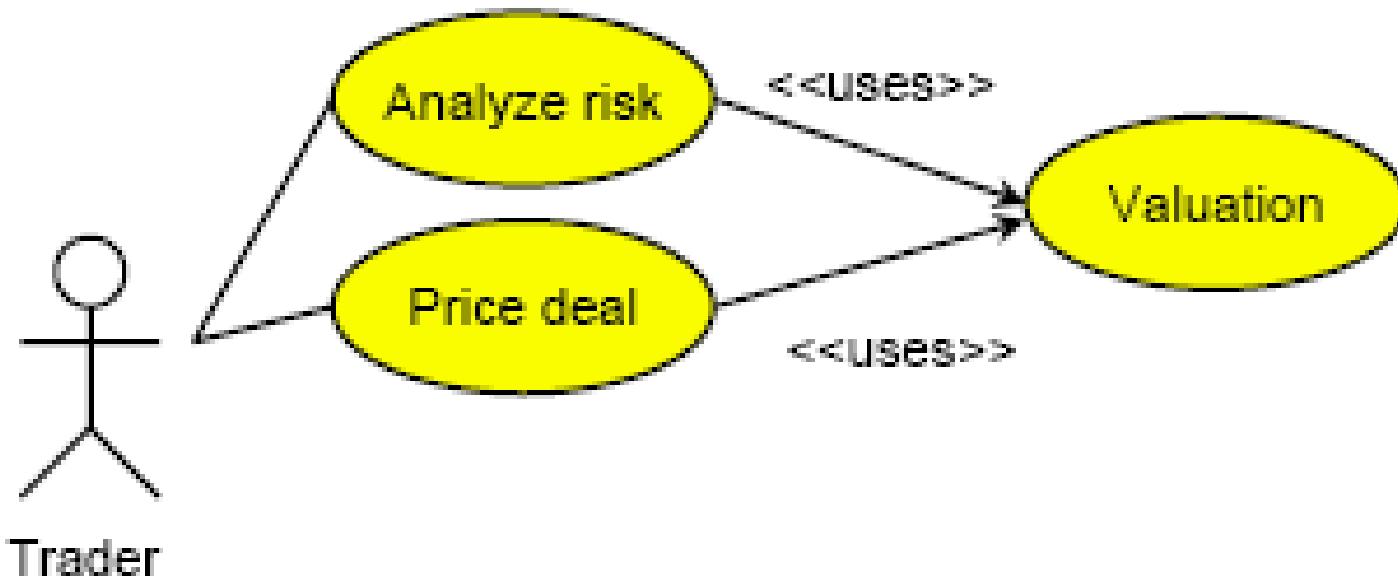


- **<<extend>>** relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a **<<extend>>** relationship is to the extended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

**<<extend>>** shown  
with dotted line

# Include

The **Include** relationship occurs when you have a chunk of behavior that is **similar across** several **use cases**.



# Include and Extend

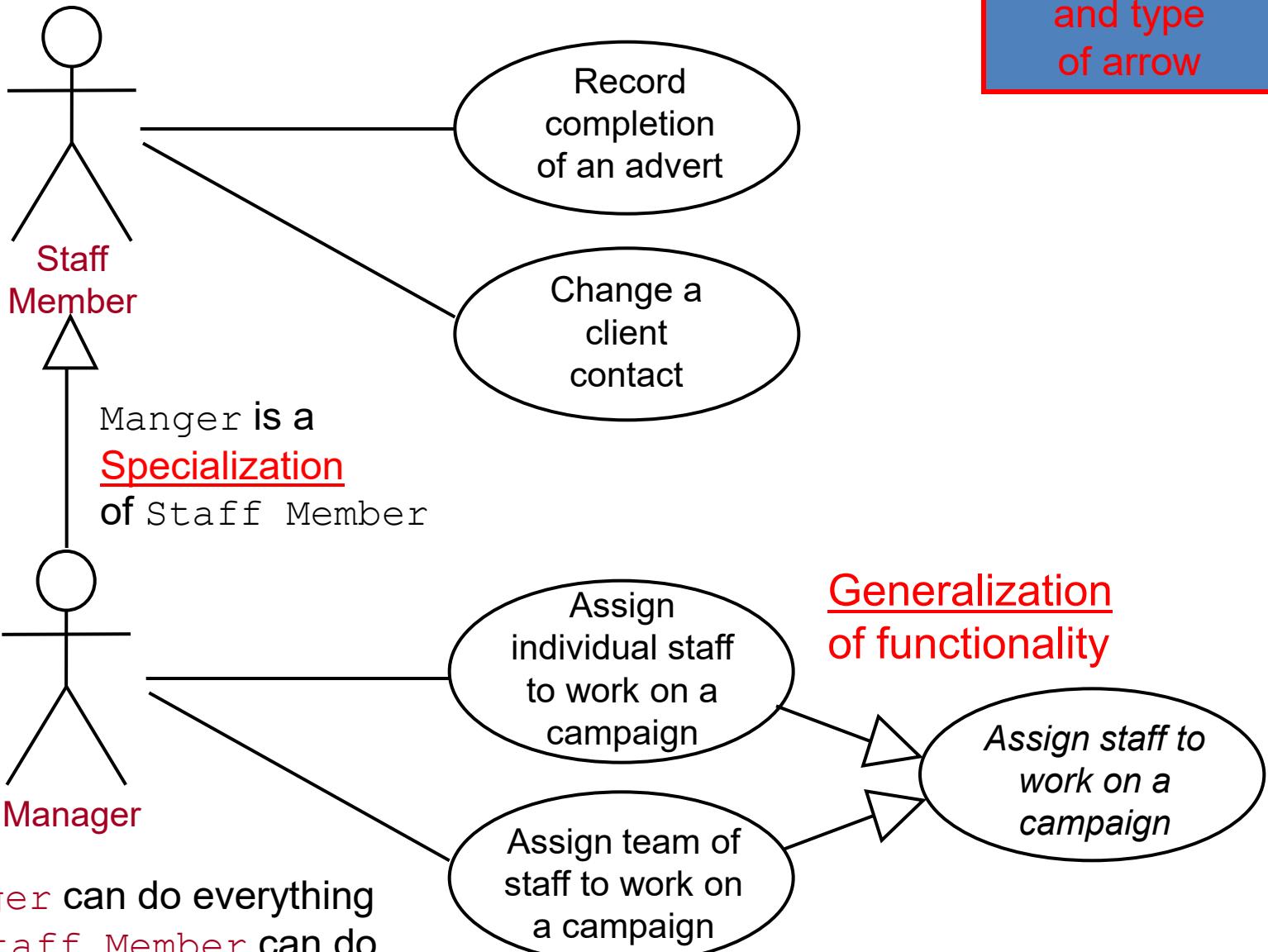
---

The similarities between **extend** and **Include** are that they both imply **factoring out** common **behavior** from several use cases to a single use case that is

- used by several other use cases or
- extended by other use cases.

Apply the following rules:

- Use **extend**, when you are describing a **variation** on normal behavior.
- Use **Include** when you want to split off **repeating** details in a use case.



**Manager** can do everything that **Staff Member** can do, and more.

---

# Drawing System Sequence Diagrams

# System Behavior

---

## Objective

- identify system events and system operations
- create system sequence diagrams for use cases

# System Behavior and UML Sequence Diagrams



It is useful to investigate and define the behavior of the software as a “black box”.

System behavior is a description of *what the system does* (without an explanation of how it does it).

Use cases describe how external actors interact with the software system. During this interaction, an actor generates events.

A request event initiates an operation upon the system.

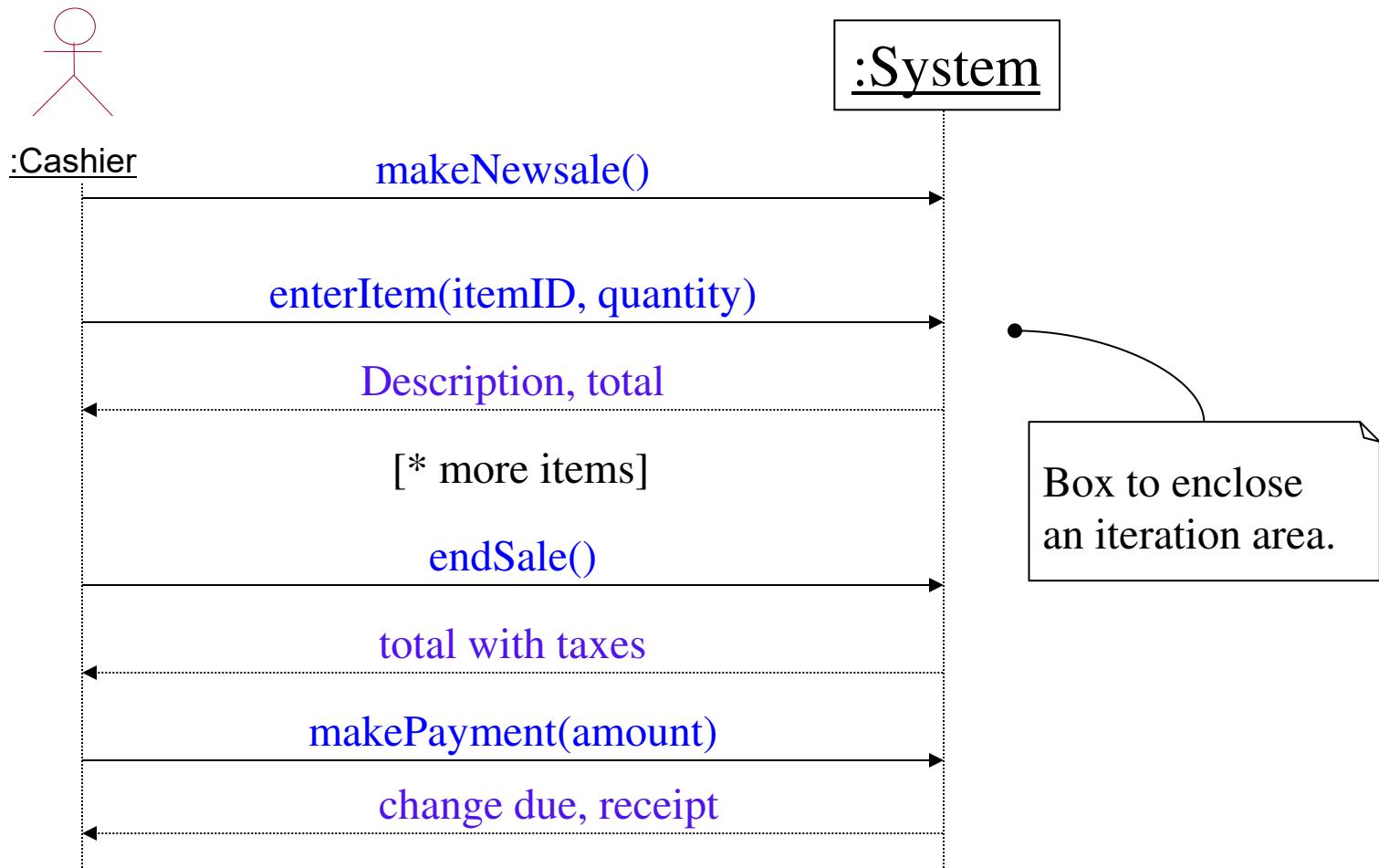
# System Behavior and System Sequence Diagrams (SSDs)

A system sequence diagram is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

All systems are treated as a black box; the diagram places emphasis on events that cross the system boundary from actors to systems.

# System sequence diagram-Example

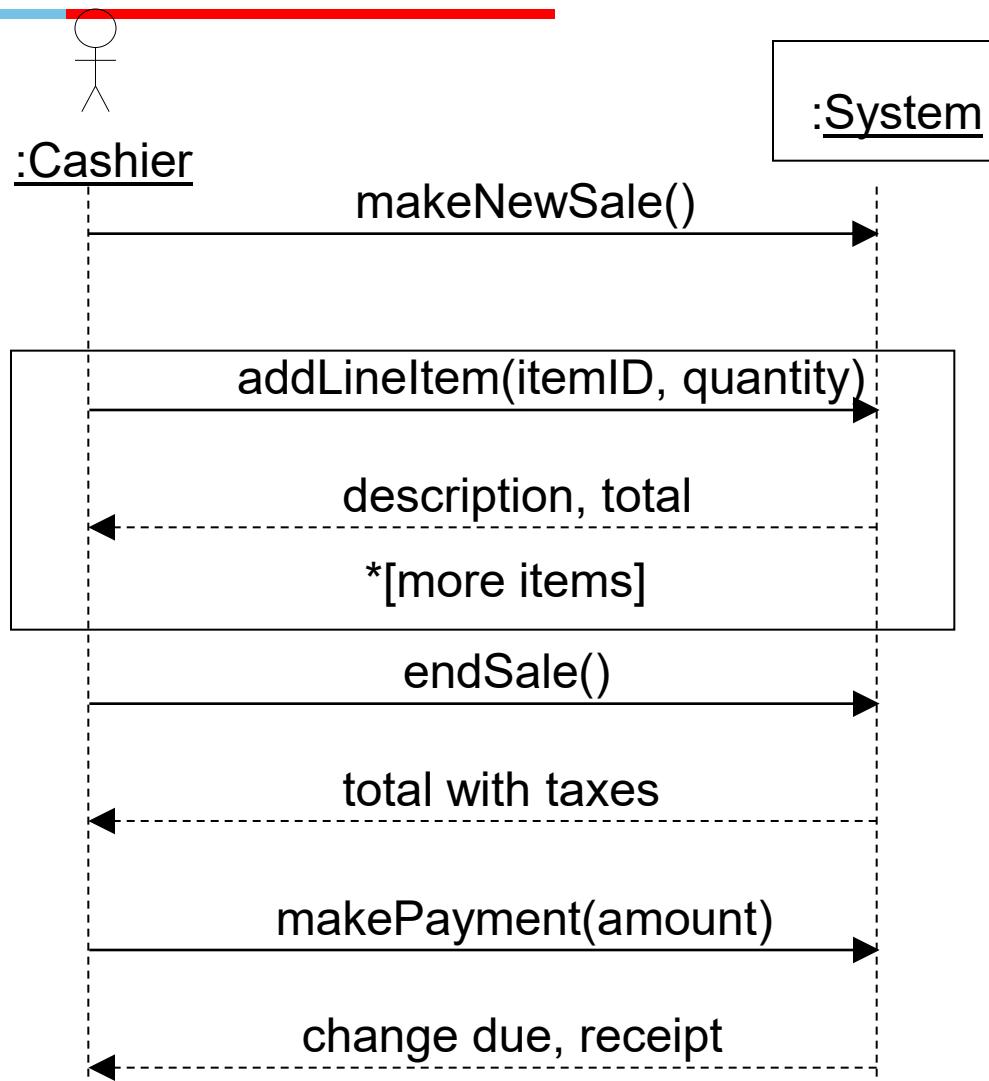
## Process Sale scenario



# SSD and Use Cases

## Simple cash-only Process Sale Scenario

1. Customer arrives at a POS checkout with goods to purchase.
  2. Cashier starts a new sale.
  3. Cashier enters item identifier.
  4. System records sale line item, and presents item description, price and running total.  
cashier repeats steps 3-4 until indicates done.
  5. System presents total with taxes calculated.
- ...



# System sequence diagrams [1]

---

SSD drawing occurs during the analysis phase of a development cycle; dependent on the creation of the use cases and identification of concepts.

- A system sequence diagram illustrates *events* from *actors* to *systems* and the external response of the system
- UML notation - Sequence Diagram *not* System Sequence Diagram.

# System sequence diagrams [2]

---

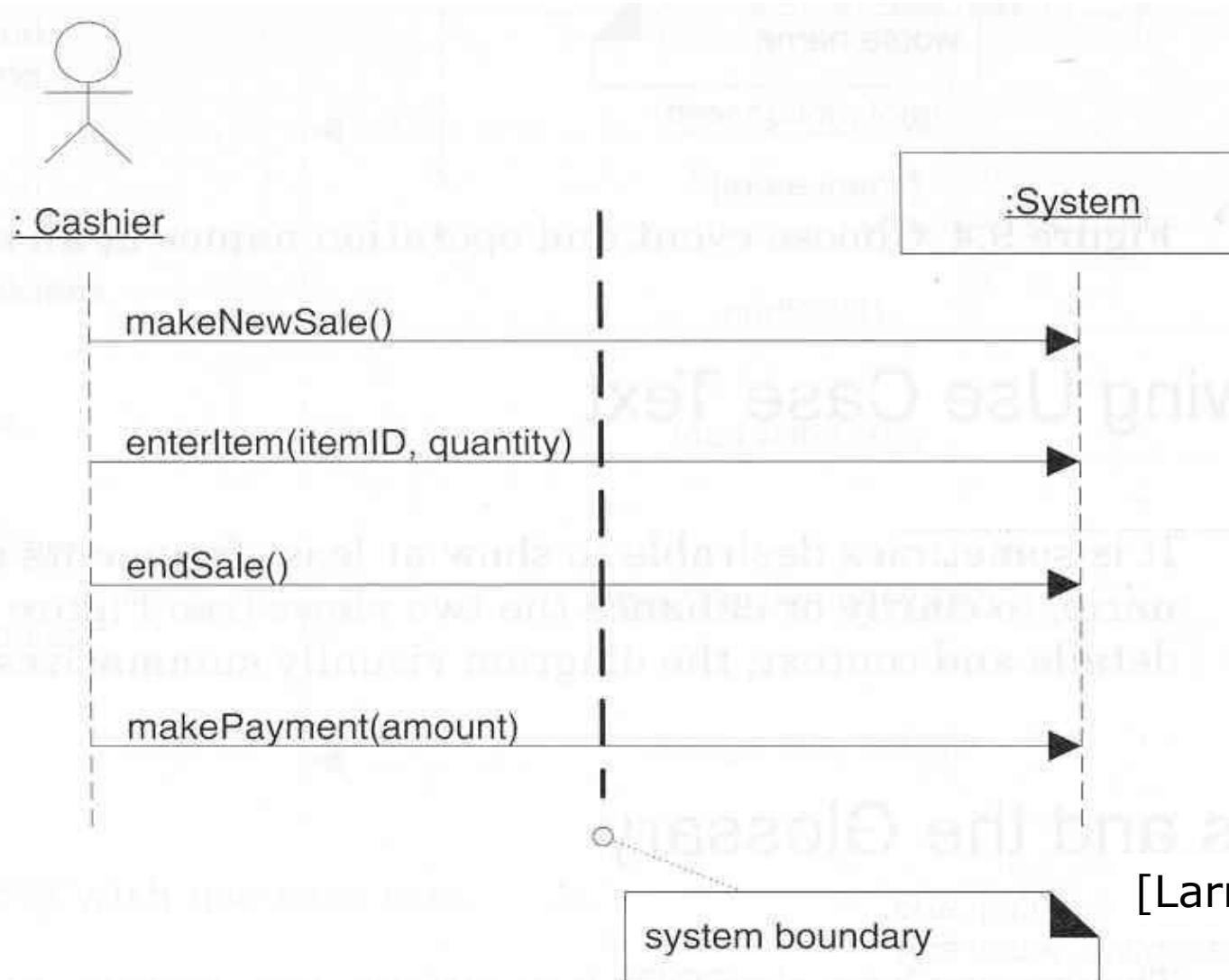
- One diagram depicts one scenario. This is the *main success* scenario.
- Frequent or complex alternate scenarios could also be illustrated.
- A system is treated as a *black box*.
- SSD is often accompanied by a textual description of the scenario to the left of the diagram.

# System sequence diagram [3]

Identify the system boundary...what is inside and what is outside.

- System event: An external event that directly stimulates the (software) system.
- Events are initiated by actors.
- Name an event at the level of *intent* and *not* using their physical input medium or interface widgets.
  - *enterItem()* is better than *scan()*.
- Keep the system response at an abstract level.
  - *description, total* is preferred over *display description and total on the POS screen*.

# The System Boundary



# Agate Case Study

Agate is an advertising agency in Birmingham. Agate deals with other companies that it calls clients. A record is kept of each client company. Clients have advertising campaign, and a record is kept of every campaign. Each campaign includes one or more adverts. Agate nominates members of creative team, which work on campaign.

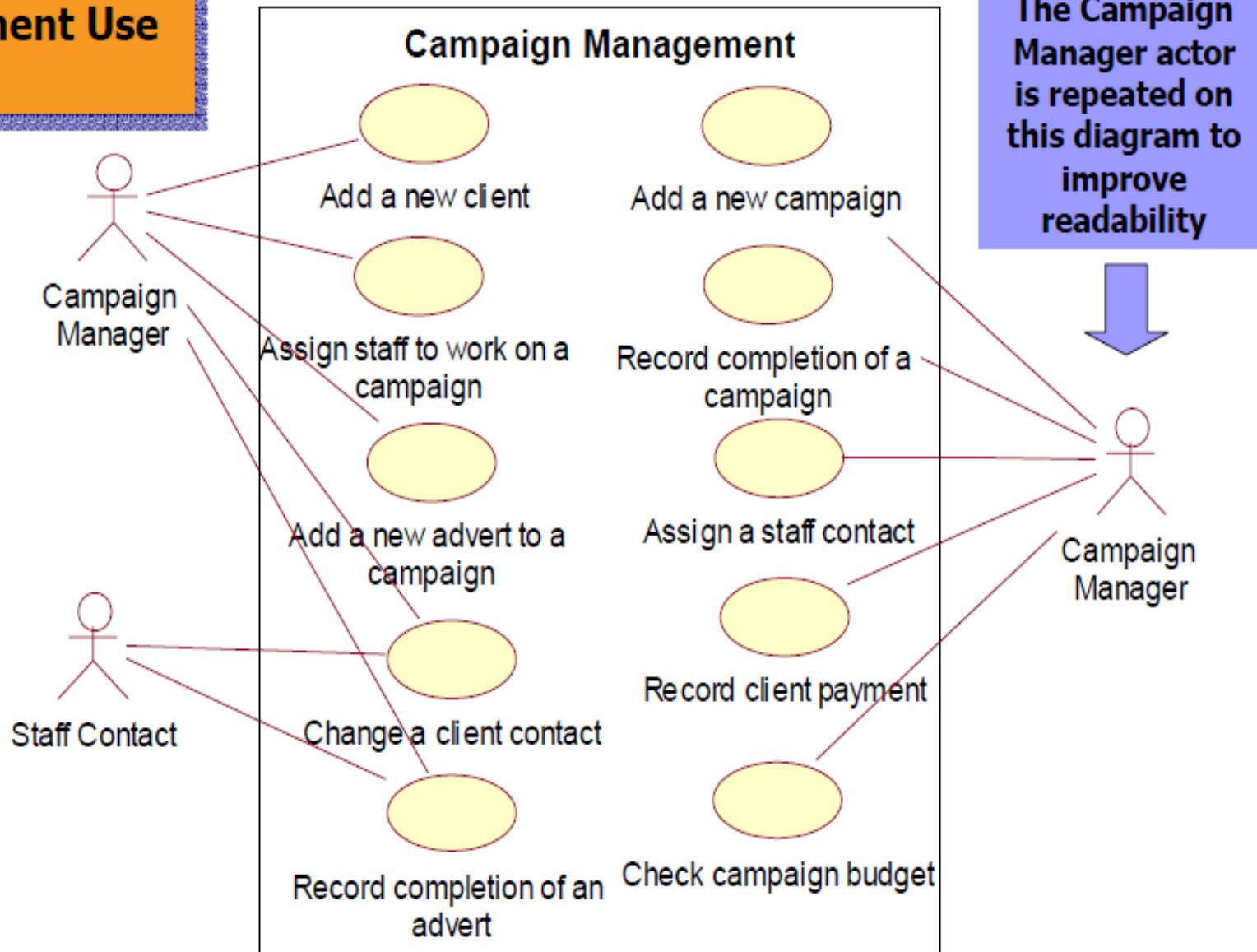
One member of the creative team manages each campaign. Staff may be working on more than one project at a time. When a campaign starts, the manager responsible estimates the likely cost of the client and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. When the campaign is completed, an actual completion date and the actual cost are recorded. When the client pays, the date paid is recorded. The manager checks the campaign budget periodically.

The system should also hold the staff grades and calculate staff salaries.

# Sample Requirements List

No.	Requirement	Use Case(s)
1.	To record names, address and contact details for each client.	Add a new client
2.	To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.	Add a new campaign
3.	To provide information that can be used in the separate accounts systems for invoicing clients for campaigns.	Record completion of a campaign
4.	To record payments for campaign that are also recorded in the separate accounts system.	Record client payment
5.	To record which staff are working on which campaigns, including the campaign manager for each campaign.	Assign staff to work on a campaign

# Campaign Management Use Cases





**BITS Pilani**  
Pilani Campus

# BITS Pilani presentation

Dr. Yashvardhan Sharma  
Computer Science and Information Systems





# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.4**

# Today's Agenda

---

- Use Case Modeling
- System Sequence Diagrams
- Activity Diagrams
- Domain Model

# Identify Use Cases

---

Capture the specific ways of using the system as dialogues between an actor and the system.

Use cases are used to

- Capture system requirements
- Communicate with end users and Subject Matter Experts
- Test the system

# Specifying Use Cases

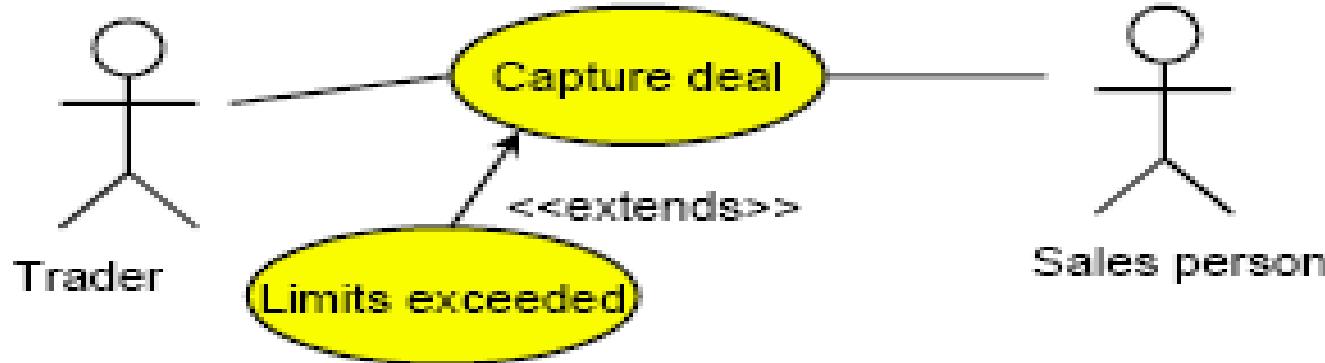
---

Create a written document for each Use Case

- Clearly define intent of the Use Case
- Define Main Success Scenario (Happy Path)
- Define any alternate action paths
- Use format of Stimulus: Response
- Each specification must be testable
- Write from actor's perspective, in actor's vocabulary

# Extend

Use the **extend** relationship when you have a use case that is **similar** to another use case but does a bit more.



- In this example the basic use case is Capture deal.
- In case certain limits for a deal are exceeded the additional use case Limits exceeded is performed.

# Extend

---

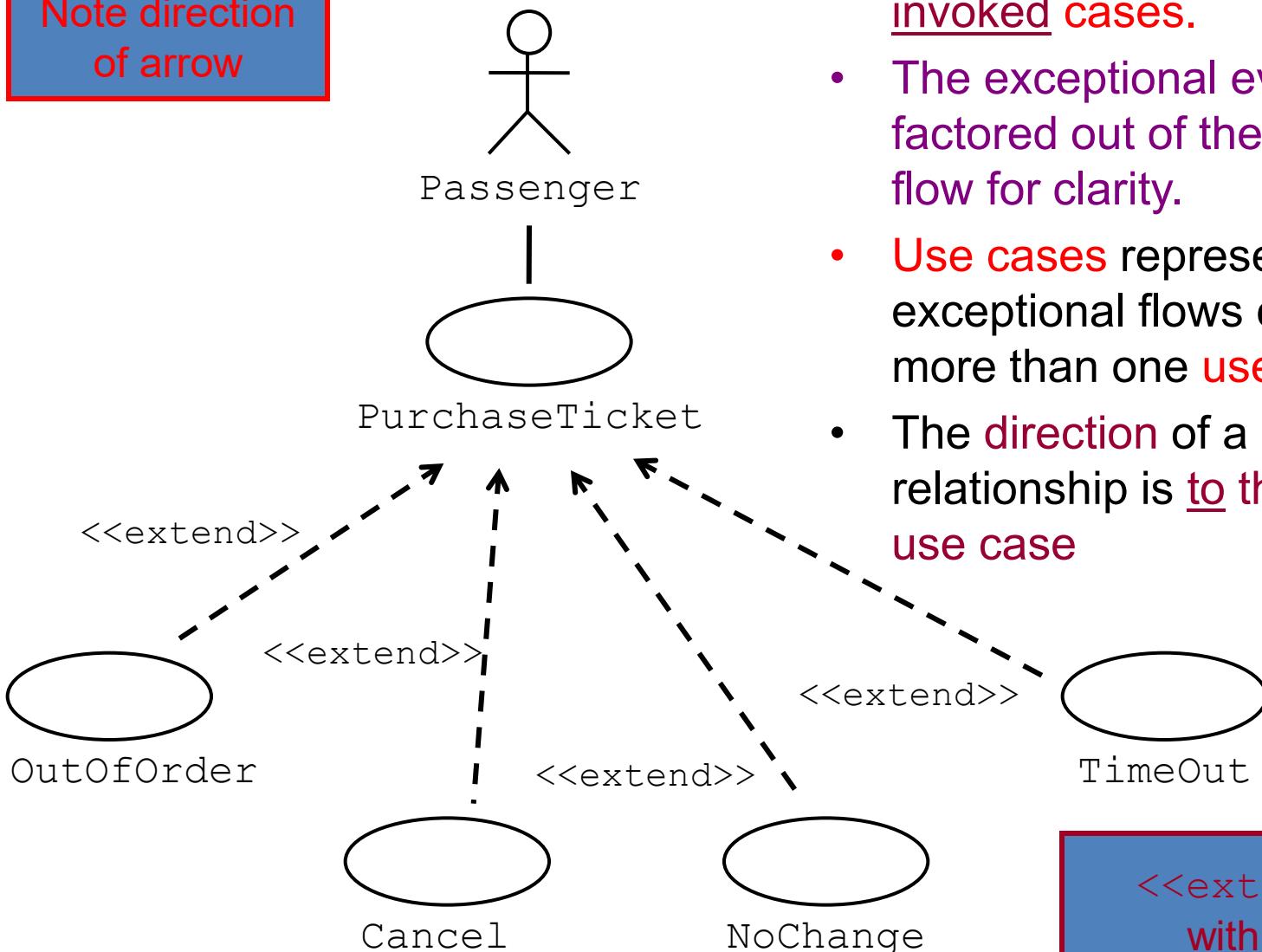
Extensions are used instead of modeling every **extra** case by a single **use case**.

Extensions are used instead of creating a **complex use case** that covers all variations.

How do you address case variation?

- Capture the simple, **normal** use case **first**.
- For **every step** in that use case ask: “What could go wrong here?”
- Plot all **variations** as extensions of the given use case.

Note direction  
of arrow



- **<<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

**<<extend>>** shown  
with dotted line

# Include

The **Include** relationship occurs when you have a chunk of behavior that is **similar across** several **use cases**.



# Include and Extend

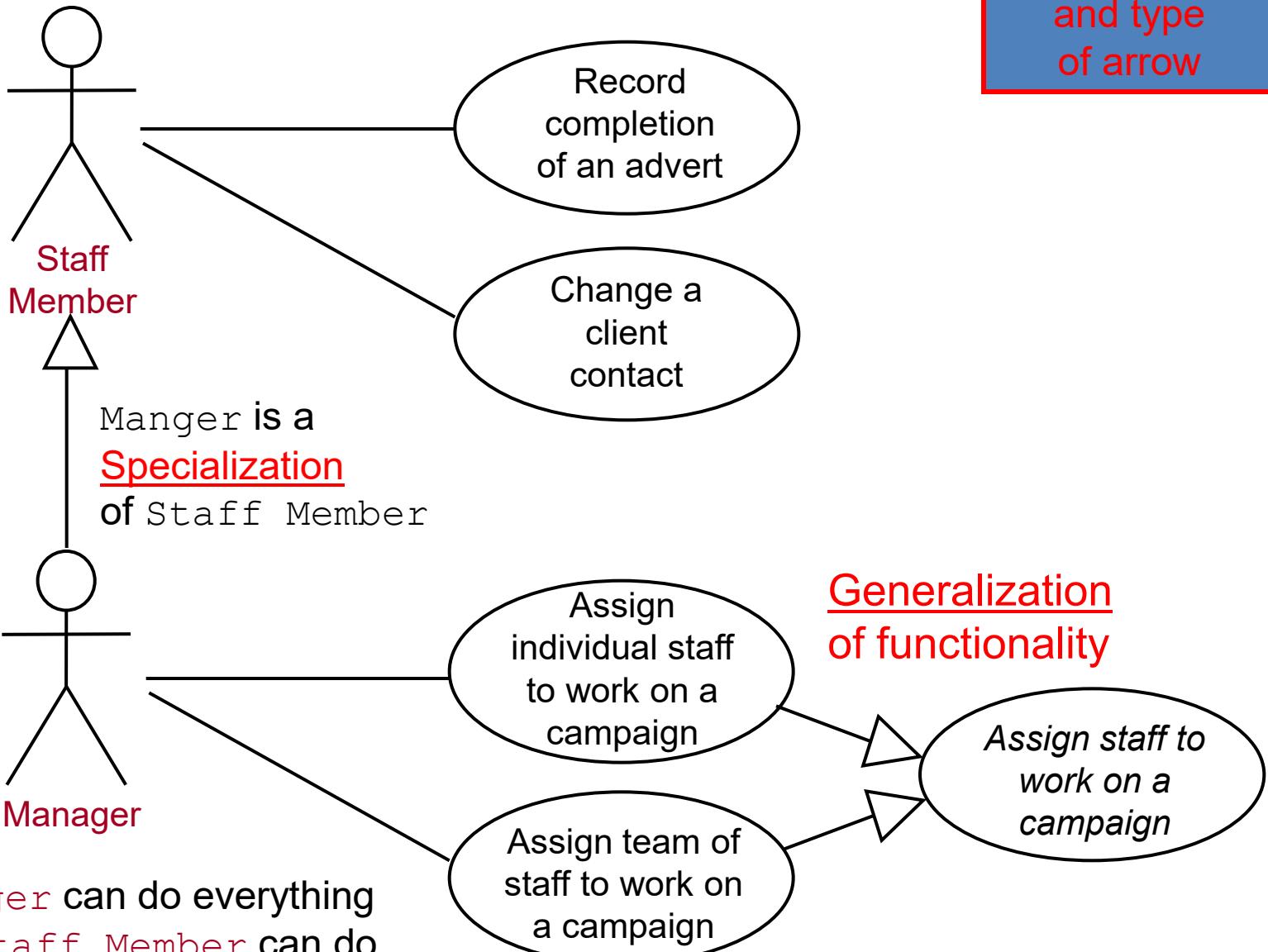
---

The similarities between **extend** and **Include** are that they both imply **factoring out** common **behavior** from several use cases to a single use case that is

- used by several other use cases or
- extended by other use cases.

Apply the following rules:

- Use **extend**, when you are describing a **variation** on normal behavior.
- Use **Include** when you want to split off **repeating** details in a use case.



---

# Drawing System Sequence Diagrams

# System Behavior

---

## Objective

- identify system events and system operations
- create system sequence diagrams for use cases

# System Behavior and UML Sequence Diagrams



It is useful to investigate and define the behavior of the software as a “black box”.

System behavior is a description of *what the system does* (without an explanation of how it does it).

Use cases describe how external actors interact with the software system. During this interaction, an actor generates events.

A request event initiates an operation upon the system.

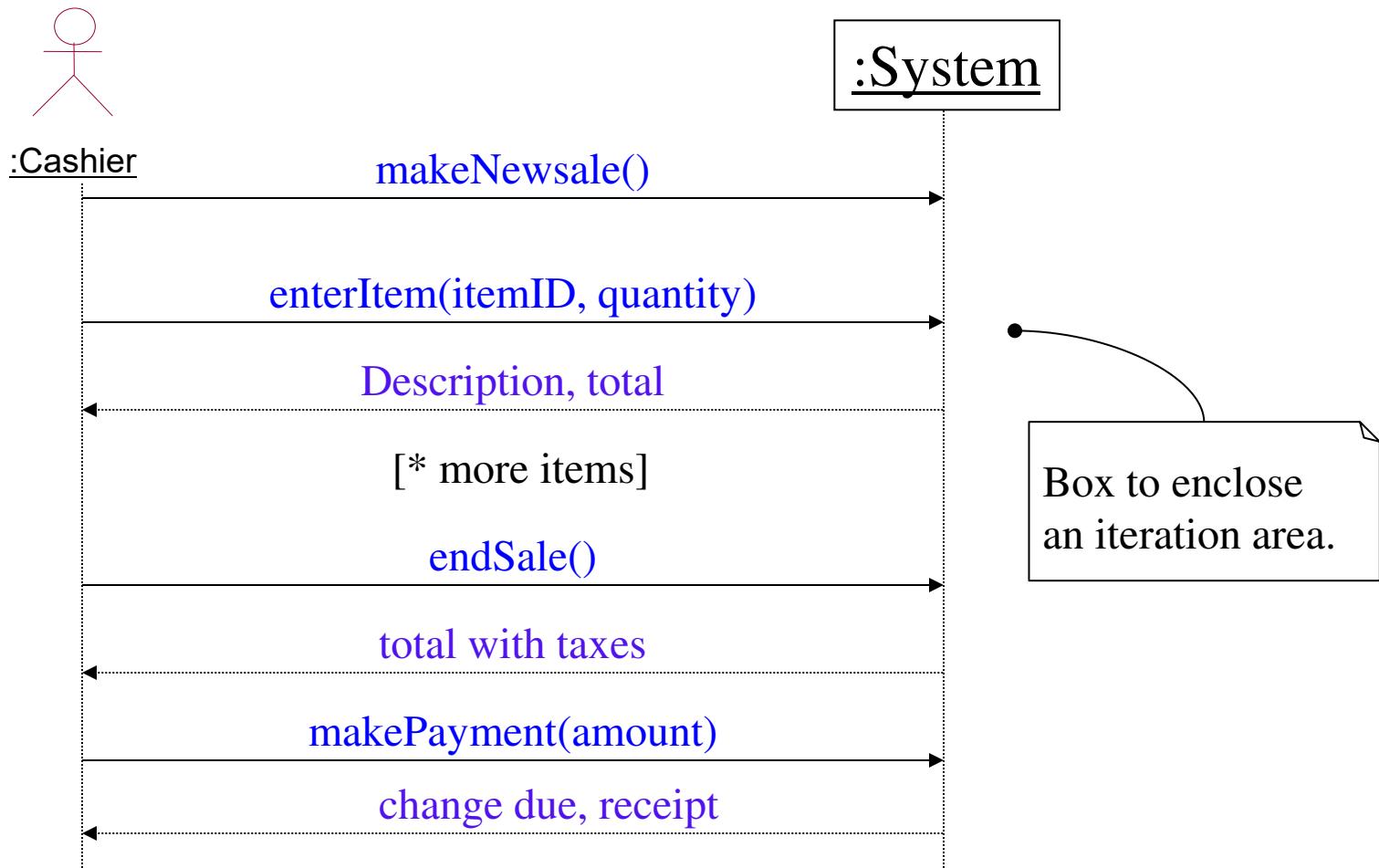
# System Behavior and System Sequence Diagrams (SSDs)

A system sequence diagram is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

All systems are treated as a black box; the diagram places emphasis on events that cross the system boundary from actors to systems.

# System sequence diagram-Example

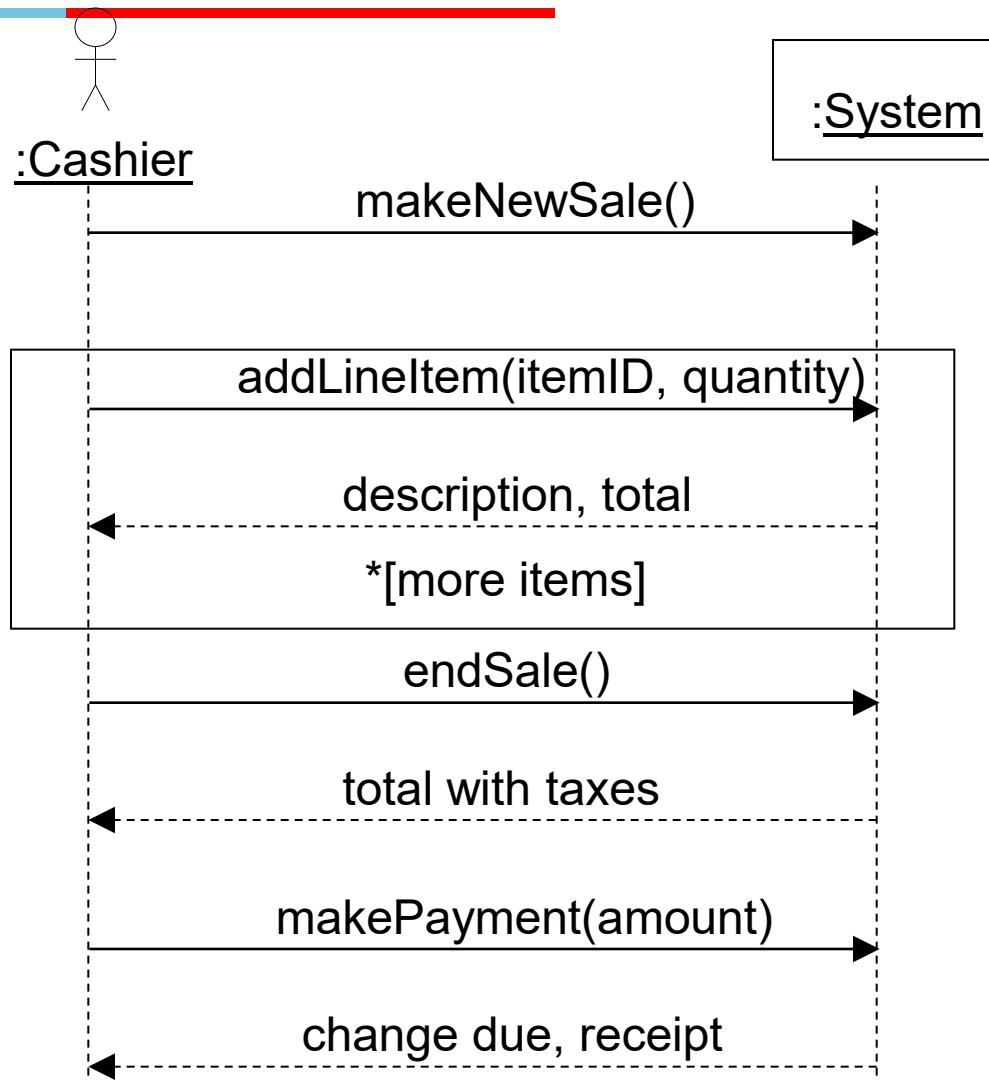
## Process Sale scenario



# SSD and Use Cases

## Simple cash-only Process Sale Scenario

1. Customer arrives at a POS checkout with goods to purchase.
  2. Cashier starts a new sale.
  3. Cashier enters item identifier.
  4. System records sale line item, and presents item description, price and running total.  
cashier repeats steps 3-4 until indicates done.
  5. System presents total with taxes calculated.
- ...



# System sequence diagrams [1]

---

SSD drawing occurs during the analysis phase of a development cycle; dependent on the creation of the use cases and identification of concepts.

- A system sequence diagram illustrates *events* from *actors* to *systems* and the external response of the system
- UML notation - Sequence Diagram *not* System Sequence Diagram.

# System sequence diagrams [2]

---

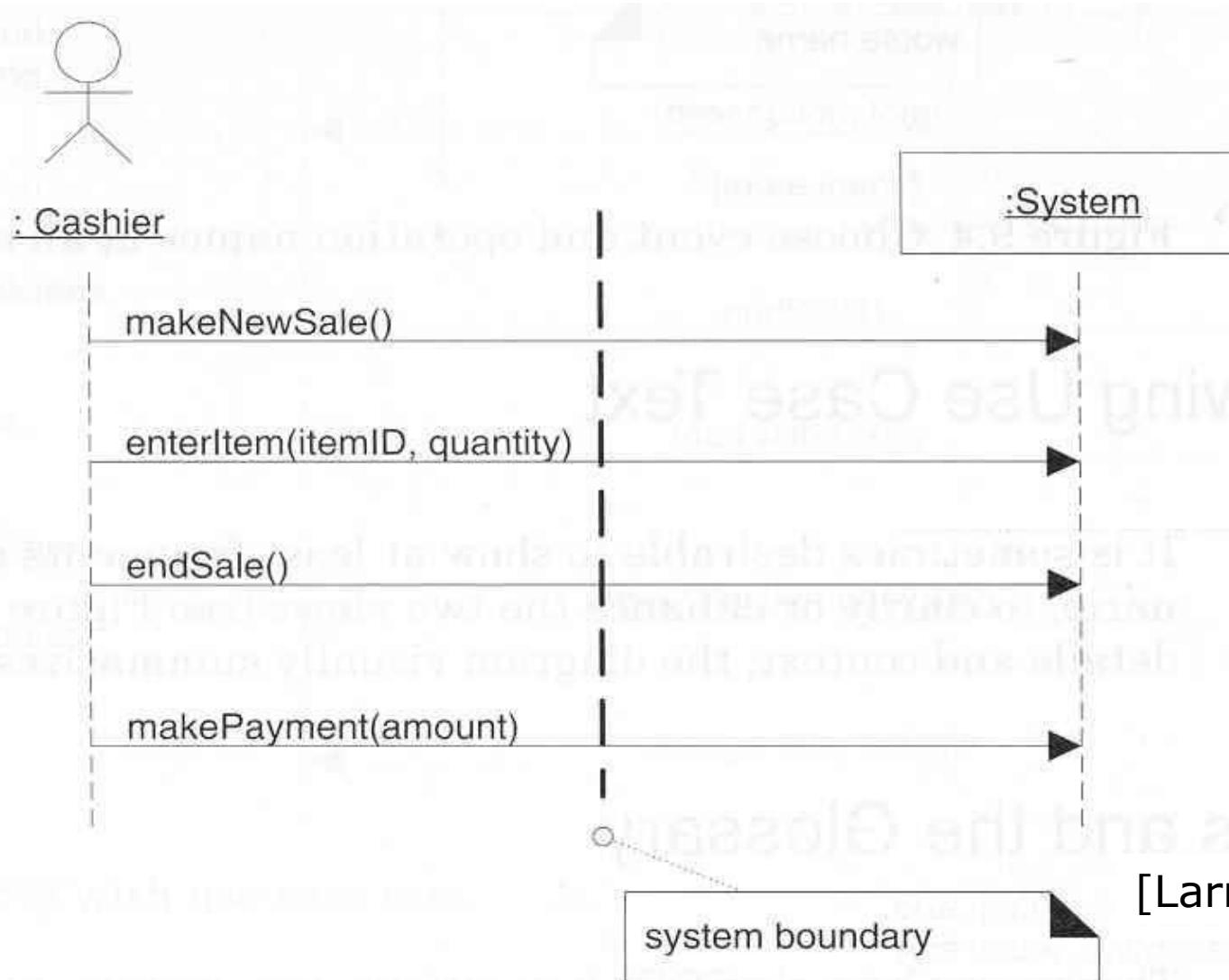
- One diagram depicts one scenario. This is the *main success* scenario.
- Frequent or complex alternate scenarios could also be illustrated.
- A system is treated as a *black box*.
- SSD is often accompanied by a textual description of the scenario to the left of the diagram.

# System sequence diagram [3]

Identify the system boundary...what is inside and what is outside.

- System event: An external event that directly stimulates the (software) system.
- Events are initiated by actors.
- Name an event at the level of *intent* and *not* using their physical input medium or interface widgets.
  - *enterItem()* is better than *scan()*.
- Keep the system response at an abstract level.
  - *description, total* is preferred over *display description and total on the POS screen*.

# The System Boundary



[Larman, 2002]

# Agate Case Study

Agate is an advertising agency in Birmingham. Agate deals with other companies that it calls clients. A record is kept of each client company. Clients have advertising campaign, and a record is kept of every campaign. Each campaign includes one or more adverts. Agate nominates members of creative team, which work on campaign.

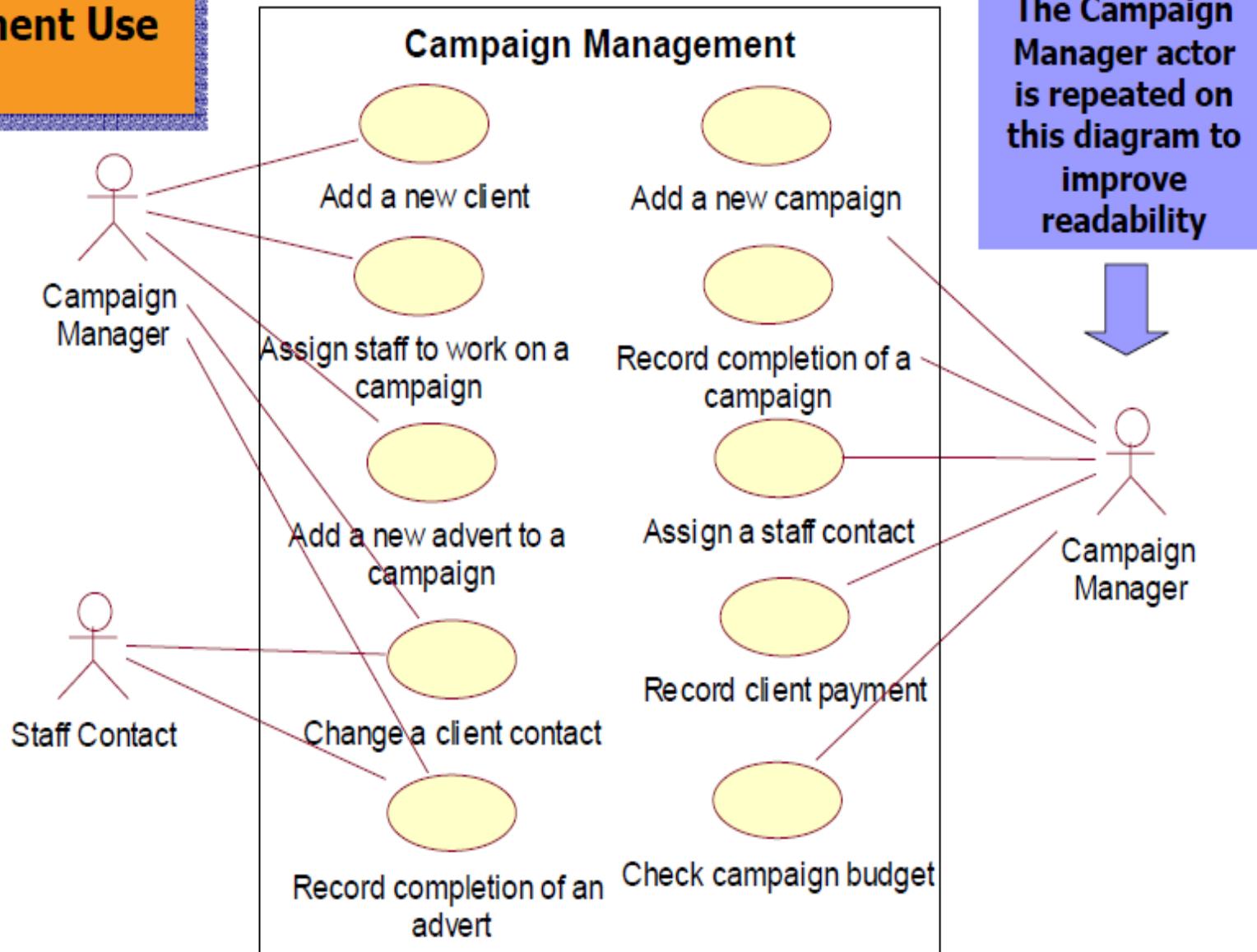
One member of the creative team manages each campaign. Staff may be working on more than one project at a time. When a campaign starts, the manager responsible estimates the likely cost of the client and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. When the campaign is completed, an actual completion date and the actual cost are recorded. When the client pays, the date paid is recorded. The manager checks the campaign budget periodically.

The system should also hold the staff grades and calculate staff salaries.

# Sample Requirements List

No.	Requirement	Use Case(s)
1.	To record names, address and contact details for each client.	Add a new client
2.	To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.	Add a new campaign
3.	To provide information that can be used in the separate accounts systems for invoicing clients for campaigns.	Record completion of a campaign
4.	To record payments for campaign that are also recorded in the separate accounts system.	Record client payment
5.	To record which staff are working on which campaigns, including the campaign manager for each campaign.	Assign staff to work on a campaign

# Campaign Management Use Cases



# Activity Diagrams

---

## Purpose

- to model a task (for example in business modelling)
- to describe a function of a system represented by a use case
- to describe the logic of an operation
- to model the activities that make up the life cycle in the Unified Process

- Model dynamic behavior of a system as a flowchart.
- Description of a business process, by activities and their I/O.

# Drawing Activity Diagrams

---

## Purpose

- Draw the activity flow of a system. Activity can be described as an operation of the system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.
- Captures dynamic behavior of the system as like other behavioral diagrams but activity diagram show message flow from one activity to another.

# UML Activity Diagram

---

An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations.

An activity diagram shows actions and control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent.

Understand the rules and style guidelines for activity diagram

Be able to create functional models using activity diagrams. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

# BPM With Activity Diagrams

---

A number of activities support a business process across several departments

Activity diagrams model the behavior in a business process

- Sophisticated data flow diagrams
- Addresses Parallel concurrent activities and complex processes

# How to draw Activity Diagram

---

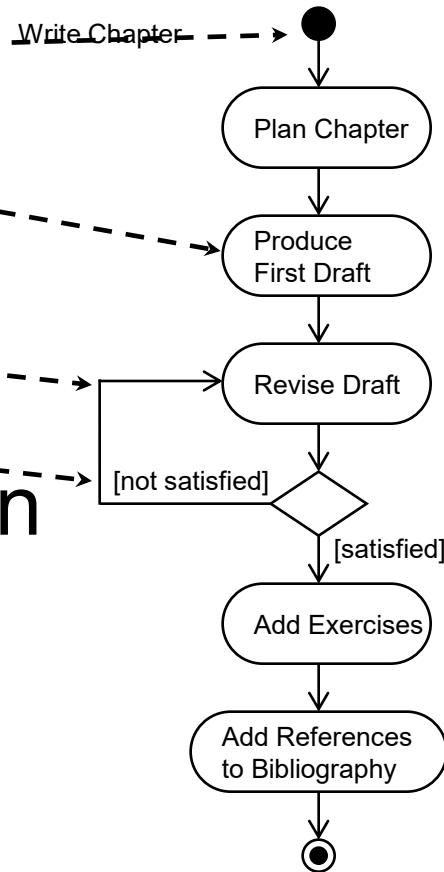
Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram.

The main element of an activity diagram is the ***activity*** itself. An activity is a function performed by the system.

After identifying the activities we need to understand how they are ***associated*** with ***constraints*** and ***conditions***.

# Diagrams in UML

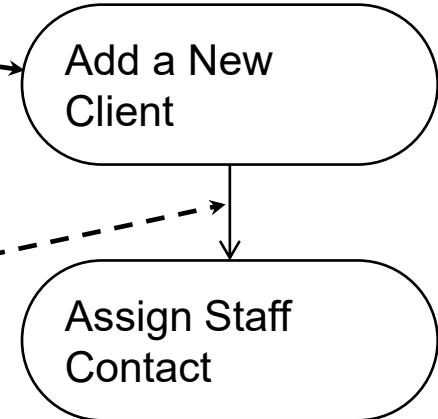
- UML diagrams consist of:
  - icons
  - two-dimensional symbols
  - paths
  - Strings
- UML diagrams are defined in the UML specification.



# Notation of Activity Diagrams

## Activities

- rectangle with rounded ends
- meaningful name



## Transitions

- arrows with open arrowheads

# Notation of Activity Diagrams

**Start state**

- black circle

**Decision points**

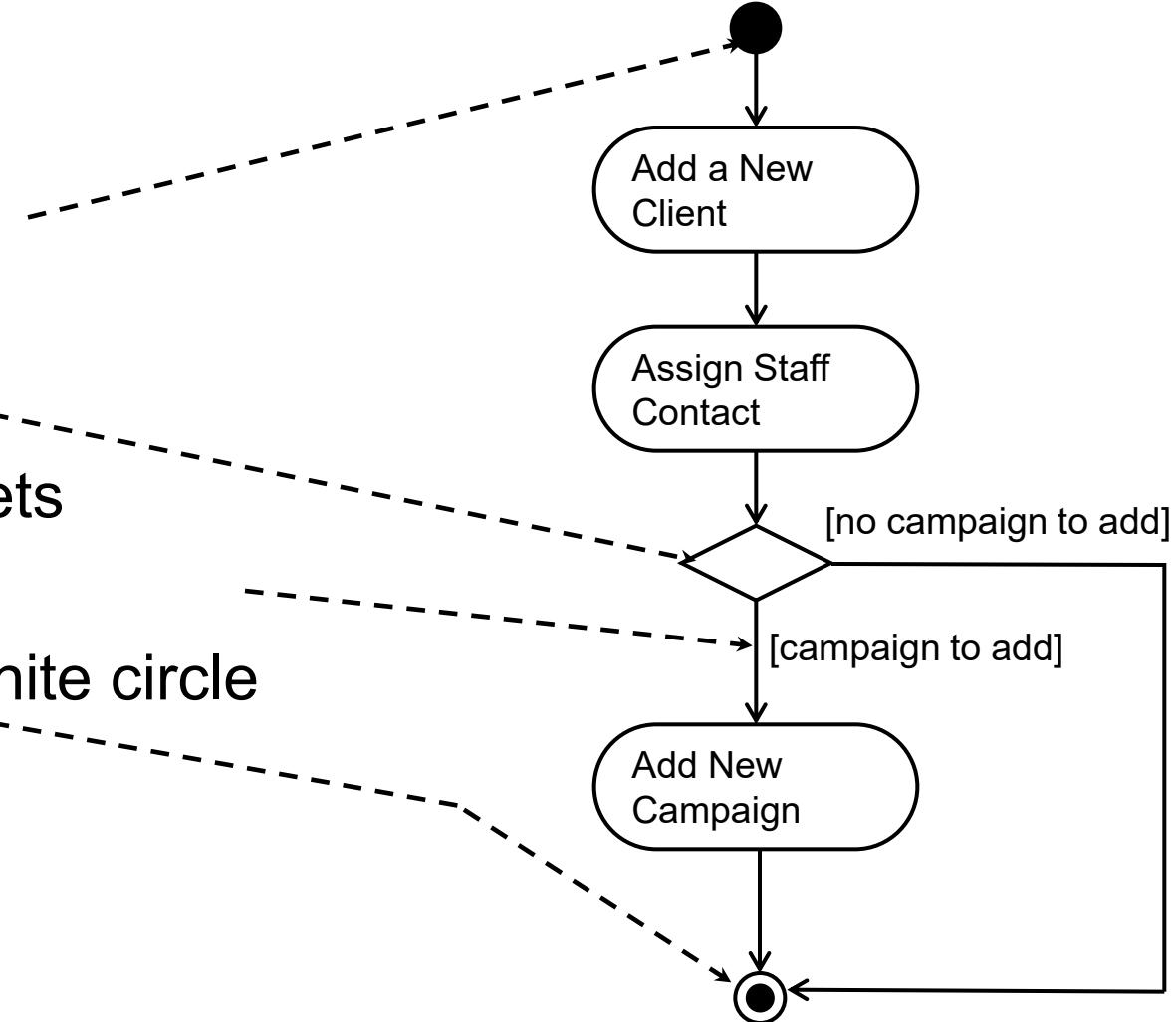
- diamond

**Guard conditions**

- in square brackets

**Final state**

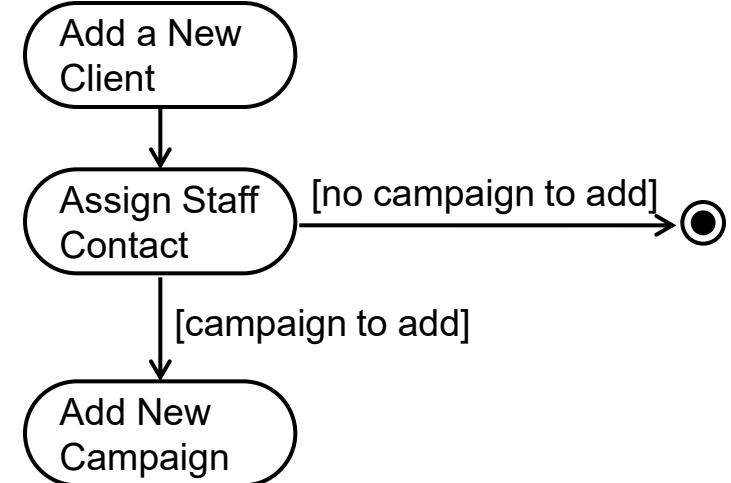
- black circle in white circle



# Notation of Activity Diagrams

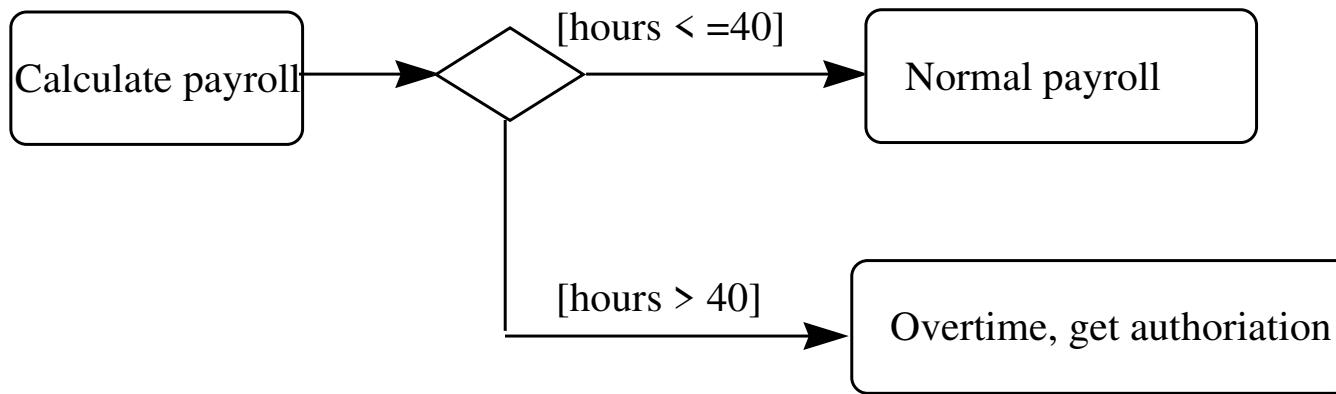
## Alternative notation for branching:

- alternative transitions are shown leaving the activity with guard conditions

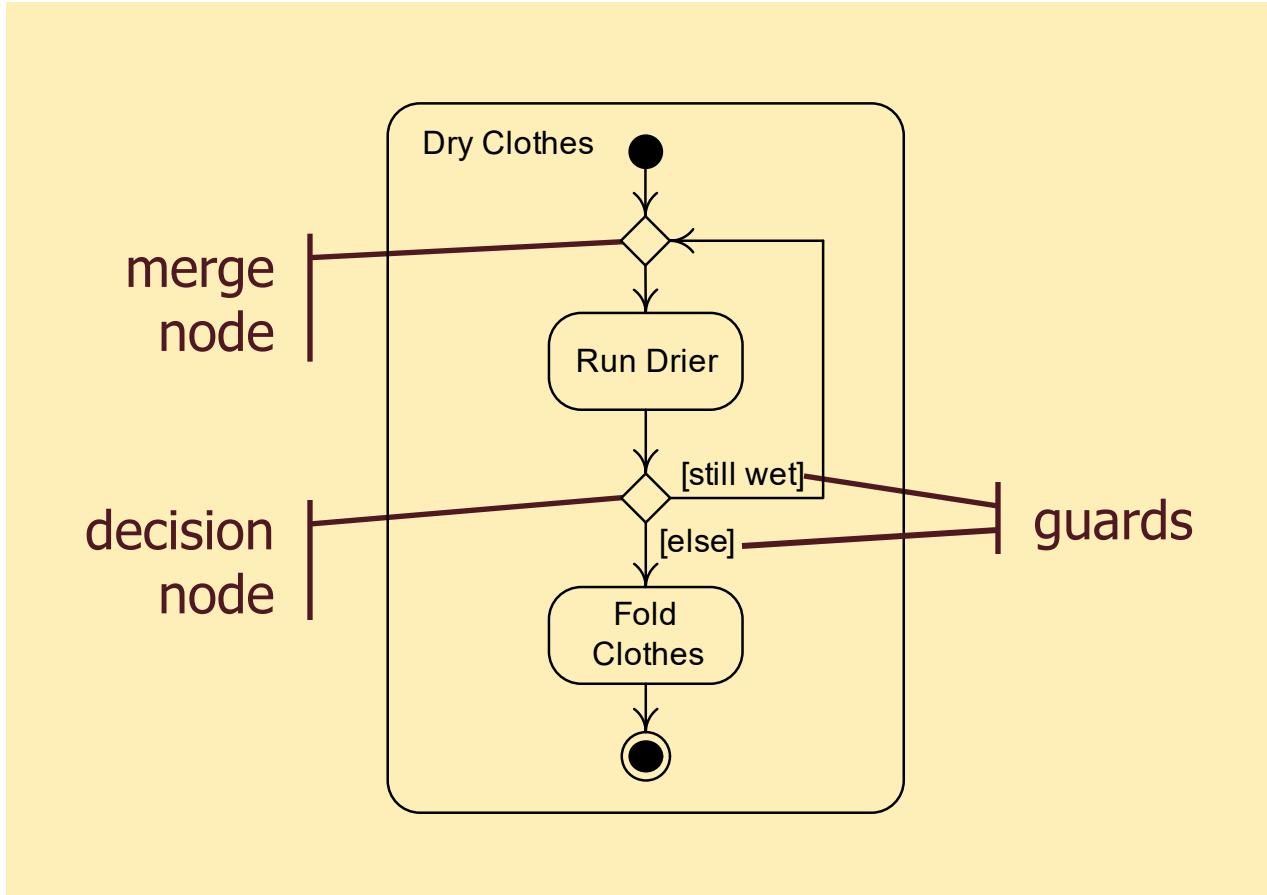


Note that guard conditions do not have to be mutually exclusive, but it is advisable that they should be

# UML Activity Decision

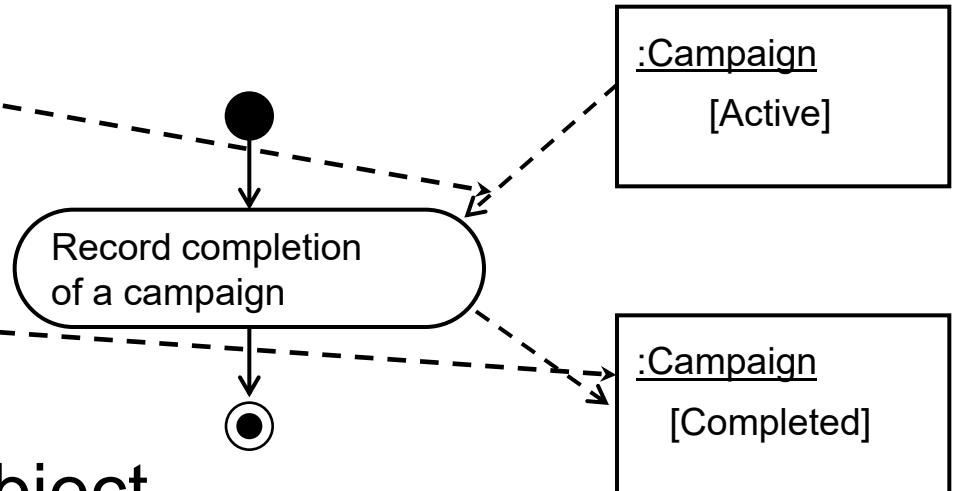


# Branching Nodes



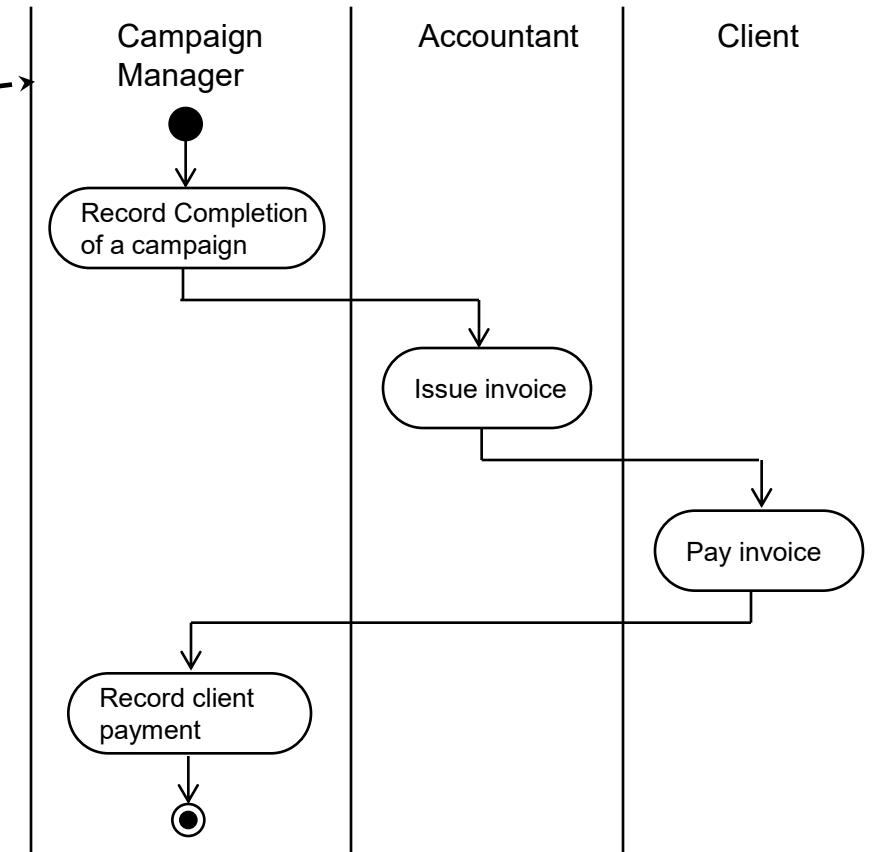
# Notation of Activity Diagrams

- Object flows
  - dashed arrow
- Objects
  - rectangle
  - with name of object underlined
  - *optionally shows the state of the object in square brackets*

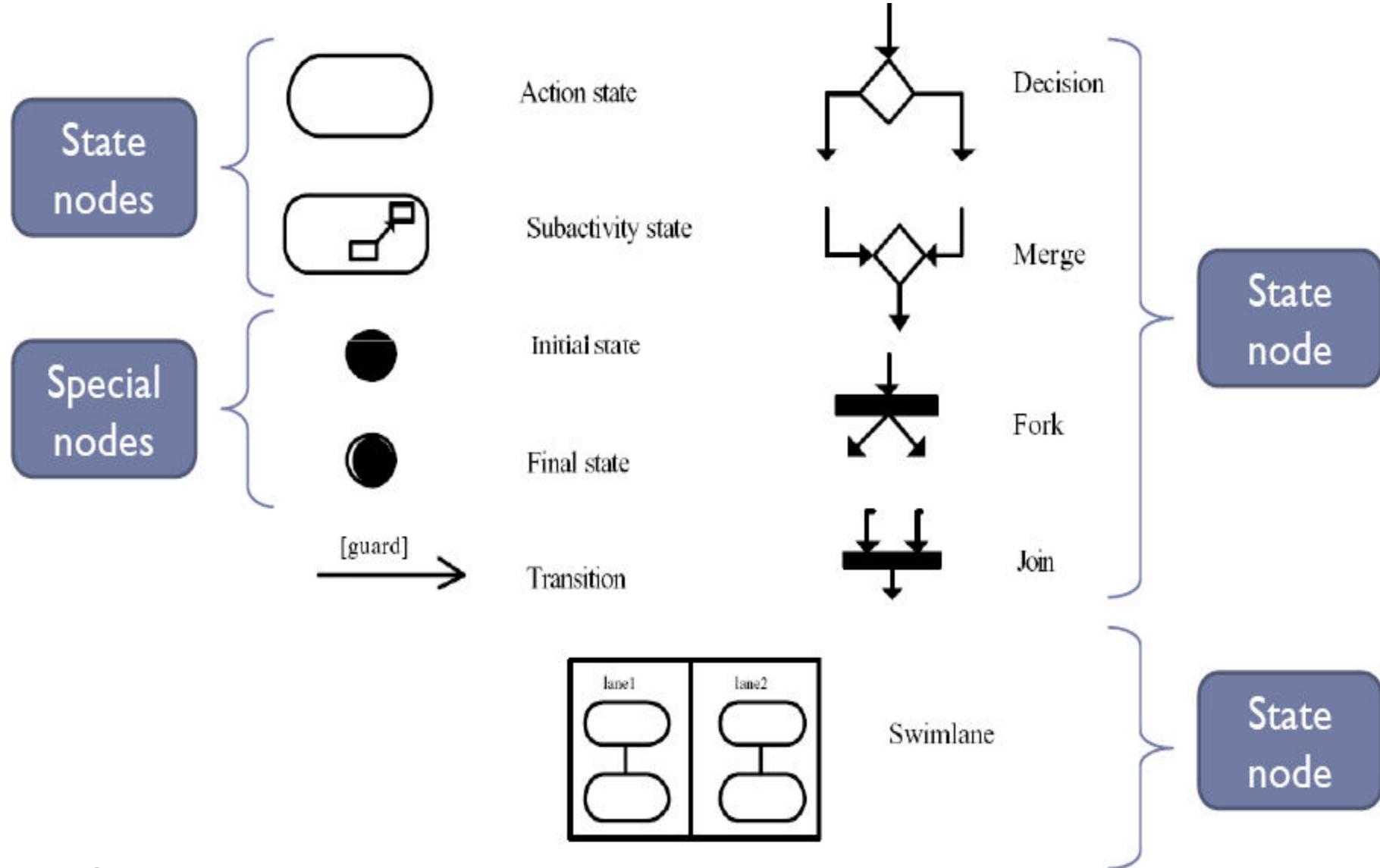


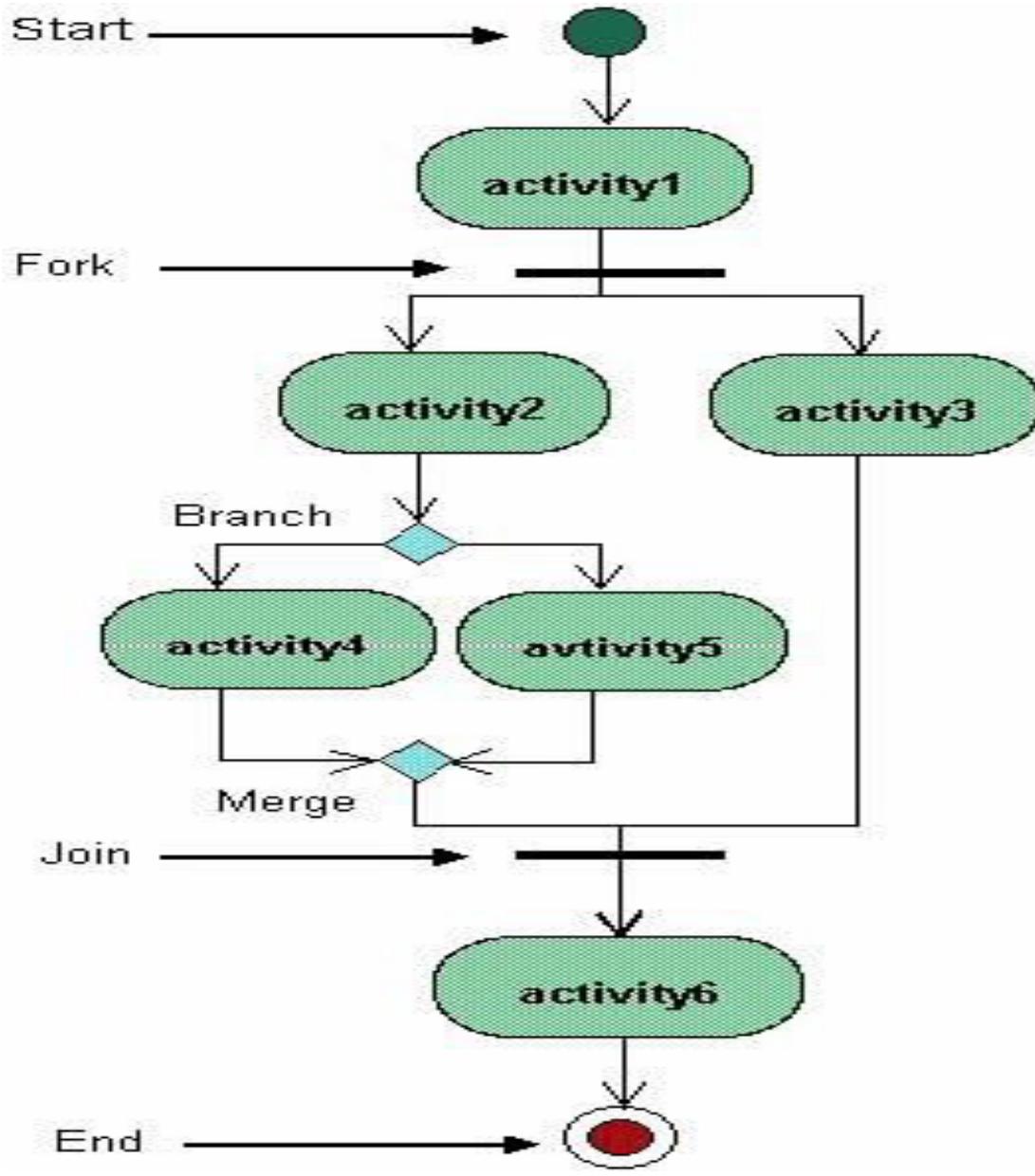
# Notation of Activity Diagrams

- **Swimlanes**
  - vertical columns
  - labelled with the person, organisation or department responsible for the activities in that column

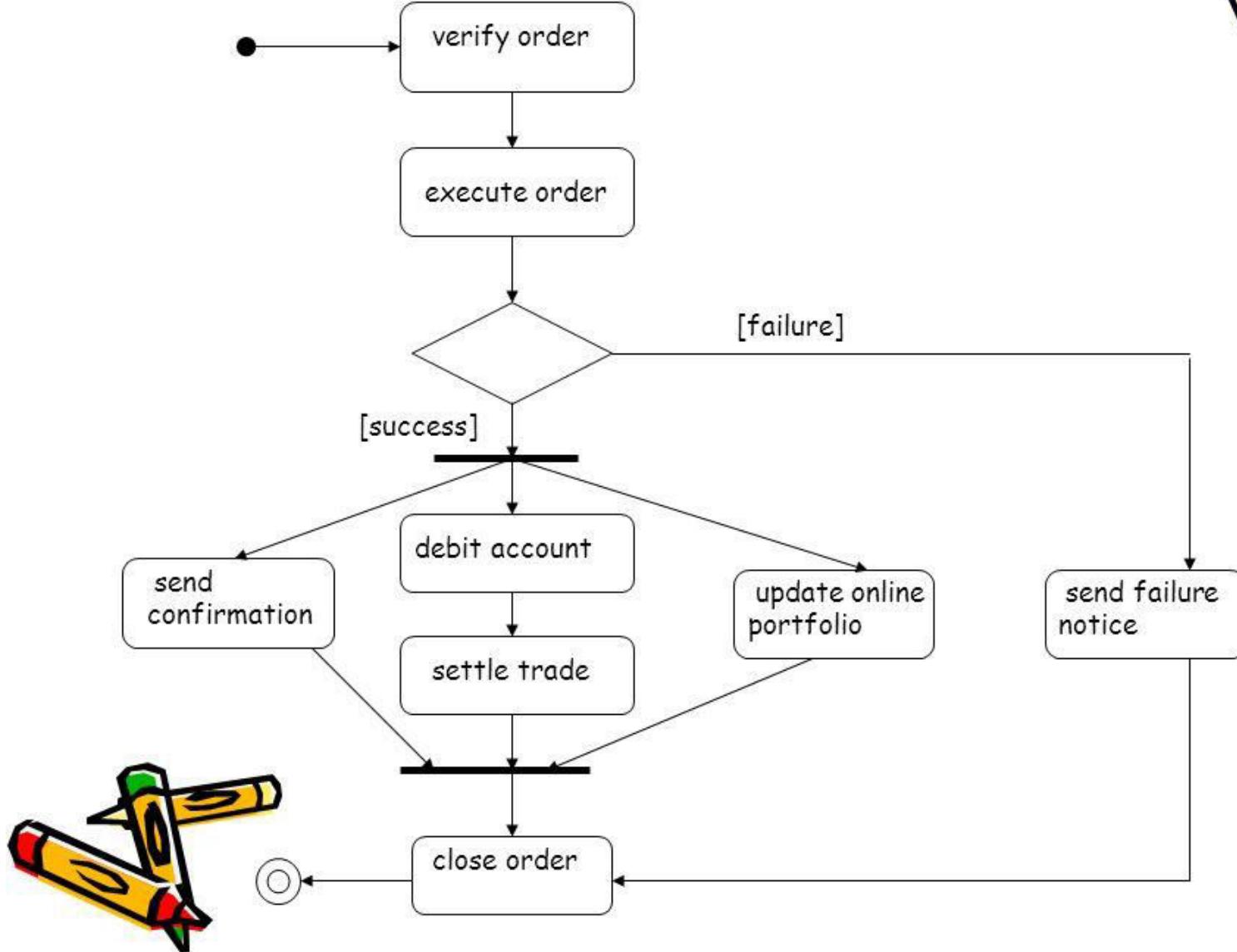


# Subset of structural elements of ADs





**Activity diagram for stock trade processing.** An activity diagram shows the sequence of steps that make up a complex process



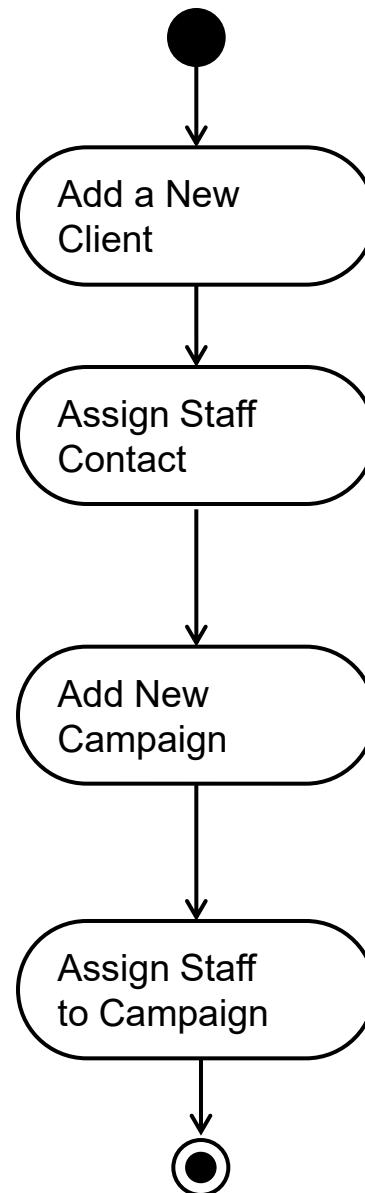
# Drawing Activity Diagrams

- What is the purpose?
  - This will influence the kind of activities that are shown
- What is being shown in the diagram?
  - What is the name of the business process, use case or operation?
- What level of detail is required?
  - Is it high level or more detailed?

# Drawing Activity Diagrams

- Identify activities
  - What happens when a new client is added in the Agate system?
    - Add a New Client
    - Assign Staff Contact
    - Add New Campaign
    - Assign Staff to Campaign
- Organise the activities in order with transitions

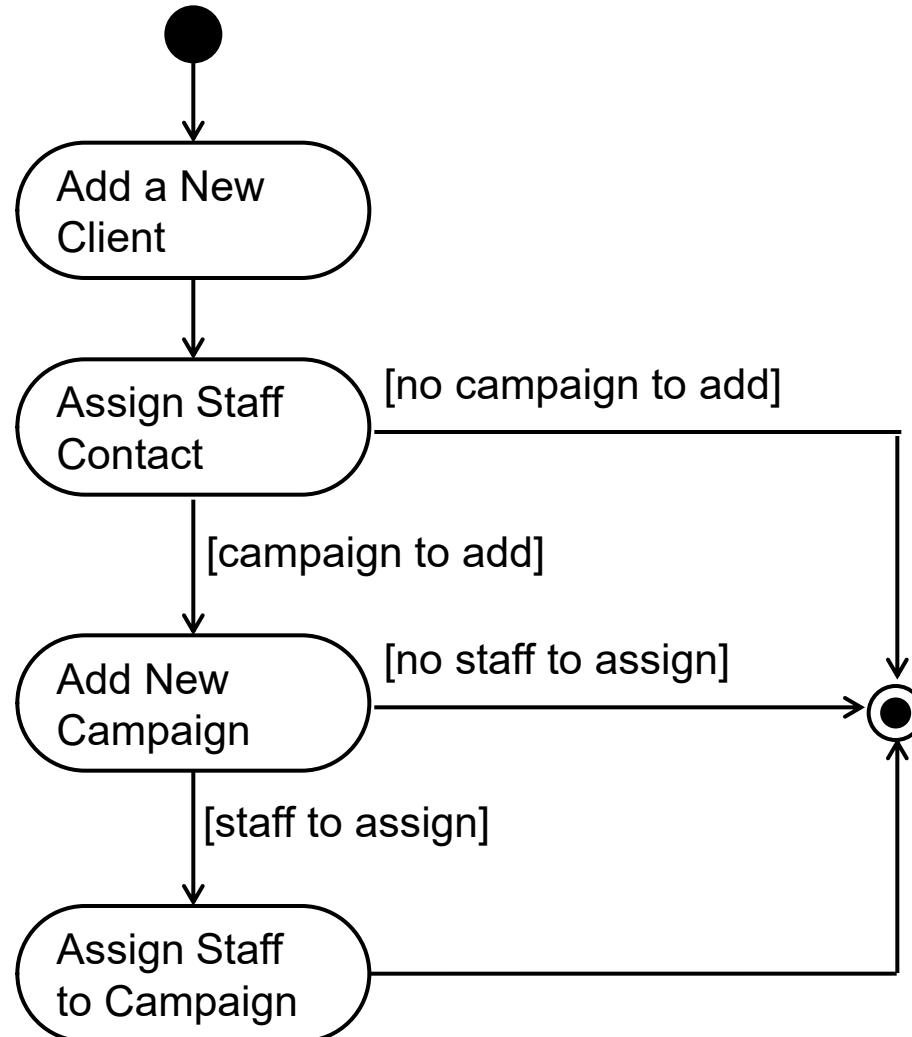
# Drawing Activity Diagrams



# Drawing Activity Diagrams

- Identify any alternative transitions and the conditions on them
  - sometimes there is a new campaign to add for a new client, sometimes not
  - sometimes they will want to assign staff to the campaign, sometimes not
- Add transitions and guard conditions to the diagram

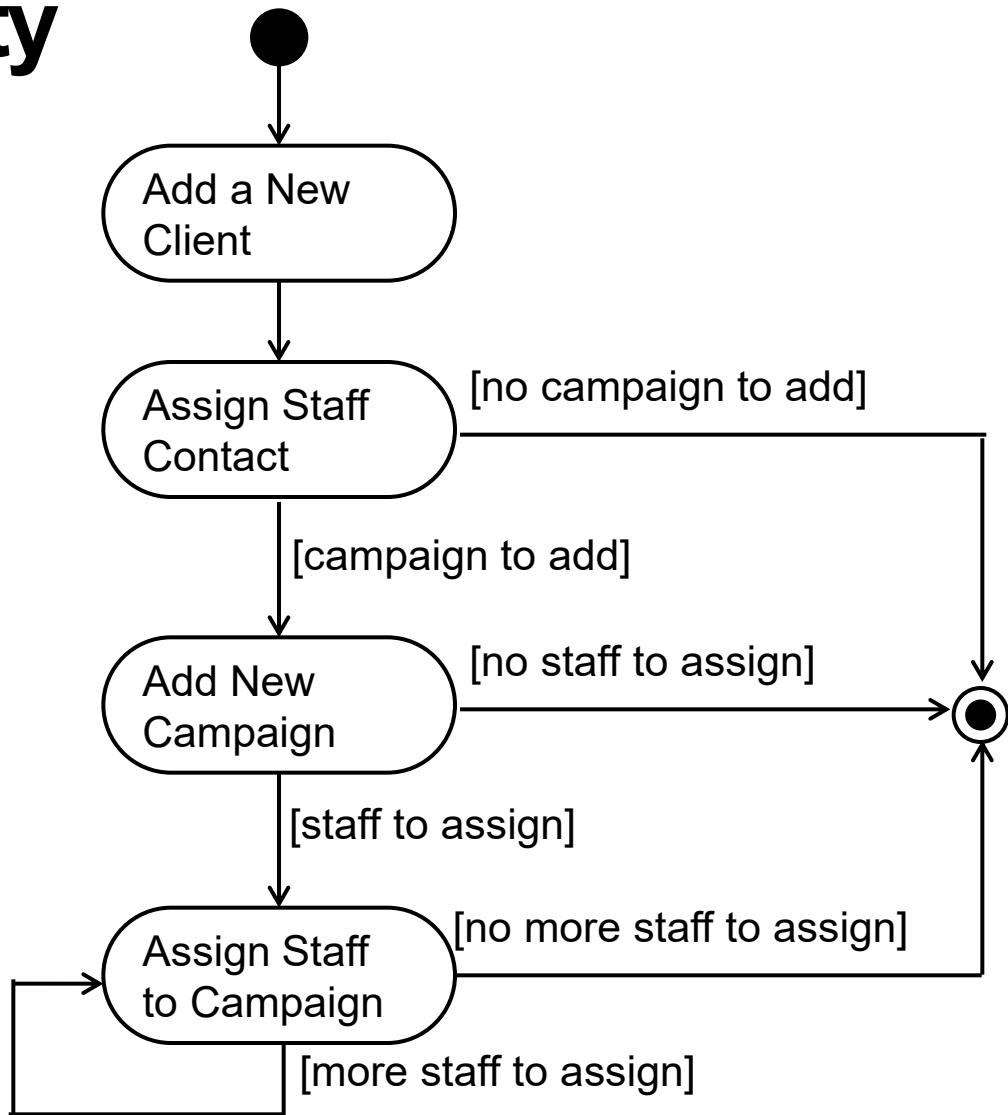
# Drawing Activity Diagrams



# Drawing Activity Diagrams

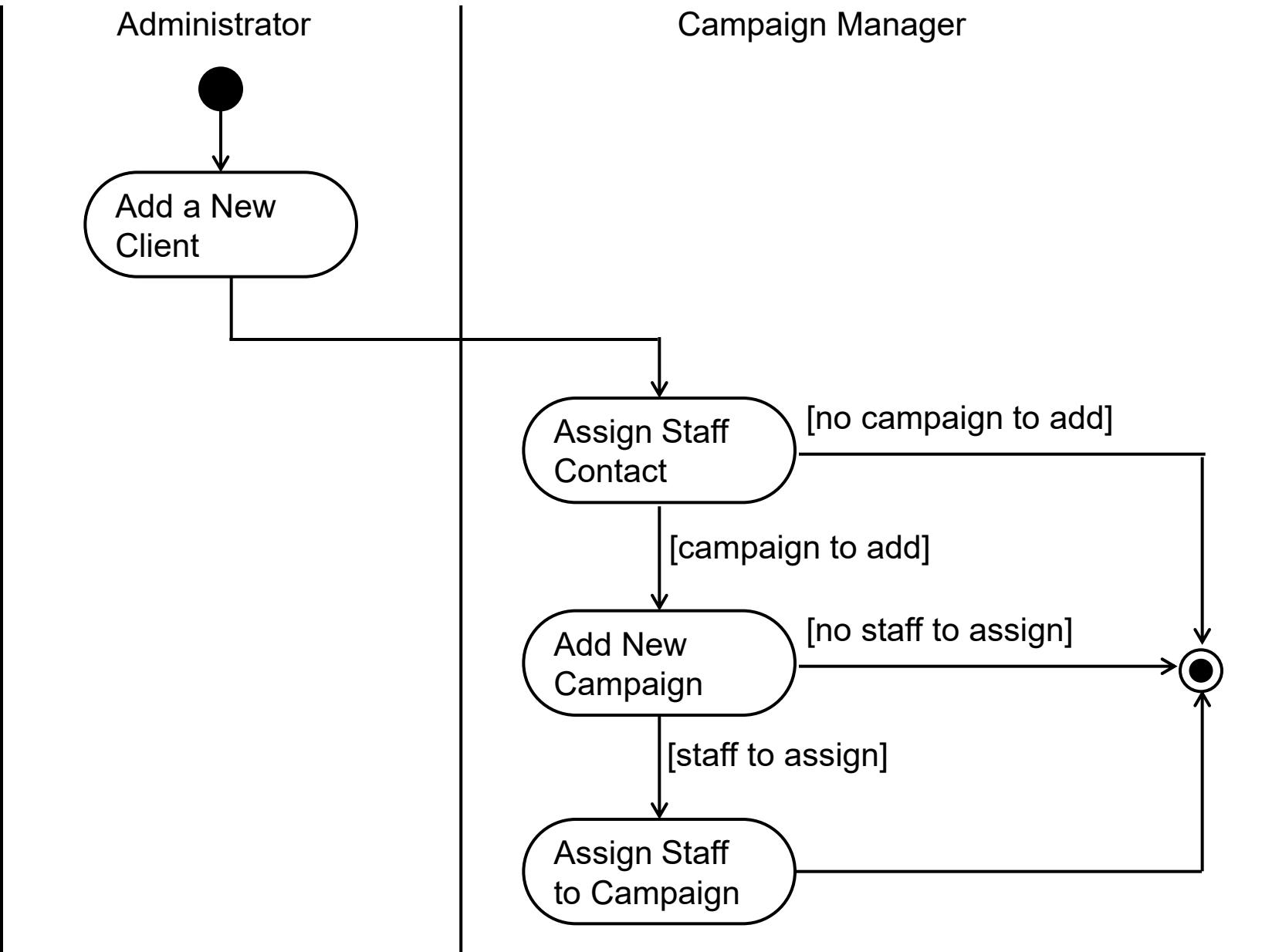
- Identify any processes that are repeated
  - they will want to assign staff to the campaign until there are no more staff to add
- Add transitions and guard conditions to the diagram

# Drawing Activity Diagrams

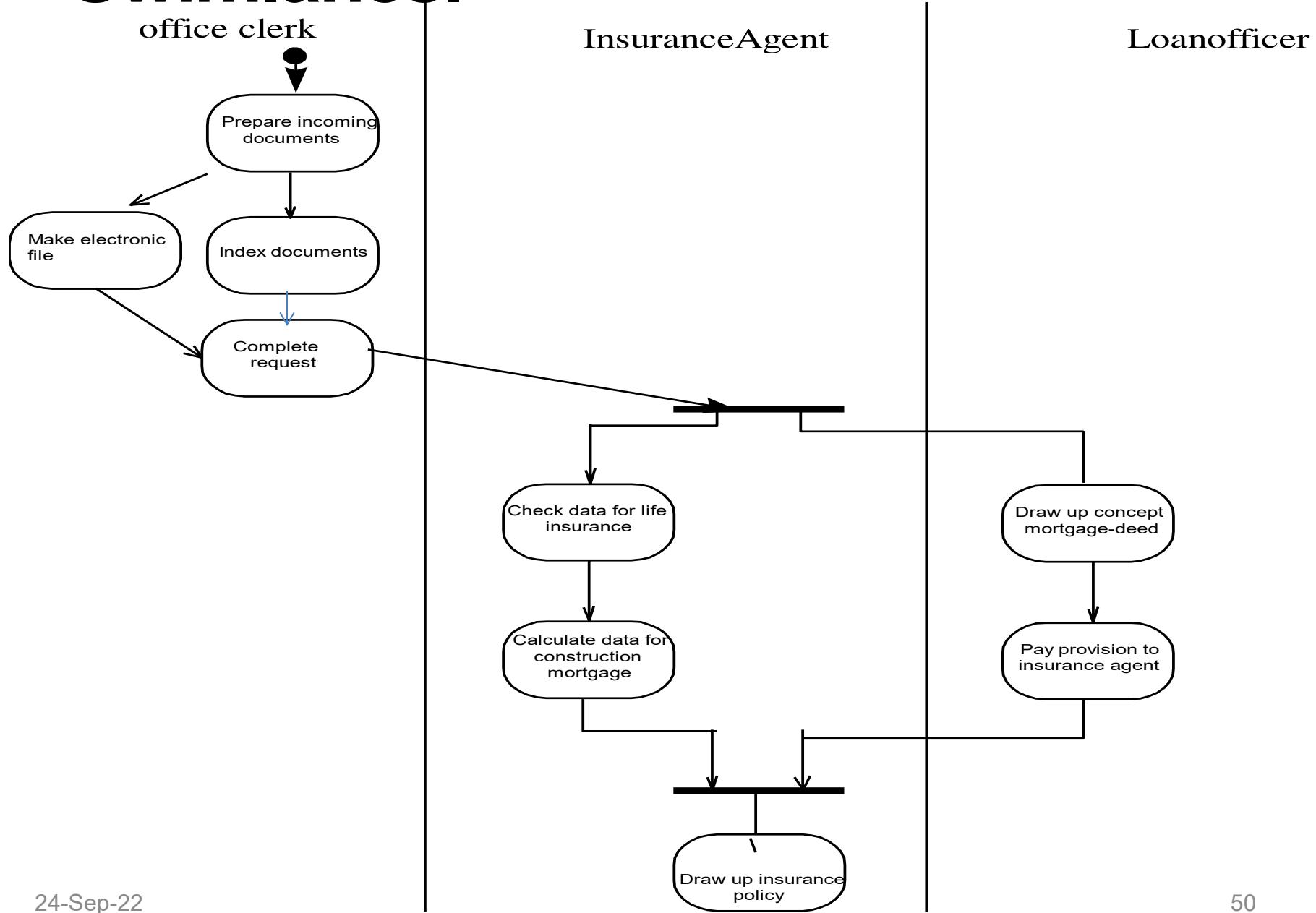


# Drawing Activity Diagrams

- Are all the activities carried out by the same person, organisation or department?
- If not, then add swimlanes to show the responsibilities
- Name the swimlanes
- Show each activity in the appropriate swimlane



# Swimlanes.

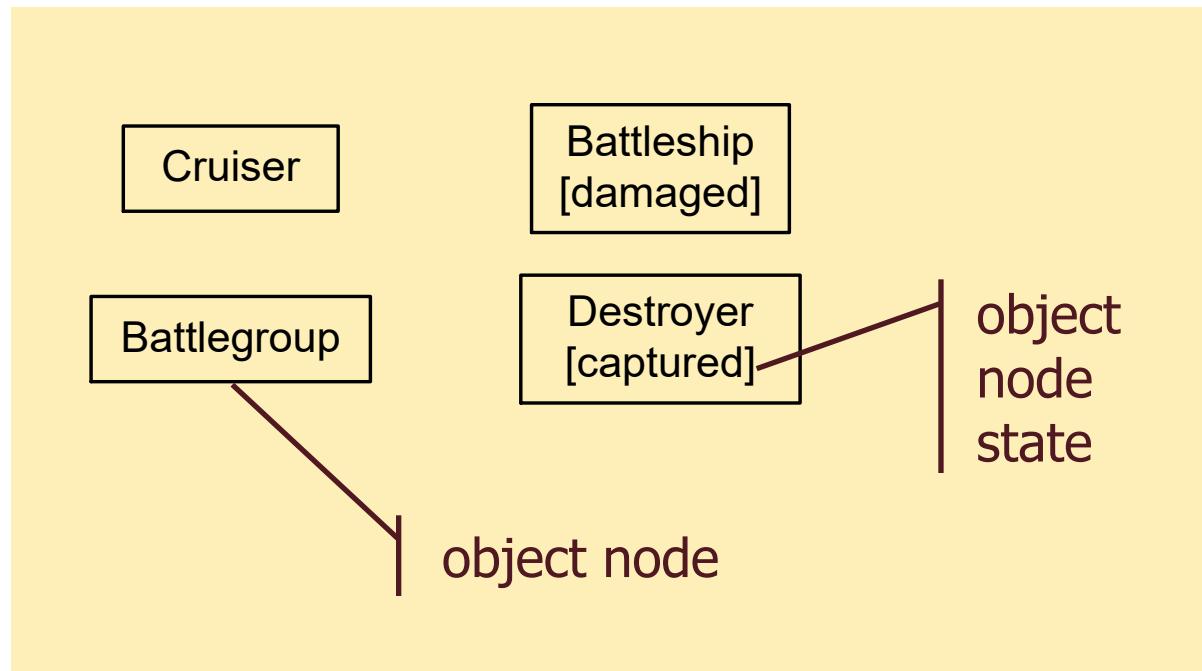


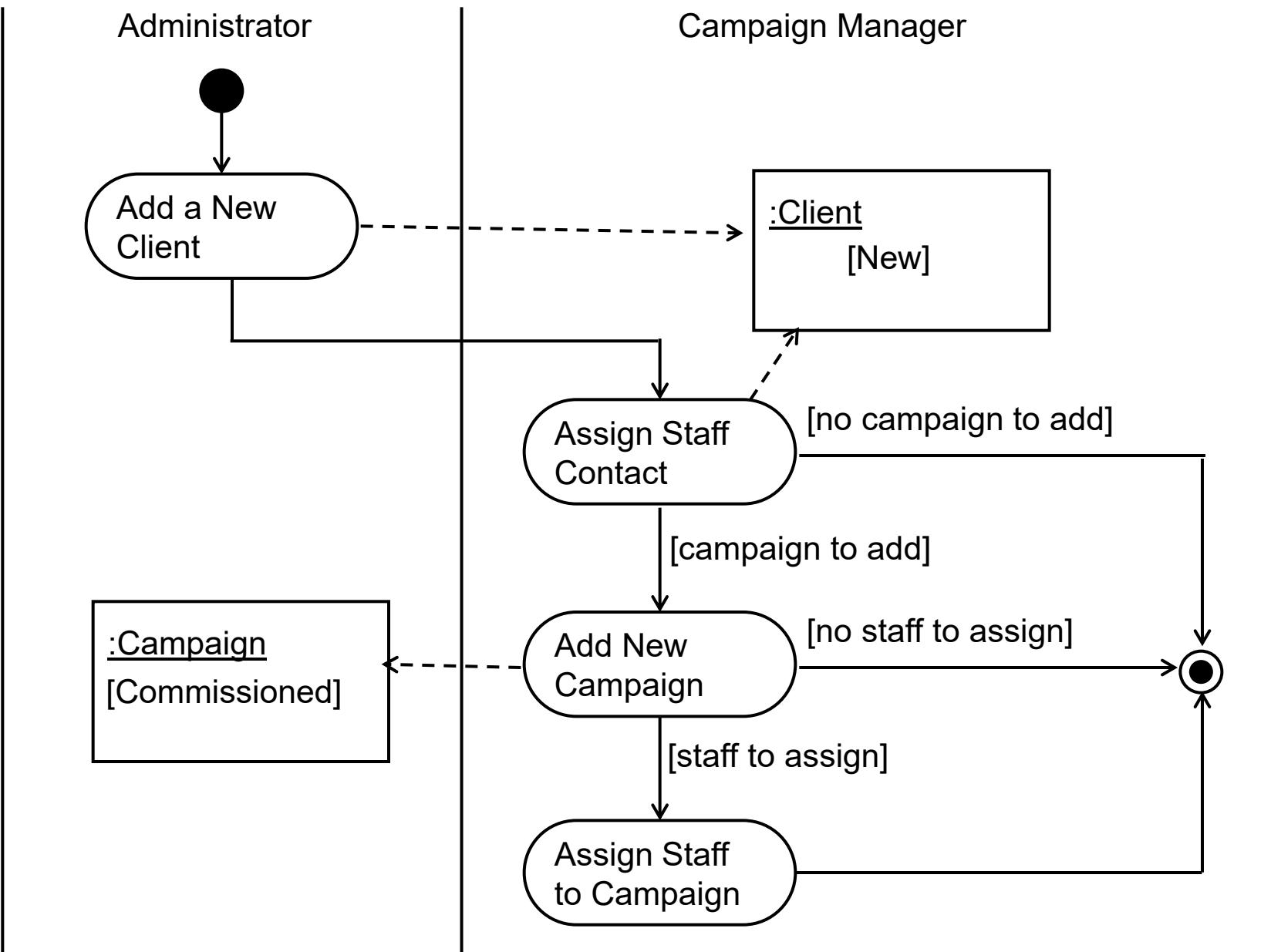
# Drawing Activity Diagrams

- Are there any object flows and objects to show?
  - these can be documents that are created or updated in a business activity diagram
  - these can be object instances that change state in an operation or a use case
- Add the object flows and objects

# Object Nodes

Data and objects are shown as object nodes.



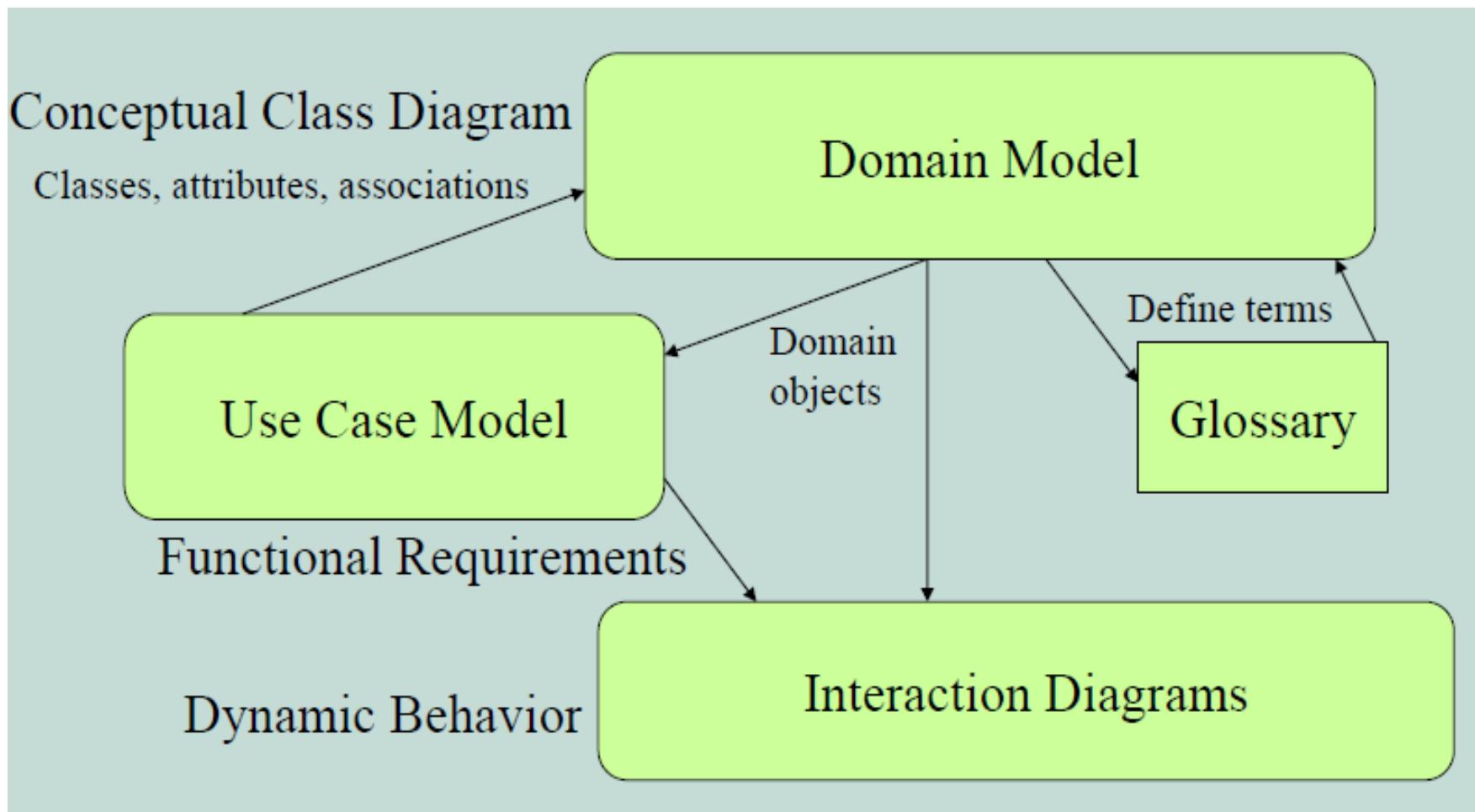


# Domain Model

# Domain model: What is it?

- Illustrates meaningful *concepts* in the *problem domain*.
- Usually expressed in the form of *static* diagrams (in Rational Rose this implies **a high-level class diagram**).
- Is a representation of *real-world* things; not software components (of the system under development).
- **No operations** are defined or specified in the domain model.
- The model shows concepts, associations between concepts, and attributes of concepts.
- Serves as a **source of designing software objects**.

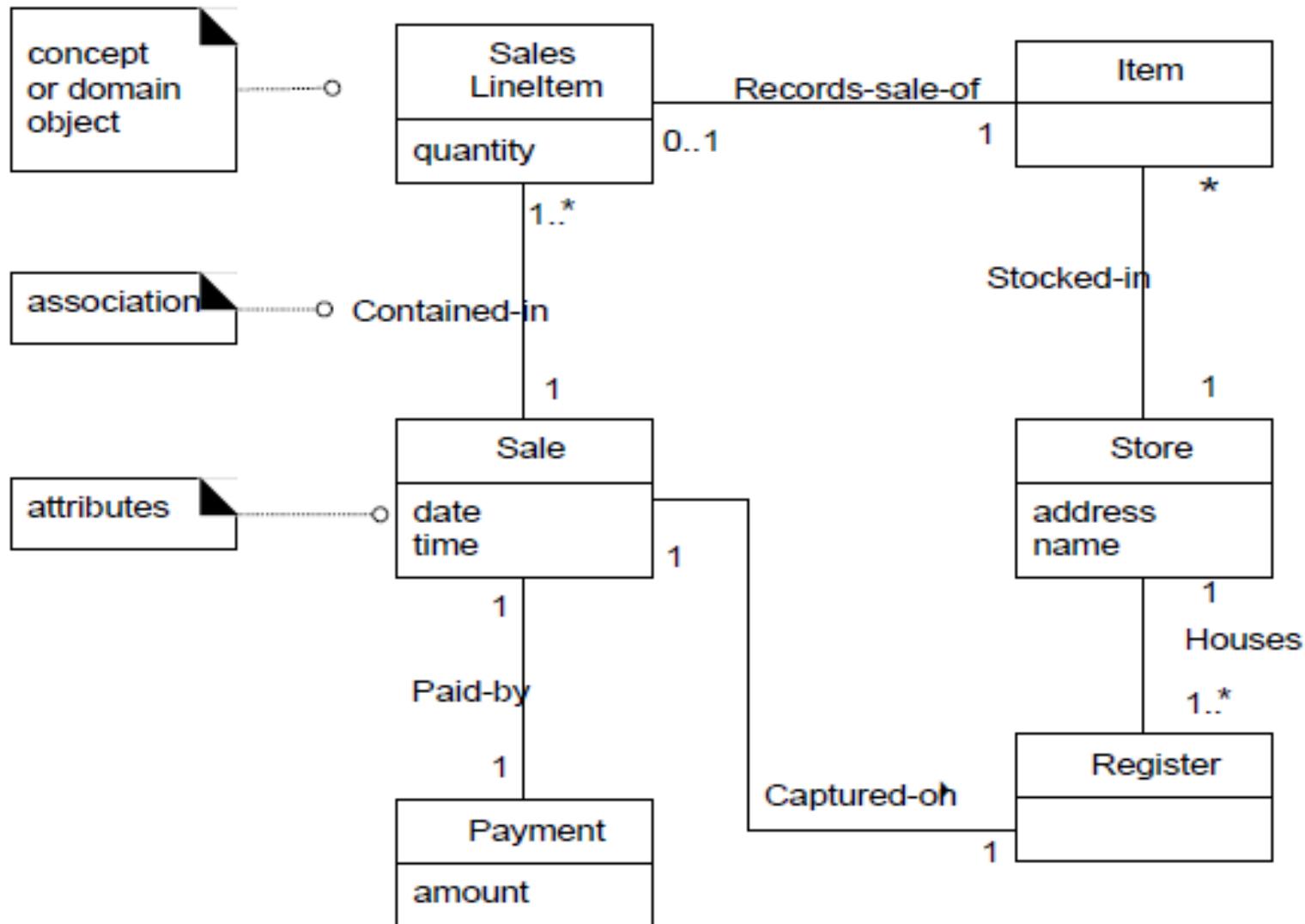
# Domain Model Relationships



# Why do a domain model?

- Gives a conceptual framework of the things in the problem space
- Helps you think – focus on concepts
- Provides a glossary of terms – noun based
- It is a static view - meaning it allows us convey time invariant business rules
- Further foundation for use case/workflow modeling
- Based on the defined structure, we can describe the state of the problem domain at any time.

# Partial Domain Model of POST



# A Domain Model is the most important OO artifact

- Its development entails identifying a rich set of conceptual classes, and is at the heart of object oriented analysis.
- It is a visual representation of the decomposition of a domain into individual conceptual classes or objects.
- It is a visual dictionary of noteworthy abstractions.

# Domain Models within in UP

- Domain Model is primarily created during elaboration iterations not in inception phase, when the need is highest to understand the noteworthy concepts and map some to software classes during design work.

# Decomposition:

- A central distinction between Object-oriented analysis and structured analysis is the division by objects rather than by functions during decomposition.
- During each iteration, only objects in current scenarios or different concepts are considered for addition to the domain model.

# Features of a domain model

- **Domain classes** – each domain class denotes a type of object.
- **Attributes** – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.
- **Associations** – an association is a relationship between two (or more) domain classes that describe links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.
- **Additional rules** – complex rules that cannot be shown symbolically can be shown with attached notes.

# Domain model: How to construct?

- Objectives
  - identify concepts (conceptual classes) related to current development cycle requirements
  - create initial conceptual (domain) model
    - Add Associations necessary to record relationships for which there is a need to preserve some memory
    - add the attributes necessary

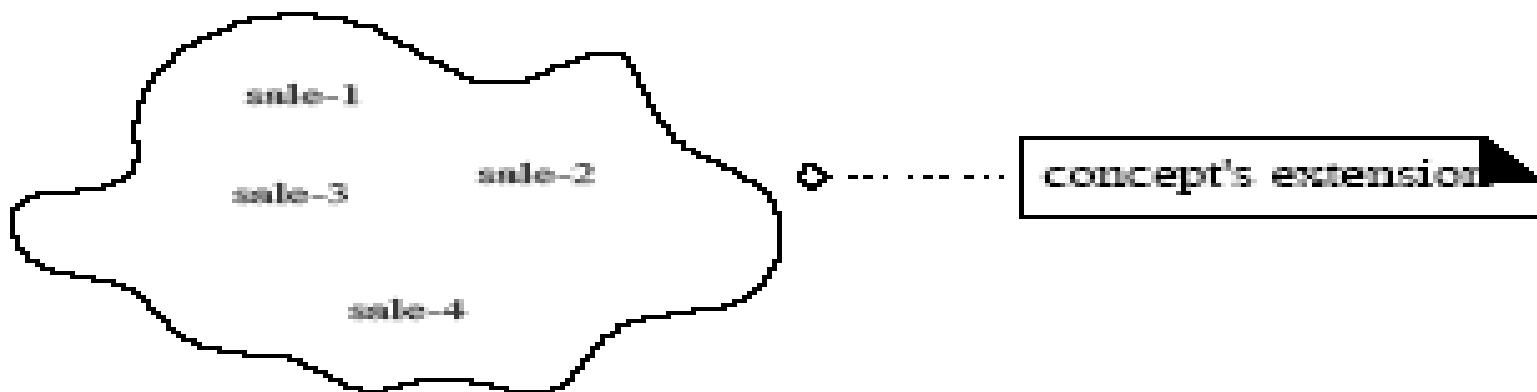
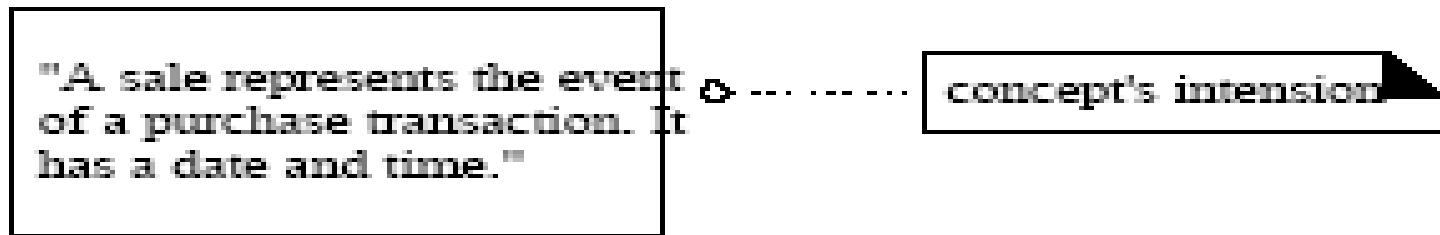
# Domain classes?

- Each domain class is an idea/thing/object. It is a descriptor for a set of things that share common features. Classes can be:-
  - ***Business objects*** - represent things that are manipulated in the business e.g. Order.
  - ***Real world objects*** – things that the business keeps track of e.g. Contact, Site.
  - ***Events that transpire/revealed*** - e.g. sale and payment.

# Think of Conceptual Classes in terms of:

- Symbol—words or images representing a conceptual class.
- Intension—the definition of a conceptual class.
- Extension—the set of examples to which the conceptual class applies.
- Symbols and Intensions are the practical considerations when creating a domain model.

# *Symbol - Intension - Extension*



# Conceptual Class Identification:

- It is better to overspecify a domain with lots of fine-grained conceptual classes than it is to underspecify it.
- Unlike data modeling, it is valid to include concepts for which there are no attributes, or which have a purely behavioral role rather than an informational role.

# Conceptual Class Identification:

- *Strategies to Identify Conceptual Classes:*
  - Reuse or modify existing domain model

There are many published, well-crafted domain models.  
Like Inventory, Finance, Healthcare etc.
  - Use a conceptual class category list

Make a list of all candidate conceptual classes
  - Identify noun phrases

Identify nouns phrases in textual descriptions of a domain ( use cases, or other documents)

# Finding concepts: Category List

- Finding concepts using the concept category list :
  - **Physical objects**: register, airplane, blood pressure monitor
  - **Places**: airport, hospital, Store
  - **Catalogs**: Product Catalog

# Conceptual Classes Category List

- Physical or tangible objects
  - Register, Airplane
- Specifications, or descriptions of things
  - ProductSpecification, FlightDescription
- Places
  - Store, Airport
- Transactions
  - Sale, Payment, Reservation
- Transaction items
  - SalesLineItem
- Roles of people
  - Cashier, Pilot
- Containers of other things
  - Store, Hangar, Airplane
- Things in a container
  - Item, Passenger
- Computer or electro mechanical systems
  - CreditPaymentAuthorizationSystem, AirTrafficControl
- Catalogs
  - ProductCatalog, PartsCatalog
- Organizations
  - SalesDepartment, Airline

## Where identify conceptual classes from noun phrases (NP)

- Vision and Scope, Glossary and Use Cases are good for this type of linguistic analysis
- However:
- Words may be ambiguous or synonymous
- Noun phrases may also be attributes or parameters rather than classes:
  - If it stores state information or it has multiple behaviors, then it's a class
  - If it's just a number or a string, then it's probably an attribute

# Finding concepts using Noun Phrases: refer to use cases

- Examine use case descriptions.
- Example: *Process Sale* use case:
  - Main success scenario:
    - *Customer* arrives at a *POS* checkout counter.
    - *Cashier* starts a new *sale*.
    - *Cashier* enters an *item* ID.
    - System records *sale line item*. It then presents a description of the *item*, its price, and a running total.
    - ....
    - ....
  - **Possible source of confusion:** Is it an *attribute* or a *concept*?
    - If X is not a number or a text then it probably is a *conceptual class*.

# From NPs to classes or attributes

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.

S C V R O O A O A

If it is, then the customer can check the account balance, deposit cash, and withdraw cash.

S R V O A V O A V O A

Checking the balance simply displays the account balance.

S M O A V O A

Depositing asks the customer to enter the amount, then updates the account balance.

S M V O R V O A V O A

Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,

S M O A V O R O A V S C V O A

the account balance is updated. The ATM prints the customer's account balance on a receipt.

O A V S C V O A O

Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? Then it's an actor, '**R**'.
- Is it also a verb (such as 'deposit')? Then it may be a method, '**M**'.
- Is it a simple value, such as 'color' (string) or 'money' (number)?  
Then it is probably an attribute, '**A**'.
- Which NPs are unmarked? Make it '**C**' for class.
- Verbs can also be classes, for example:  
**Deposit** is a class if it retains state information

# Finding concepts: Examples

- Are these concepts or attributes?
  - Store
  - Flight
  - Price
- Use terms familiar to those in the problem domain.  
POST or register?
- Concepts from “Unreal” world ?
- Example – Telecommunications (requires a high degree of abstraction)
  - Message, Connection
  - Port, Dialog, Route, Protocol

# Finding concepts: Examples

As an example, should *store* be an attribute of *Sale*, or a separate conceptual class *Store*?



or... ?

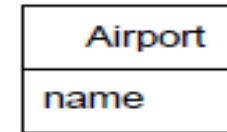
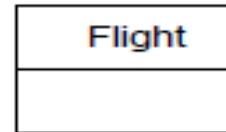


In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.

As another example, consider the domain of airline reservations. Should *destination* be an attribute of *Flight*, or a separate conceptual class *Airport*?



or... ?



In the real world, a destination airport is not considered a number or text—it is a massive thing that occupies space. Therefore, *Airport* should be a concept.

If in doubt, make it a separate concept. Attributes should be fairly rare in a domain model.

# ***What about Sales Receipt?***

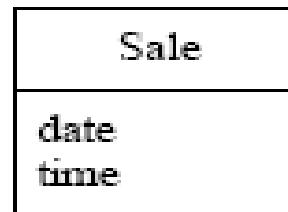
- Should it be included in the model?
  - It's common in the real world system
  - It's a sales report. Reports not explicitly stated in the use cases have little value in this model.
  - However, when a Customer wishes to return an item, it is an important object in the domain.
  - Since this development cycle doesn't include the Return Items use case, we'll leave it out of this CM

# Concepts in POST domain

- POST
- Item
- Sale
- Store
- Payment
- SalesLineItem
- Product Specification
- ProductCatalog
- Customer
- Cashier
- Manager

# *Do's and Don'ts in Conc. Model*

Do:



real-world concept,  
not a software class

avoid



software artifact; not  
part of conceptual model

Don't:

avoid



software class; not part  
of conceptual model



**BITS Pilani**  
Pilani Campus

# BITS Pilani presentation

Dr. Yashvardhan Sharma  
Computer Science and Information Systems





# **SS ZG514/SE Z512**

# **Object Oriented Analysis and Design**

## **Lecture No.6**

# Today's Agenda

---

- Object Oriented Analysis
- Ways to do Object Oriented Analysis
- Domain Model

# Understanding Analysis

- In software engineering, analysis is the process of converting the user requirements to system specification.
- System specification (logic structure) is the developer's view of the system.
- ***Function-oriented analysis*** – concentrating on the **decomposition** of complex functions to simple ones.
- ***Object-oriented analysis*** – **identifying objects** and the relationship between objects.

# Understanding Analysis

- ▶ **The goal in Analysis is to understand the problem**
  - ❖ and to begin to develop a visual model of what you are trying to build, independent of implementation and technology concerns.
- ▶ **Analysis focuses on translating the functional requirements into software concepts.**
  - ❖ The idea is to get a rough cut at the objects that comprise our system, but focusing on behavior (and therefore encapsulation).
  - ❖ We then move very quickly, nearly seamlessly into “design” and the other concerns.

# Analysis Versus Design

- ❖ Focus on understanding the problem
  - ❖ Idealized Behavior
  - ❖ System structure
  - ❖ Functional requirements
  - ❖ A small model
- ❖ Focus on understanding the solution
  - ❖ Operations and Attributes
  - ❖ Performance
  - ❖ Close to real code
  - ❖ Object lifecycles
  - ❖ Non-functional requirements
  - ❖ A large model

Analysis

Design

# Overview of Analysis

- Analysis is an iterative process... make a 'first cut' conceptual model and then iteratively refine it as your understanding of the problem increases.
- Brief Steps in the object analysis process
  - 1. Identifying the classes
  - 2. Identifying the classes attributes
  - 3. Identifying relationships and collaborations between classes

# Object Oriented Analysis

- ▶ Identifying objects:
  - ❖ Using concepts, CRC cards etc.
- ▶ Organising the objects:
  - ❖ classifying the objects identified, so similar objects can later be defined in the same class.
- ▶ Identifying relationships between objects:
  - ❖ this helps to determine inputs and outputs of an object.
- ▶ Defining objects internally:
  - ❖ information held within the objects.

# Three ways to do Object Oriented Analysis

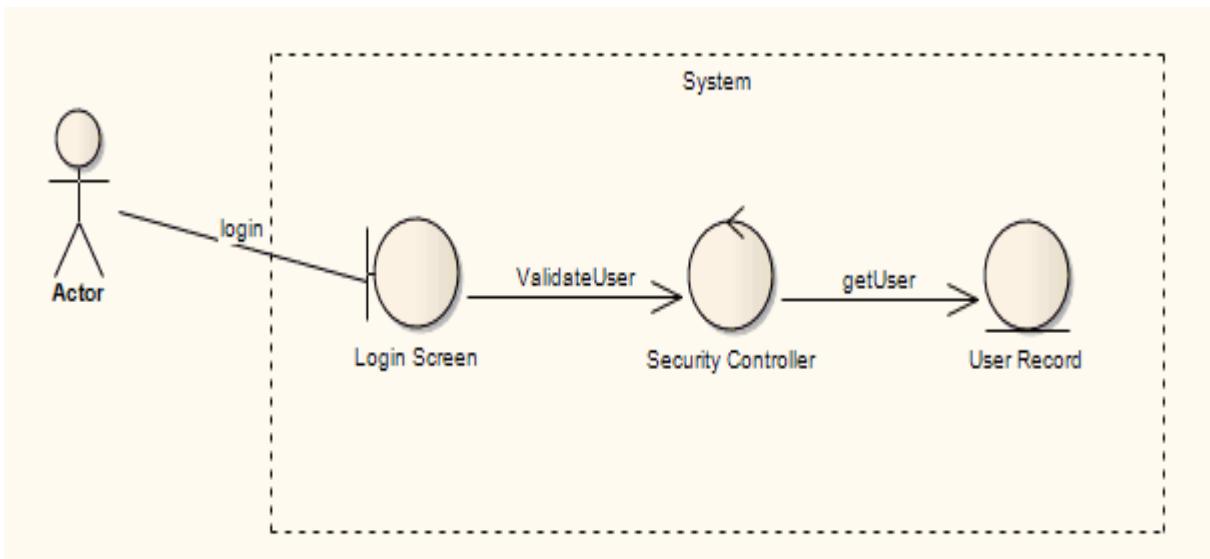
- Conceptual model (Larman)
  - ❖ Produce a “light” class diagram.
- CRC cards (Beck, Cunningham)
  - ❖ Index cards and role playing.

# Three ways to do Object Oriented Analysis (1)

- ▶ Analysis model with stereotypes (Jacobson).

Stereotypes used are:

- ❖ Boundaries (for a system boundary (for example, a Login screen))
- ❖ Entities (the element is a persistent or data element)
- ❖ Control ( to specify an element is a controller of some process).



# Object Identification

- Textual analysis of use-case information
  - Nouns suggest classes
- Creates a rough first cut
- Common object list
- Incidents
- Roles

# Case study: a resort reservation system

Design the software to handle reservations at Blue Lake Resort. The resort comprises several cottages and two meeting rooms. Cottages can comprise from one to three beds. The first meeting room has a capacity of 20 persons, the second, 40 persons. Cottages can be booked by the night; meeting rooms can be booked by the hour. Rates for cottages are expressed per person, per night; rates for meeting rooms are expressed per hour.

# Case study: a resort reservation system

Clients can book cottages and meeting rooms in advance by providing a phone number and a valid credit card information. Customers can express preferences for specific cottages. Cancellation of reservations is possible but requires 24 hours notice. An administrative charge applies to all cancellations. Every morning, a summary of the bookings for the previous day is printed out and the related information is erased from the computer; a list of the cottages and meeting rooms that will require cleaning is printed.

# Identification of Classes[1]

- Step 1: identify candidate classes
  - *Using nouns*: extract objects from the nouns in the requirements
  - *Data flow*: start with inputs and determine what objects are needed to transform the inputs to the outputs

# Identification of Classes[2]

- *Using the things to be modeled:* identify individual or group things such as persons, roles, organizations, logs, reports... in the application domain that is to be modeled; map to corresponding classes.

# Identification of Classes[3]

- *Using previous experience*: object-oriented domain analysis, application framework, class hierarchies, and personal experience

# Noun identification

- Identify candidate classes by extracting all nouns out of the requirements/specifications of the system.
- Don't be too selective at first; write down every class that comes to mind. A selection process will follow.

# Noun identification: Case Study

Design the software to handle reservations at *Blue Lake Resort*. The resort comprises several cottages and two meeting rooms. Cottages can comprise from one to three beds. The first meeting room has a capacity of 20 persons, the second, 40 persons. Cottages can be booked by the night; meeting rooms can be booked by the hour. Rates for cottages are expressed per person, per night; rates for meeting rooms are expressed per hour.

# Noun identification: Case Study

Clients can book cottages and meeting rooms in advance by providing a phone number and a valid credit card information. Customers can express preferences for specific cottages. Cancellation of reservations is possible but requires 24 hours notice. An administrative charge applies to all cancellations. Every morning, a summary of the bookings for the previous day is printed out and the related information is erased from the computer; a list of the cottages and meeting rooms that will require cleaning is printed.

# Keeping the right candidate classes

Having made a tentative list of classes, eliminate unnecessary or incorrect classes according to the following criteria:

- **Redundant classes**: nouns that express the same information
- **Irrelevant classes**: nouns that have nothing to do with the problem
- **Vague classes**: nouns with ill-defined boundaries or which are too broad
- **Attributes**: nouns referring to something simple with no interesting behavior, which is simply an attribute of another class
- **Events or operations**: nouns referring to something done to the system (sometimes events or operations are to be kept: check if they have state, behavior and identity).
- **Outside the scope of the system**: nouns that do not refer to something inside the system

# Refined list: Case Study

- **Good classes:** *Cottage, Meeting room, Client*
- **Bad classes:** *Software, Blue Lake Resort, Bed, Capacity, Night, Hour, Rate, Phone number, Credit card information, Customer, Preference, Cancellation, Notice, Administrative charge, Morning, Day, Information, Computer*
- **Don't know:** *Reservation, Resort, Summary, List*

# More Class Identification Tips

- Adjectives can suggest different kinds of objects or different use of the same object. If the adjectives indicate different behavior then make a new class.
- Be wary of sentences in the passive voice, or those whose subjects are not part of the system. Recast the sentence in the active voice and see if it uncovers other classes.  
e.g.: "a summary of the bookings for the previous day is printed out". Missing from this sentence is the subject or who is doing the printing.

# More Class Identification Tips

- Keeping two lists (strong candidates and weaker ones) is a useful technique to avoid losing information while still sorting between things you are sure about and things that have yet to be settled.  
Additional techniques in Step 2 (identifying responsibilities and attributes) and Step 3(identifying relationships and attributes) will help.

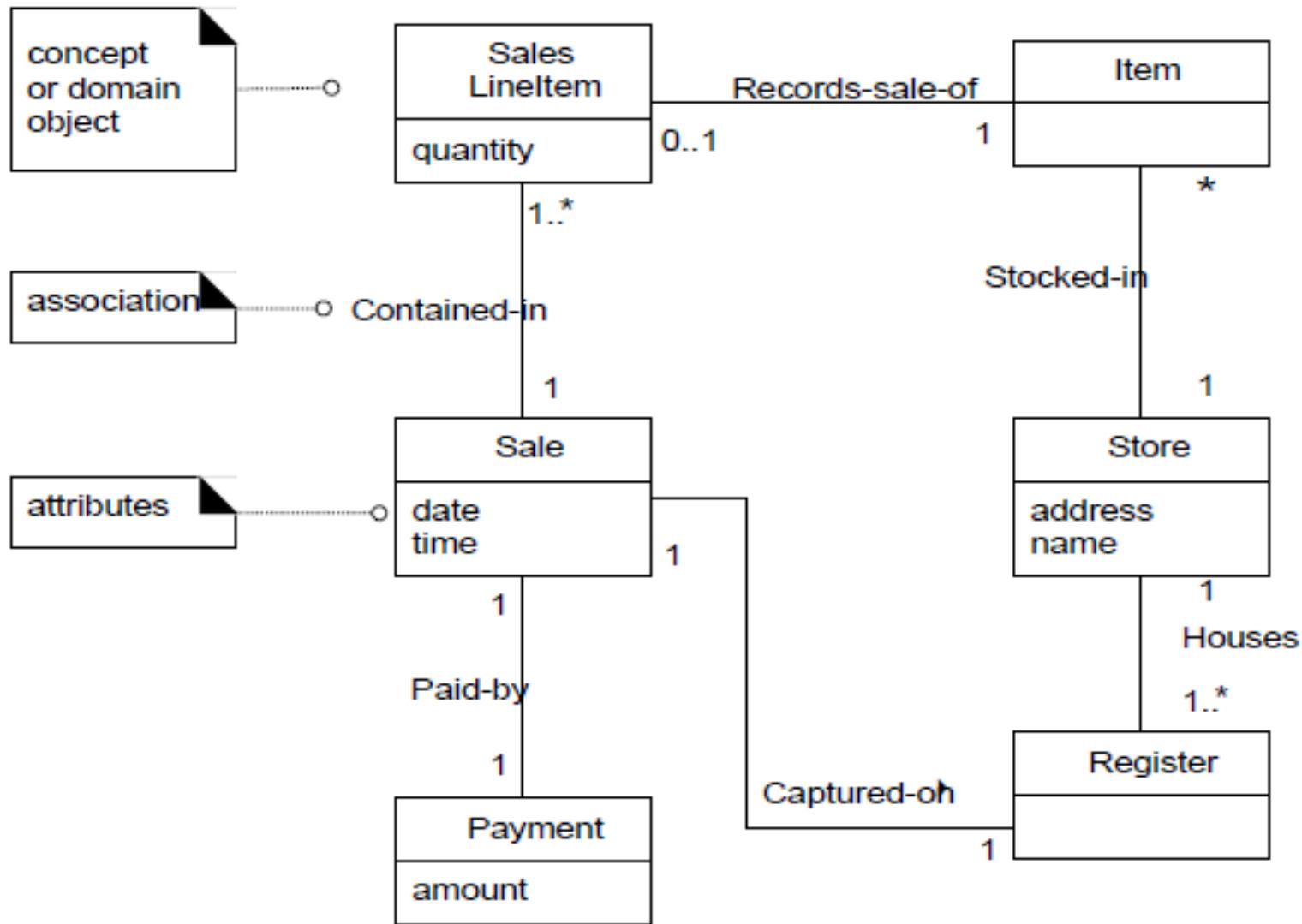
# ATM Case Study

- Design the software to support a computerized banking network including both human cashiers and automatic teller machine (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicated directly with their own bank's computers. Human cashiers enter account and transaction data.

# ATM Case Study (Cont.)

- Automatic teller machines communicate with a central computer that clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate recordkeeping and security provisions. The system must handle concurrent accesses to the same account correctly.
- The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

# Partial Domain Model of POST



# A Domain Model is the most important OO artifact

- Its development entails identifying a rich set of conceptual classes, and is at the heart of object oriented analysis.
- It is a visual representation of the decomposition of a domain into individual conceptual classes or objects.
- It is a visual dictionary of noteworthy abstractions.

# Domain Models within in UP

- Domain Model is primarily created during elaboration iterations not in inception phase, when the need is highest to understand the noteworthy concepts and map some to software classes during design work.

# Decomposition:

- A central distinction between Object-oriented analysis and structured analysis is the division by objects rather than by functions during decomposition.
- During each iteration, only objects in current scenarios or different concepts are considered for addition to the domain model.

# Features of a domain model

- **Domain classes** – each domain class denotes a type of object.
- **Attributes** – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.
- **Associations** – an association is a relationship between two (or more) domain classes that describe links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.
- **Additional rules** – complex rules that cannot be shown symbolically can be shown with attached notes.

# Domain model: How to construct?

- Objectives
  - identify concepts (conceptual classes) related to current development cycle requirements
  - create initial conceptual (domain) model
    - Add Associations necessary to record relationships for which there is a need to preserve some memory
    - add the attributes necessary

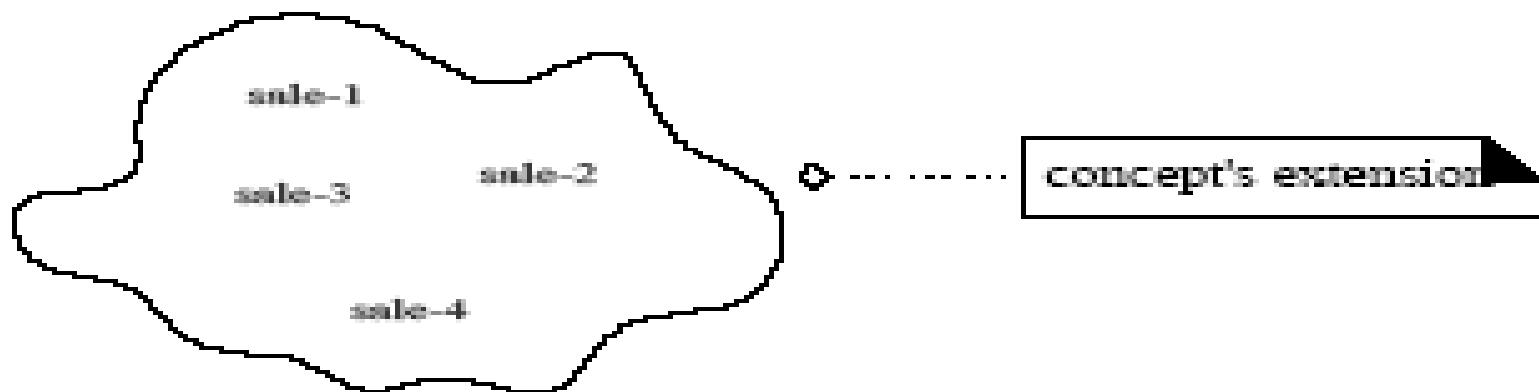
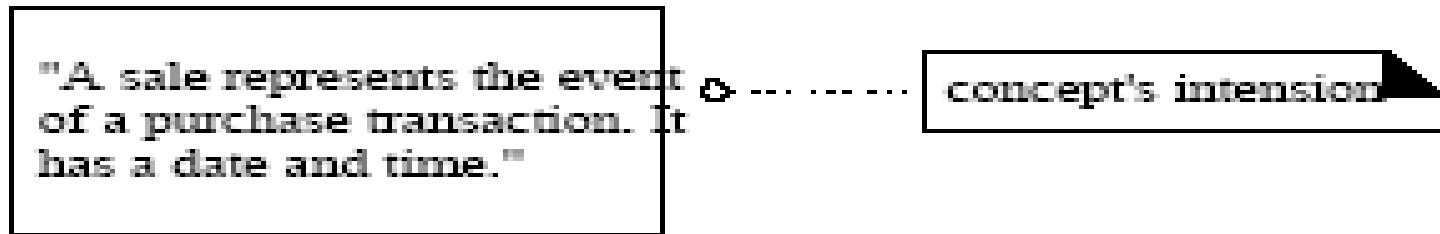
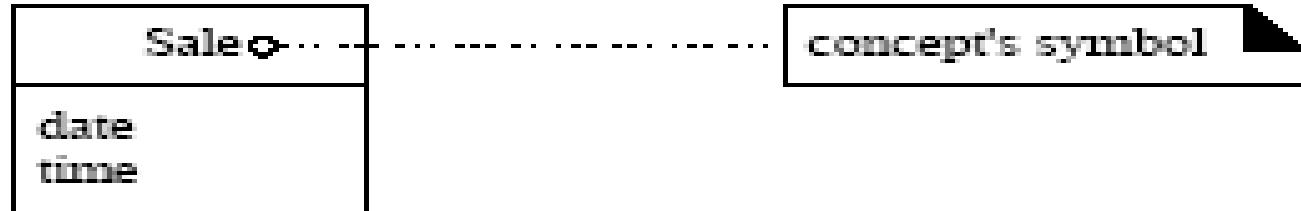
# Domain classes?

- Each domain class is an idea/thing/object. It is a descriptor for a set of things that share common features. Classes can be:-
  - ***Business objects*** - represent things that are manipulated in the business e.g. Order.
  - ***Real world objects*** – things that the business keeps track of e.g. Contact, Site.
  - ***Events that transpire/revealed*** - e.g. sale and payment.

# Think of Conceptual Classes in terms of:

- Symbol—words or images representing a conceptual class.
- Intension—the definition of a conceptual class.
- Extension—the set of examples to which the conceptual class applies.
- Symbols and Intensions are the practical considerations when creating a domain model.

# *Symbol - Intension - Extension*



# Conceptual Class Identification:

- It is better to overspecify a domain with lots of fine-grained conceptual classes than it is to underspecify it.
- Unlike data modeling, it is valid to include concepts for which there are no attributes, or which have a purely behavioral role rather than an informational role.

# Conceptual Class Identification:

- *Strategies to Identify Conceptual Classes:*
  - Reuse or modify existing domain model

There are many published, well-crafted domain models.  
Like Inventory, Finance, Healthcare etc.
  - Use a conceptual class category list

Make a list of all candidate conceptual classes
  - Identify noun phrases

Identify nouns phrases in textual descriptions of a domain ( use cases, or other documents)

# Finding concepts: Category List

- Finding concepts using the concept category list :
  - **Physical objects**: register, airplane, blood pressure monitor
  - **Places**: airport, hospital, Store
  - **Catalogs**: Product Catalog

# Conceptual Classes Category List

- Physical or tangible objects
  - Register, Airplane
- Specifications, or descriptions of things
  - ProductSpecification, FlightDescription
- Places
  - Store, Airport
- Transactions
  - Sale, Payment, Reservation
- Transaction items
  - SalesLineItem
- Roles of people
  - Cashier, Pilot
- Containers of other things
  - Store, Hangar, Airplane
- Things in a container
  - Item, Passenger
- Computer or electro mechanical systems
  - CreditPaymentAuthorizationSystem, AirTrafficControl
- Catalogs
  - ProductCatalog, PartsCatalog
- Organizations
  - SalesDepartment, Airline

# Where identify conceptual classes from noun phrases (NP)

- Vision and Scope, Glossary and Use Cases are good for this type of linguistic analysis
- However:
- Words may be ambiguous or synonymous
- Noun phrases may also be attributes or parameters rather than classes:
  - If it stores state information or it has multiple behaviors, then it's a class
  - If it's just a number or a string, then it's probably an attribute

# Finding concepts using Noun Phrases: refer to use cases

- Examine use case descriptions.
- Example: *Process Sale* use case:
  - Main success scenario:
    - *Customer* arrives at a *POS* checkout counter.
    - *Cashier* starts a new *sale*.
    - *Cashier* enters an *item* ID.
    - System records *sale line item*. It then presents a description of the *item*, its price, and a running total.
    - ....
    - ....
  - **Possible source of confusion:** Is it an *attribute* or a *concept*?
    - If X is not a number or a text then it probably is a *conceptual class*.

# From NPs to classes or attributes

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.

S C V R O O A O A

If it is, then the customer can check the account balance, deposit cash, and withdraw cash.

S R V O A V O A V O A

Checking the balance simply displays the account balance.

S M O A V O A

Depositing asks the customer to enter the amount, then updates the account balance.

S M V O R V O A V O A

Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,

S M O A V O R O A V S C V O A

the account balance is updated. The ATM prints the customer's account balance on a receipt.

O A V S C V O A O

Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? Then it's an actor, '**R**'.
- Is it also a verb (such as 'deposit')? Then it may be a method, '**M**'.
- Is it a simple value, such as 'color' (string) or 'money' (number)?  
Then it is probably an attribute, '**A**'.
- Which NPs are unmarked? Make it '**C**' for class.
- Verbs can also be classes, for example:  
**Deposit** is a class if it retains state information

# Finding concepts: Examples

- Are these concepts or attributes?
  - Store
  - Flight
  - Price
- Use terms familiar to those in the problem domain.  
POST or register?
- Concepts from “Unreal” world ?
- Example – Telecommunications (requires a high degree of abstraction)
  - Message, Connection
  - Port, Dialog, Route, Protocol

# Finding concepts: Examples

As an example, should *store* be an attribute of *Sale*, or a separate conceptual class *Store*?



or... ?

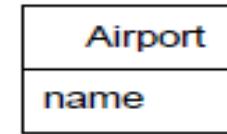
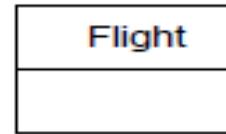


In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.

As another example, consider the domain of airline reservations. Should *destination* be an attribute of *Flight*, or a separate conceptual class *Airport*?



or... ?



In the real world, a destination airport is not considered a number or text—it is a massive thing that occupies space. Therefore, *Airport* should be a concept.

If in doubt, make it a separate concept. Attributes should be fairly rare in a domain model.

# ***What about Sales Receipt?***

- Should it be included in the model?
  - It's common in the real world system
  - It's a sales report. Reports not explicitly stated in the use cases have little value in this model.
  - However, when a Customer wishes to return an item, it is an important object in the domain.
  - Since this development cycle doesn't include the Return Items use case, we'll leave it out of this CM

# Concepts in POST domain

- POST
- Item
- Sale
- Store
- Payment
- SalesLineItem
- Product Specification
- ProductCatalog
- Customer
- Cashier
- Manager

# *Do's and Don'ts in Conc. Model*

Do:



real-world concept,  
not a software class

avoid



software artifact; not  
part of conceptual model

Don't:

avoid



software class; not part  
of conceptual model

# ***Specification Concepts/Conceptual Classes***

- When are they needed?
  - Add a specification concept when:
    - deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained.
    - it reduces redundant or duplicated information

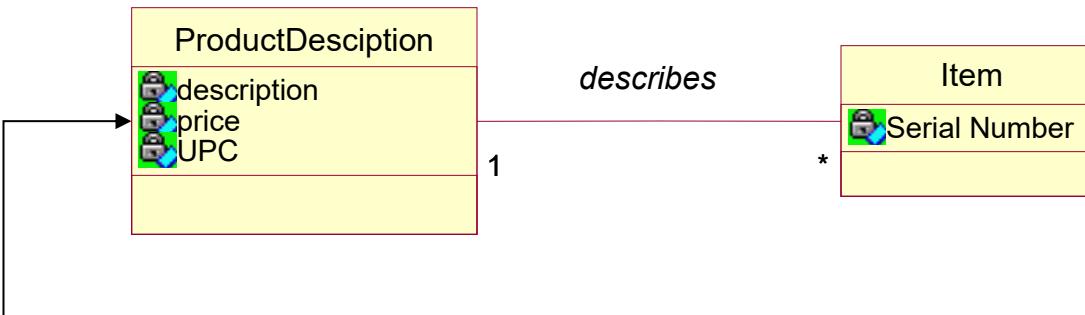
# Specification Example

- Assume that
  - an *item* instance represents a physical *item* in the store; it has a serial number
  - an *item* has a description, price and itemID which are not recorded anywhere else.
  - every time a real physical *item* is sold, a corresponding software instance of *item* is deleted from the database
- With these assumptions, what happens when the store sells out of a specific *item* like “burgers”? **How does one find out how much does the “burger” cost ?**
- Notice that the price is stored with the inventoried instances

# Specification Example – Contd.

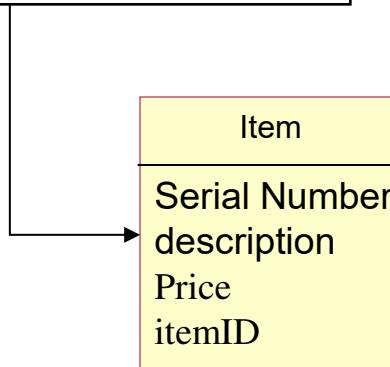
- Also notice the data is duplicated many times with each instance of the *item*.
- This example illustrates the need for a concept of objects that are specifications or descriptions of other things (often called a *Proxy or Surrogate*)
- Description or specification objects are strongly related to the things they describe.

# Specification - Example



Which of these two  
is a better choice of concepts?

XSpecification describes X



# Conceptual Models - Association

- Objective
  - Identify associations within a conceptual model
  - distinguish between need-to-know associations from comprehension-only associations

# Associations

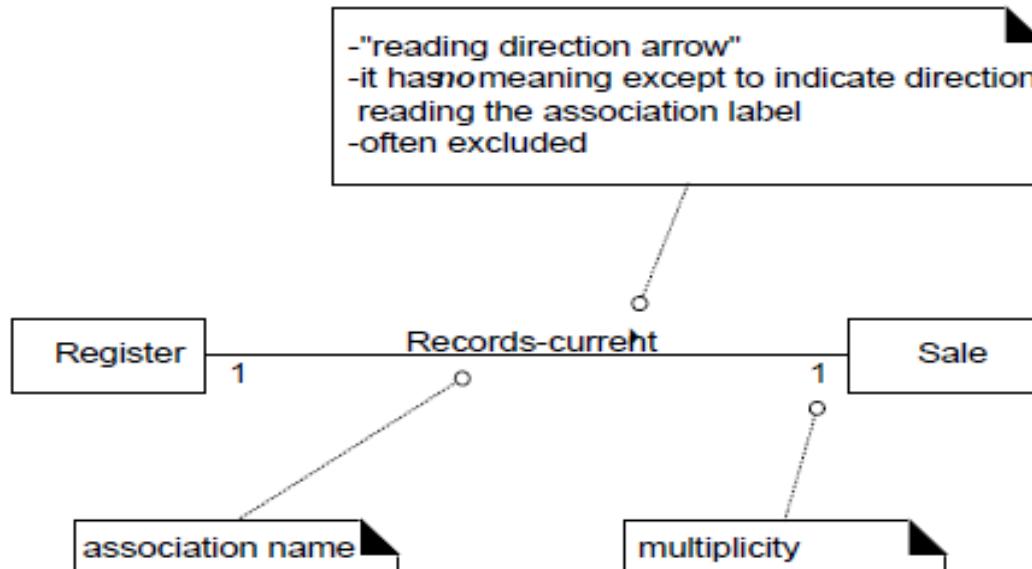
- Associations imply knowledge of a relationship that needs to be preserved over time
  - could be milliseconds or years!
  - such associations are *need-to-know* assocs.
- Associations derived from the Common Associations List (Pg 155-156, Larman).
- Represented as a solid line between objects
  - the association is inherently bi-directional
  - may contain a *cardinality* or *multiplicity* value
  - optionally contains an arrow for easy reading

## Common Associations

- A is subpart/member of B. (SaleLineItem-Sale)
- A uses or manages B. (Cashier –Register, Pilot-airplane)
- A communicates with B. (Student -Teacher)
- A is transaction related to B. (Payment -Sale)
- A is next to B. (SaleLineItem-SaleLineItem)
- A is owned by B. (Plane-Airline)
- A is an event related to B. (Sale-Store)

# Associations

- **Association** - a relationship between concepts that indicates some meaningful and interesting connection



For example, do we need to remember what *SaleLineItem* instances are associated with a *Sale instance*? *Definitely, otherwise it would not be possible to reconstruct a sale, print a receipt, or calculate a sale total.*

# Associations

- Do we need to have memory of a relationship between a current *Sale and a Manager*?
- No, the requirements do not suggest that any such relationship is needed. It is not wrong to show a relationship between a Sale and Manager, but it is not compelling or useful in the context of our requirements.

# Finding Associations

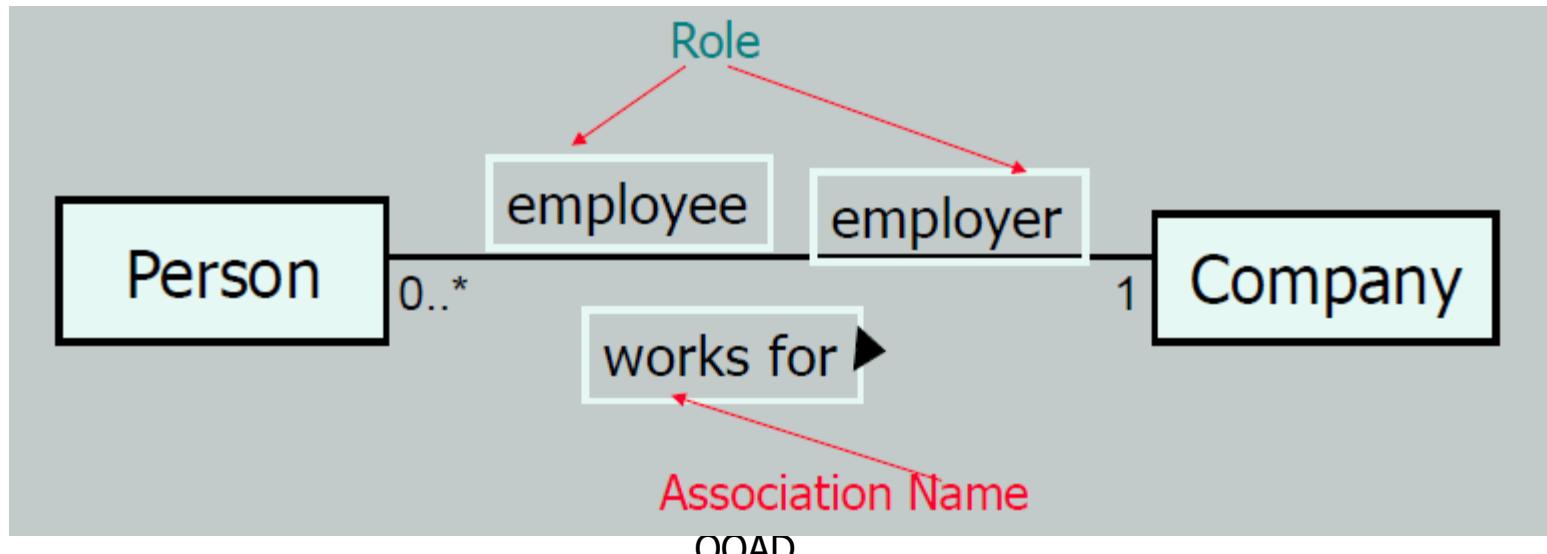
- High priority associations
  - $A$  is a physical or logical part of  $B$
  - $A$  is physically or logically contained in/on  $B$
  - $A$  is recorded in  $B$
- Other associations
  - $A$  uses or manages or controls  $B$  (*Pilot -airplane*)
  - $A$  owns  $B$  (*Airline -airplane*) or (*Register-Store*)
  - $A$  is event related to  $B$  (*Customer-Sale*)

# Association Guidelines

- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration (need-to-know associations)
- More important to identify concepts than associations
- Too many associations tend to confuse the conceptual model
- Avoid showing redundant or derivable associations

# Roles in Associations

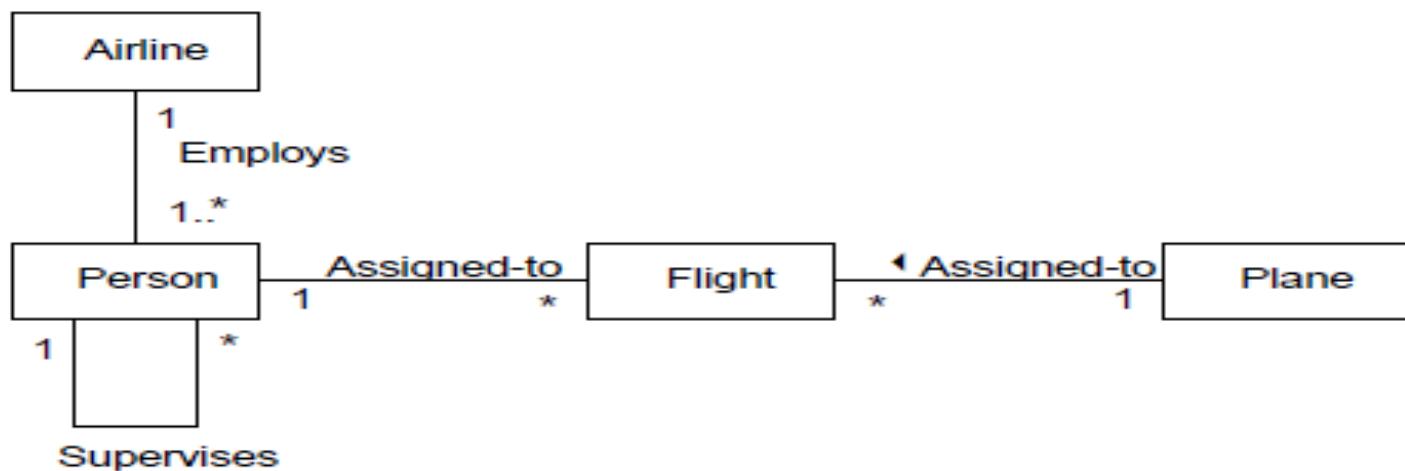
- Each of the two ends of an association is called a **role**. Roles have
  - name
  - multiplicity expression
  - navigability



# **Naming Associations**

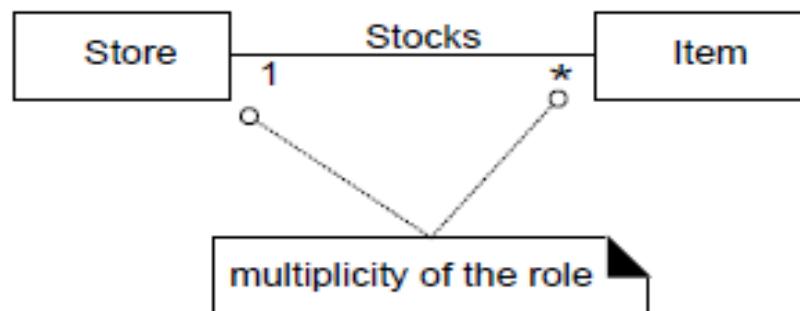
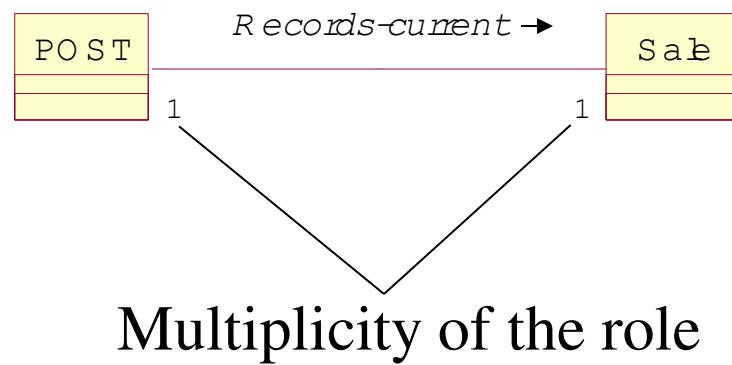
- Association names should start with a capital letter (same as concepts, objects)
- ***Noun phrases*** help identify ***objects/concepts***
- ***Verb phrases*** help identify ***associations***
- Use hyphens to separate words when a phrase is used to name something. Name should enhance meaning also. Like ***Sales uses Cashpayment instead of sale Paid -by Cashpayment***
- Legal Format are:
  - Records-current or RecordCurrent  
OOAD

# *Naming Associations*



# Multiplicity

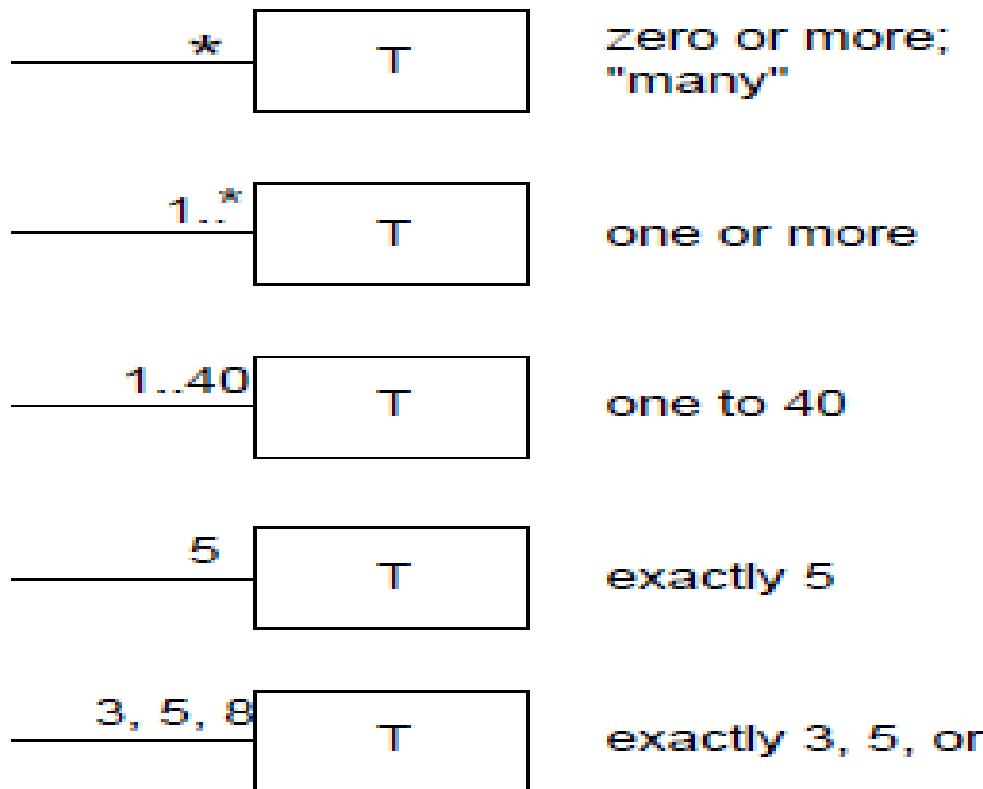
- **Multiplicity** defines how many instances of type A can be associated with one instance of type B at a particular moment in time



# Association - Multiplicity

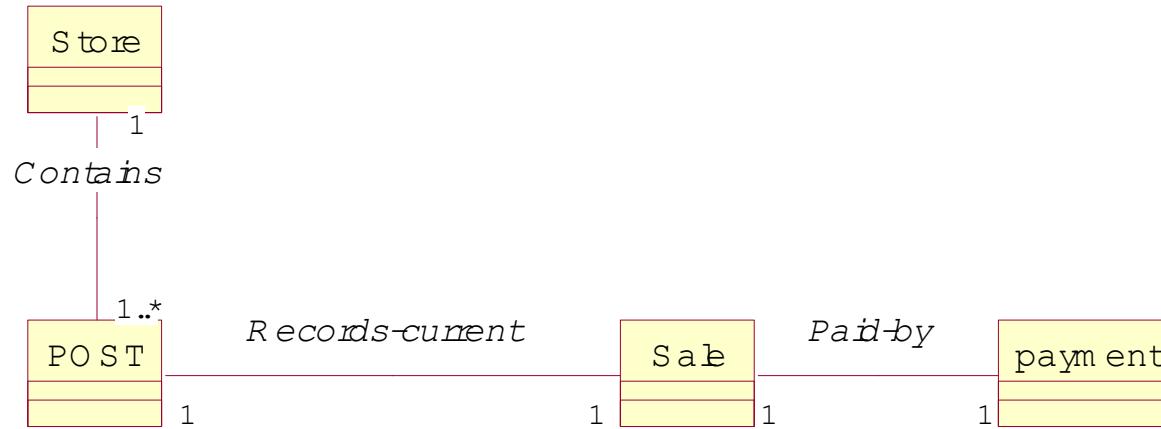
- **Multiplicity:** indicates the number of objects of one class that may be related to a single object of an associated class.
- Can be one of the following types
  - 1 to 1, 1 to 0..\*, 1 to 1..\*, 1 to n, 1 to 1..n
- The multiplicity value communicates how many instances can be validly associated with another, at a particular moment, rather than over a span of time. For example, it is possible that a used car could be repeatedly sold back to used car dealers over time. But at any particular moment, the car is only *Stocked-by one dealer*. The car is not *Stocked-by many dealers at any particular moment*.

# Association - Multiplicity



Multiplicity values.

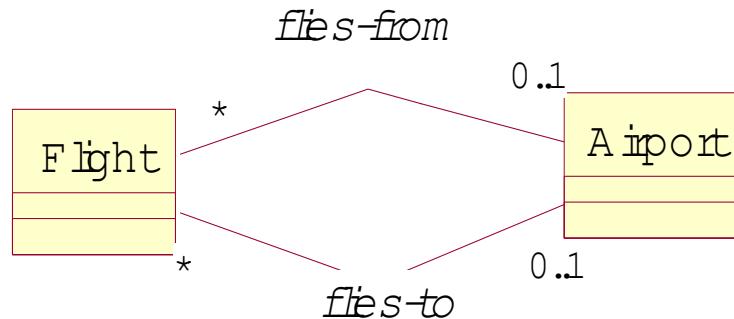
# Associations - Contd.



Associations are generally read left to right, top to bottom

# Associations - Contd.

Multiple associations between two types

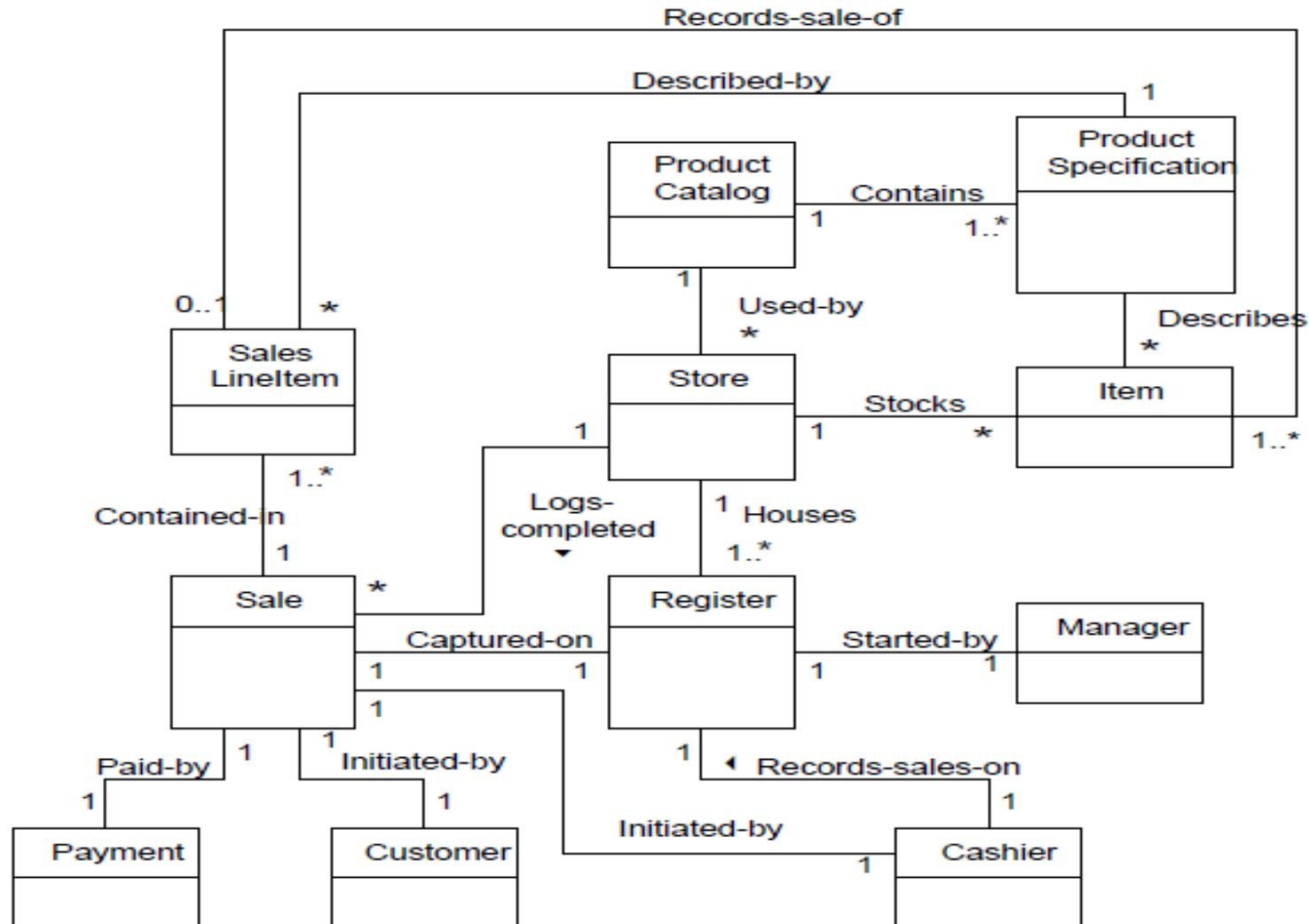


During analysis phase, an association is *not* a statement about data flows, instance variables, or object connections in the software solution, it is a statement that a relationship is meaningful in a purely conceptual sense.

# *Regarding Associations*

- Associations are real-world relationships, which may or may not be implemented in the final system.
- Conversely, we may discover associations that need to be implemented but were missed in analysis. The conceptual model should be updated!
- Usually, associations are implemented by placing an instance variable which “points to” (references) an instance of the associated class.
- Only add associations (or concepts) to the model that aid in the *comprehension(understanding)* of the system by others.
- Remember, conceptual models are communication tools, used to express concepts in the problem domain.

# Partial Domain Model of POST



# Conceptual Model - Attributes

- Objectives
  - identify attributes in a conceptual model
  - distinguish between correct and incorrect attributes

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

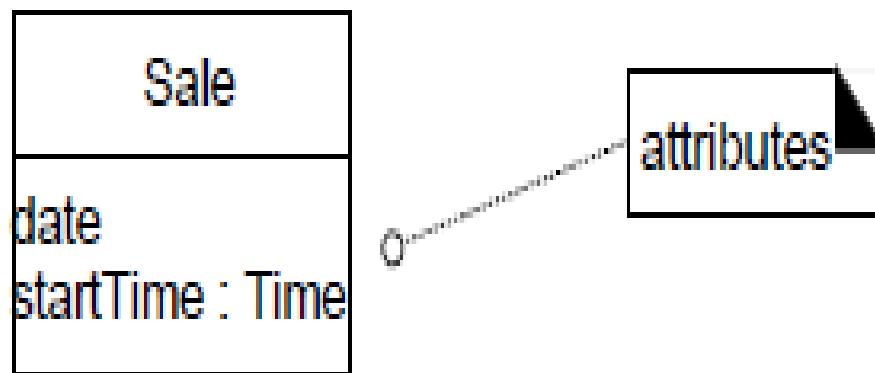
# Attributes [1]

- **Attribute** - is a logical data value of an object.  
*It is a named property of a class describing values held by each object of the class*
- **Attribute Type:** A specification of the external behavior and/or the implementation of the attribute
- Include the following attributes in a domain model
  - those for which the requirements suggest or imply a need to remember information
- For example, a **sale** receipt normally includes a date and time attribute

Attribute Name:attribute Type

# **UML Attribute Notation**

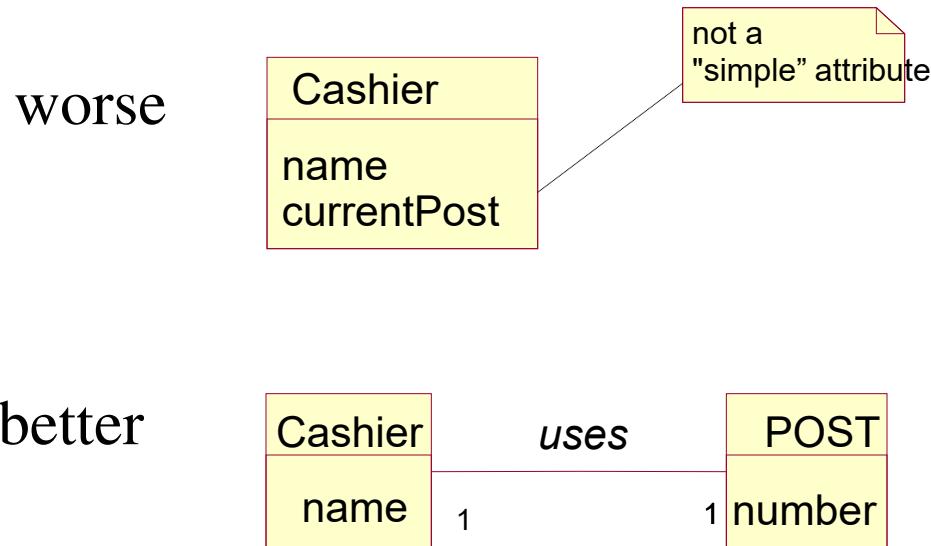
- Attributes reside in the 2nd compartment of a concept box
- Format is name: type
- Attributes should be pure data values



# Attributes [2]

- Attributes in a conceptual model should preferably be **simple attributes or pure data values**
- Common simple attribute types include
  - boolean, date, number, string, time

# Attributes: Examples



*Relate two items with associations, not attributes, in conceptual model. Avoid representing complex domain concepts as attributes; use associations.*

# Complex Attributes

- Pure data values - expressed as attributes; they do not illustrate specific behaviors;
  - Example - Phone number
  - A Person can have many Phone numbers
- Non-primitive attribute types
  - represent attributes as non-primitive types (concepts or objects) if
    - it is composed of separate sections (name of a person)
    - there are operations associated with it such as validation
    - it is a quantity with a unit (payment has a unit of currency)

# Non-primitive Data Type Classes

Represent what may initially be considered a primitive data type (such as a number or string) as a non-primitive class if:

- It is composed of separate sections.
  - phone number, name of person
- There are operations usually associated with it, such as parsing or validation.
- social security number
- It has other attributes.
- promotional price could have a start (effective) date and end date
- It is a quantity with a unit.
- payment amount has a unit of currency
- It is an abstraction of one or more types with some of these qualities.
- item identifier in the sales domain is a generalization of types such as Universal Product Code (UPC) or European Article Number (EAN)

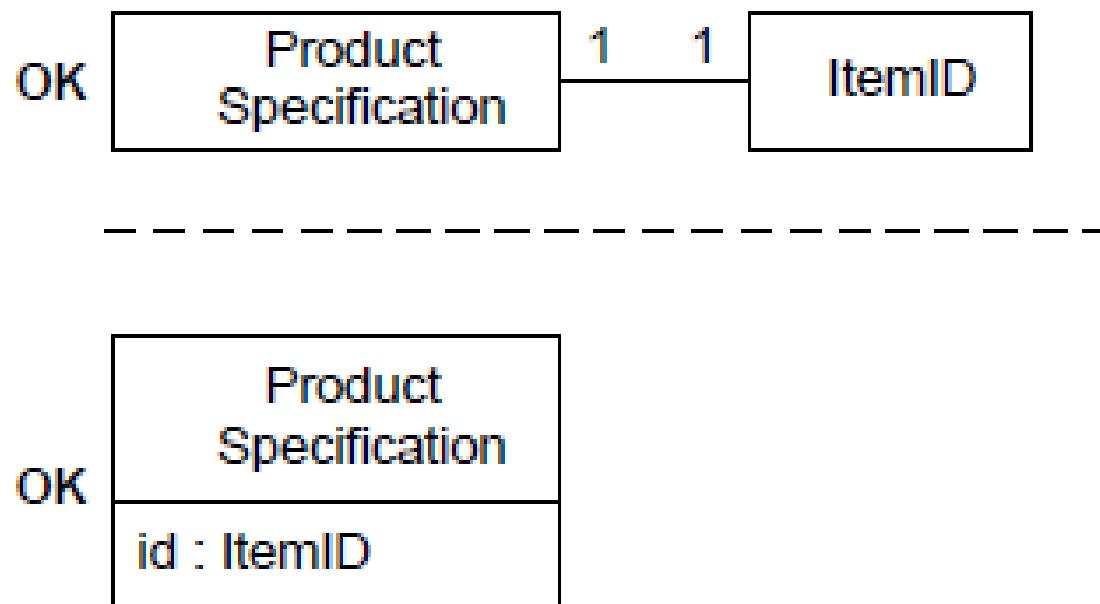
# Non-primitive Data Type Classes

- The type of an attribute may be expressed as a non-primitive class in its own right in a domain model.
- Example: The item identifier in POST is an abstraction of various common coding schemes, including UPC-A, UPC-E, and the family of EAN schemes. These numeric coding schemes have subparts identifying the manufacturer, product, country (for EAN), and a checksum digit for validation. Therefore, there should be a non-primitive *ItemID* class

# Where to Illustrate Data Type Classes?

- *Should the `ItemId` class be shown as a separate conceptual class in a domain model?*
- It depends on what you want to emphasize in the diagram. Since `ItemId` is a *data type (unique identity of instances is not important)*, it may be shown in the attribute compartment of the class box. But since it is a non-primitive class, with its own attributes and associations, it may be interesting to show it as a conceptual class in its own box. There is no correct answer; it depends on how the domain model is being used as a tool of communication, and the significance of the concept in the domain.

# Example

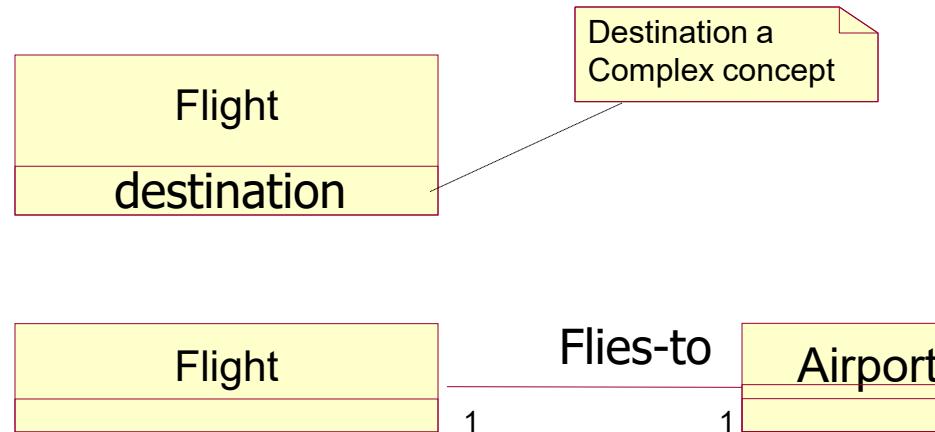


# Non-primitive Data Type Classes

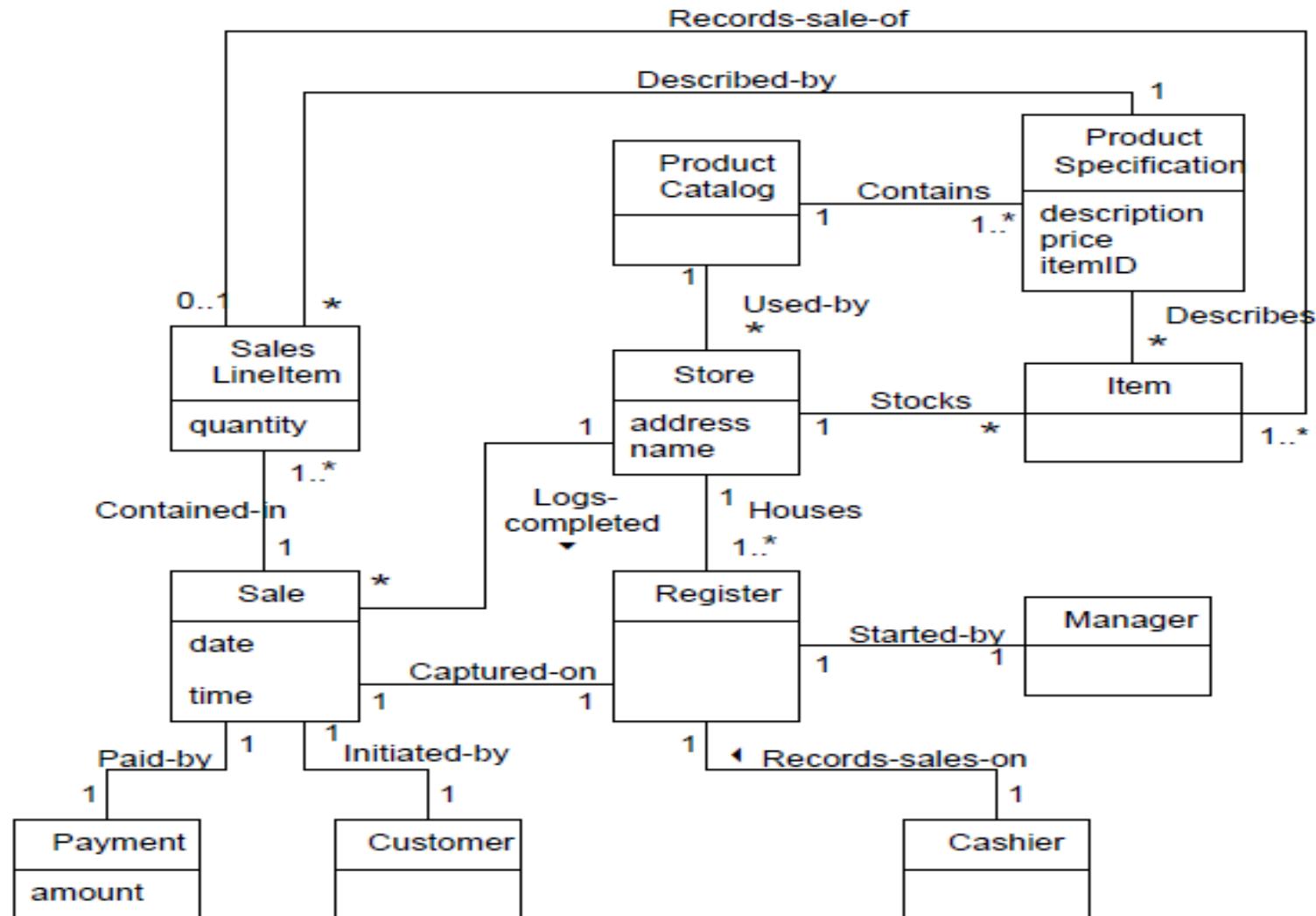
- The *price* and *amount* attributes should be non-primitive *Quantity* or *Money* classes because they are quantities in a unit of currency.
- The *address* attribute should be a non-primitive *Address* class because it has separate sections.
- The classes *ItemID*, *Address*, and *Quantity* are data types (*unique identity* of instances is not meaningful) but they are worth considering as separate classes because of their qualities.

# Example

- It is desirable to show non-primitive attributes as concepts in a conceptual model



# Partial Domain Model



# Recording terms in Glossary

- Define all terms that need clarification in a **glossary** or **model dictionary**.

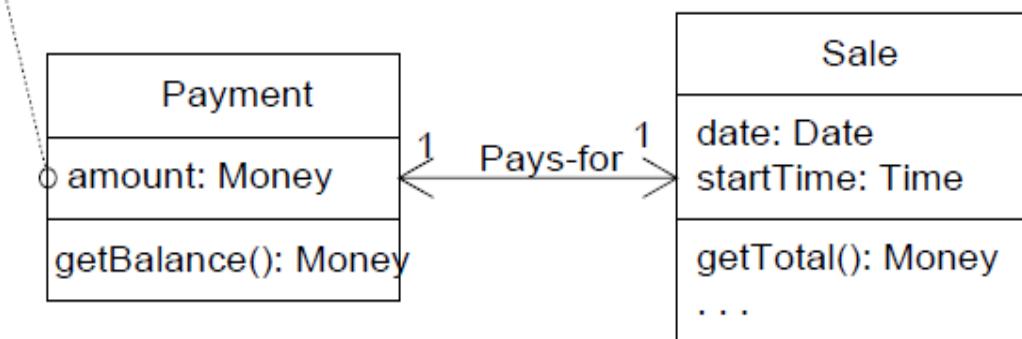
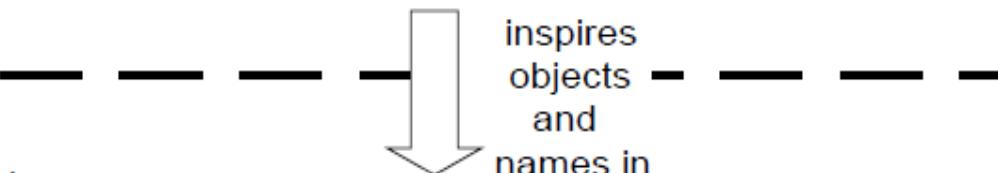
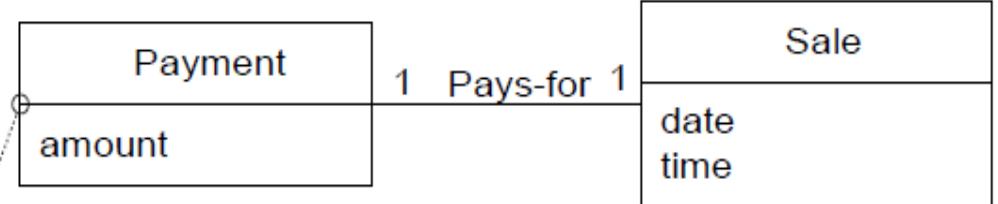
# Domain Model v/s Design Model

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former ~~inspires~~ the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

**UP Domain Model**  
Stakeholder's view of the noteworthy concepts in the domain.



**UP Design Model**

The object-oriented developer has taken inspiration from the real world in creating software classes.