



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

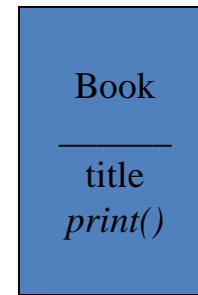
- **OO Design** - emphasis on defining the software objects; ultimately these objects are implemented in some programming language; *Book* may have a method named *print*.

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

Conceptual
model

Responsibility
assignment;

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

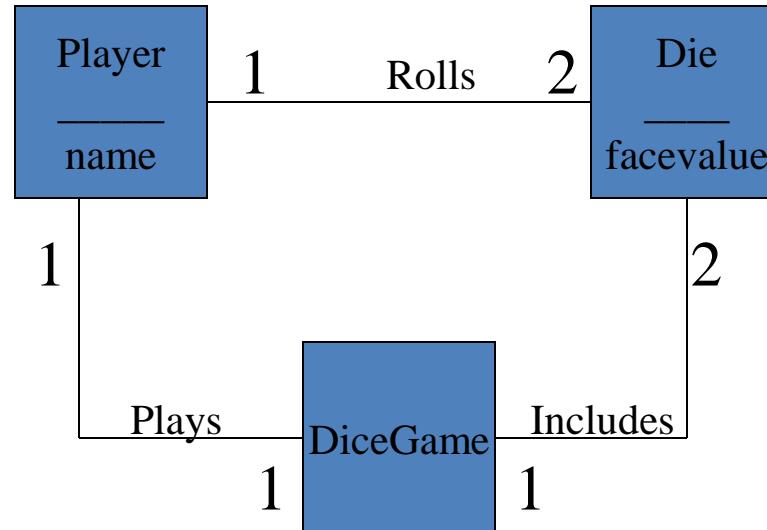
Description: This use case begins when the player picks up and rolls the die....

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

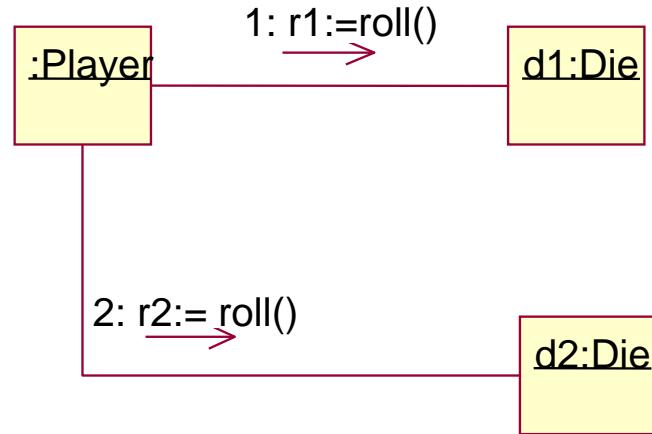


Essential step - allocating responsibility to objects and illustrating how they interact with other objects.

- 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

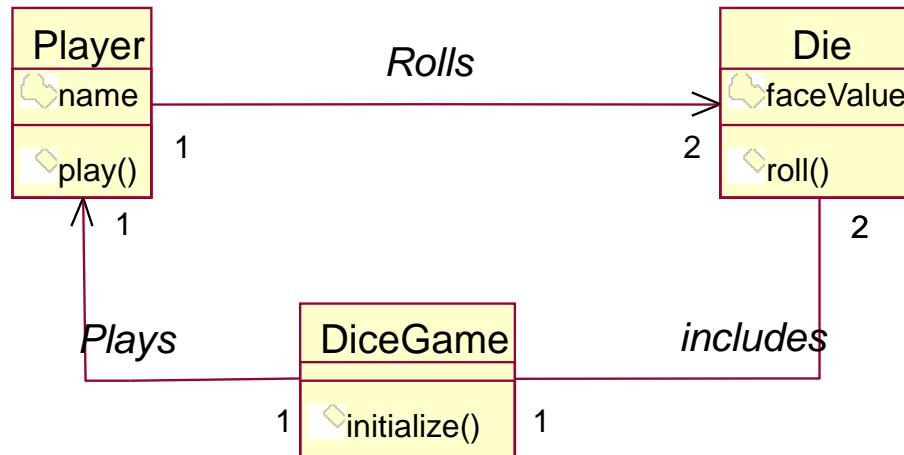
Key questions to ask

- 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

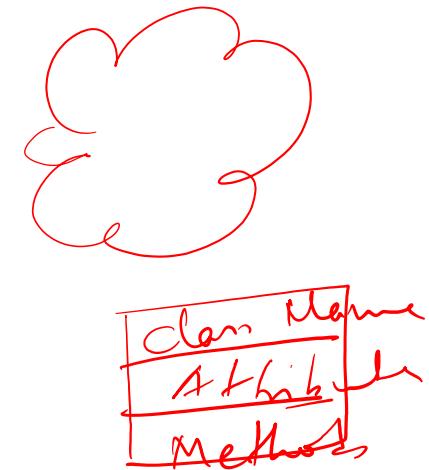
Analysis and Design models

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

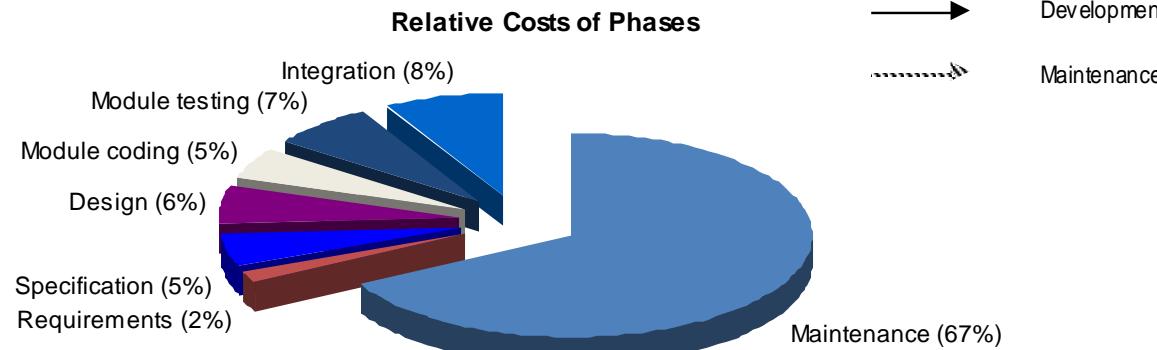
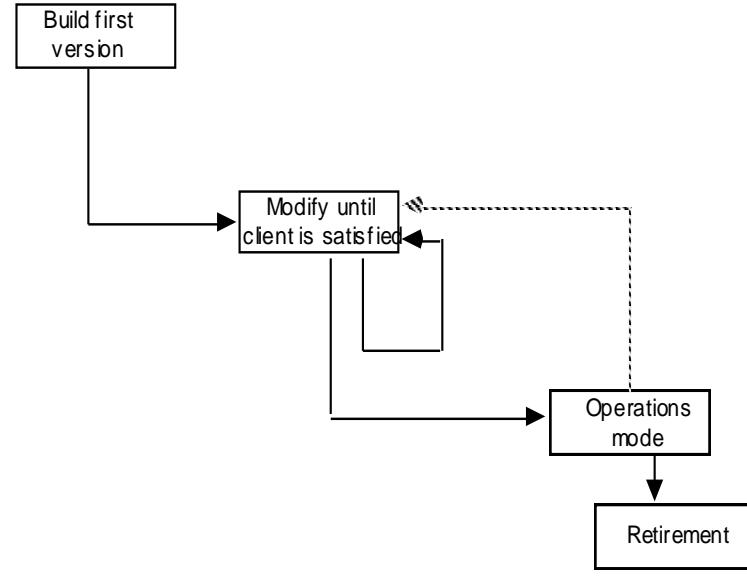
Lifecycle Models

Build-and-fix

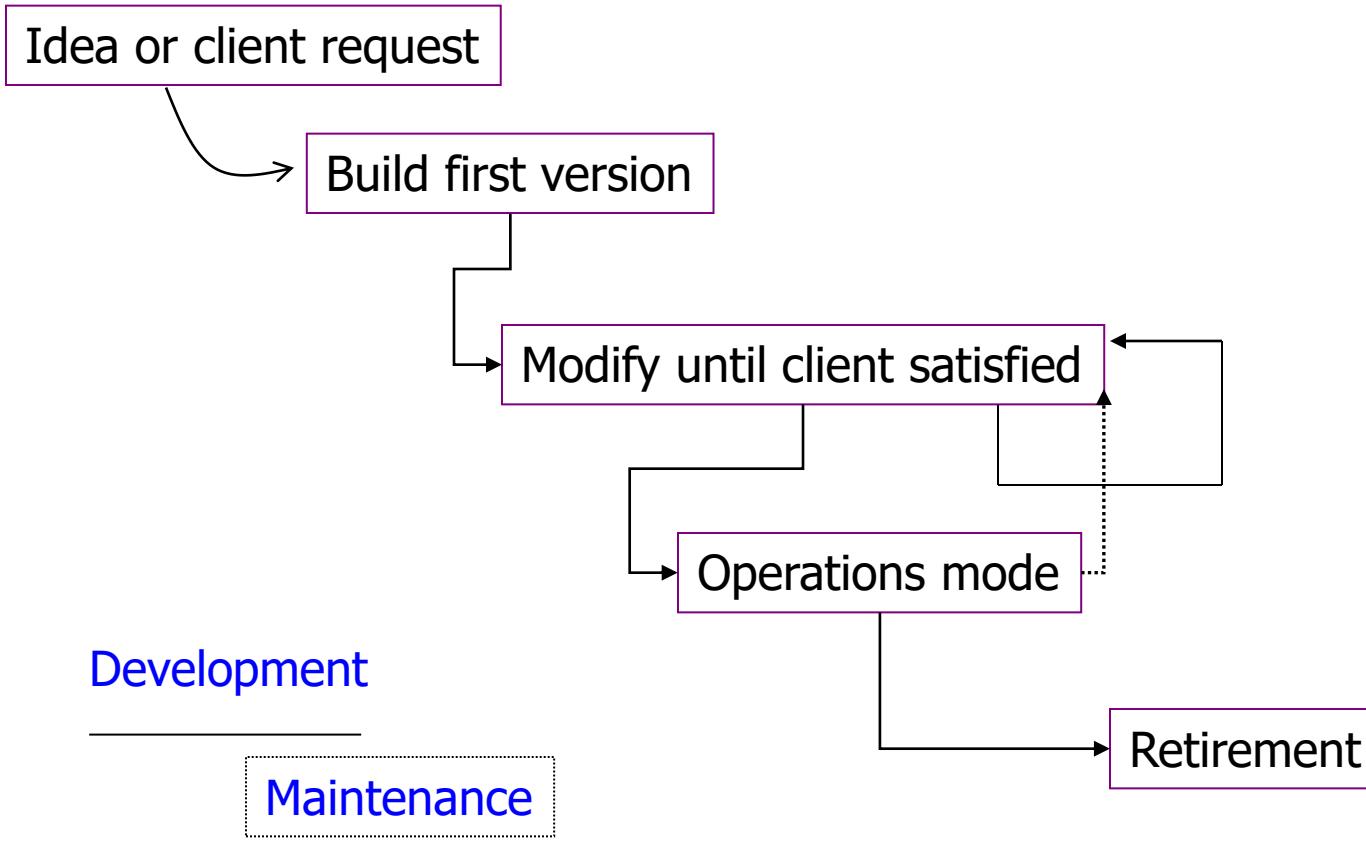
- develop system
 - without specs or design
 - modify until customer is satisfied

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.

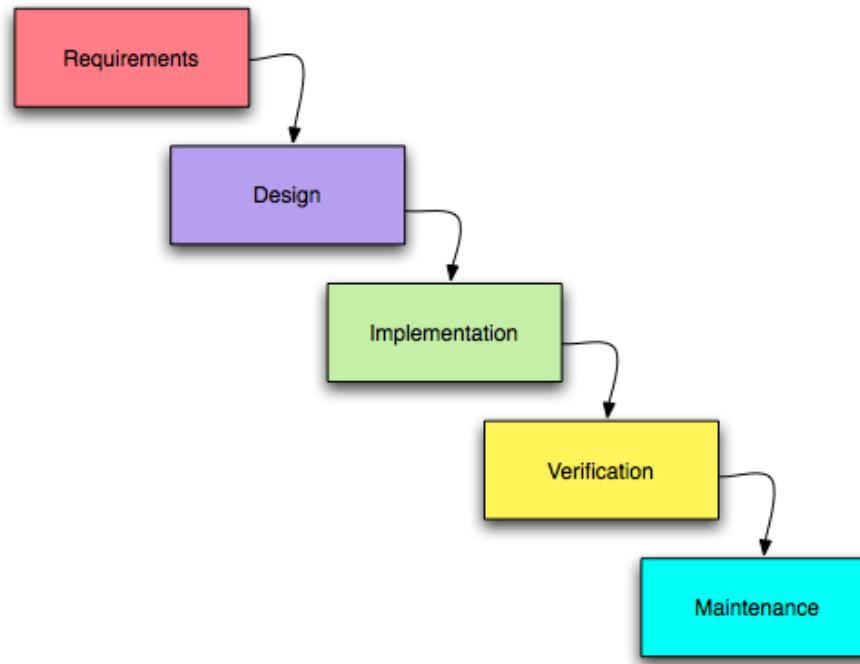
- There is no explicit design. However, a design will likely evolve in the mind of the developer.
- Modify until customer is satisfied.
- The approach might work for small programming projects [TA 162/252].

Build and fix model [3]

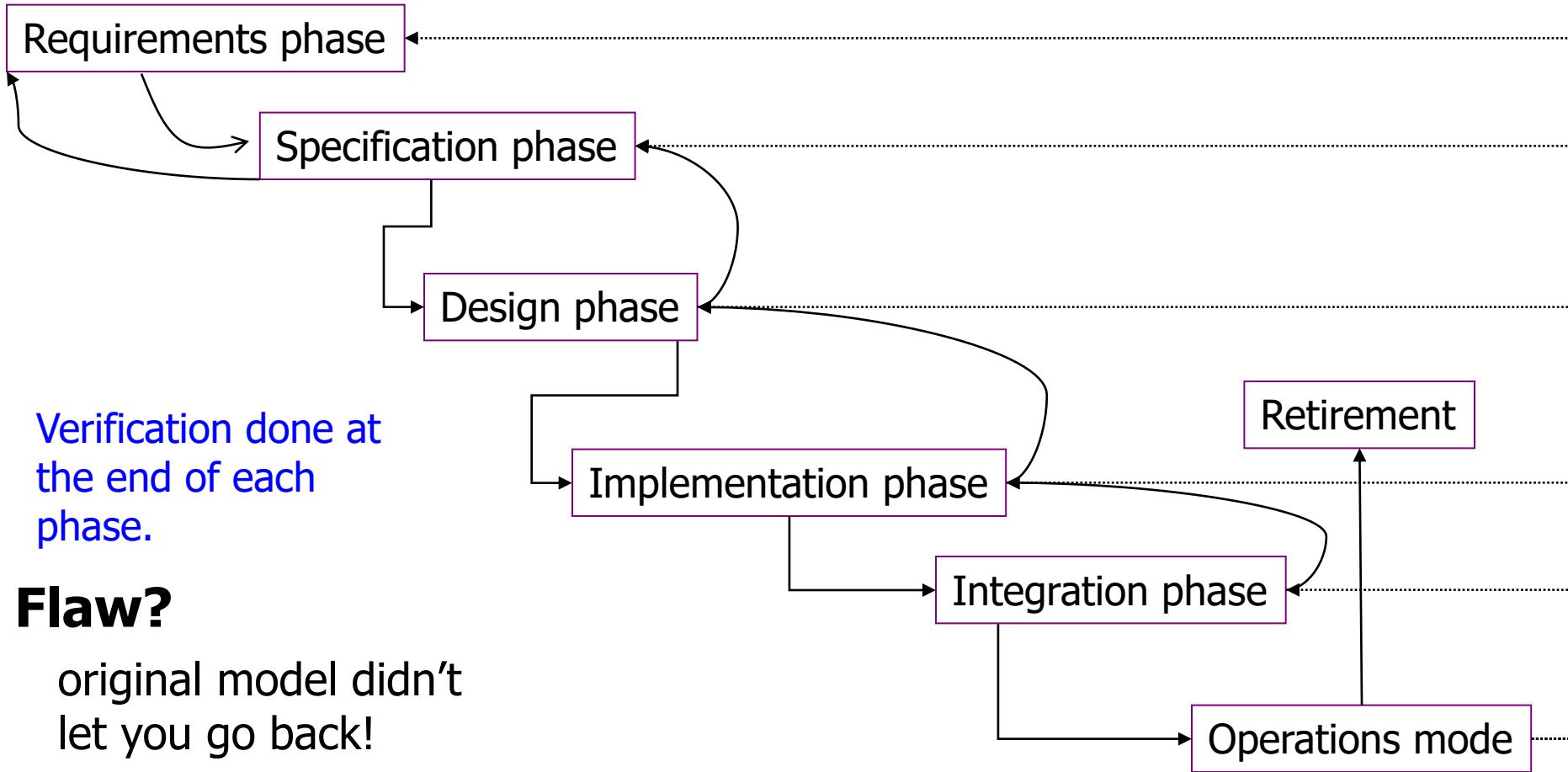
Cost of fixing an error increases as one moves away from the phase in which the error was injected.

- 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.

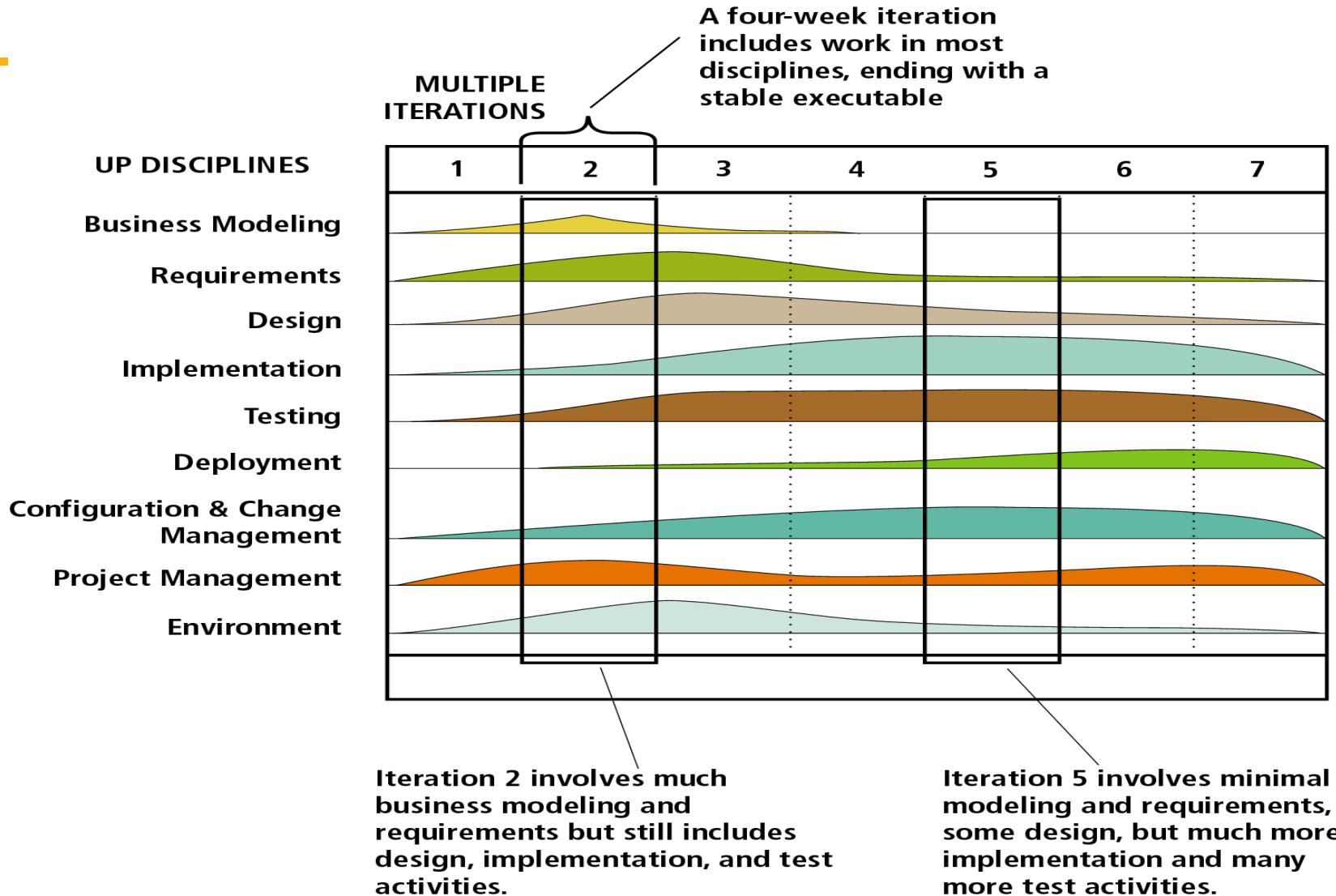
Unified Development Process [2]

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?*

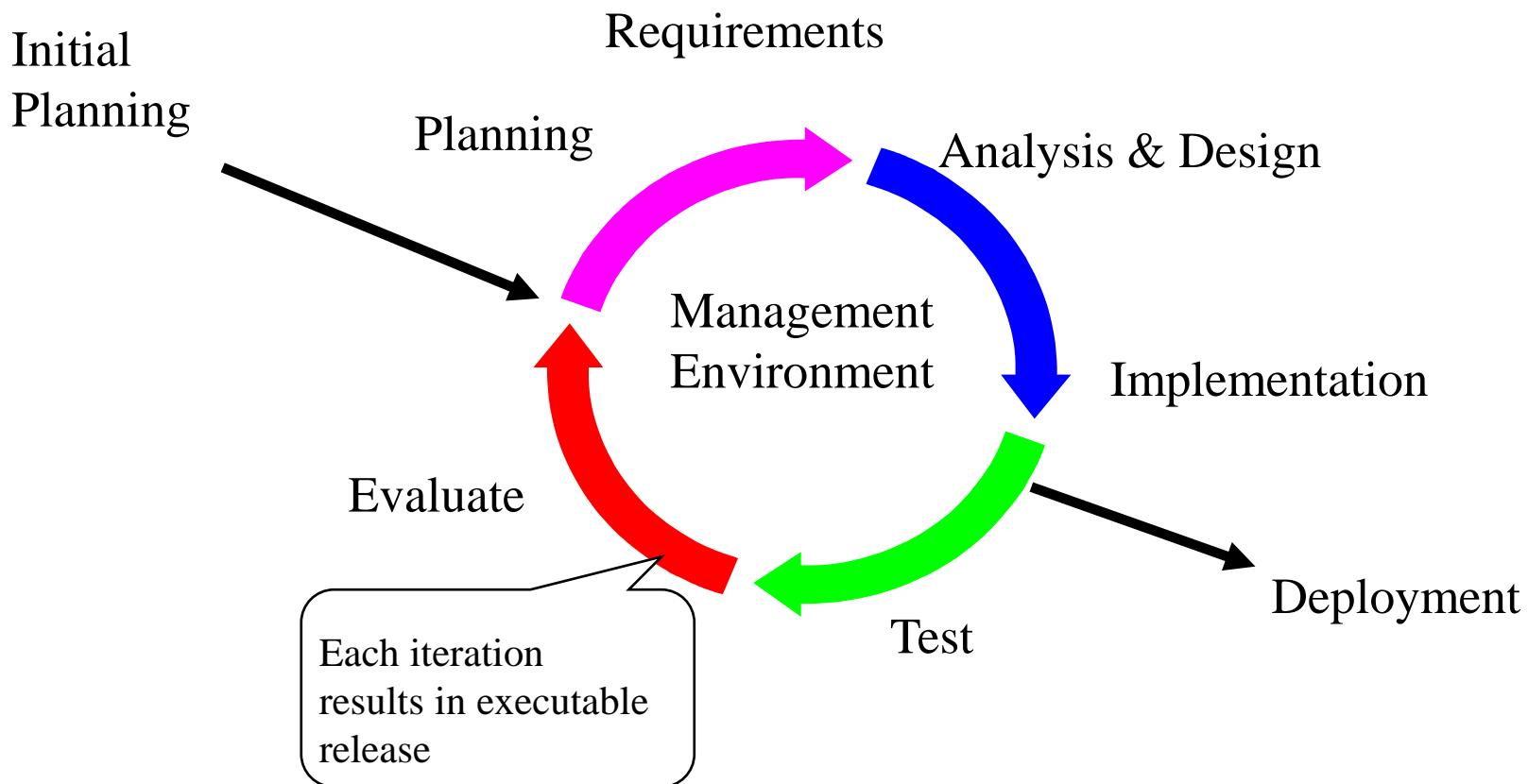
The Unified Process

- 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.

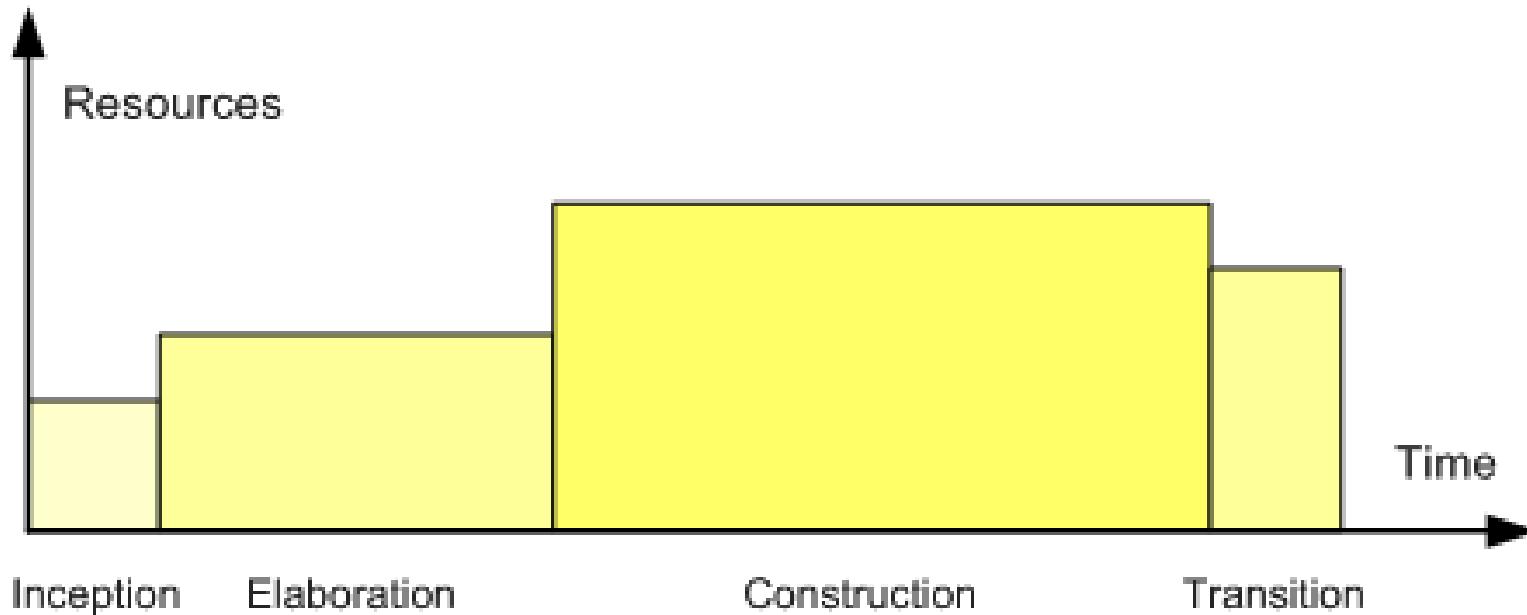
Each iteration period has its own development plan.

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.

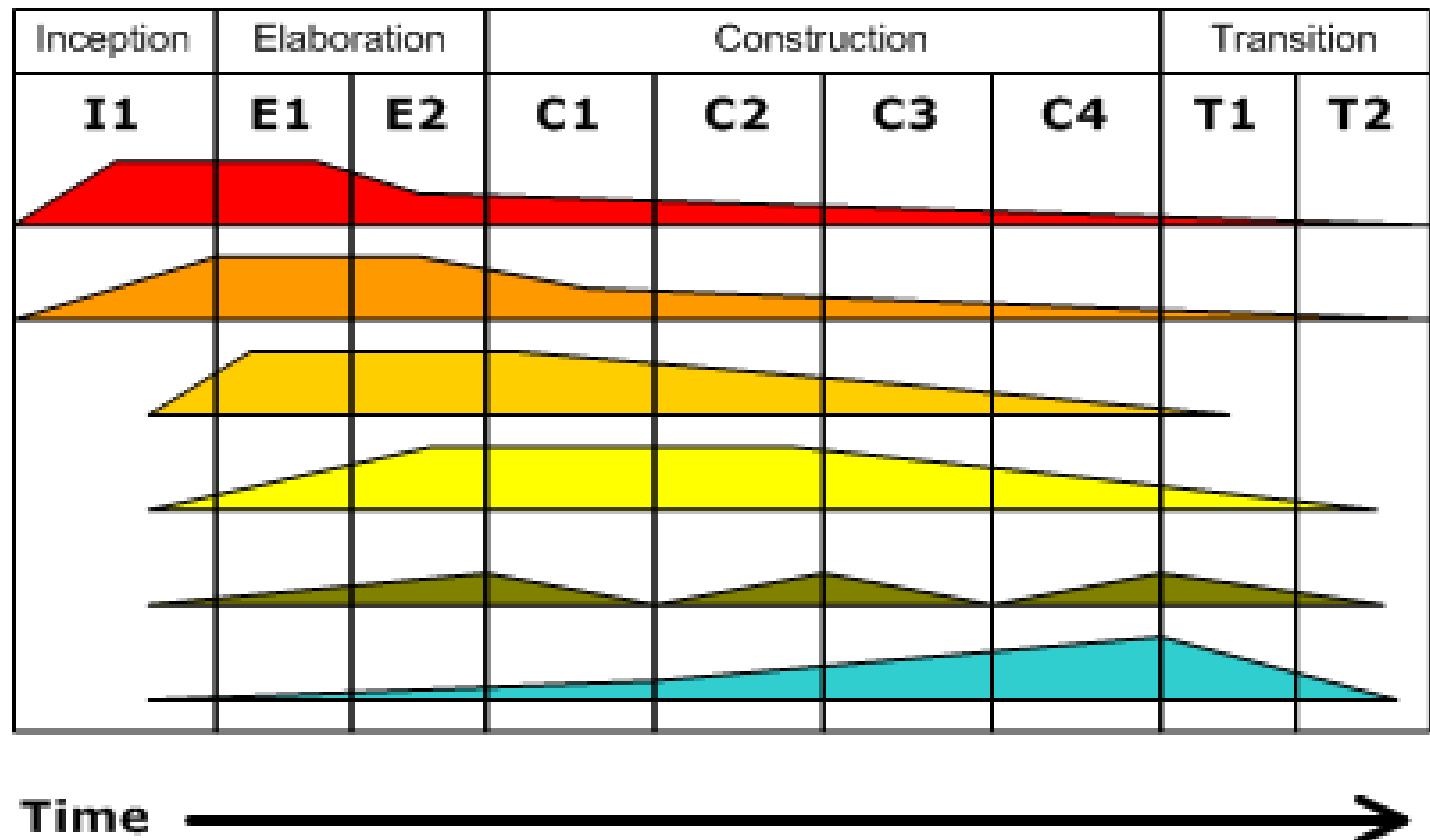
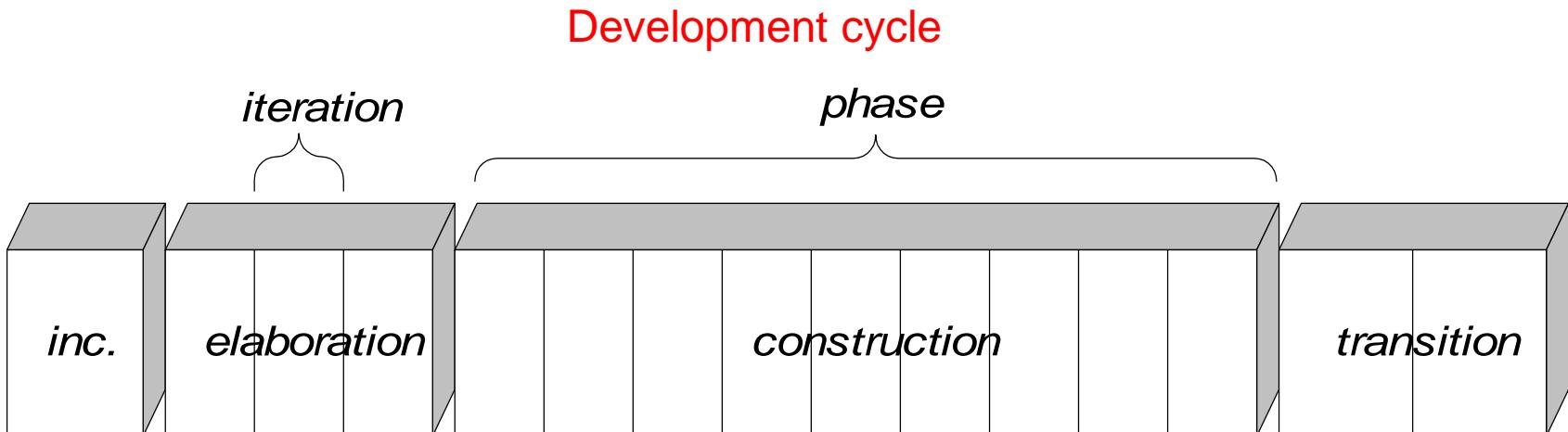


Diagram illustrating how the relative emphasis of different disciplines changes over the course of the project

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

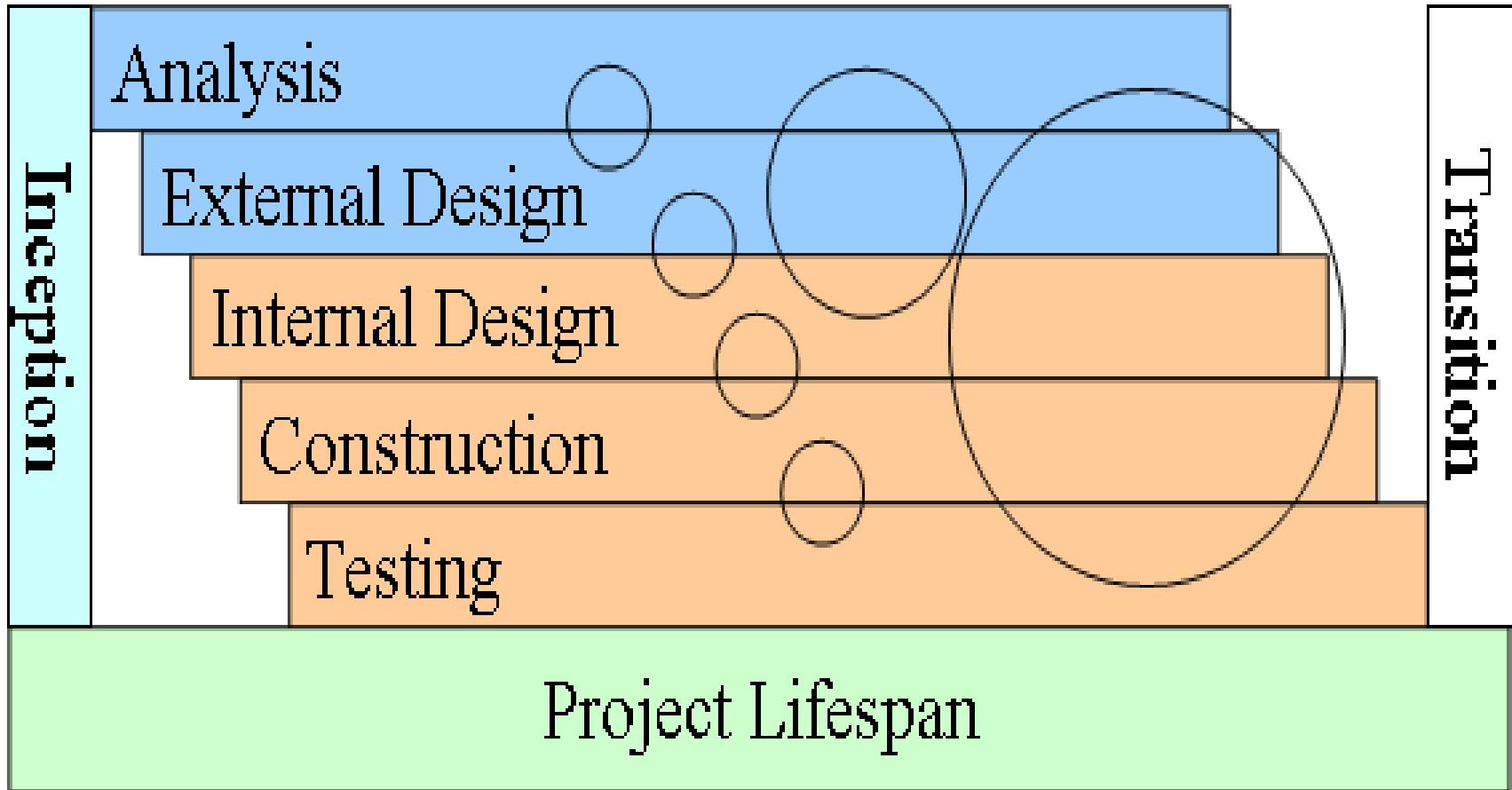
UP artifacts

- 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, 1/5th 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.

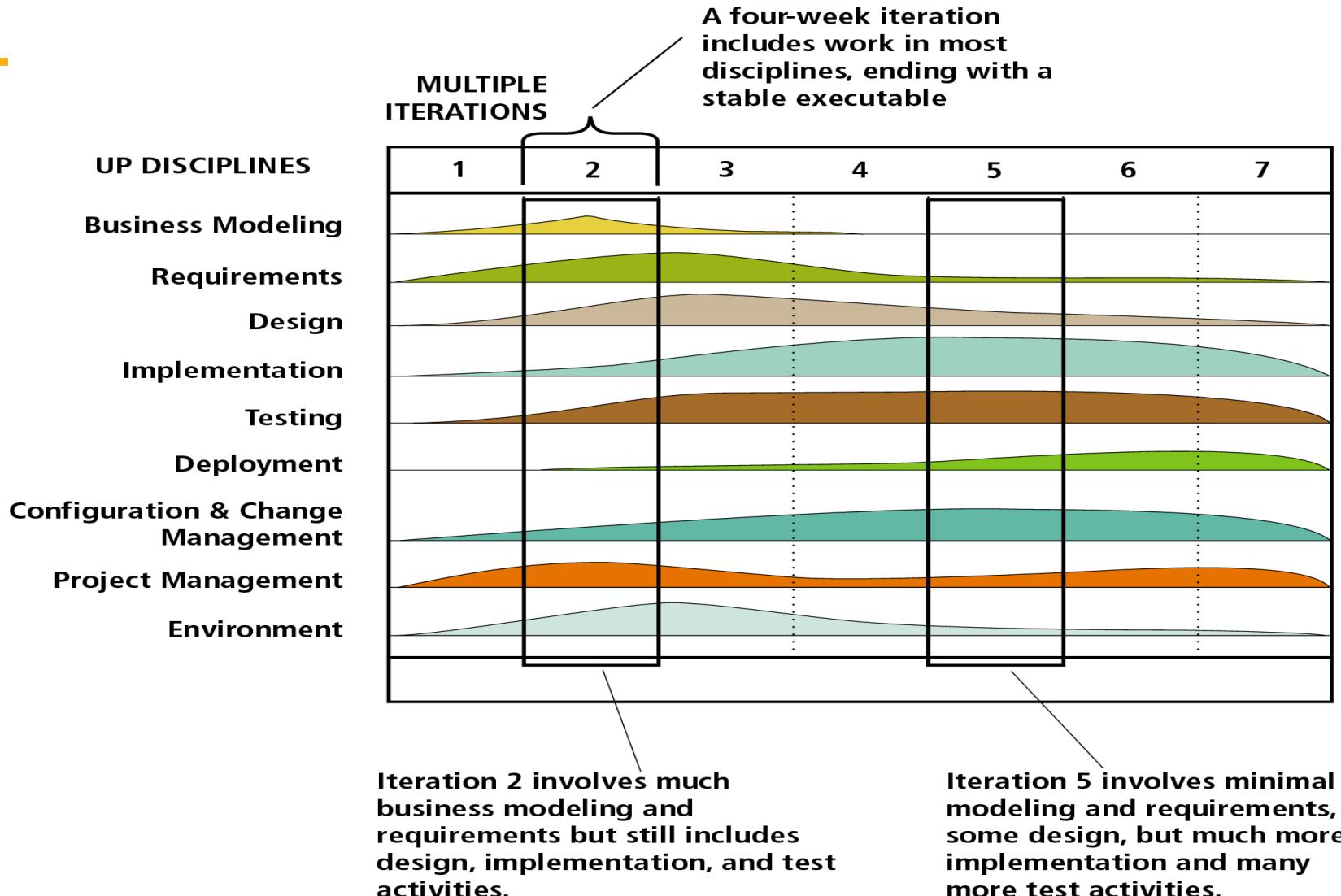
Unified Development Process [2]

- 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?*

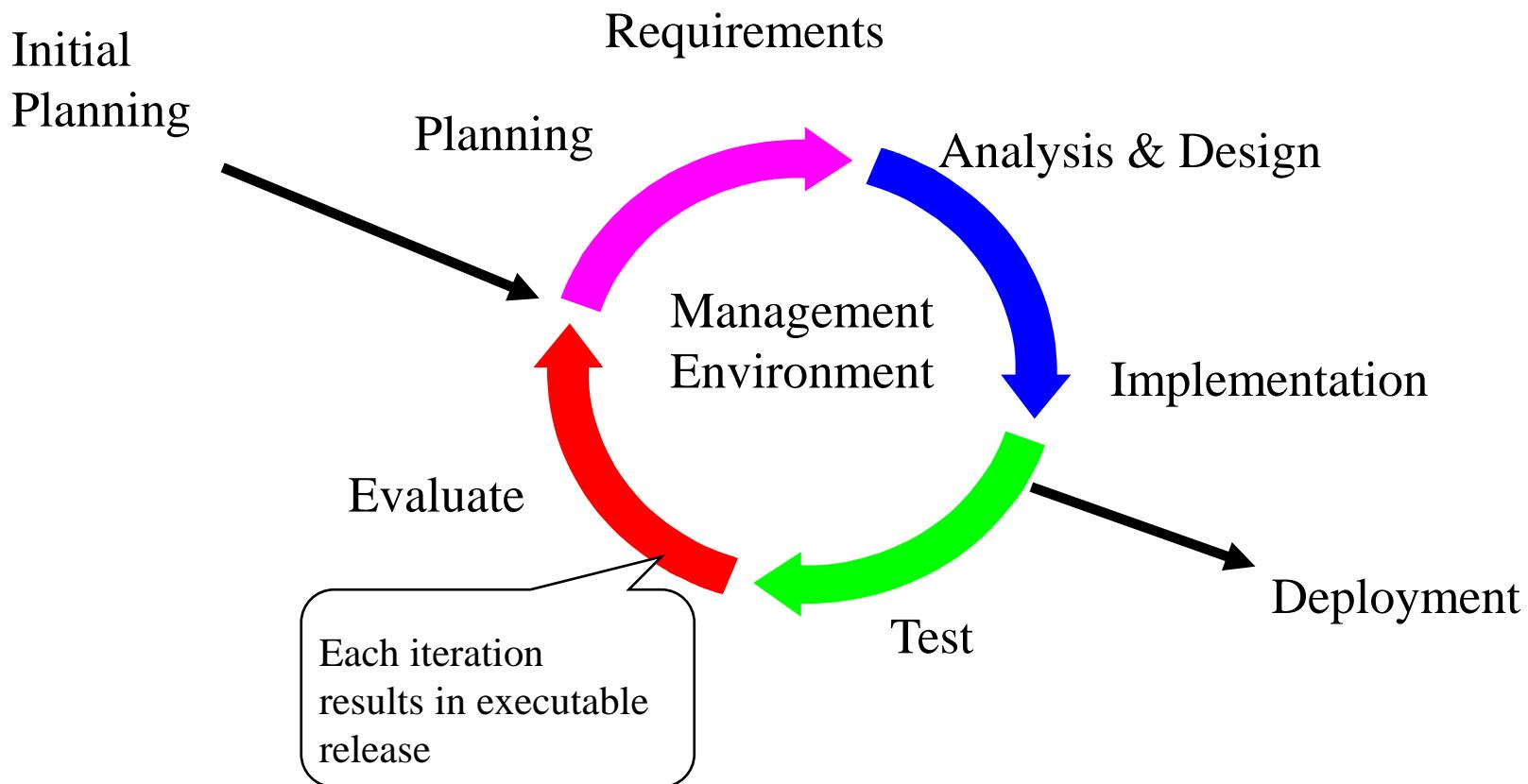
The Unified Process

- 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.

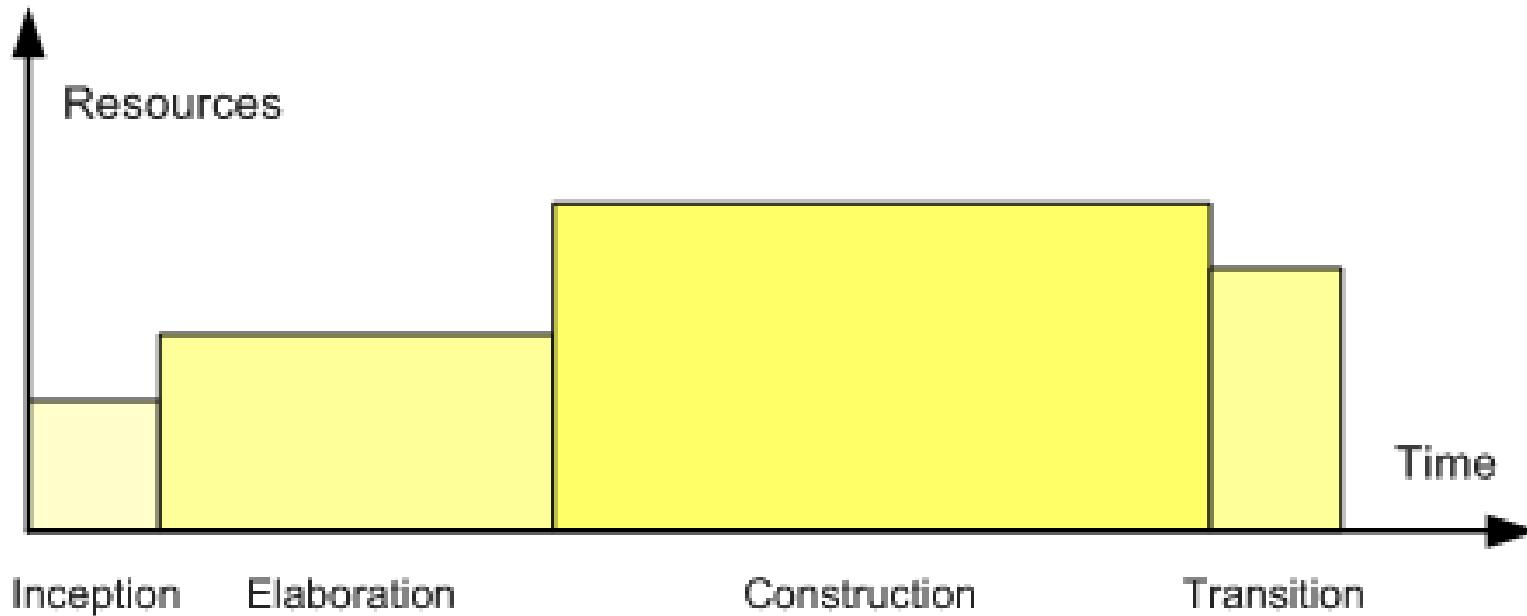
Each iteration period has its own development plan.

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.

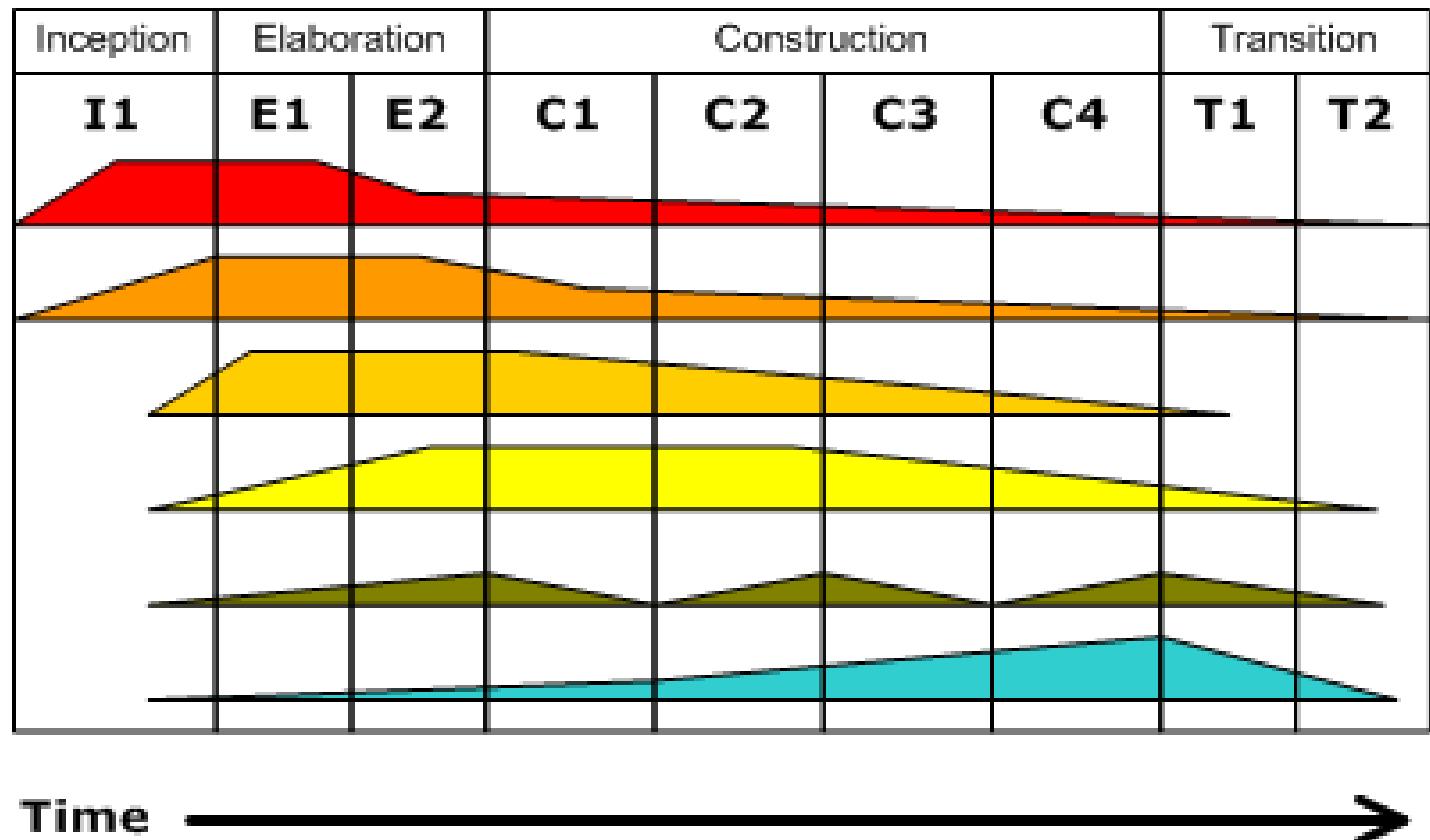
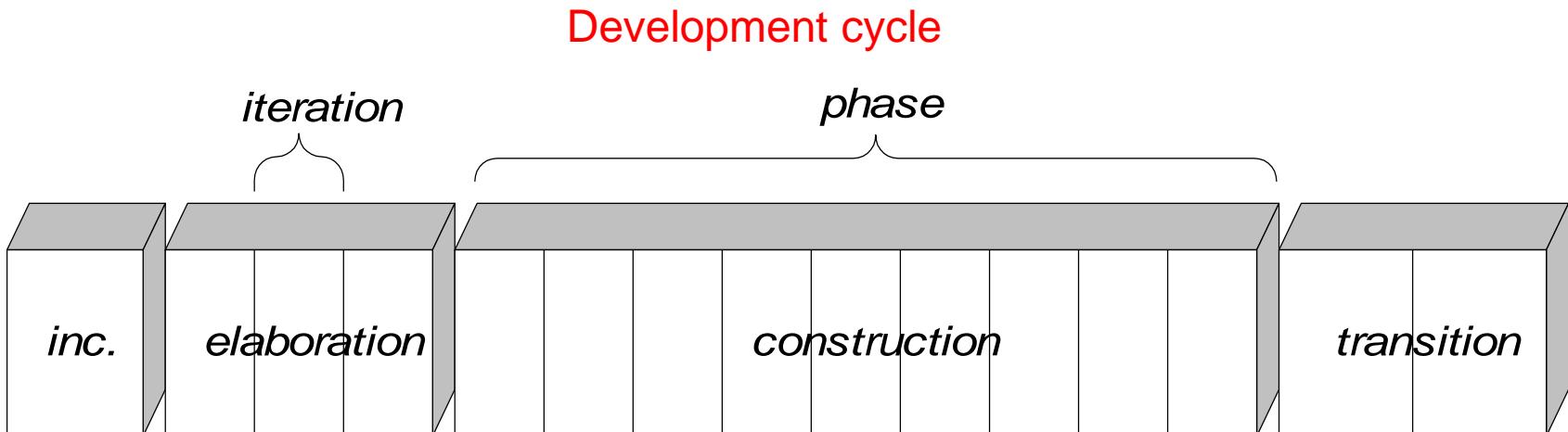


Diagram illustrating how the relative emphasis of different disciplines changes over the course of the project

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

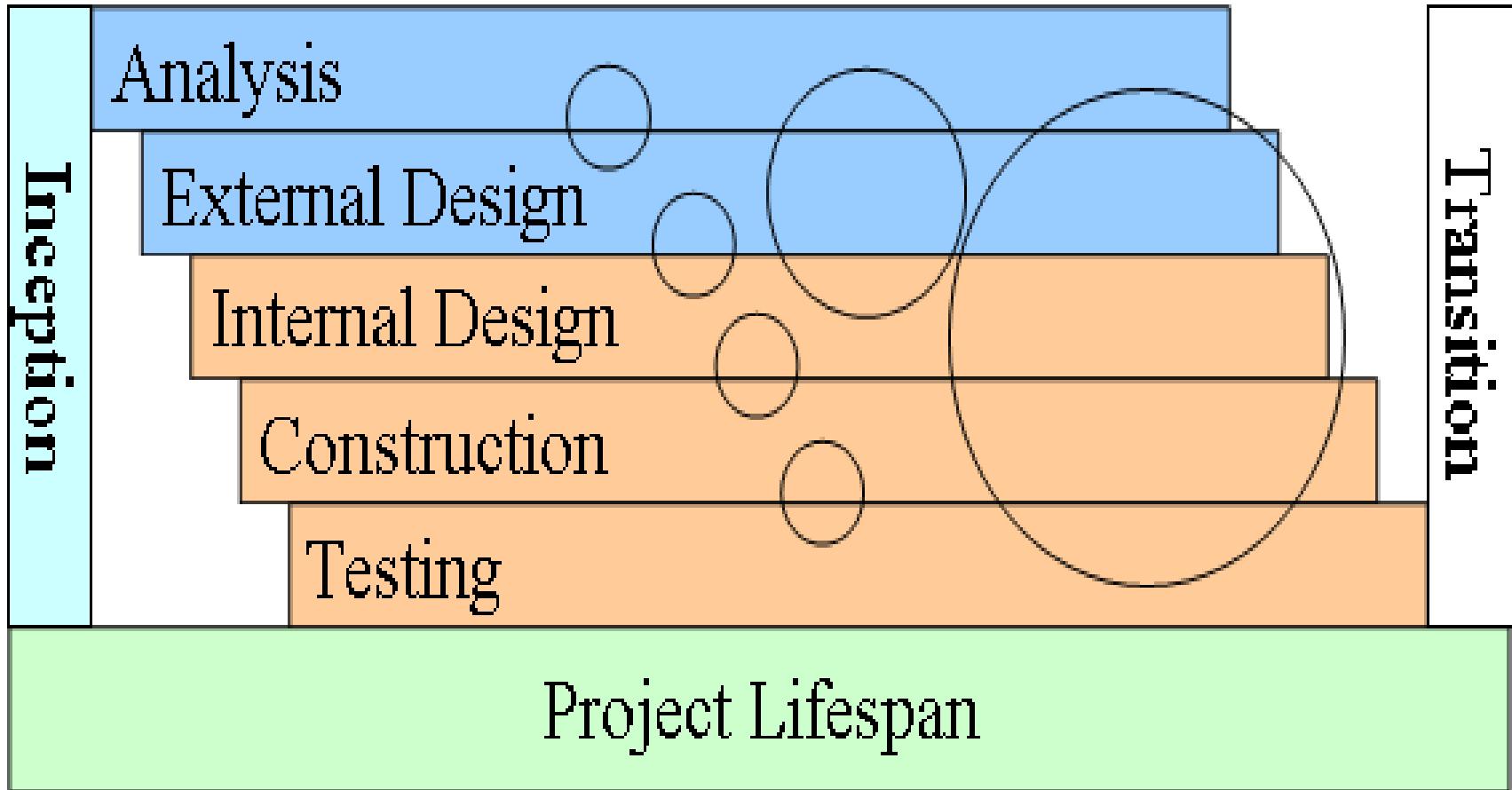
UP artifacts

- 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, 1/5th 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.

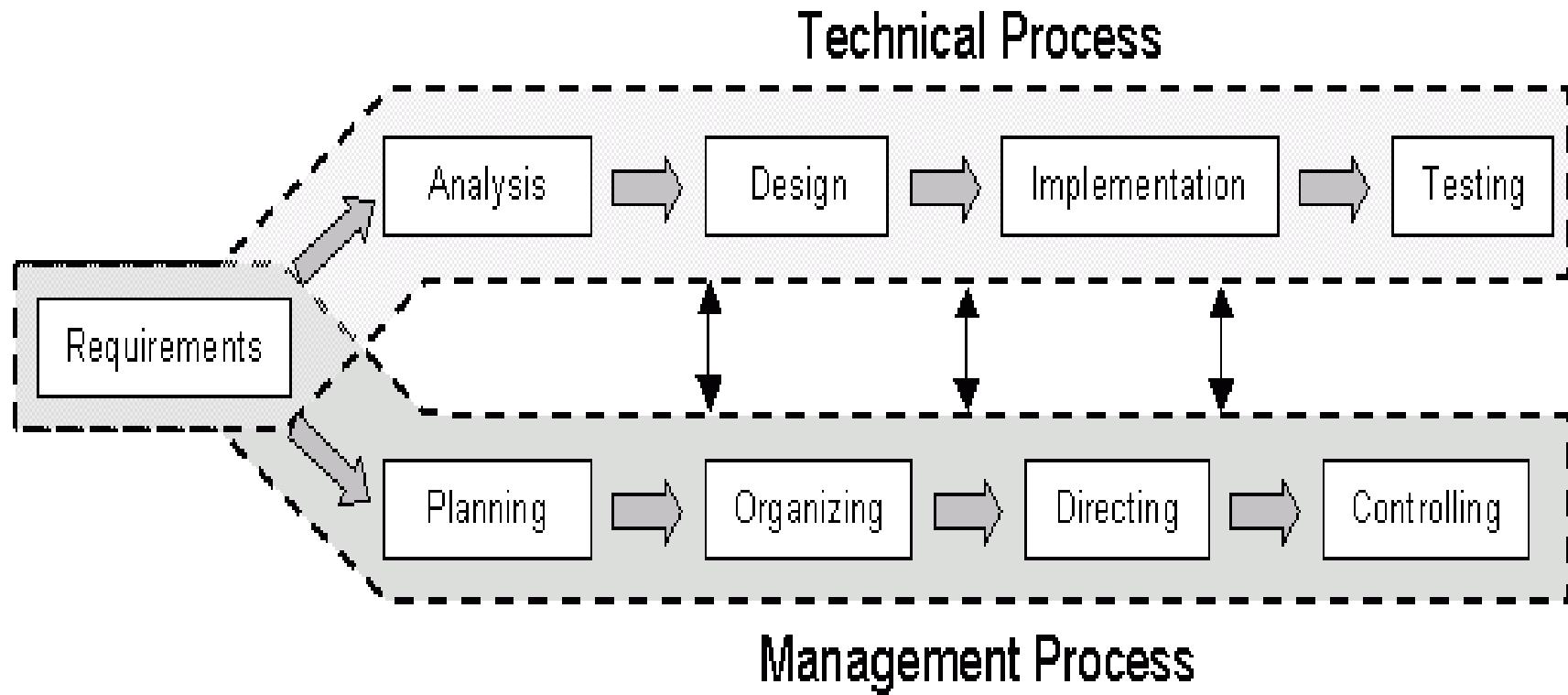
Questions

- 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).

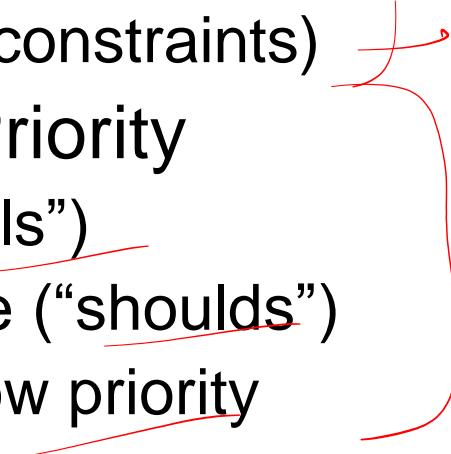
What is this Phase For?

- 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 *(S.O.R.)*
- performance
- documentation/training
- data

- resources
- security
- reliability
- portability
- maintenance
- etc.

facts
objective & achievable

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.

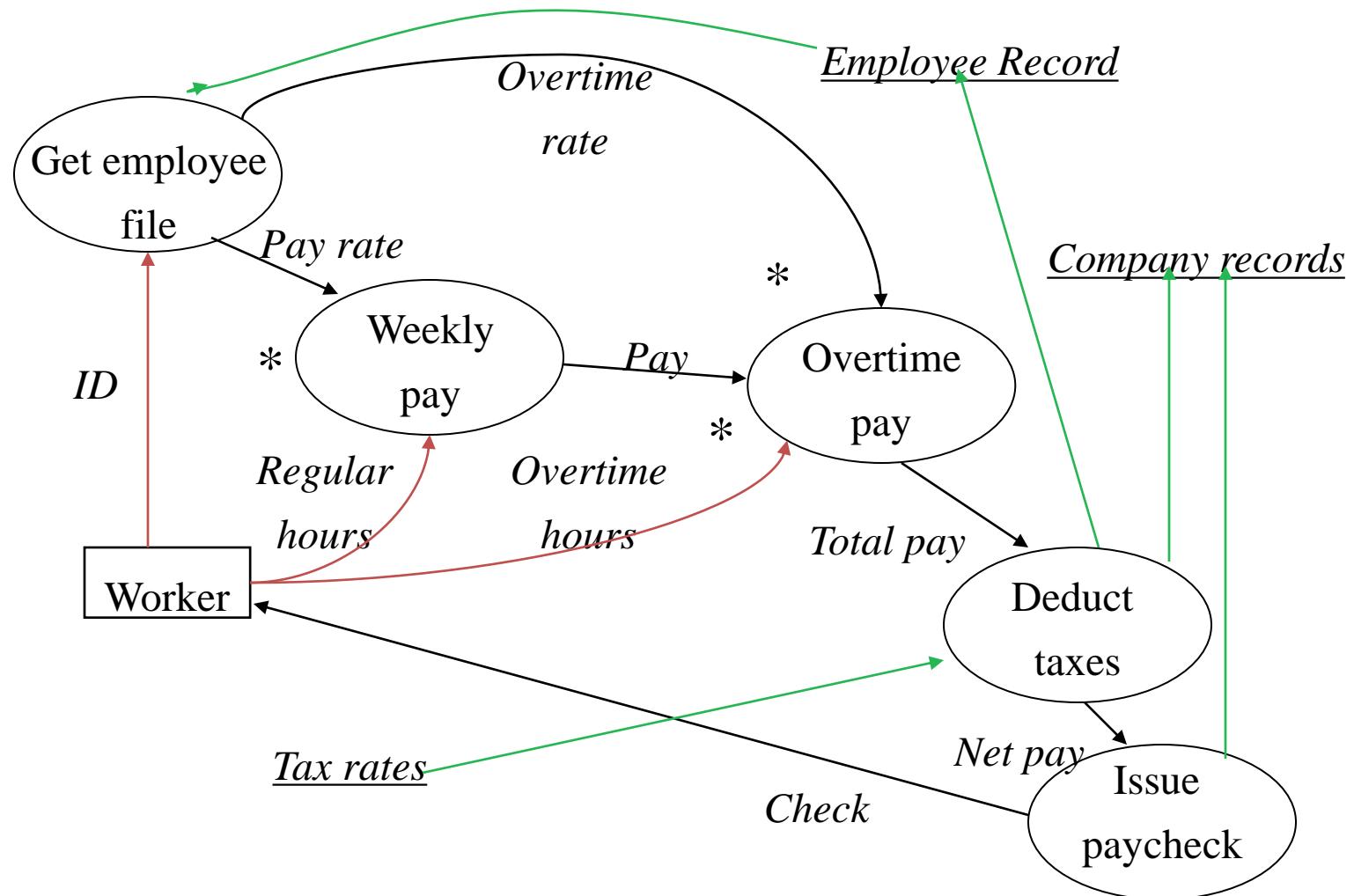
Requirements Analysis [3]

- 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]

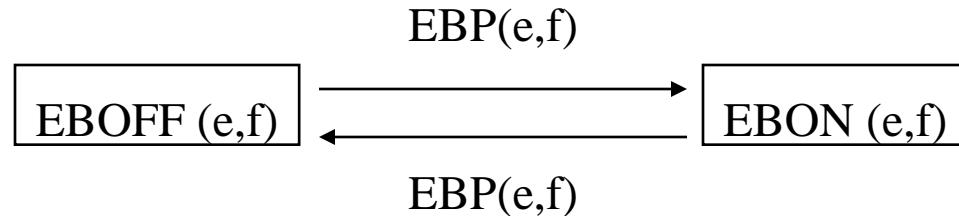
- C-requirements:
 - Use cases expressed individually and with a use case diagram. A use case specifies a collection of scenarios.
 - Sample use case: *Process sale*.
 - *Data flow diagram*:
 - Explains the flow of data items across various functions. Useful for explaining system functions. [Example 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 .

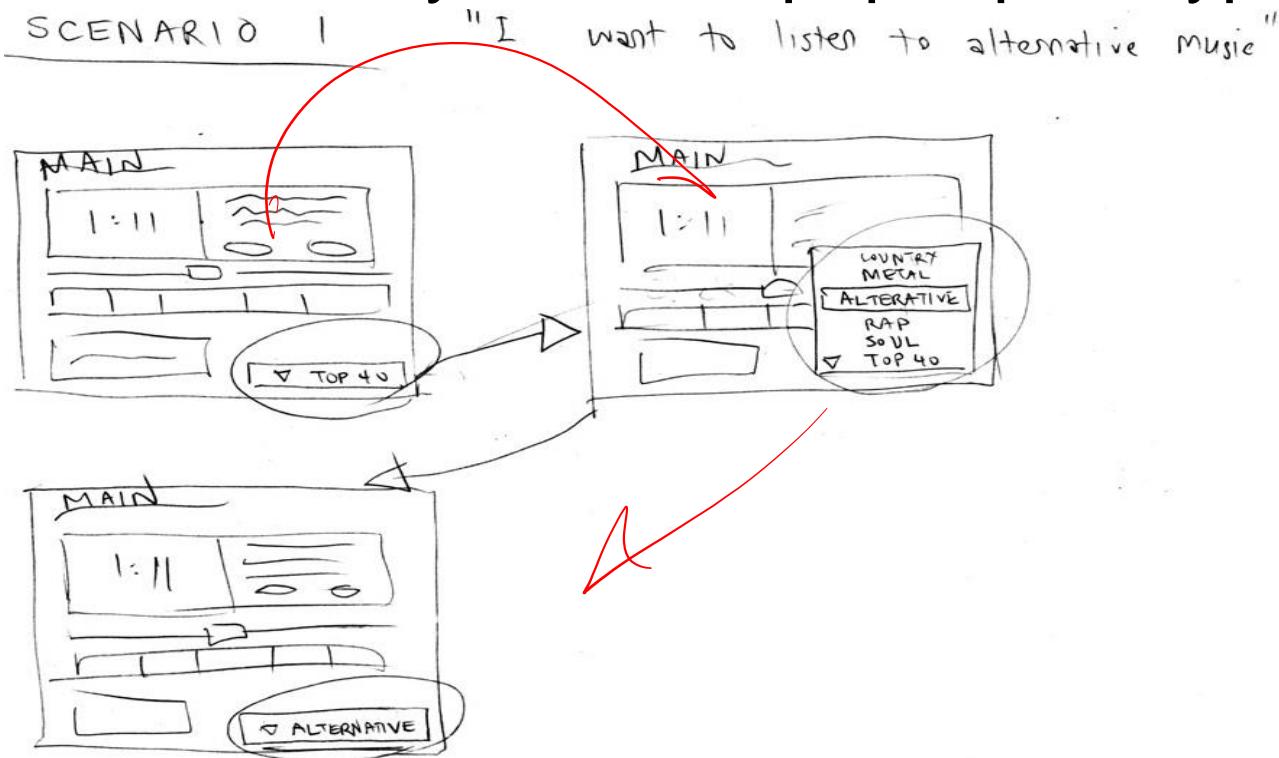
EBON (e,f): Elevator e button ON at floor f .

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



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

Traceability
Matrix

2. Traceability: Non-Functional requirements

- Isolate each non-functional requirement.
- 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

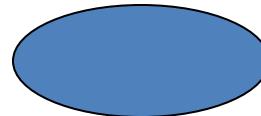
The Use Case Model

- 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

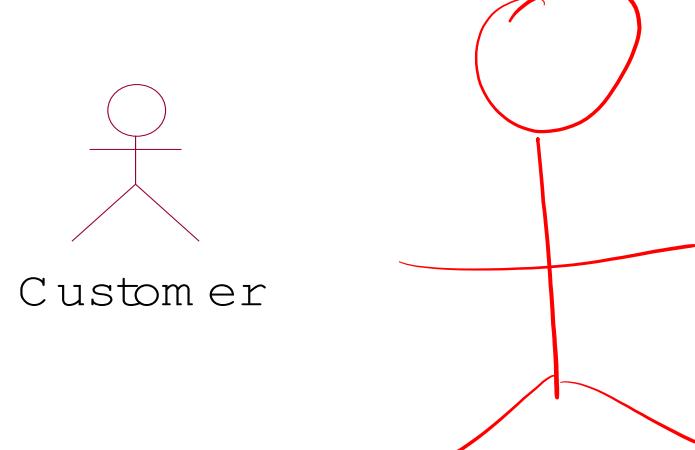
- UML icon for use case



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:



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

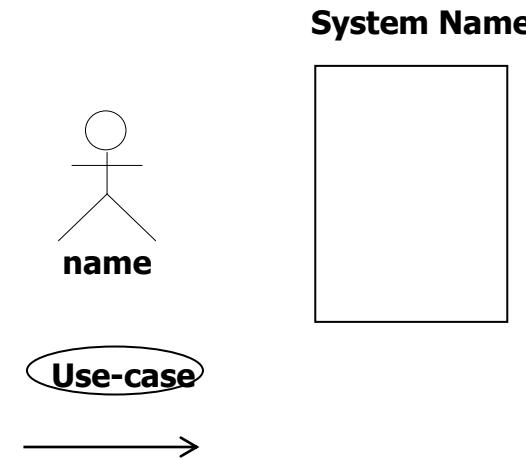
Use-Case Modeling

- 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



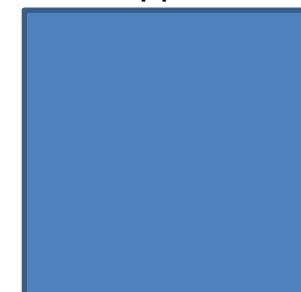
System

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.

Actor goal list

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) *Manage user*

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*)

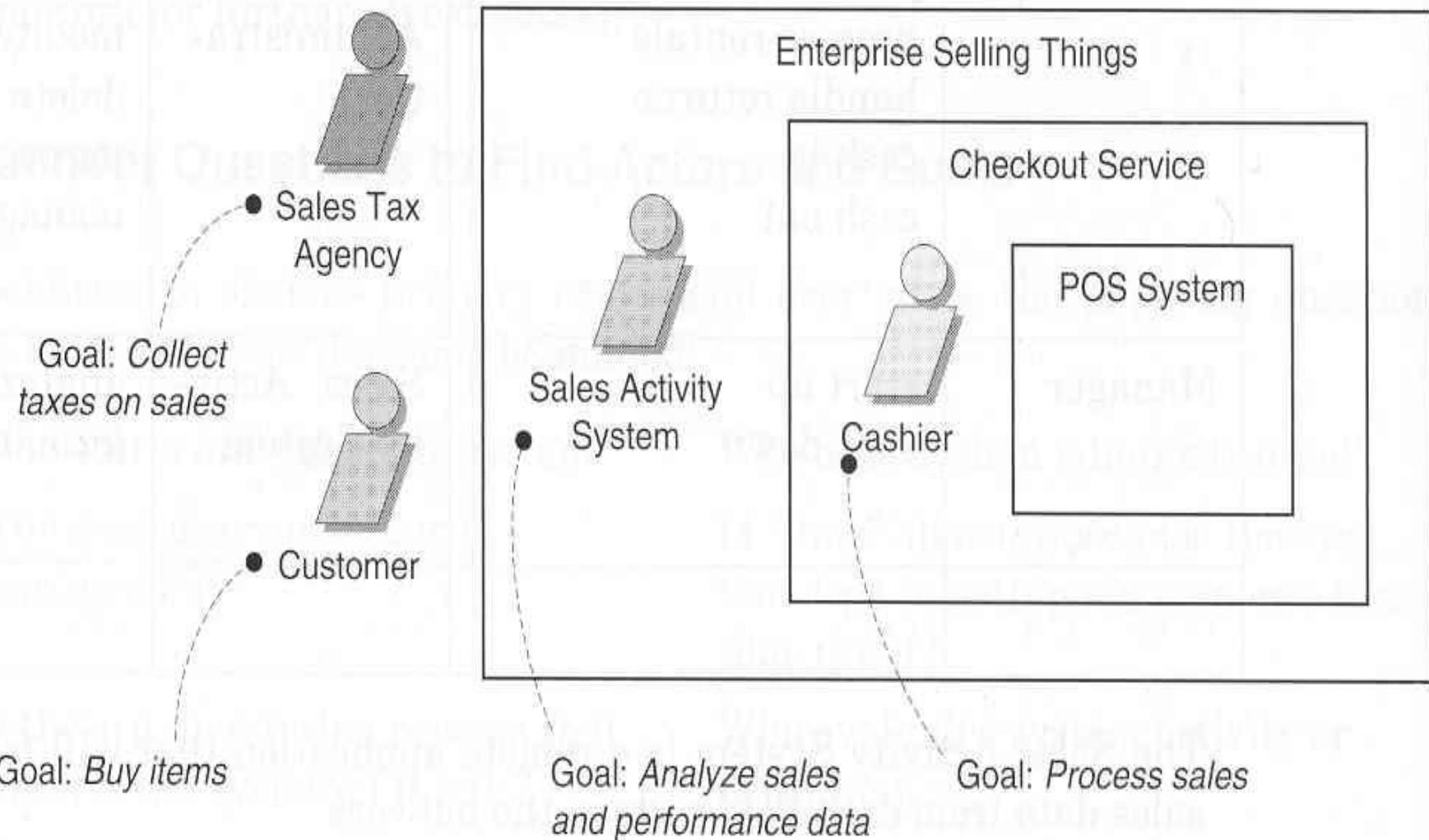
Exception to EBP

- 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

Innovate Achieve

lead



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

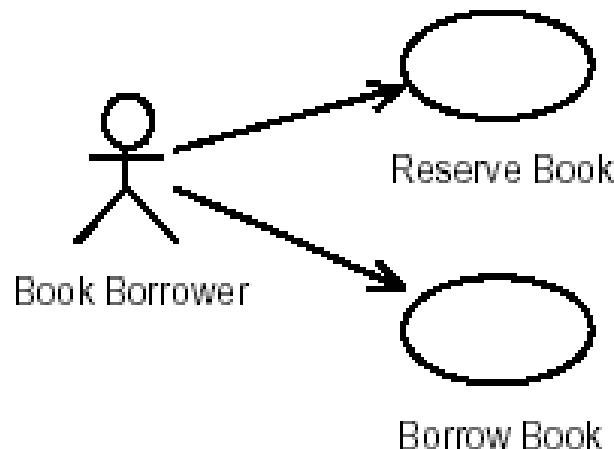
High level vs. Low Level Use cases [2]

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.

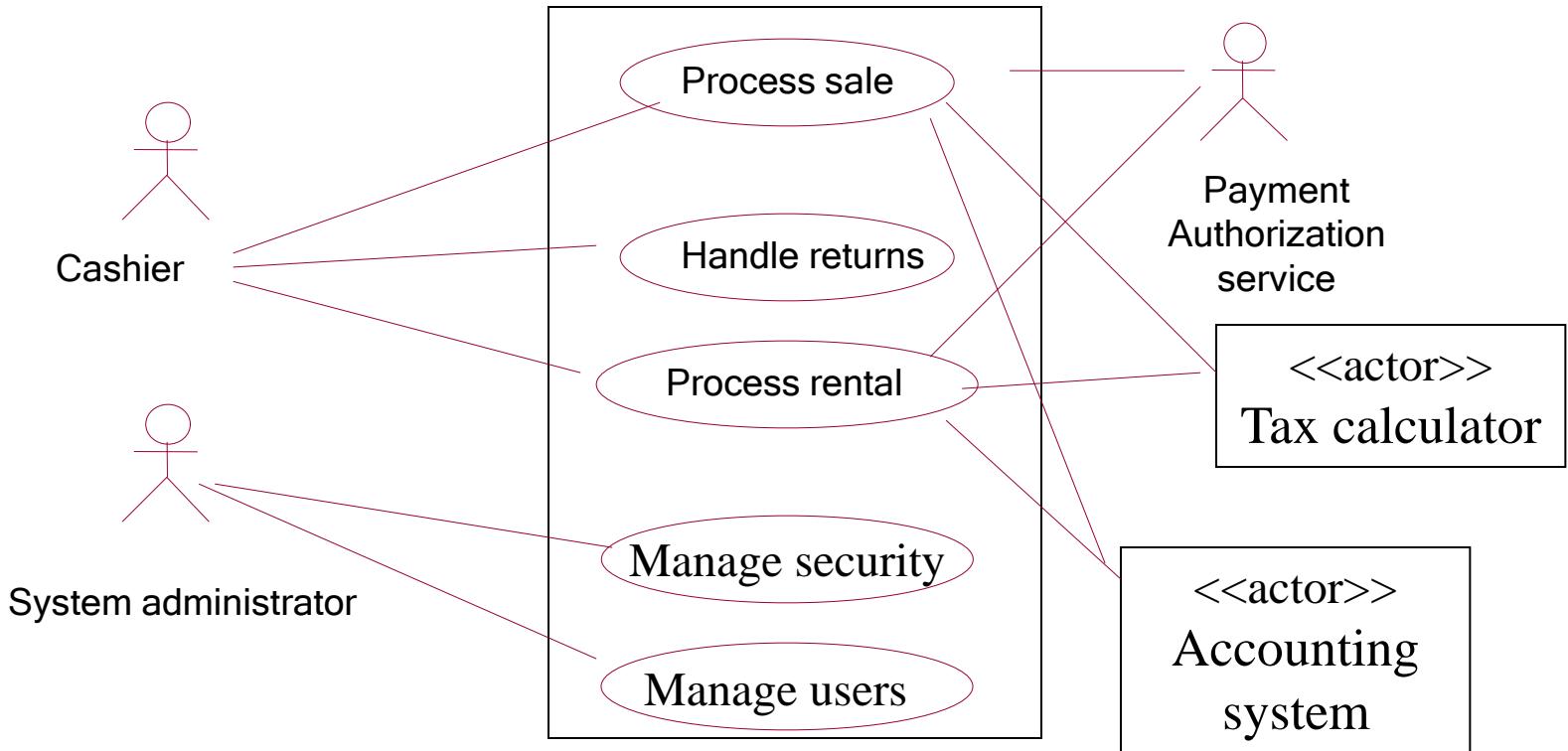
- *Use case diagrams* in *UML* are very easy to understand without knowing *UML* in detail!
 - *Actors* (external users of the system - also includes other systems)
 - *Use cases* (how the system can be used by the user)

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

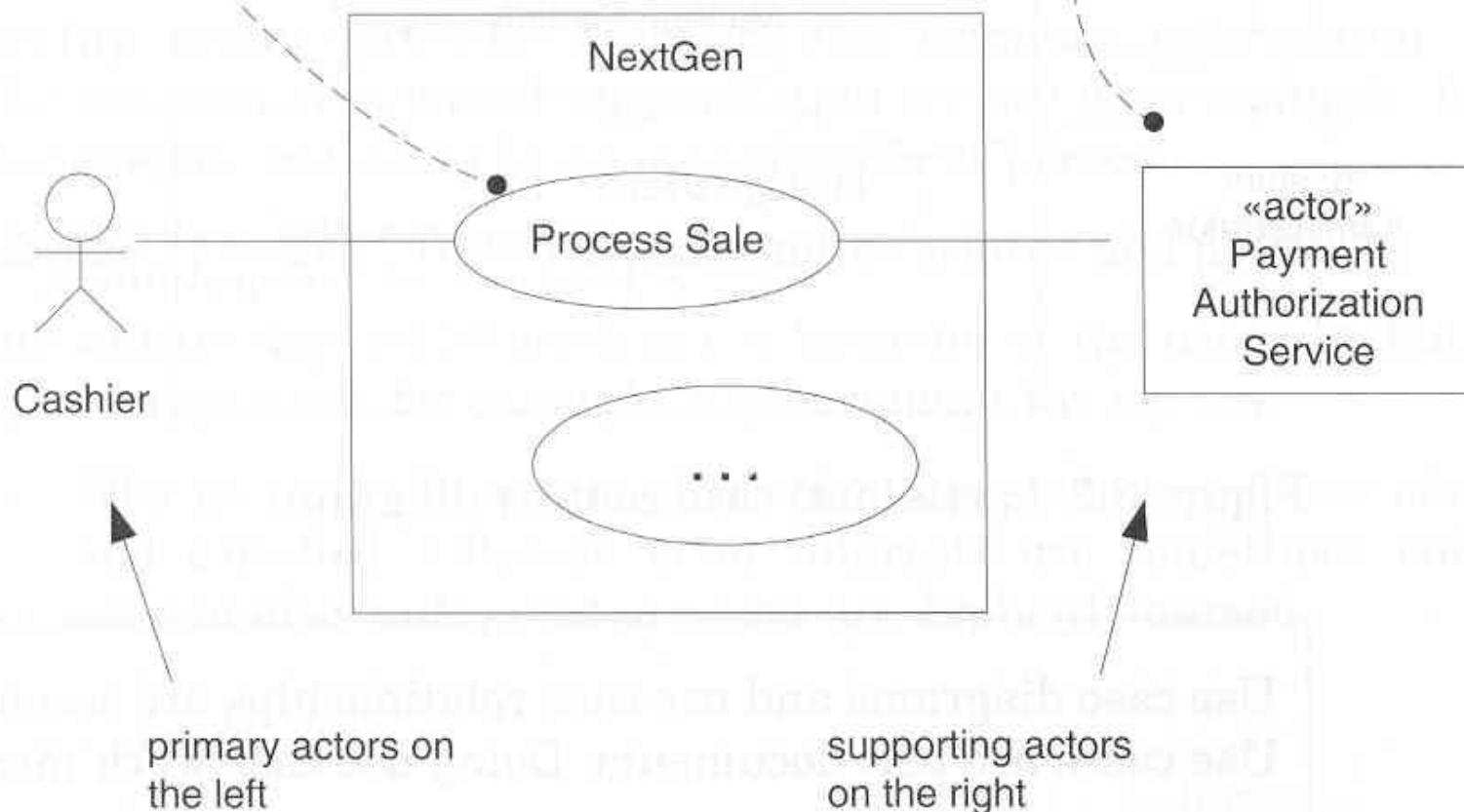
- Example: High, Medium, Low
- Example: Must have, Essential, Nice to have

- 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.

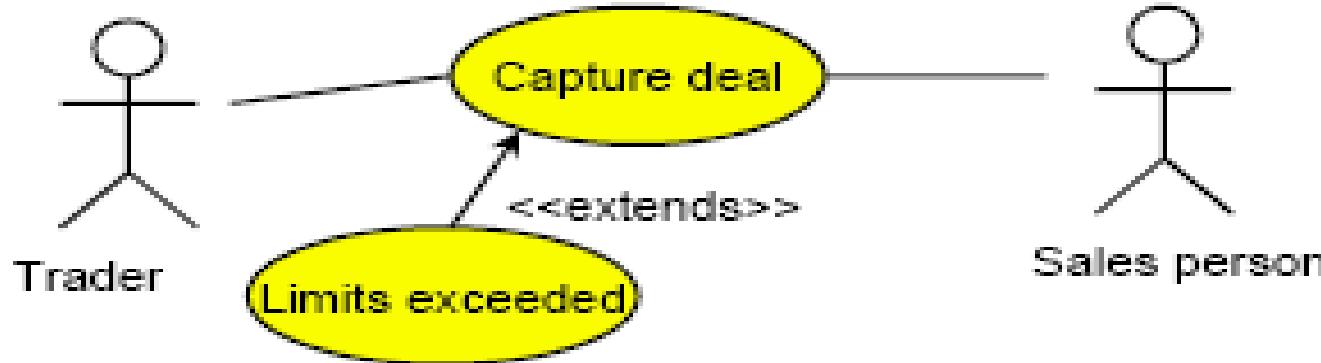
Don't worry about finding all the details

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

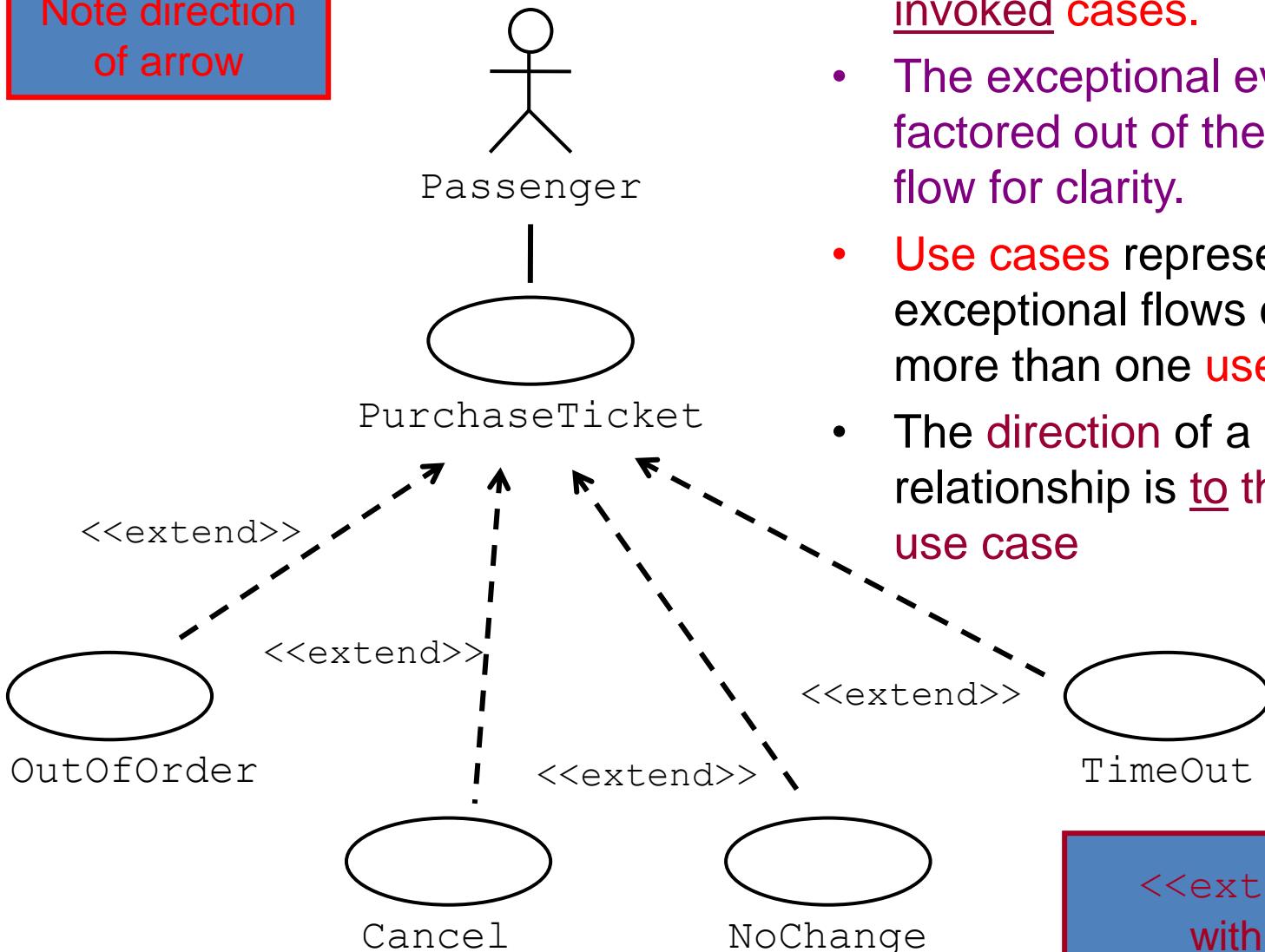
Extensions are used instead of modeling every **extra** case by a single **use case**.

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**.



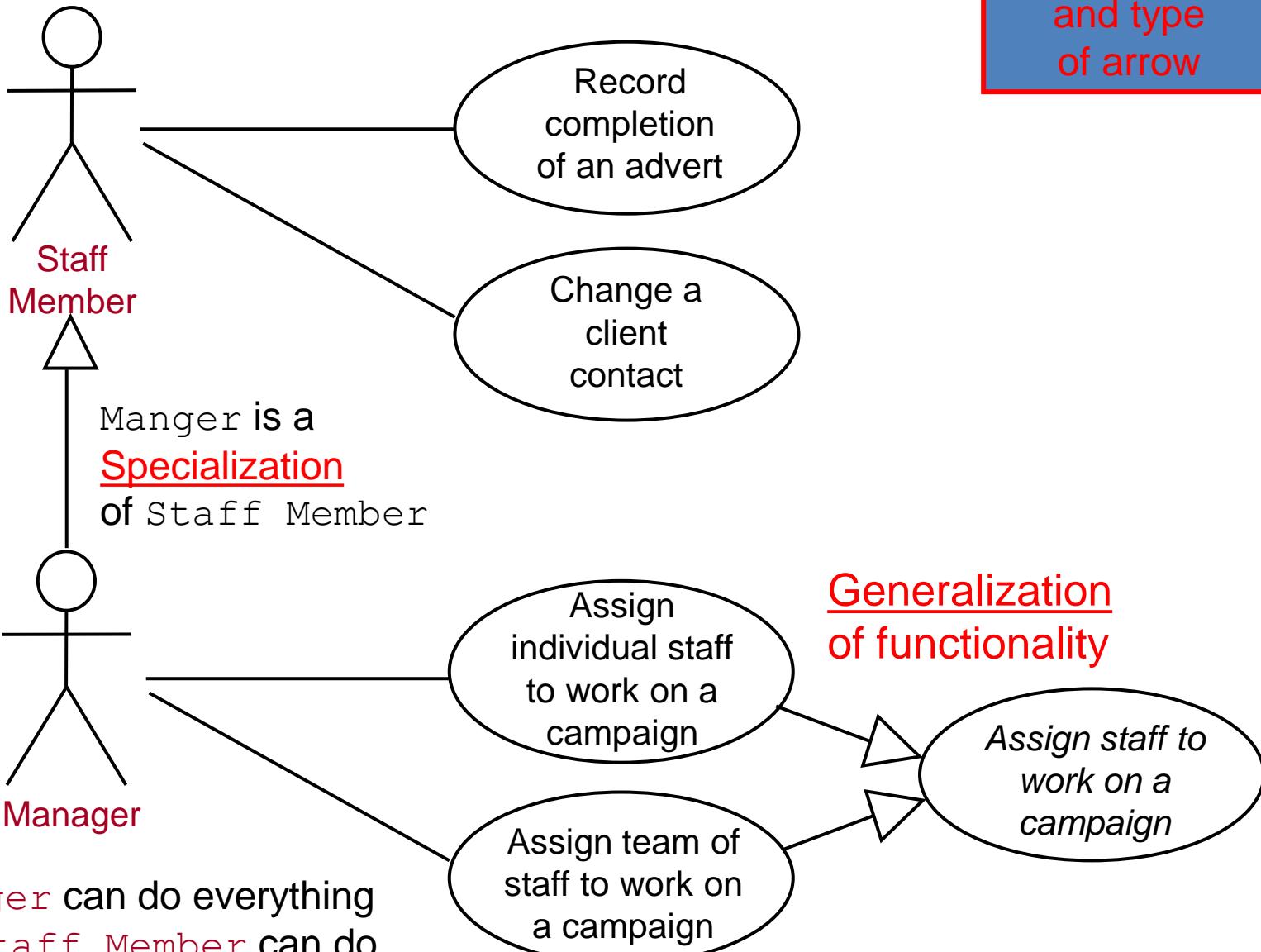
Include and Extend

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.

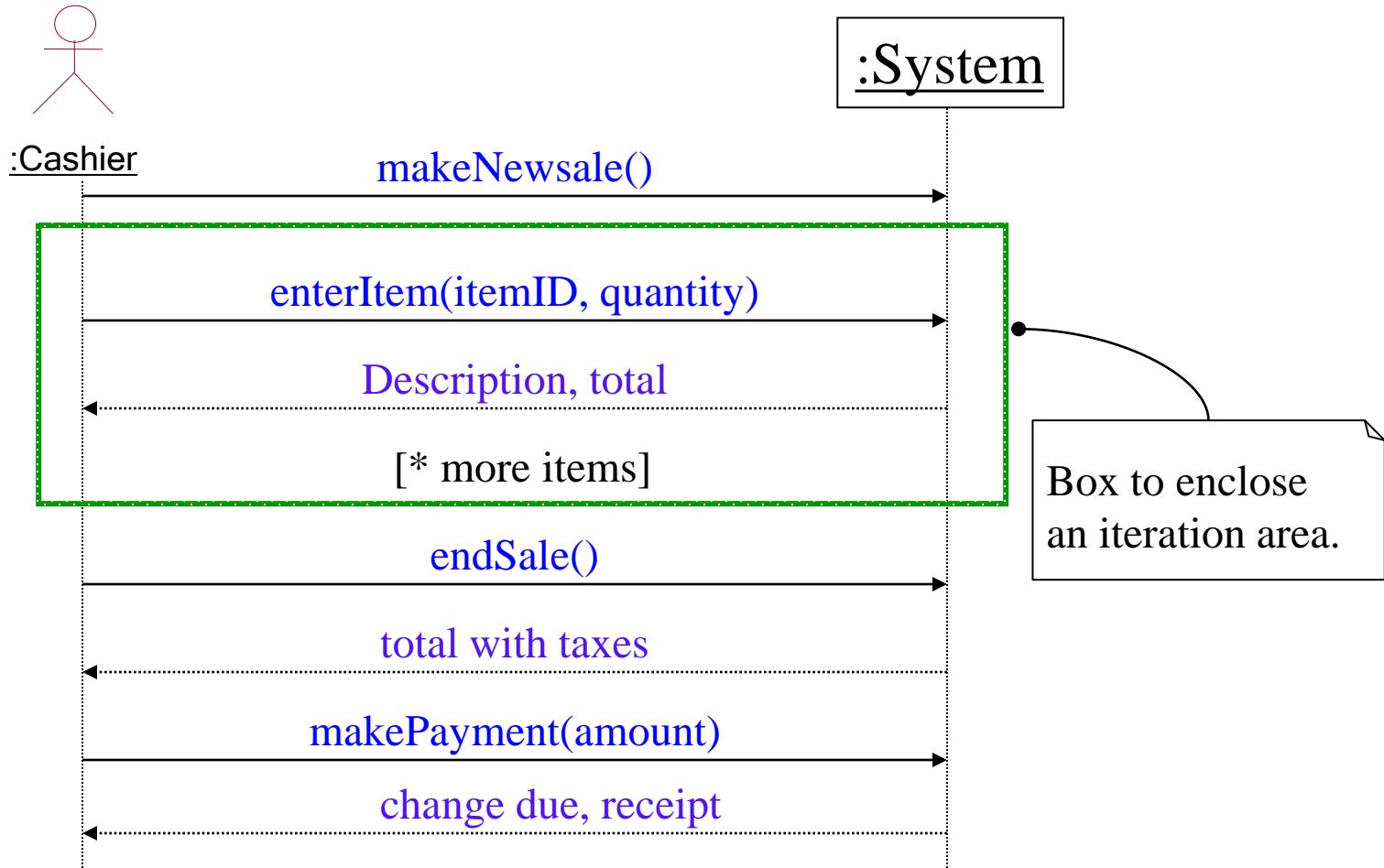
System Behavior and System Sequence Diagrams (SSDs)

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

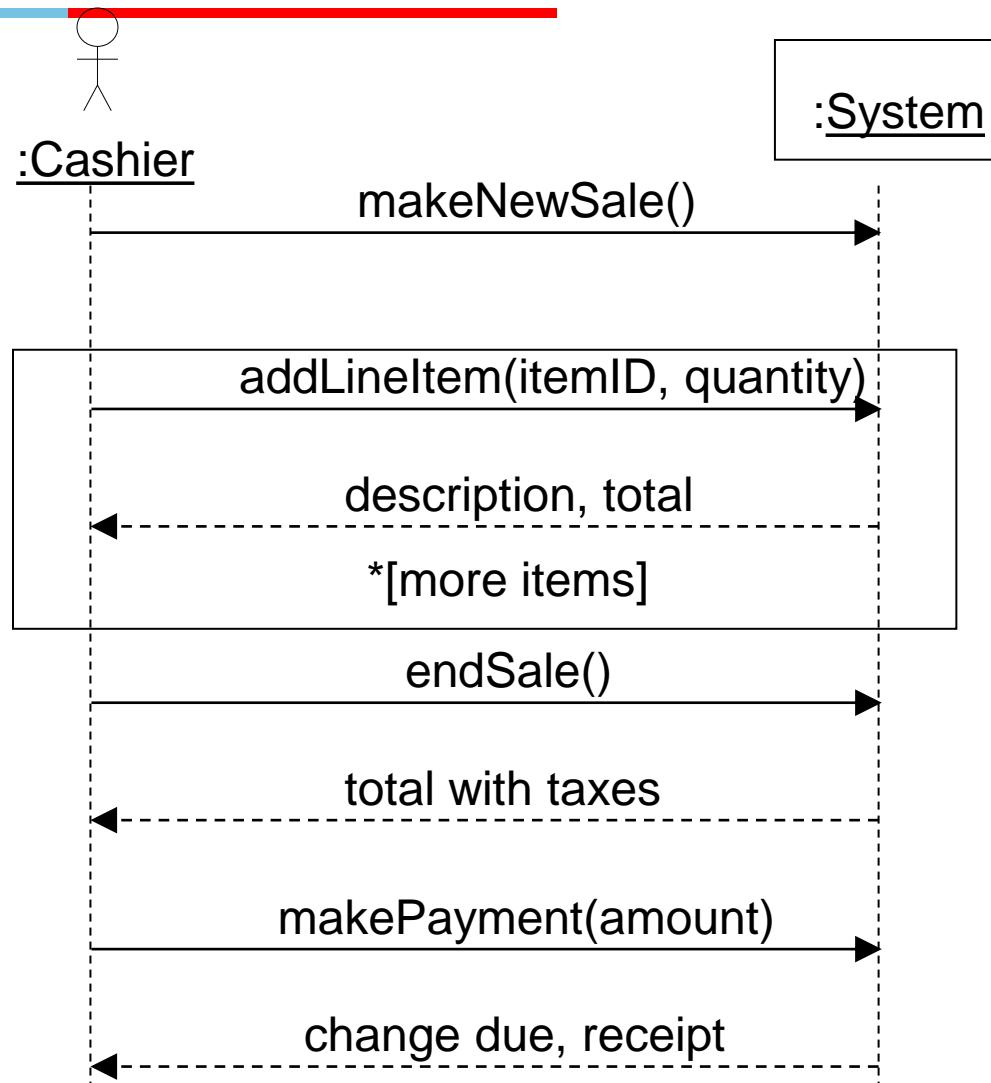
Process Sale scenario



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]

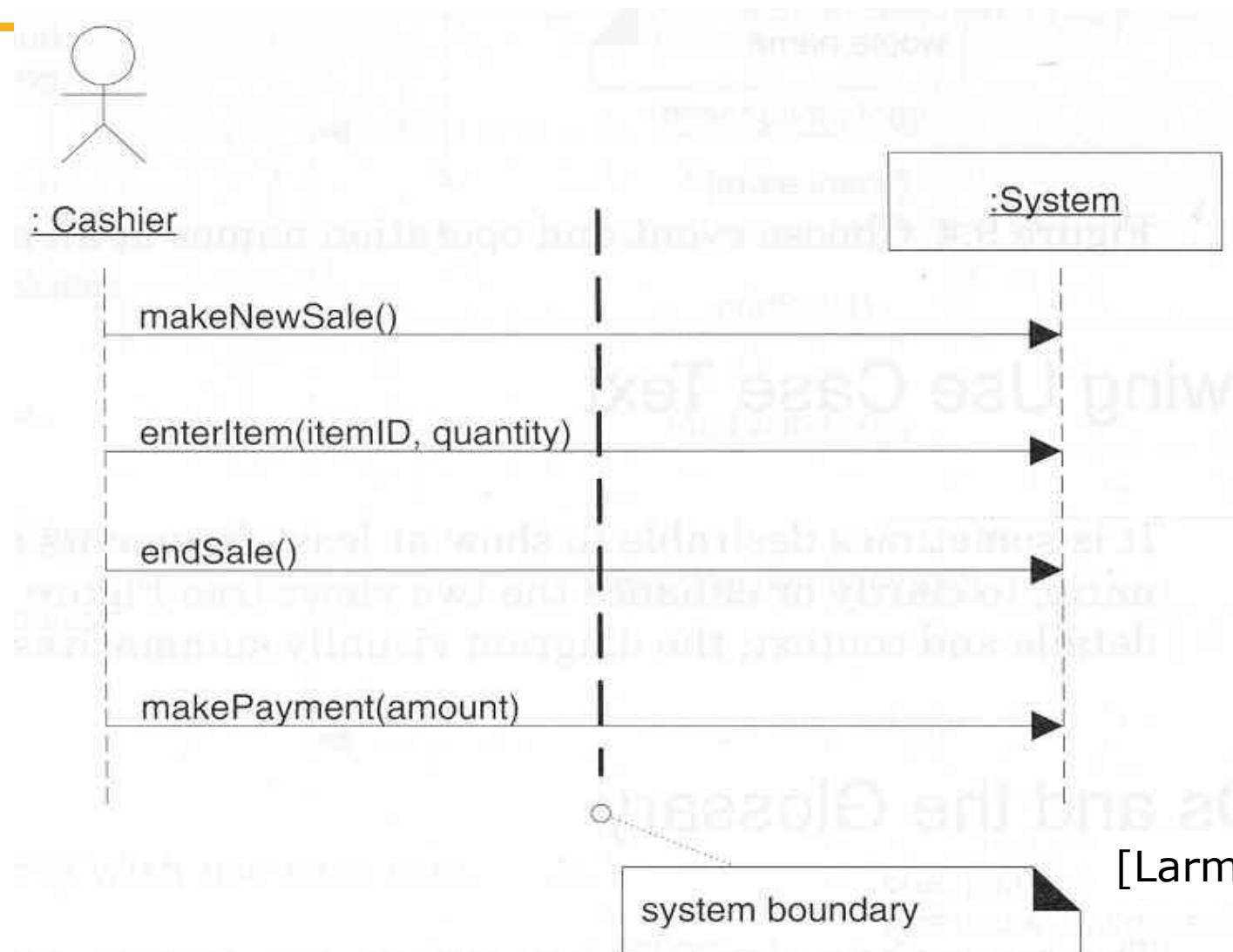
- One diagram depicts one scenario. This is the *main success* scenario.
- Frequent or complex alternate scenarios could also be illustrated.
- A system is treated as a *black box*.
- SSD is often accompanied by a textual description of the scenario to the left of the diagram.

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



[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.

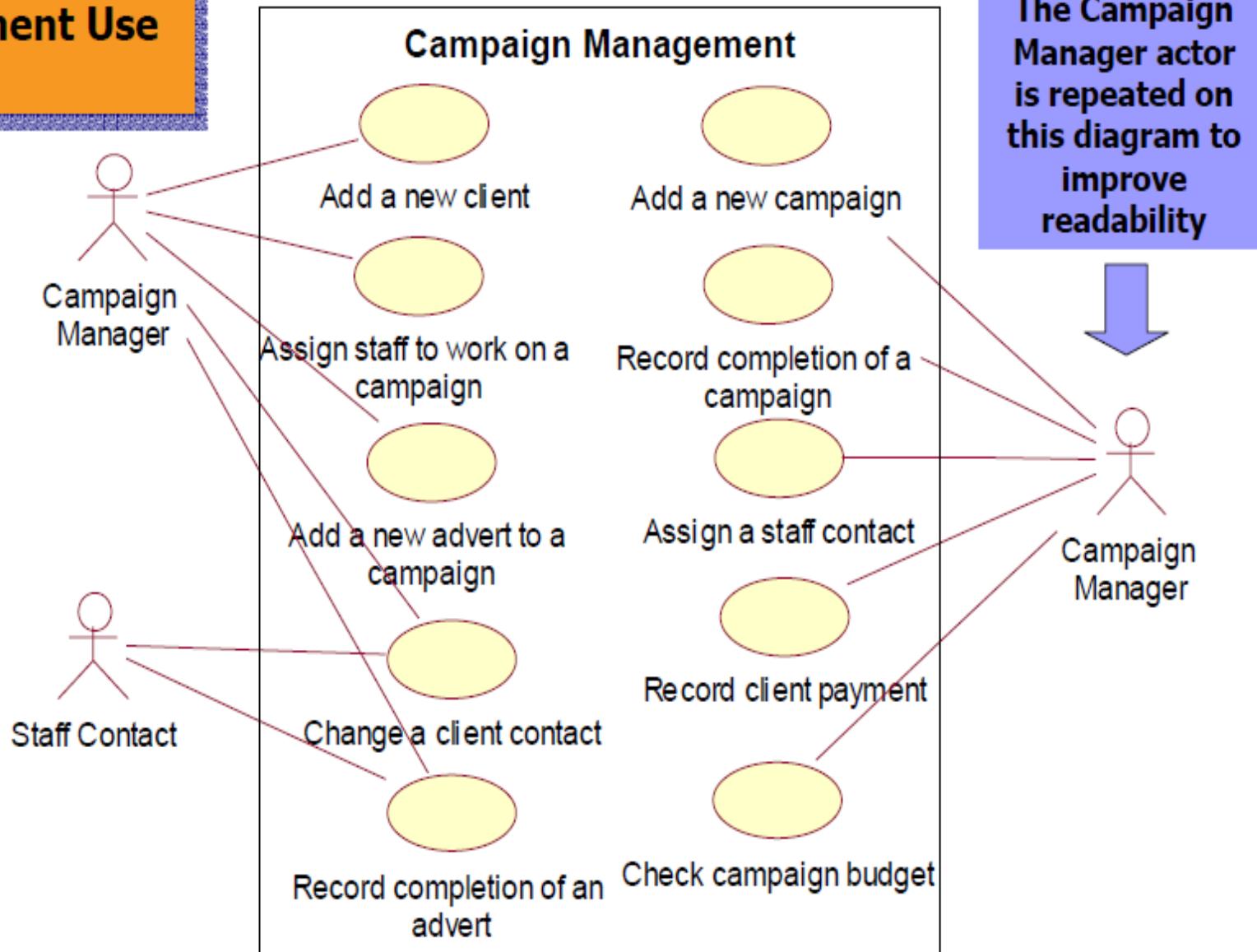
One member of the creative team manages each campaign. Staff may be working on more than one project at a time. When a campaign starts, the manager responsible estimates the likely cost of the client and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. 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





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

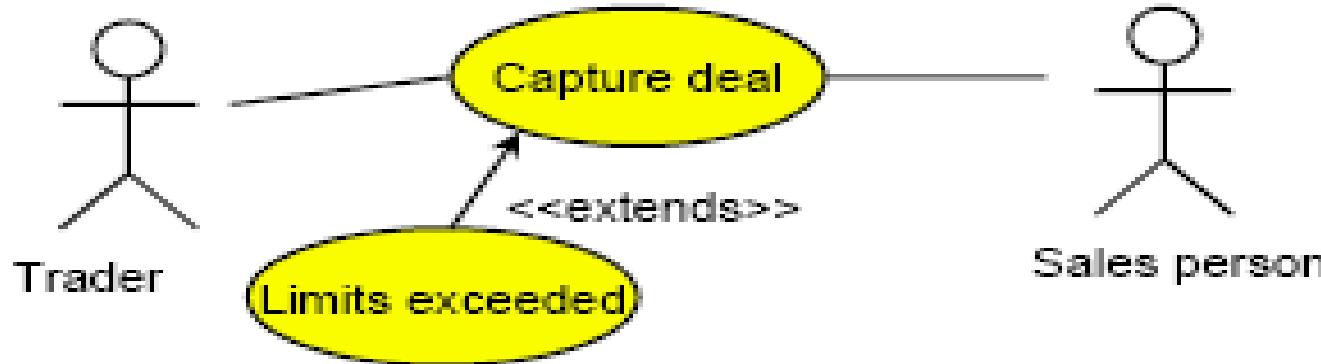
Create a written document for each Use Case

- 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

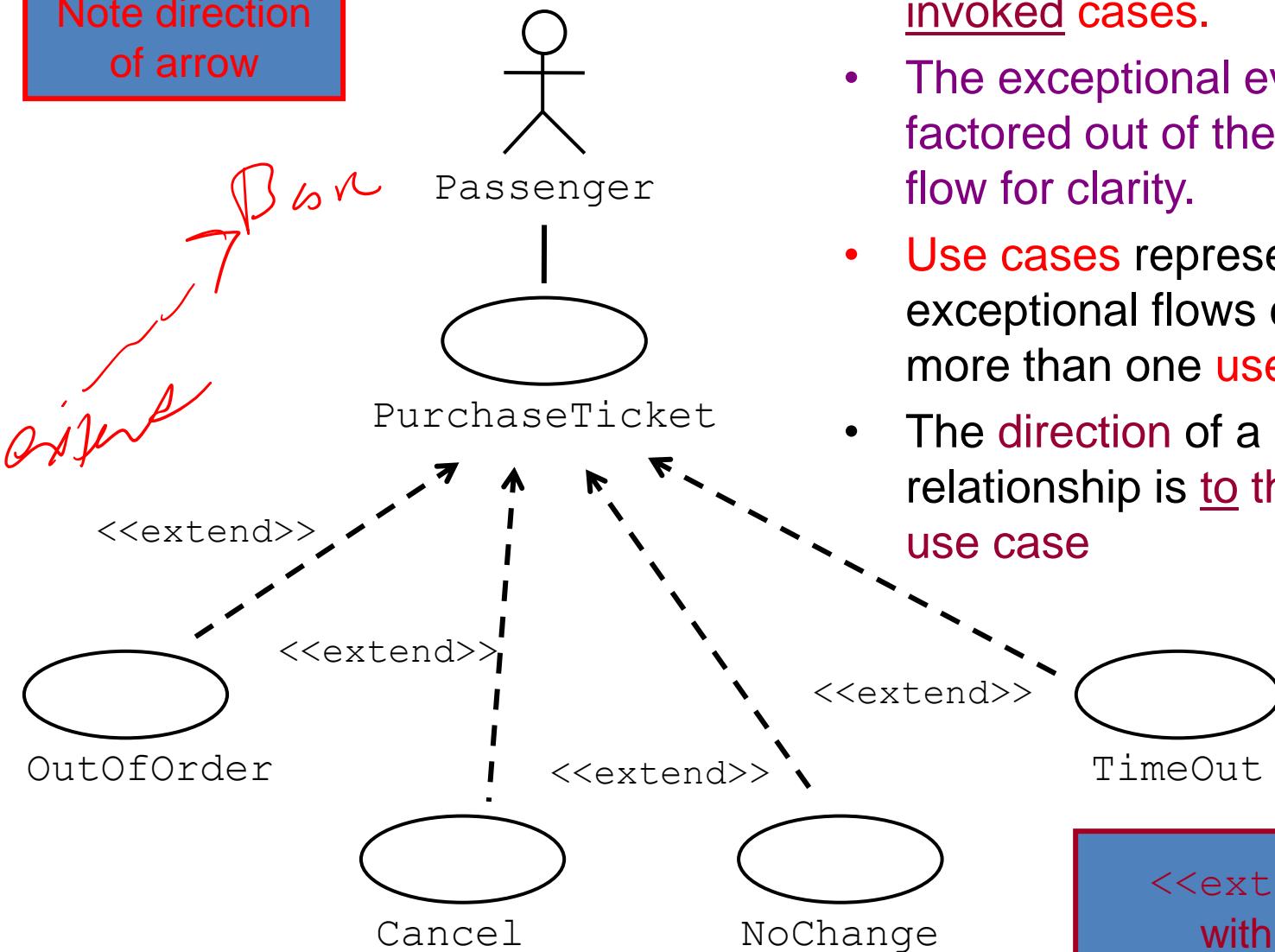
Extensions are used instead of modeling every **extra** case by a single **use case**.

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

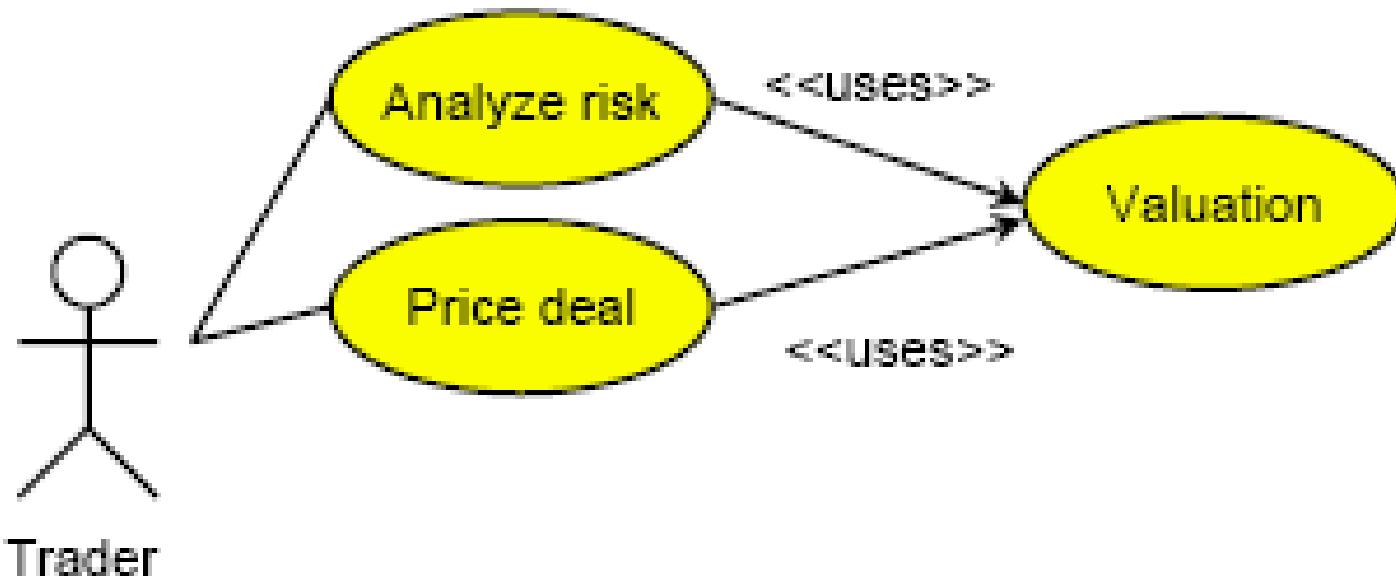


- **<<extend>>** relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one **use case**.
- The **direction** of a **<<extend>>** relationship is to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



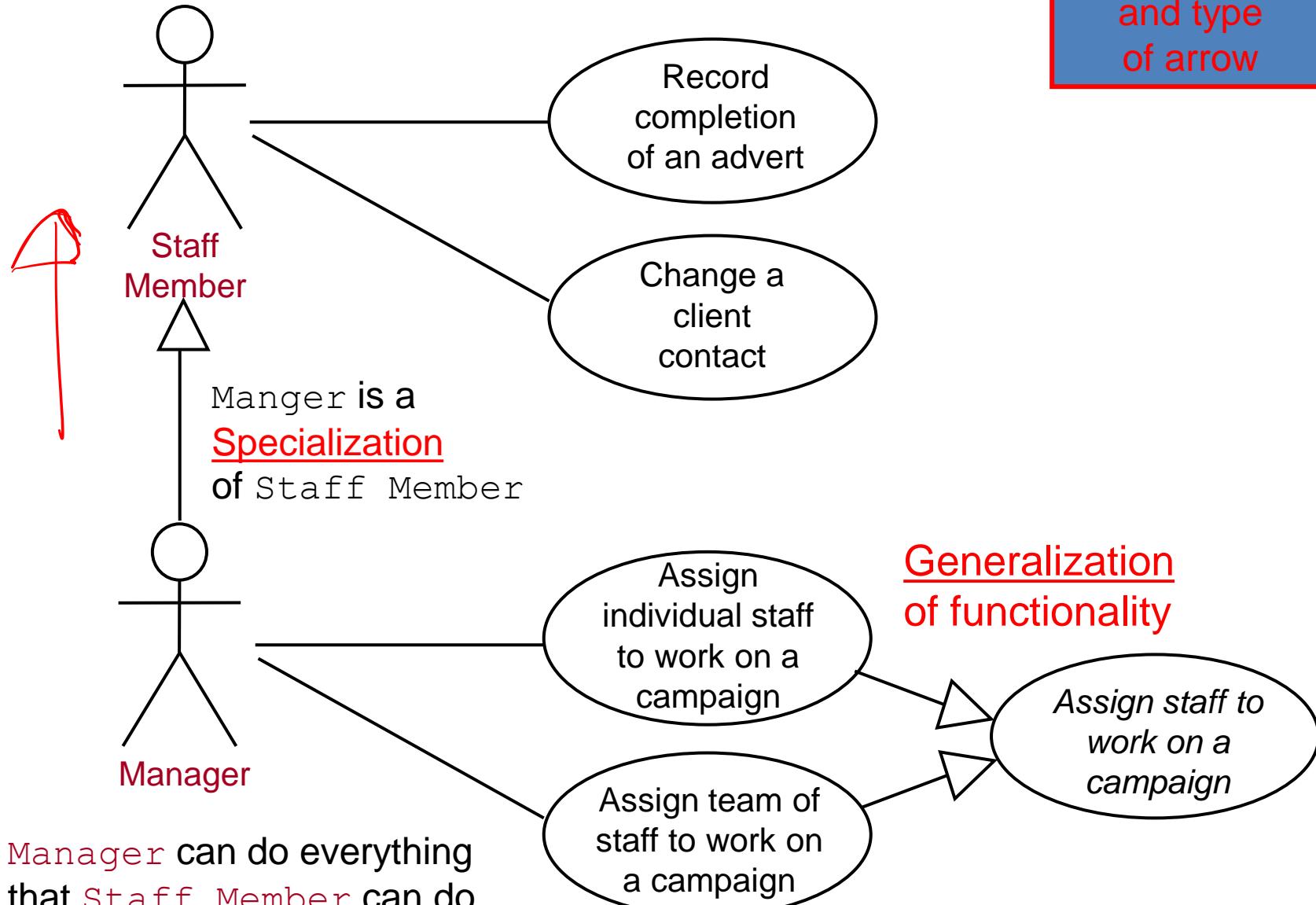
Include and Extend

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)

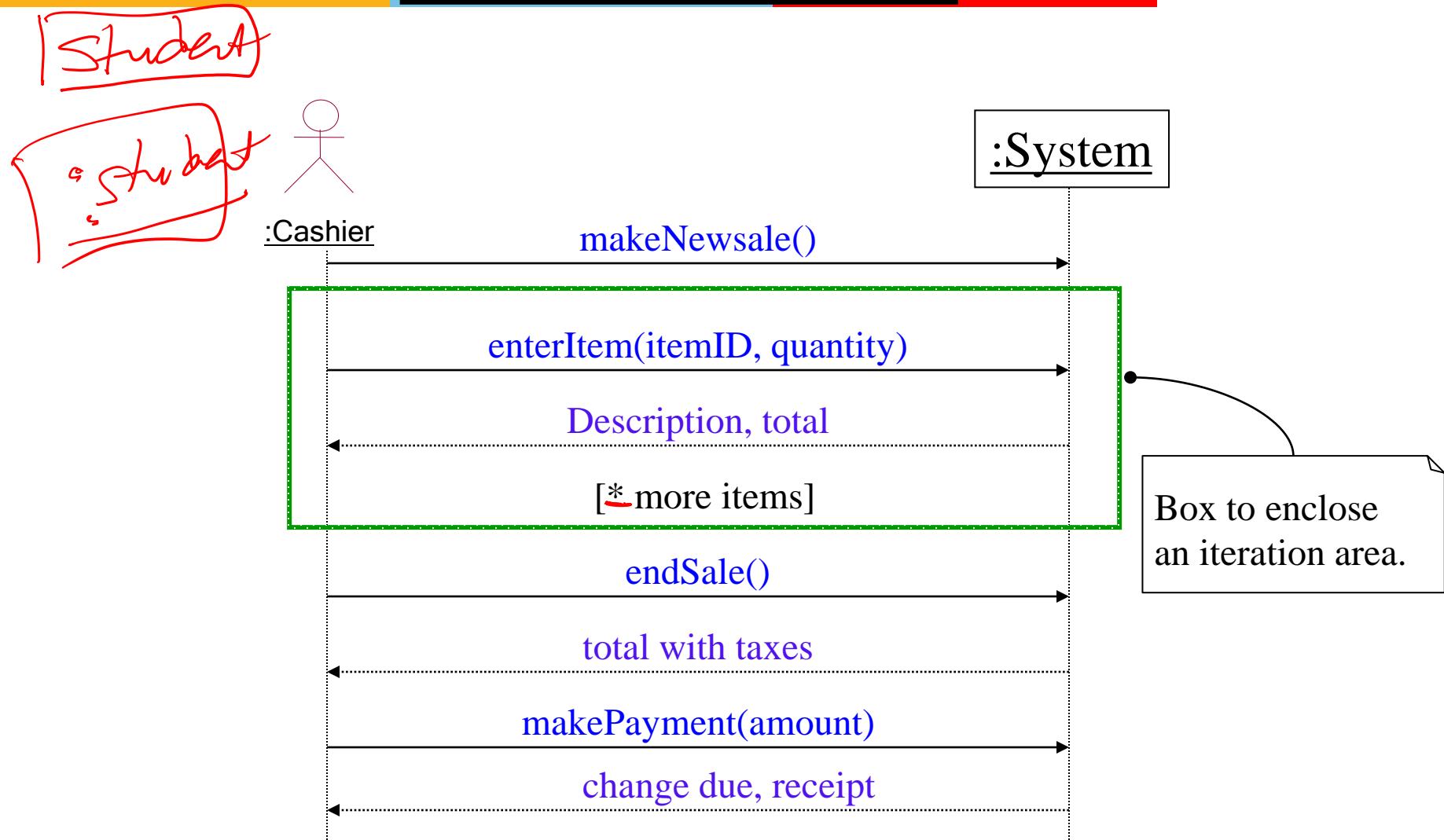
A system sequence diagram is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

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



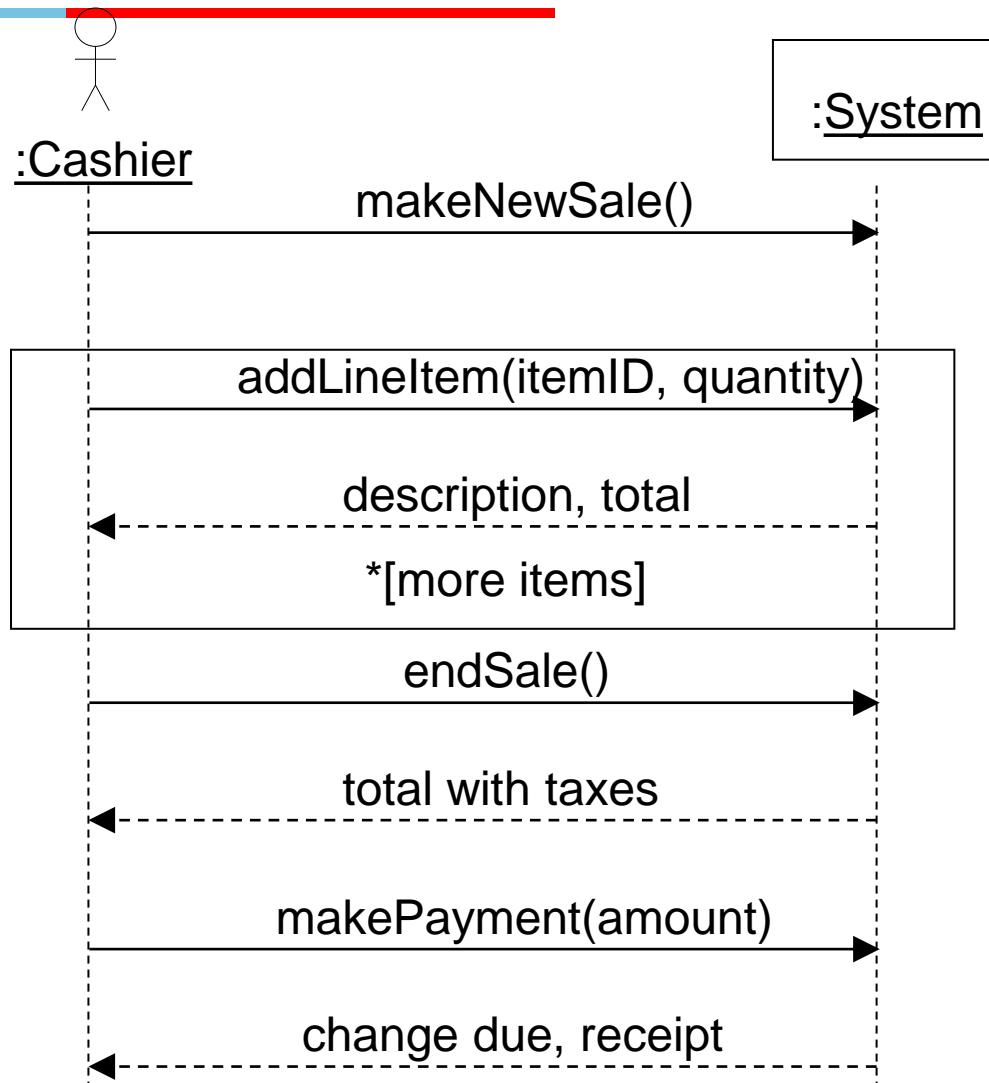
Process Sale scenario



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]

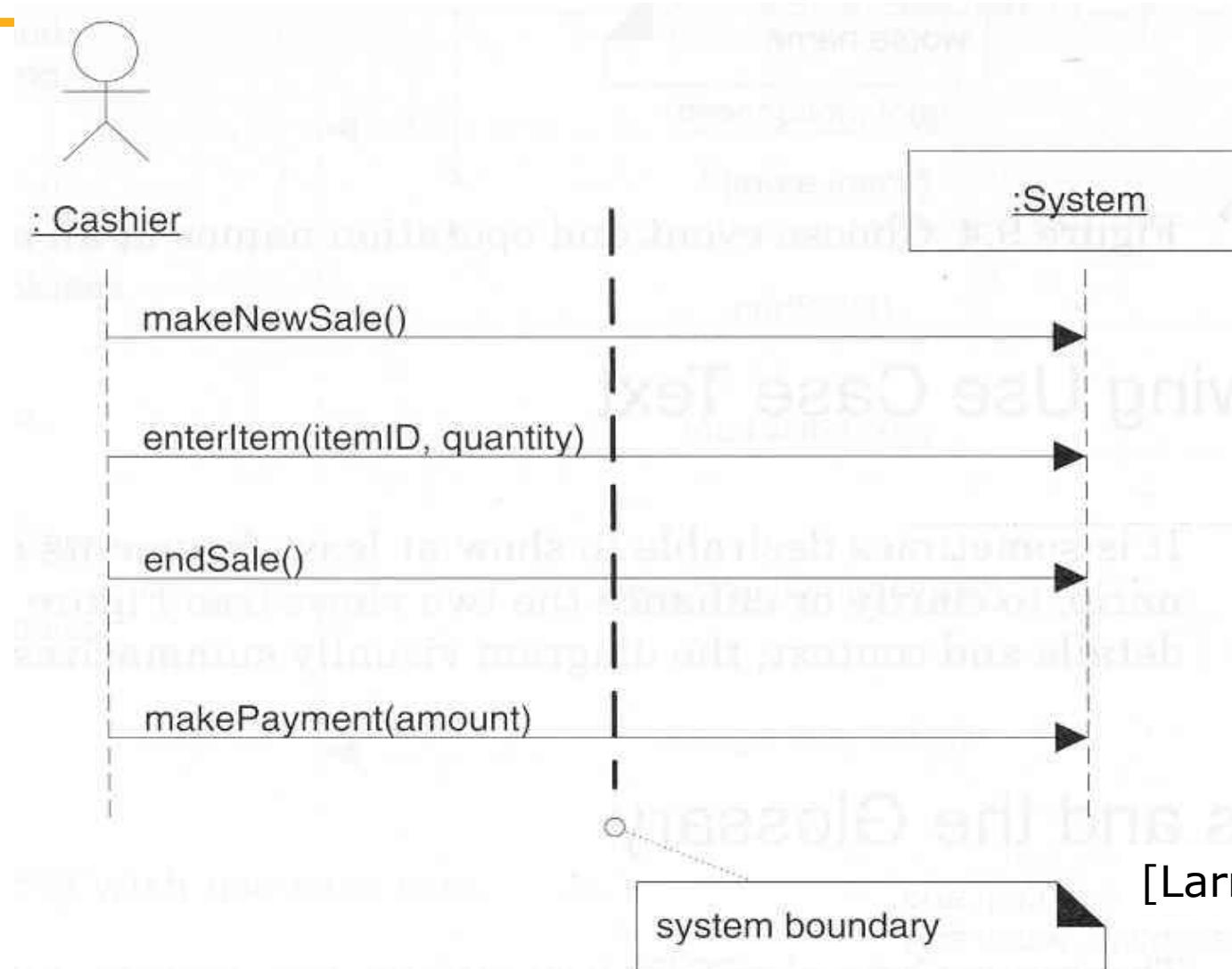
- One diagram depicts one scenario. This is the *main success* scenario.
- Frequent or complex alternate scenarios could also be illustrated.
- A system is treated as a *black box*.
- SSD is often accompanied by a textual description of the scenario to the left of the diagram.

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



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.

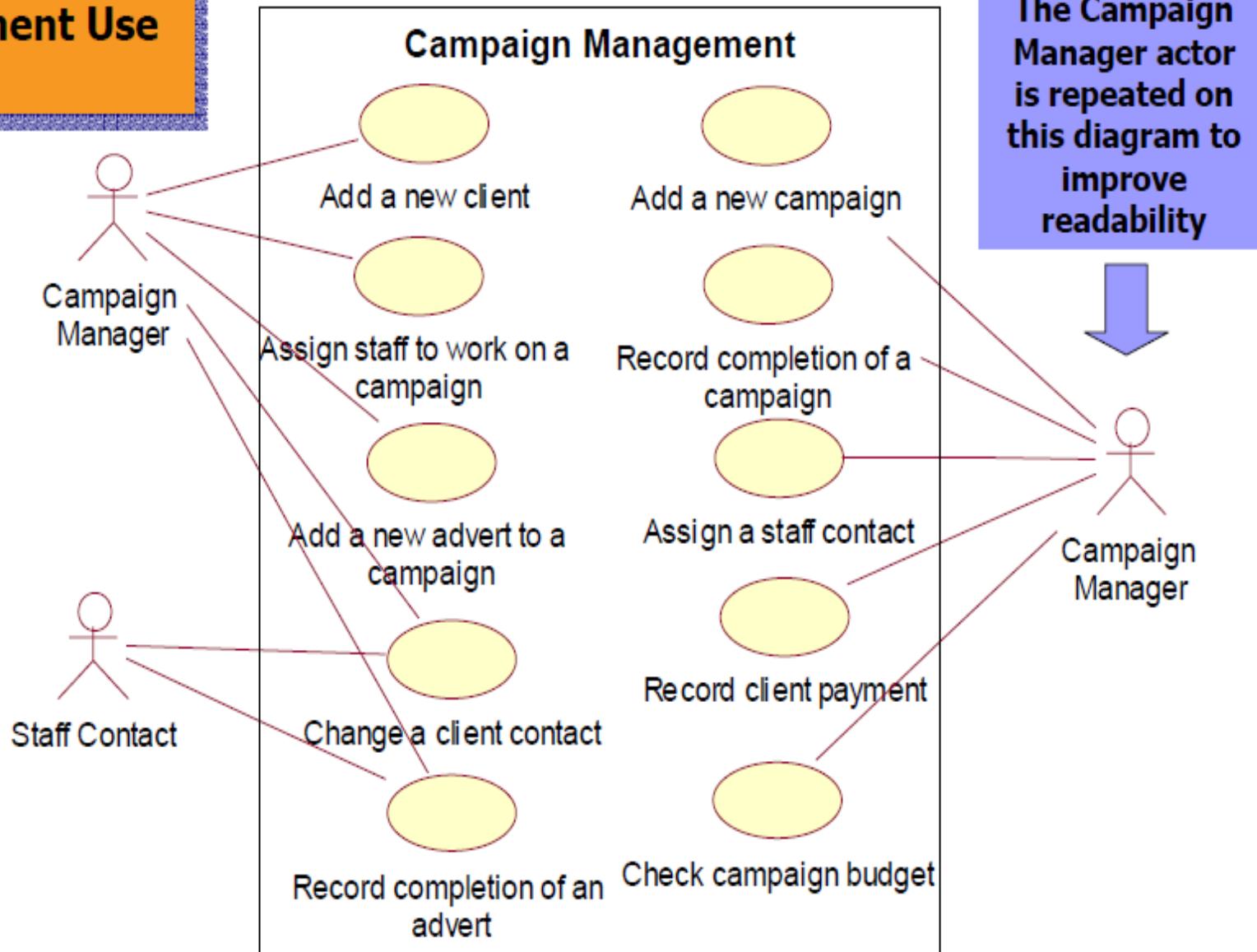
One member of the creative team manages each campaign. Staff may be working on more than one project at a time. When a campaign starts, the manager responsible estimates the likely cost of the client and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. 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

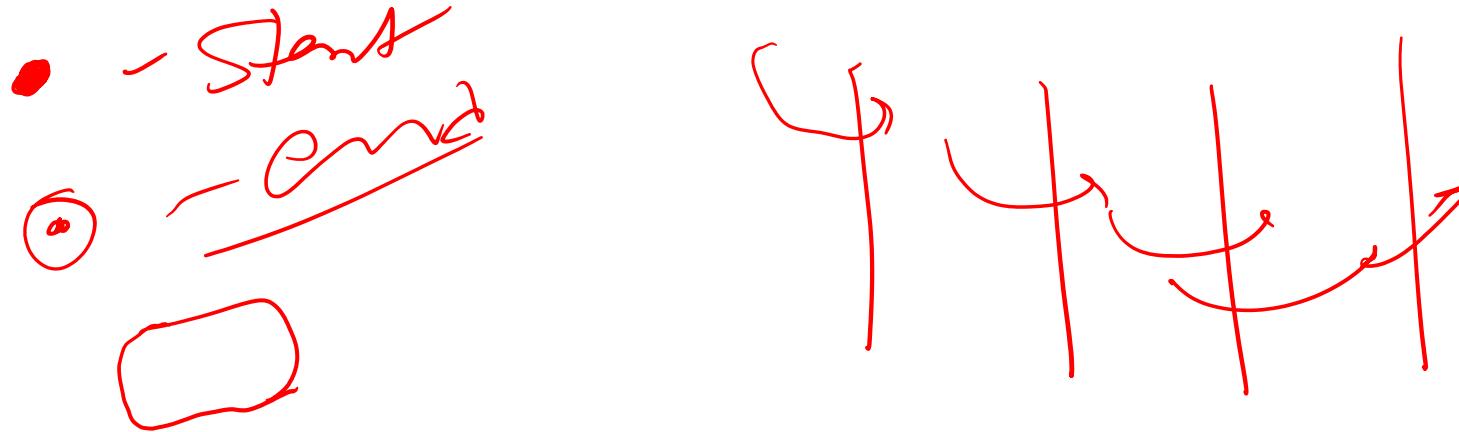
Be able to create functional models using activity diagrams. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

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

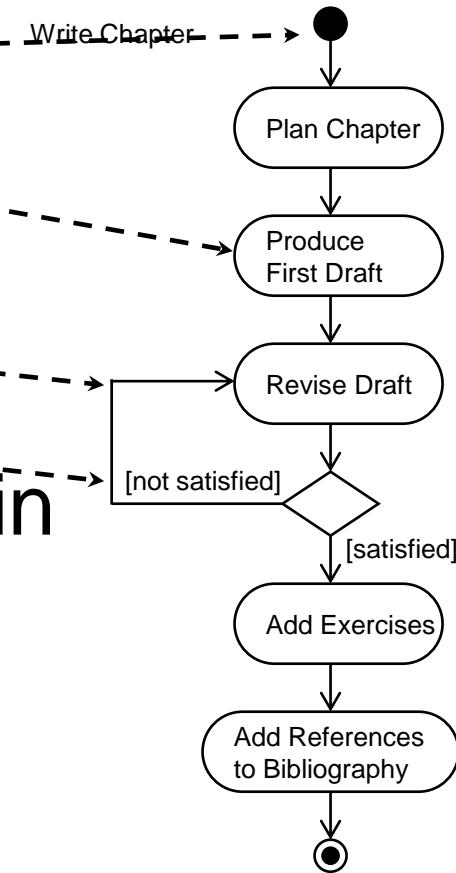
Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram.

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

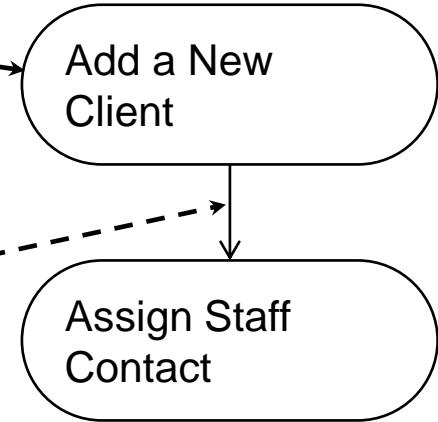
- UML diagrams consist of:
 - icons
 - two-dimensional symbols
 - paths
 - Strings
- UML diagrams are defined in the UML specification.



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

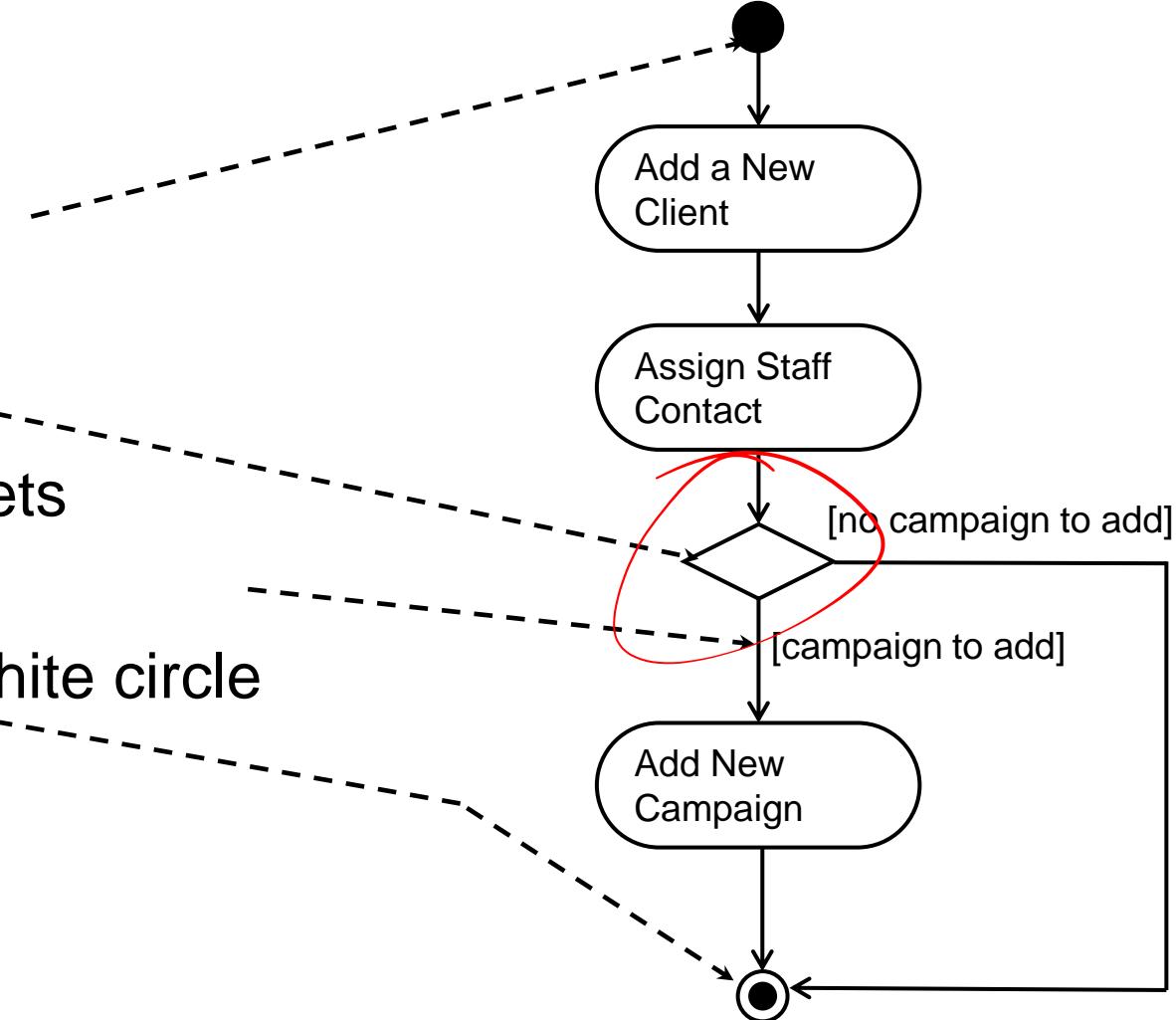
- diamond

Guard conditions

- in square brackets

Final state

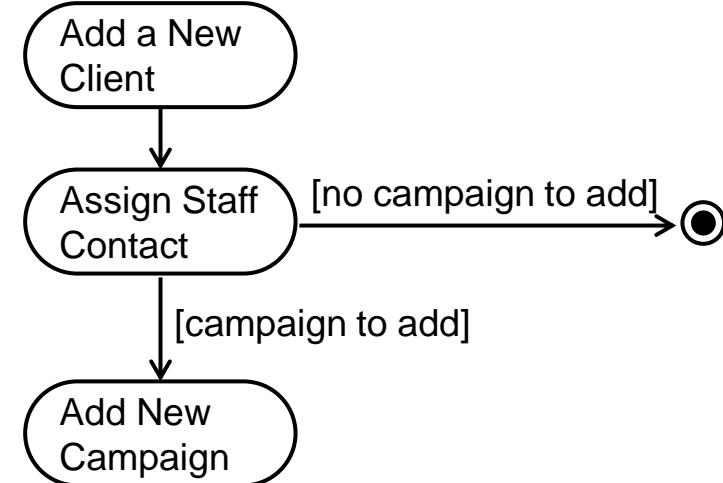
- black circle in white circle



Notation of Activity Diagrams

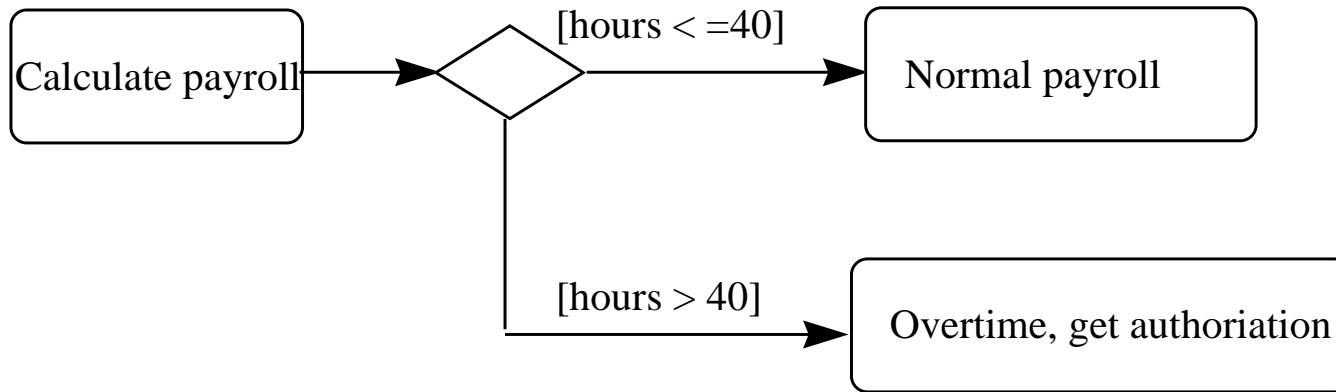
Alternative notation for branching:

- alternative transitions are shown leaving the activity with guard conditions

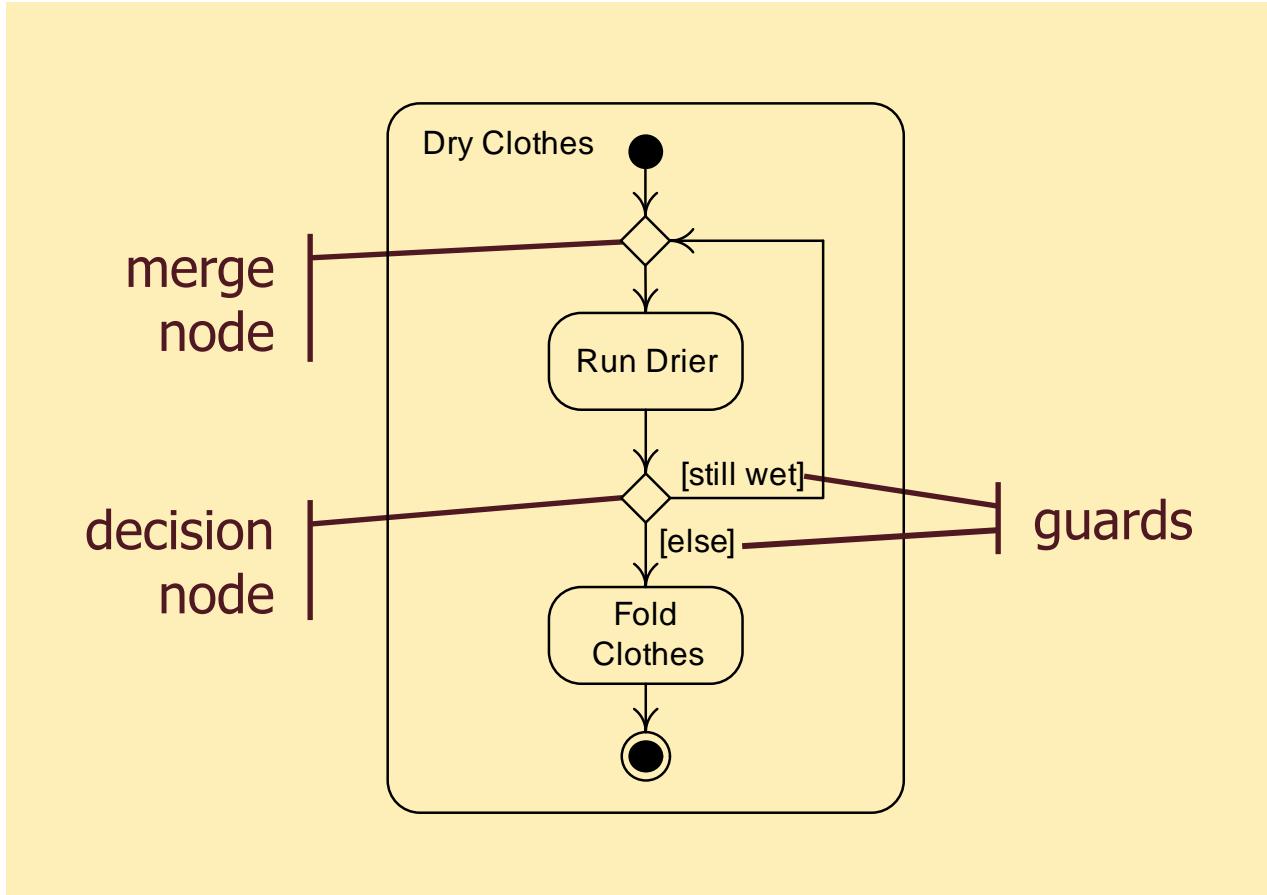


Note that guard conditions do not have to be mutually exclusive, but it is advisable that they should be

UML Activity Decision

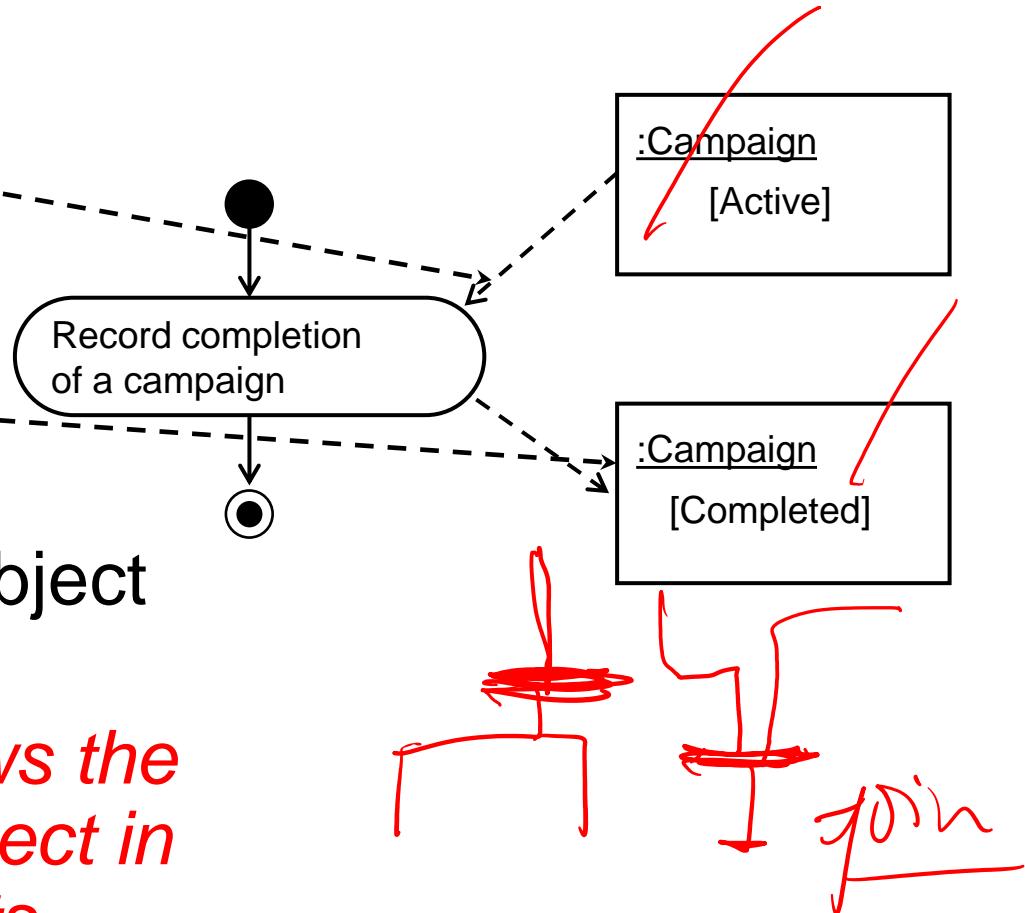


Branching Nodes



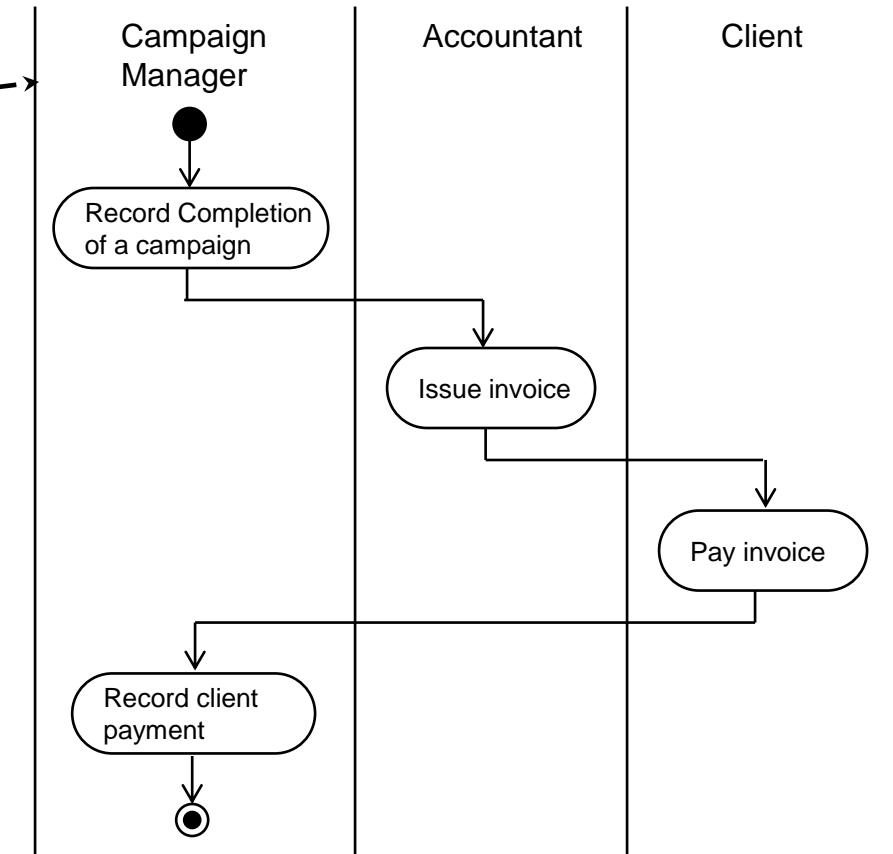
Notation of Activity Diagrams

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

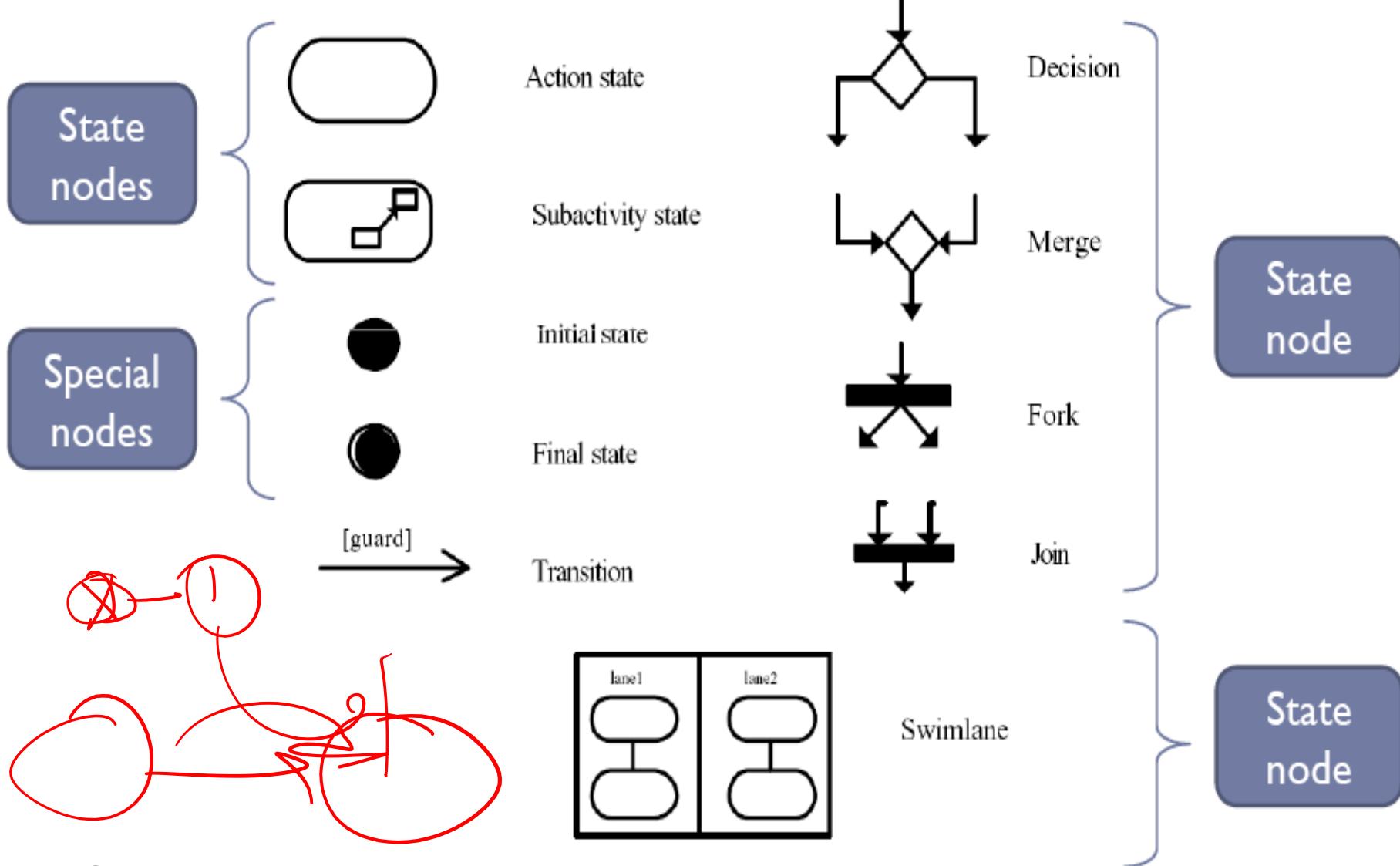


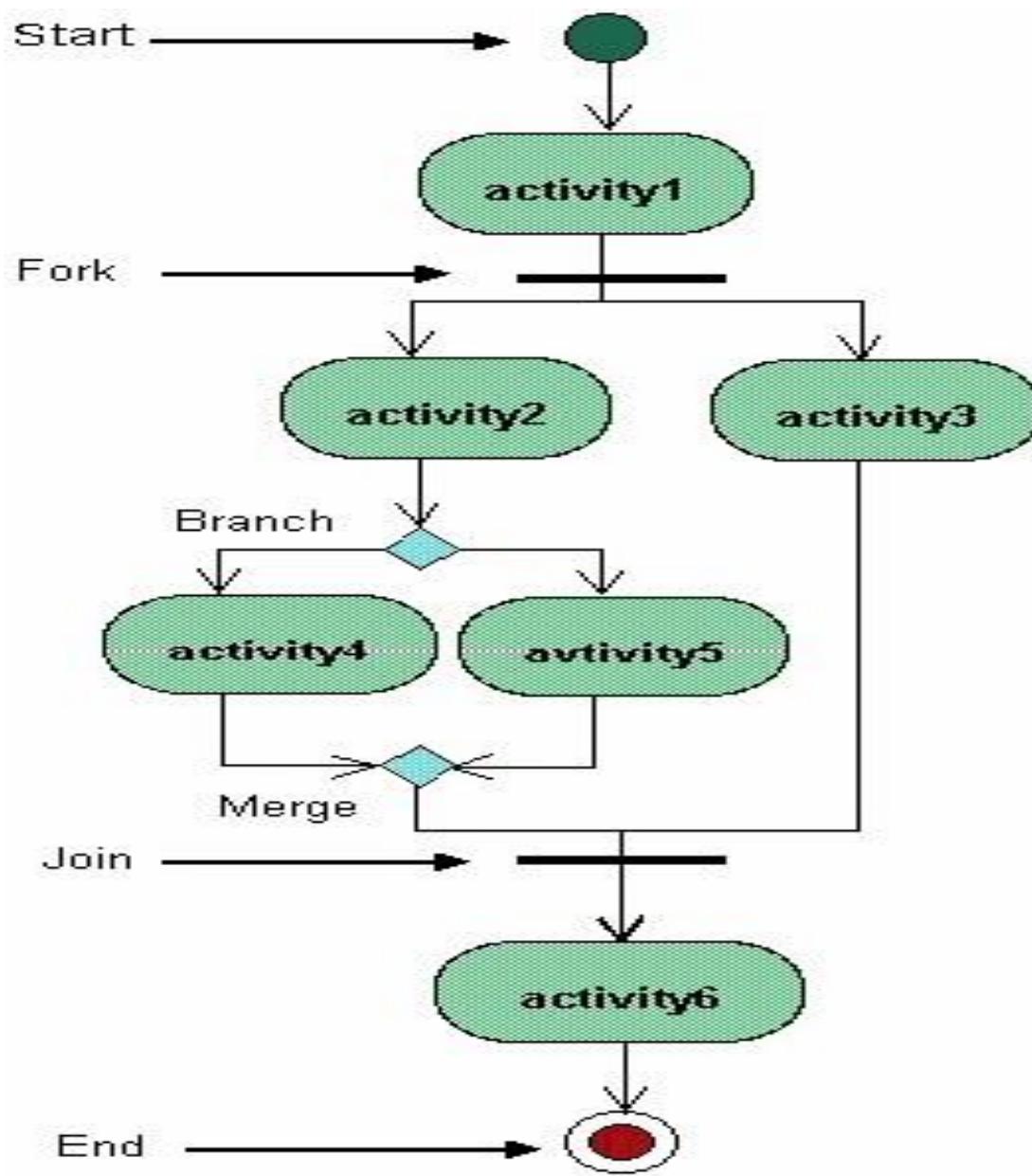
Notation of Activity Diagrams

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

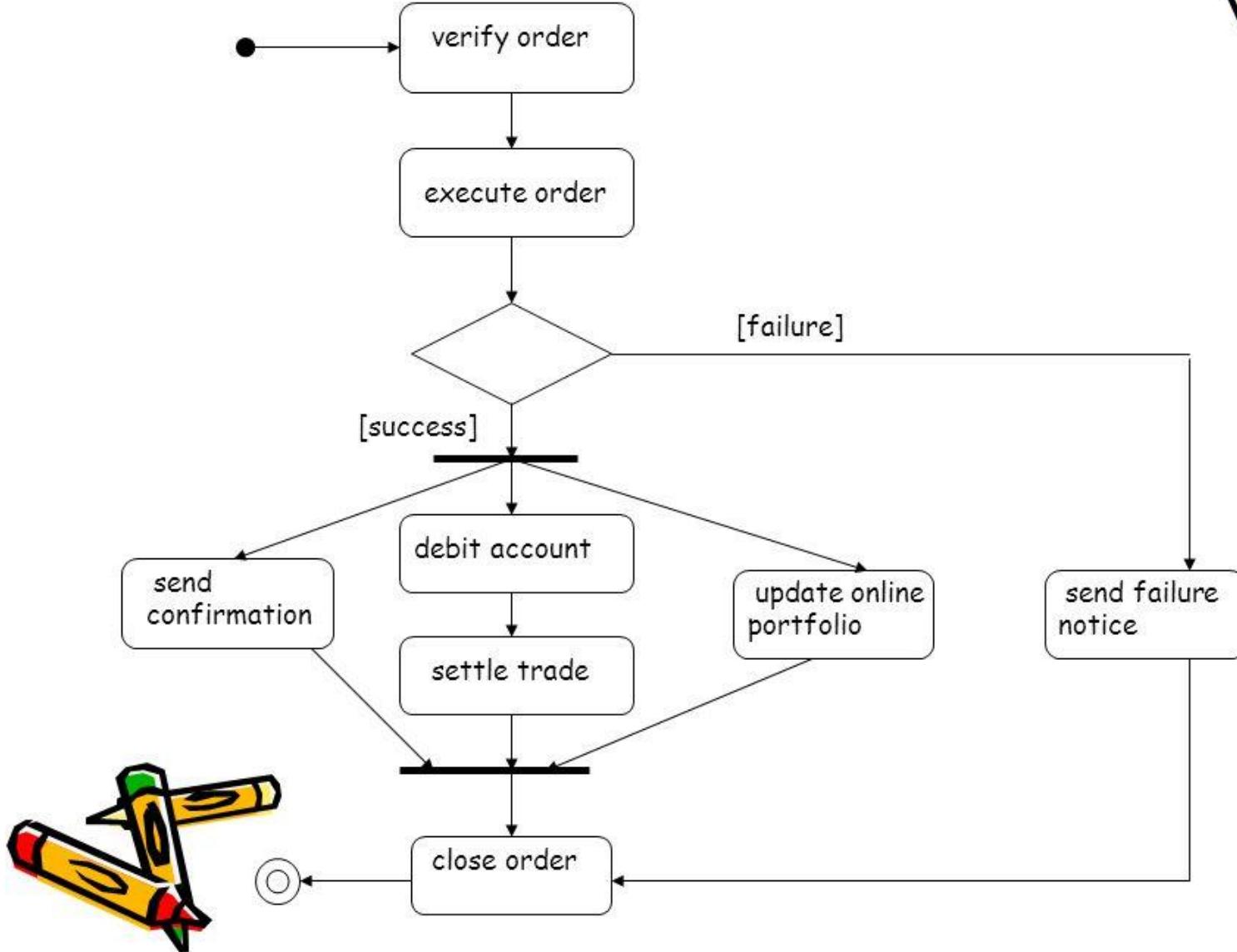


Subset of structural elements of ADs





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



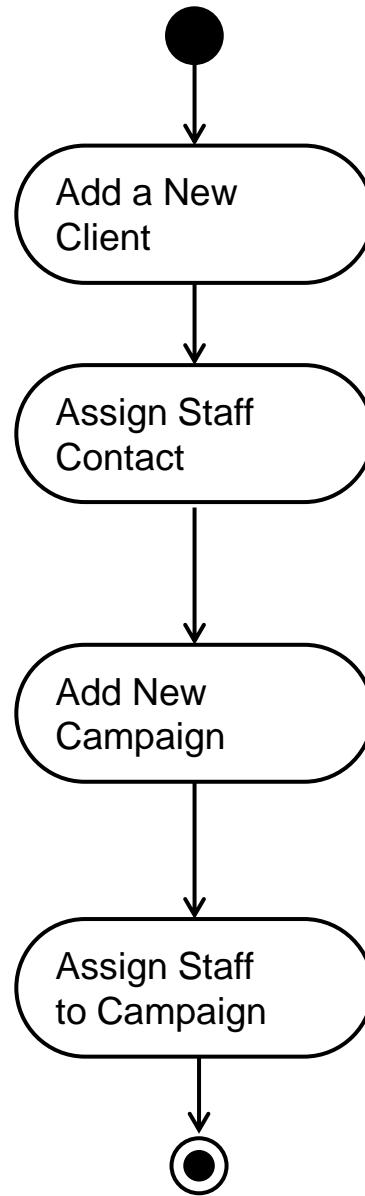
Drawing Activity Diagrams

- 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

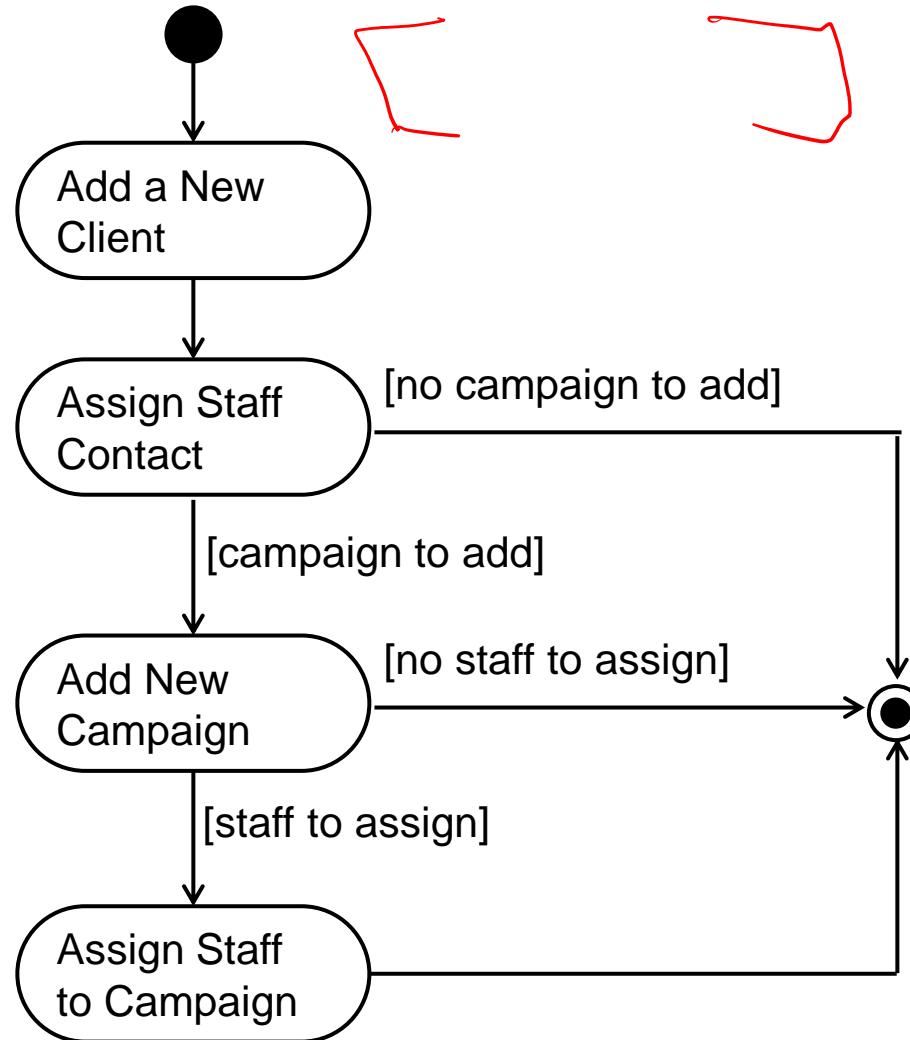


Drawing Activity Diagrams

- 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

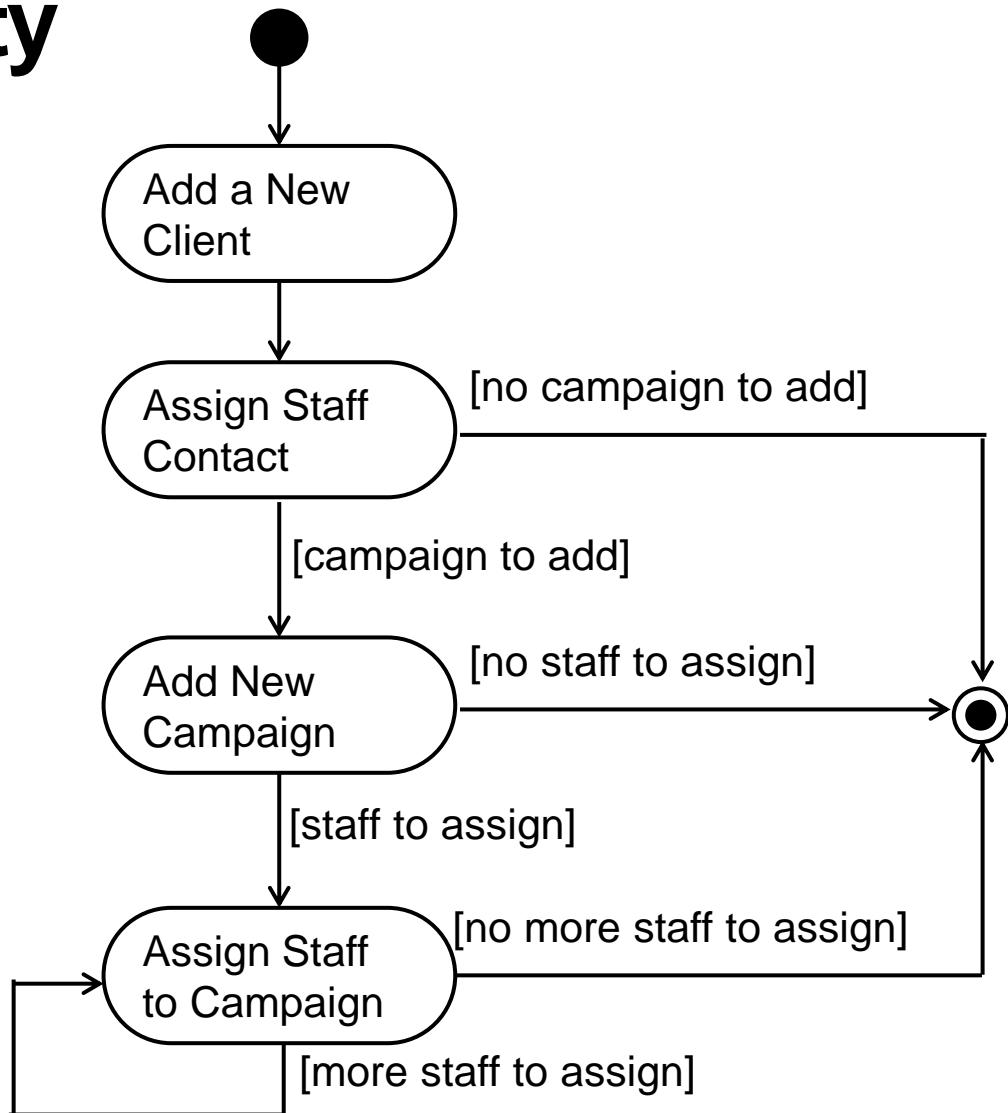
Diagrams



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

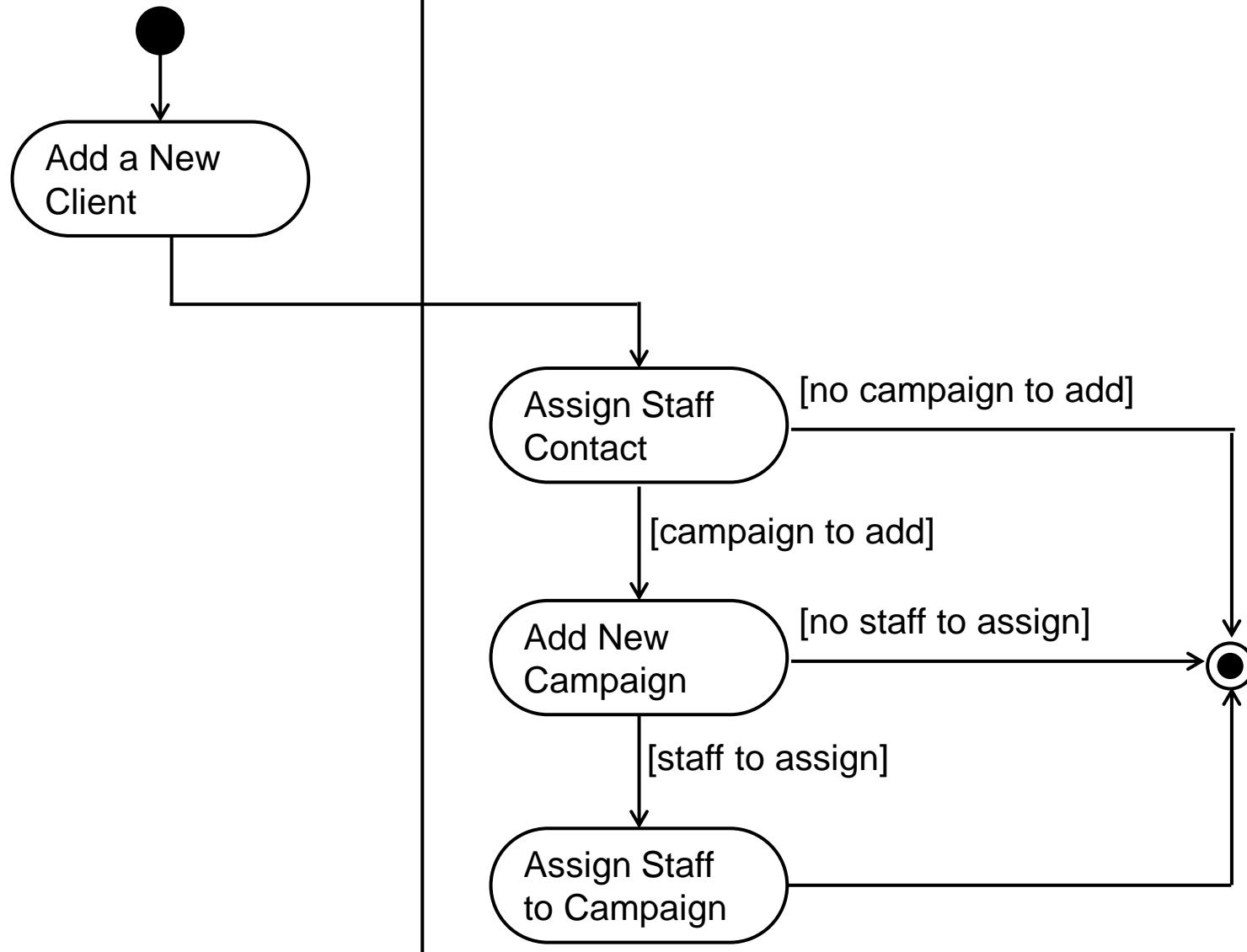


Drawing Activity Diagrams

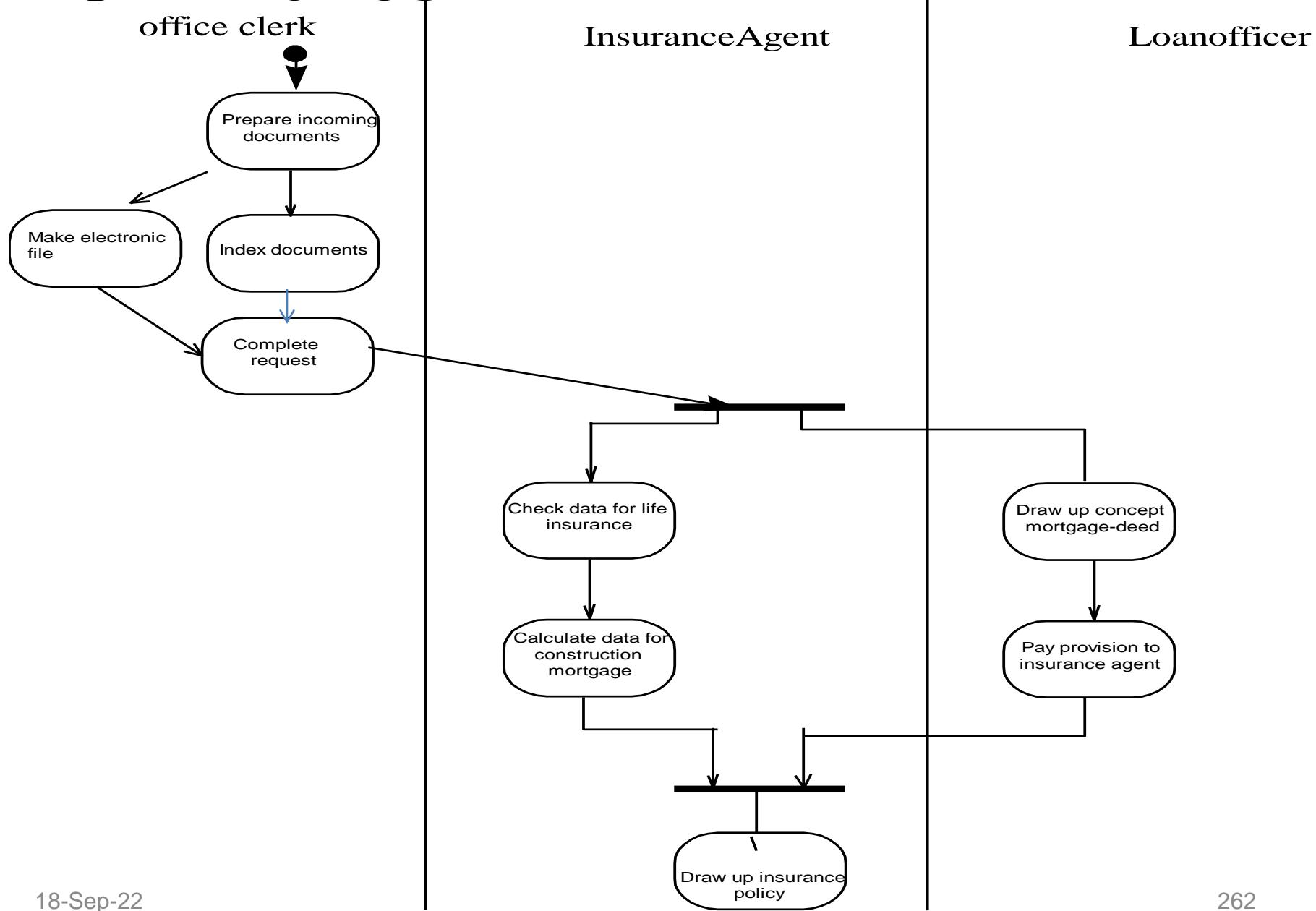
- Are all the activities carried out by the same person, organisation or department?
- If not, then add swimlanes to show the responsibilities
- Name the swimlanes
- Show each activity in the appropriate swimlane

Administrator

Campaign Manager



Swimlanes.

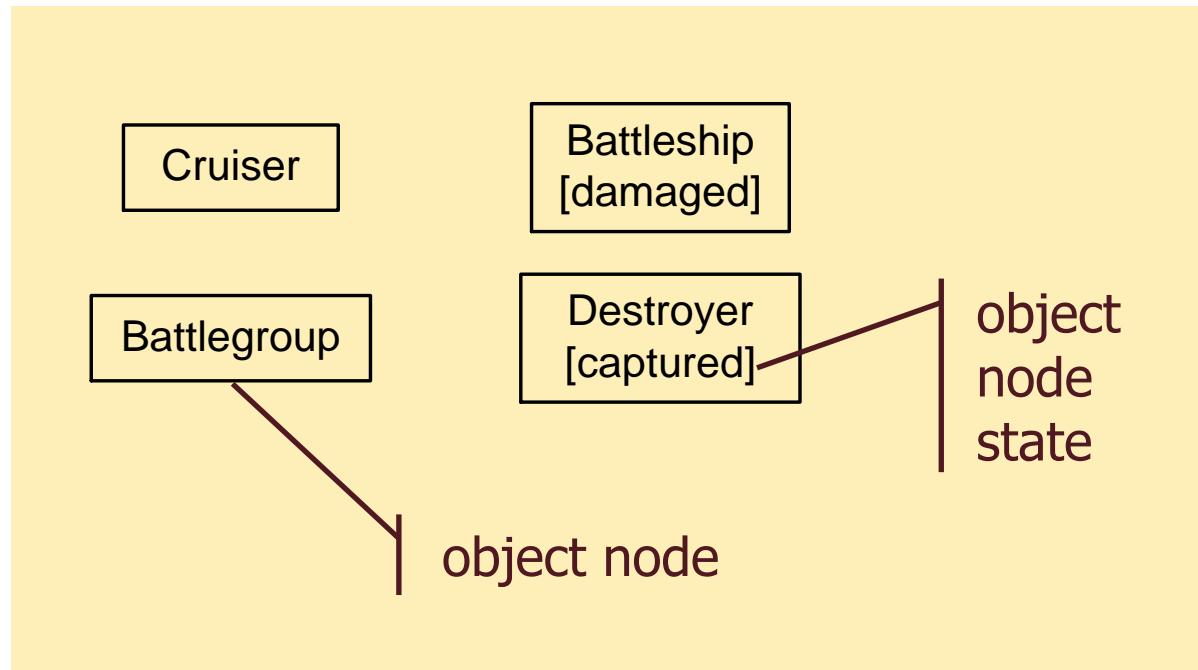


Drawing Activity Diagrams

- 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

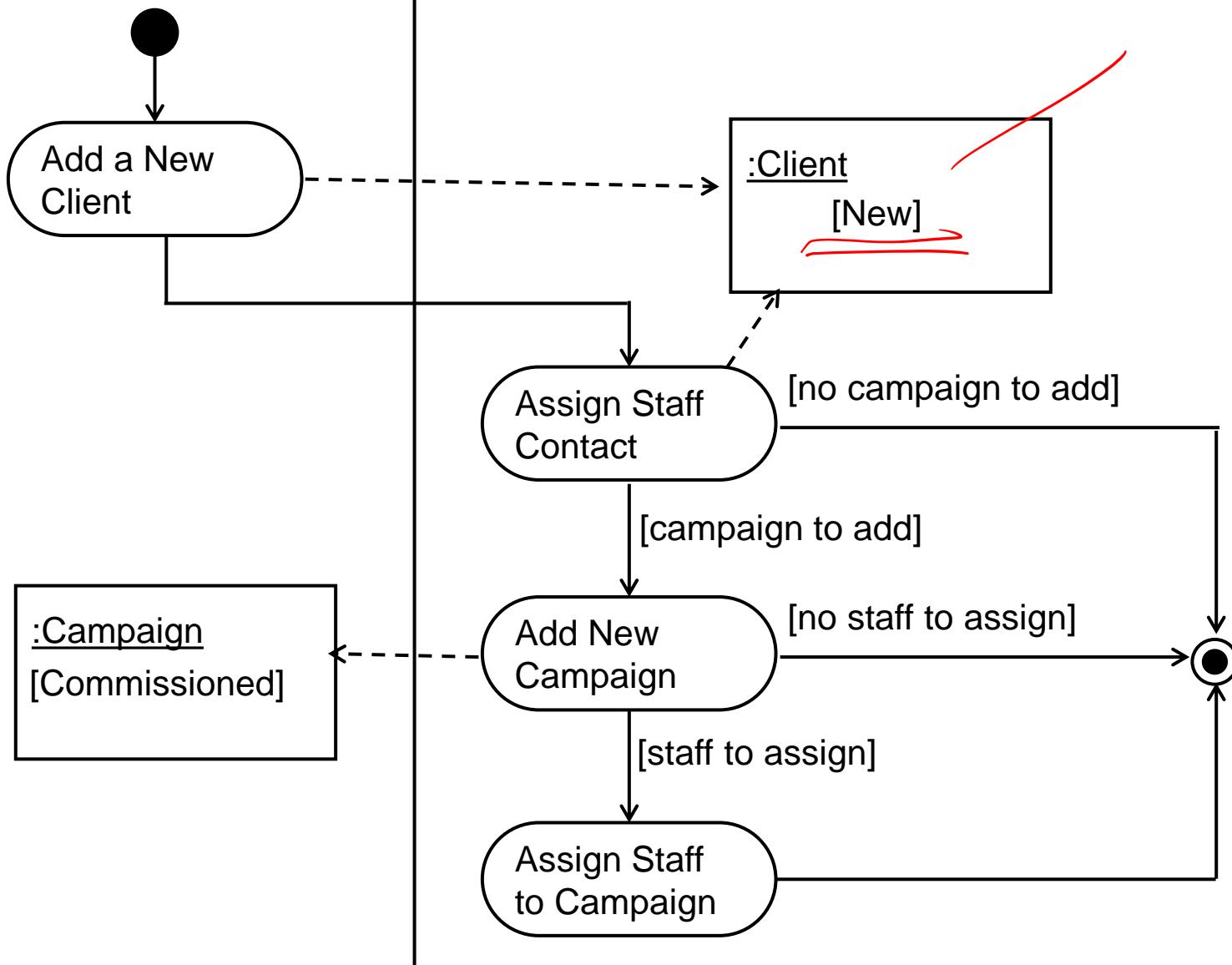
Object Nodes

Data and objects are shown as object nodes.

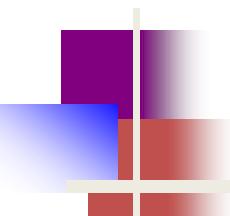


Administrator

Campaign Manager



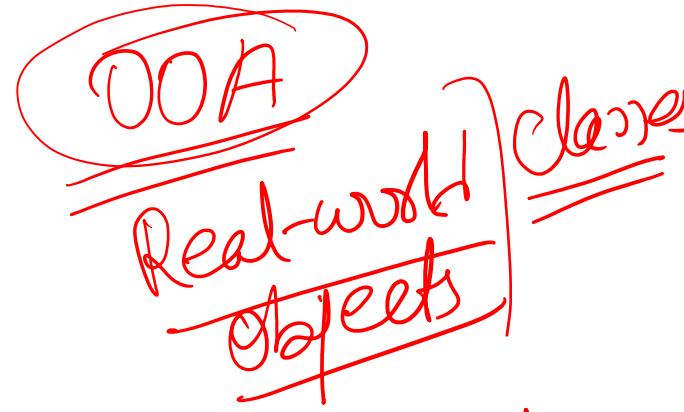
Domain Model



~~Analysis~~
~~Class Diagram~~

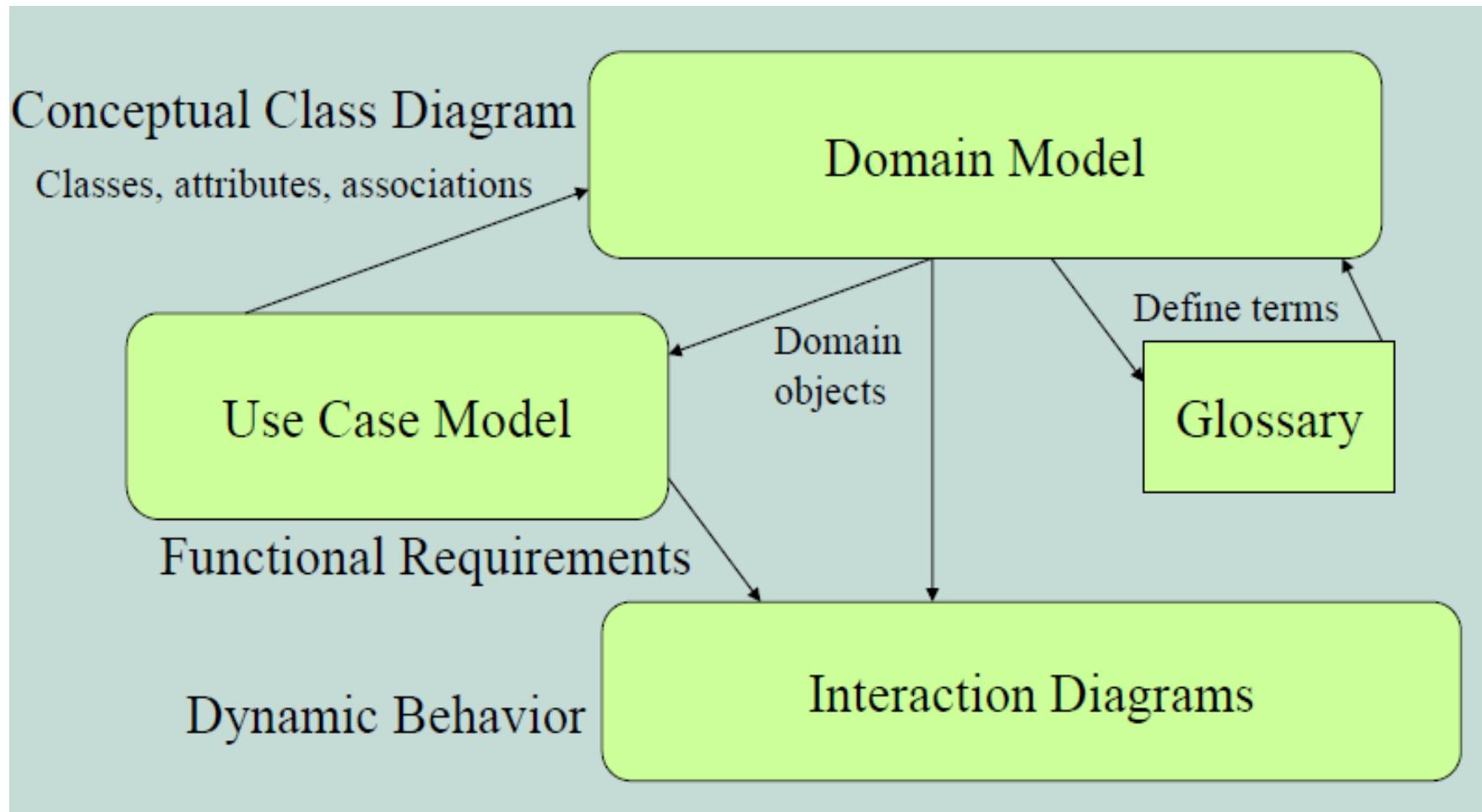
~~Conceptual~~
~~Class Diagram~~

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**.

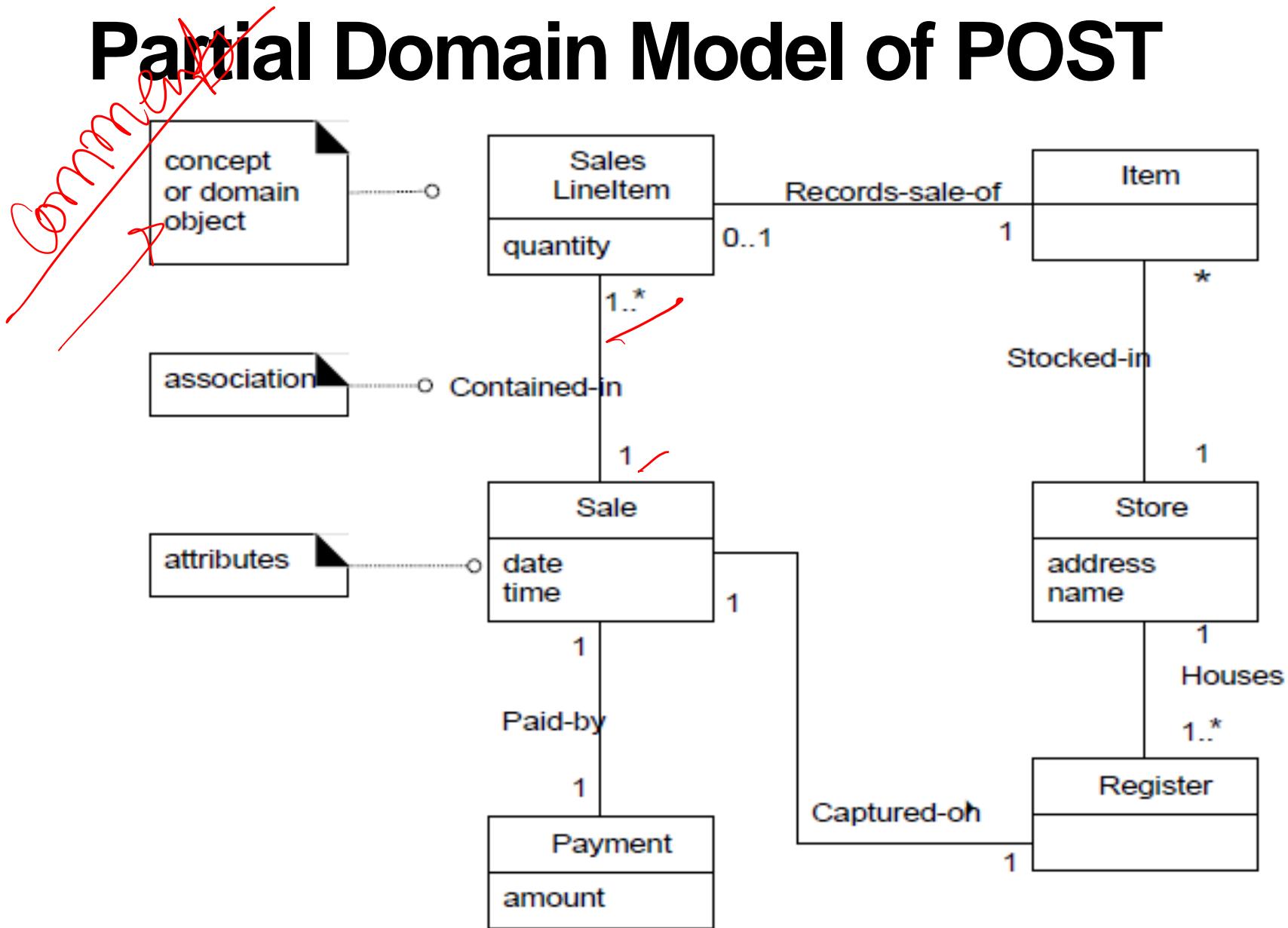
Domain Model Relationships



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 with 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

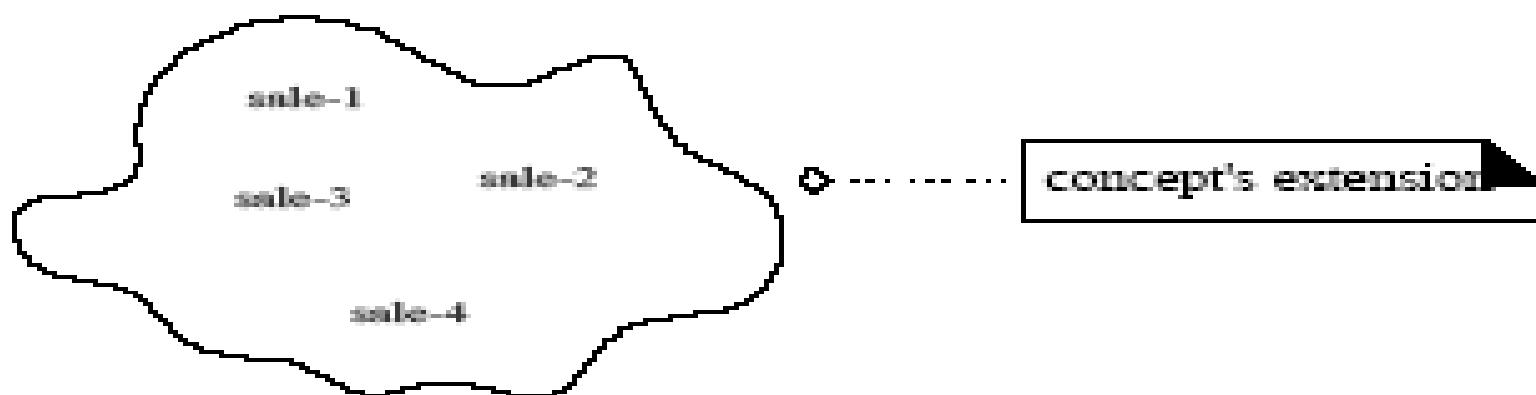
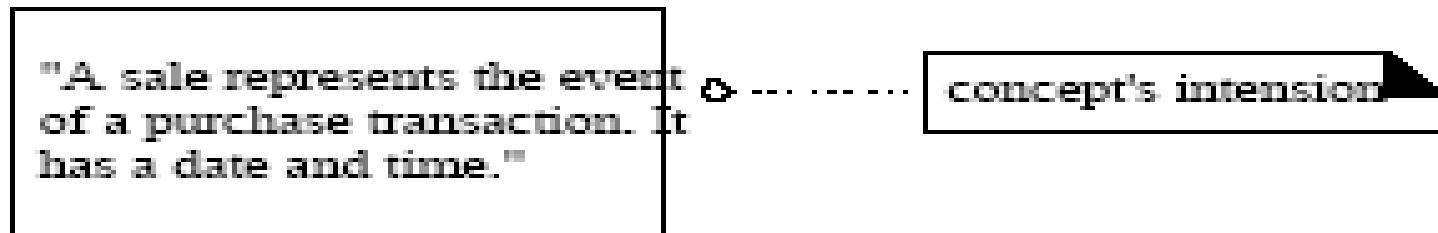
Domain classes?

- 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.

SC V RO OA OA

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

SR V OA V OA V OA

Checking the balance simply displays the account balance.

SM OA V OA

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

SM V OR V OA V OA

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

SM OA V OR OA V SC V OA

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

OA V SC V OA 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... ?

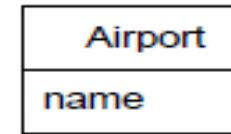
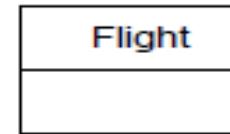


In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.

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?

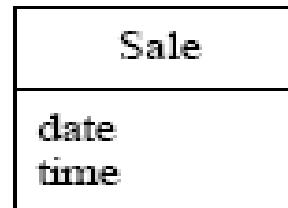
- Should it be included in the model?
 - It's common in the real world system
 - It's a sales report. Reports not explicitly stated in the use cases have little value in this model.
 - However, when a Customer wishes to return an item, it is an important object in the domain.
 - Since this development 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.

Association – real-world

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

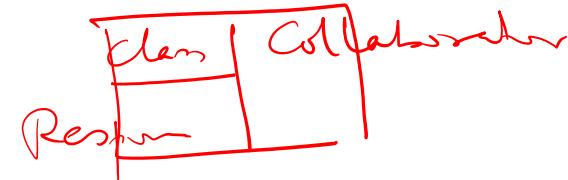
Design

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 *Real-work objects*
 - 1. Identifying the classes
 - 2. Identifying the classes attributes
 - 3. Identifying relationships and collaborations between classes

Object Oriented Analysis

Part I experience



- ▶ 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.

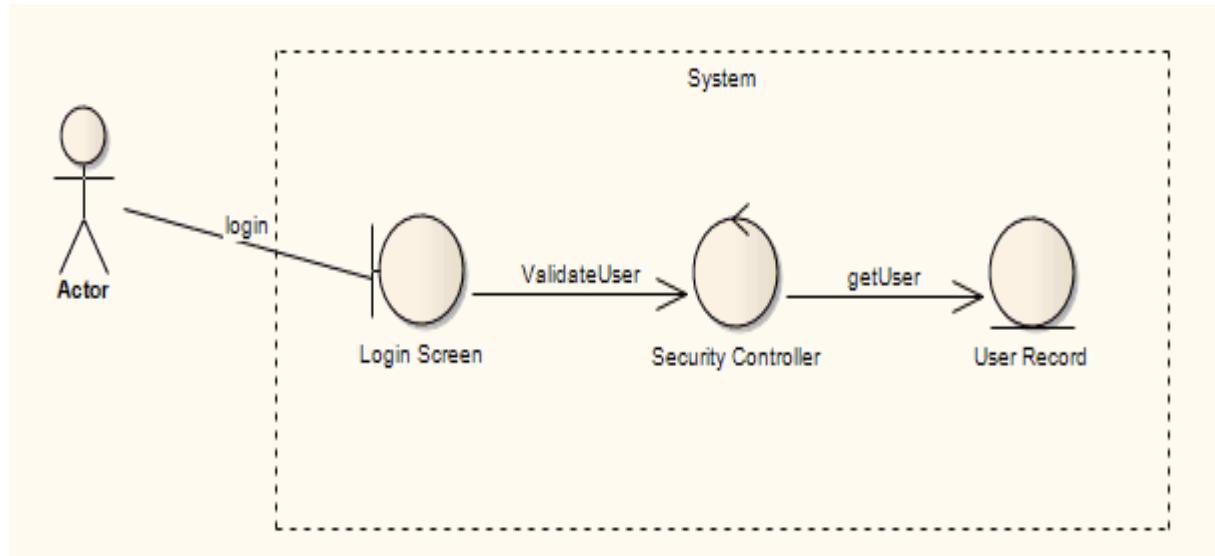
Three ways to do Object Oriented Analysis (1)

- ▶ 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).

entity



Object Identification

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

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.

Redundancy

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

Goal/Problem

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.

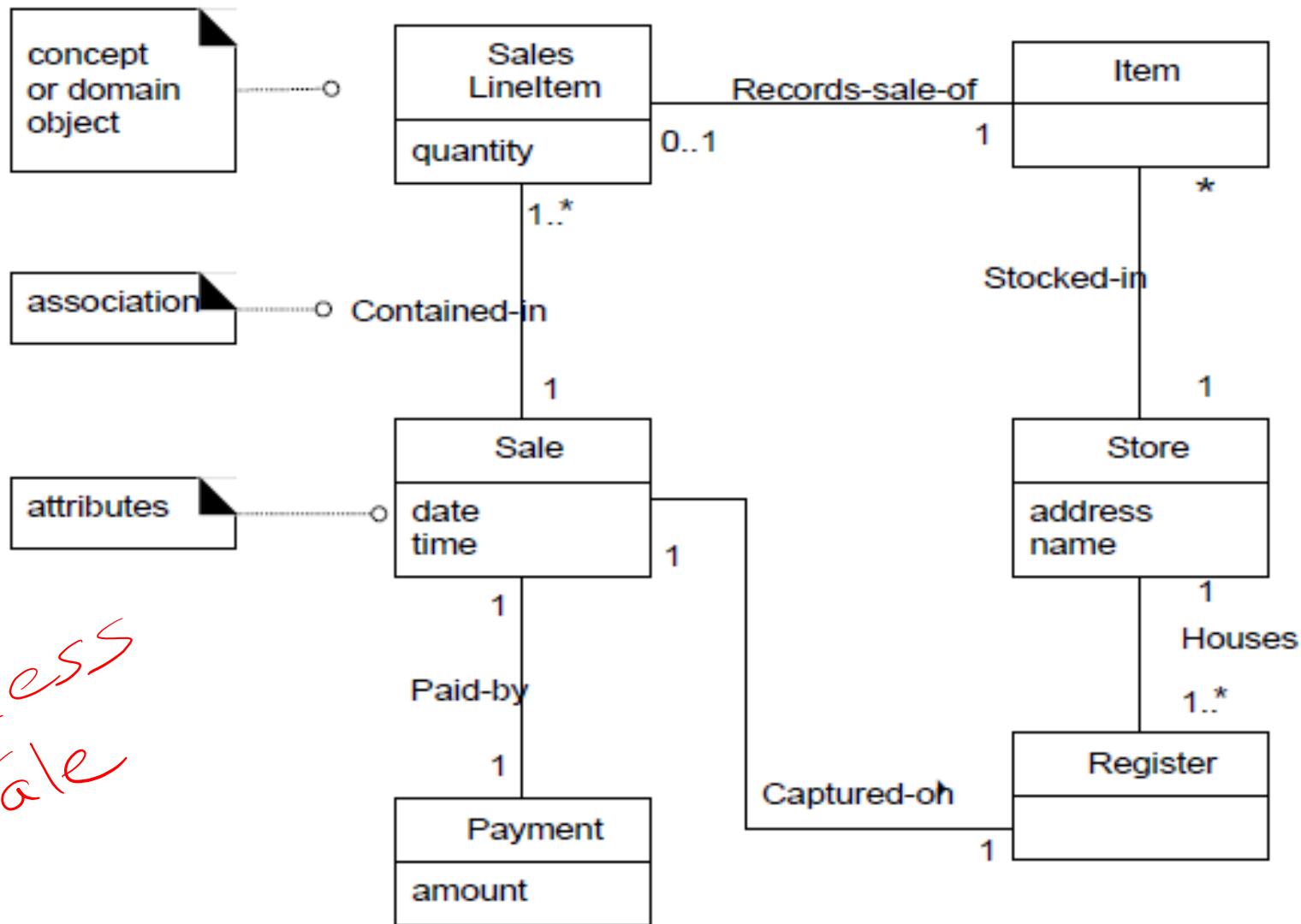
ATM Case Study

- 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



Process Sale

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 with 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.

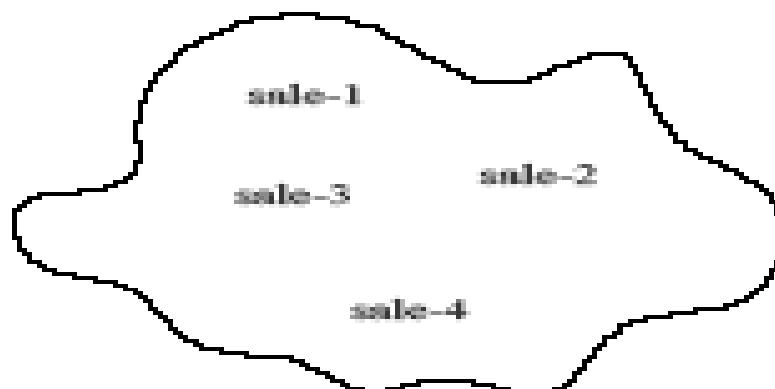
Symbol - Intension - Extension



concept's symbol

"A sale represents the event of a purchase transaction. It has a date and time."

concept's intension



concept's 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.

SC V RO OA OA

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

SR V OA V OA V OA

Checking the balance simply displays the account balance.

SM OA V OA

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

SM V OR V OA V OA

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

SM OA V OR OA V SC V OA

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

OA V SC V OA 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... ?

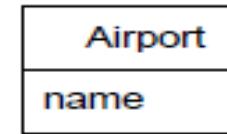
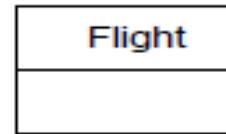


In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.

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

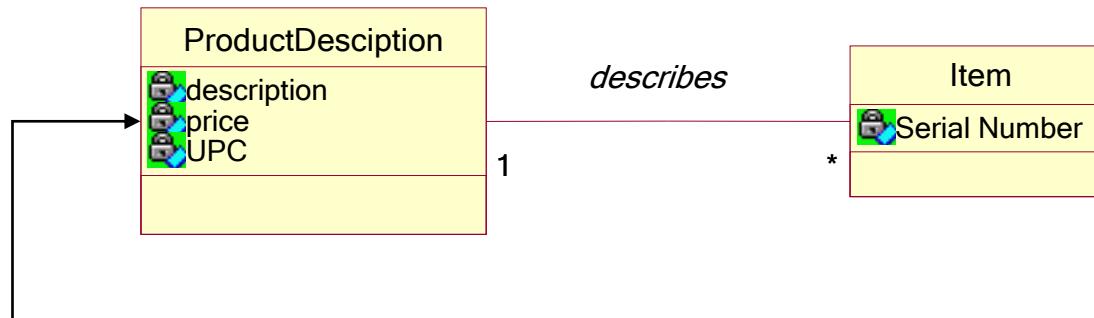
Specification Example

- 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.

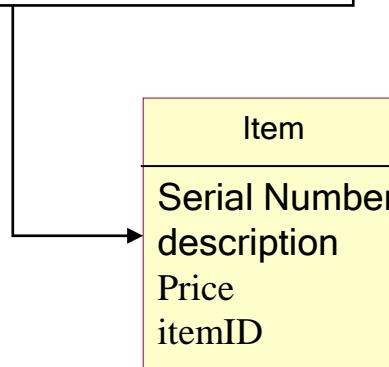
- Also notice the data is duplicated many times with each instance of the *item*.
- This example illustrates the need for a concept of objects that are specifications or descriptions of other things (often called a *Proxy or Surrogate*)
- Description or specification objects are strongly related to the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

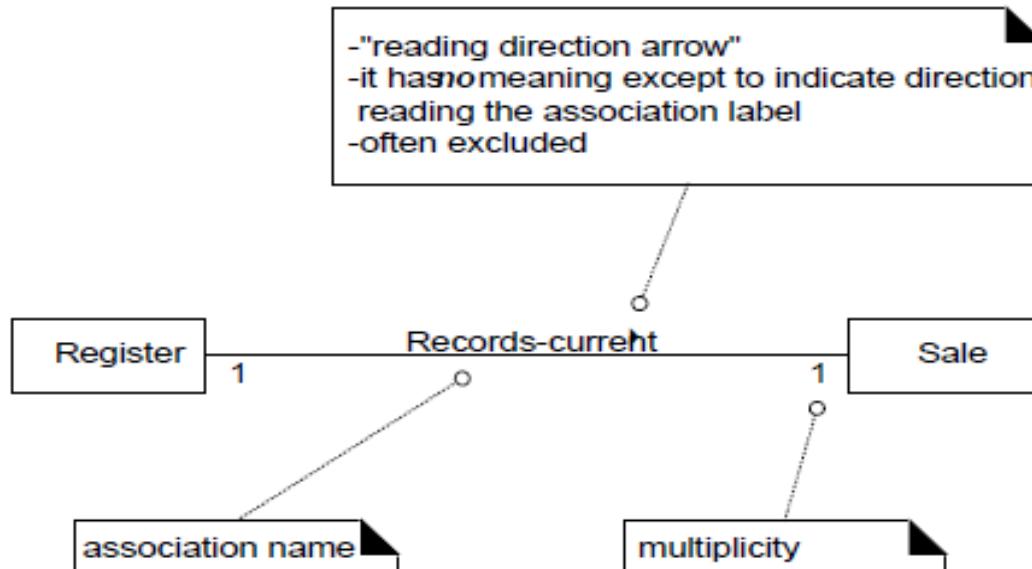
- Associations imply knowledge of a relationship that needs to be preserved over time
 - could be milliseconds or years!
 - such associations are *need-to-know* assocs.
- Associations derived from the Common Associations List (Pg 155-156, Larman).
- Represented as a solid line between objects
 - the 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

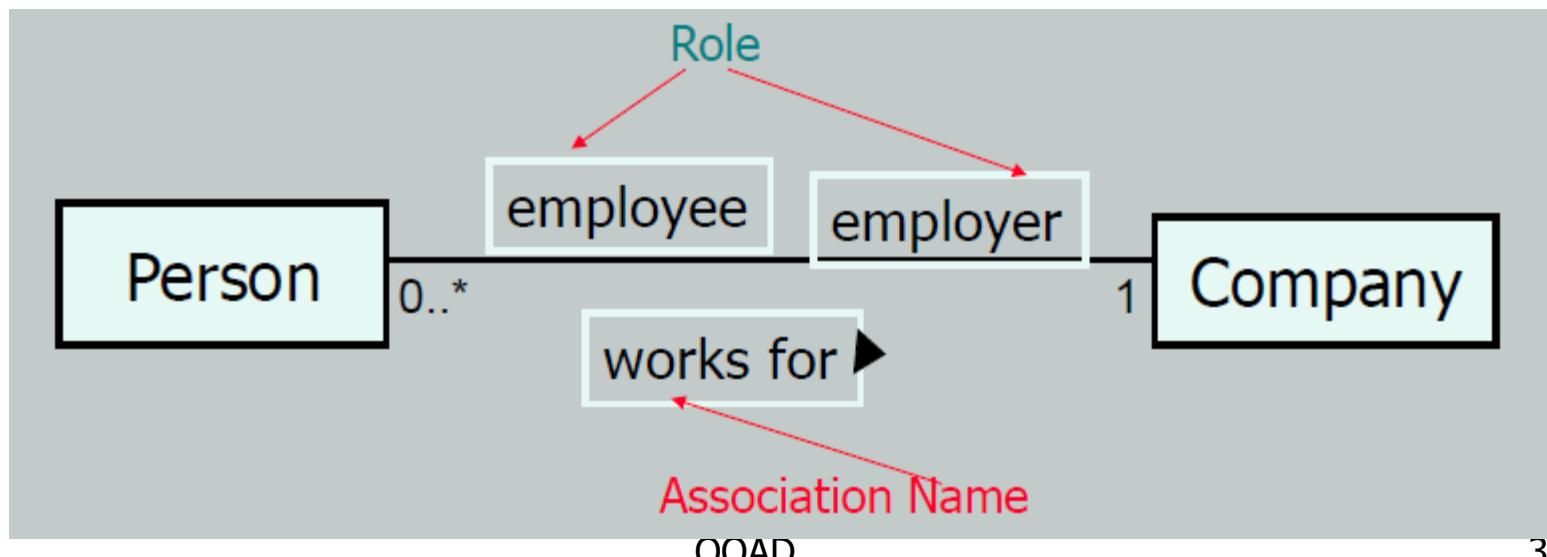
- High priority associations
 - A is a physical or logical part of B
 - A is physically or logically contained in/on B
 - A is recorded in B
- Other associations
 - A uses or manages or controls B (*Pilot -airplane*)
 - A owns B (*Airline -airplane*) or (*Register-Store*)
 - A is event related to B (*Customer-Sale*)

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

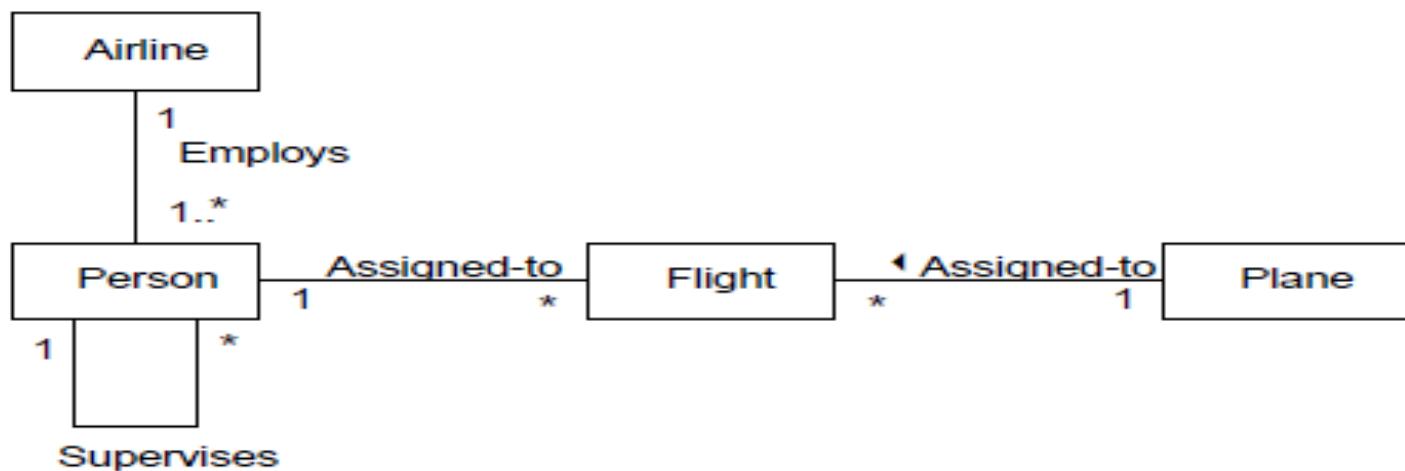
- Each of the two ends of an association is called a **role**. Roles have
 - name
 - multiplicity expression
 - navigability



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

Naming Associations

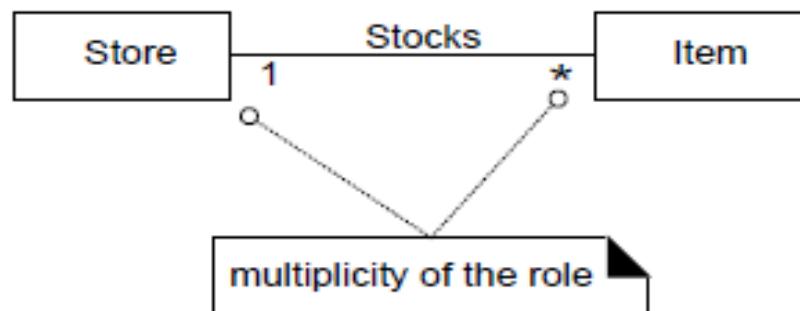


Multiplicity

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



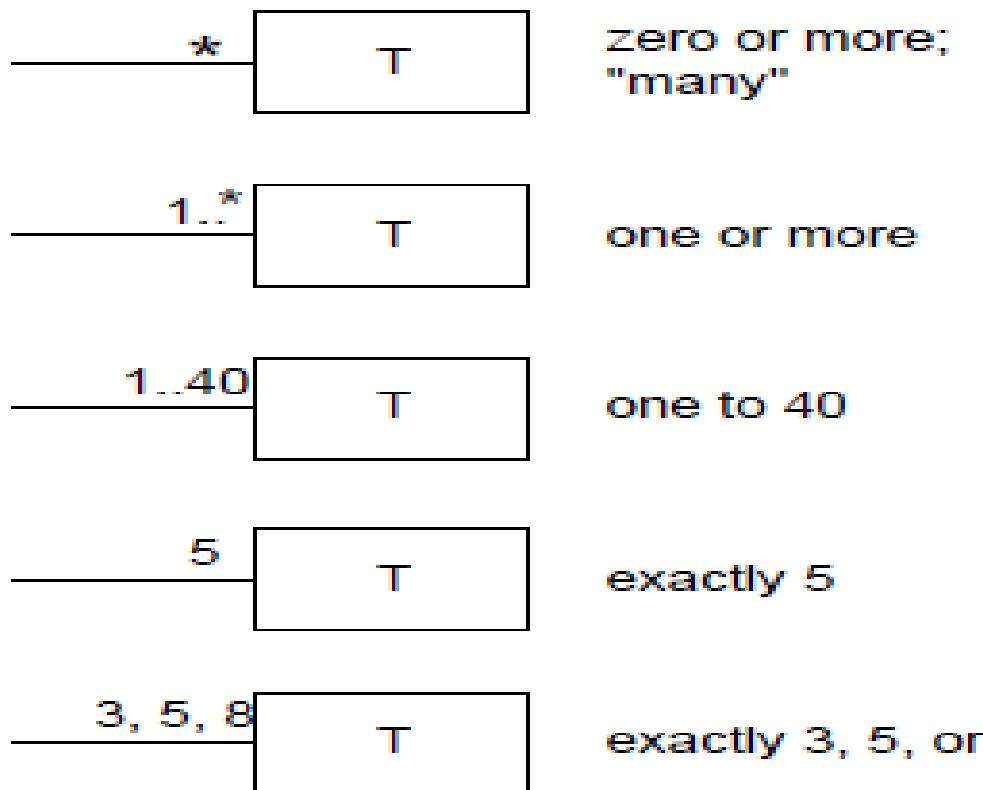
Multiplicity of the role



Association - Multiplicity

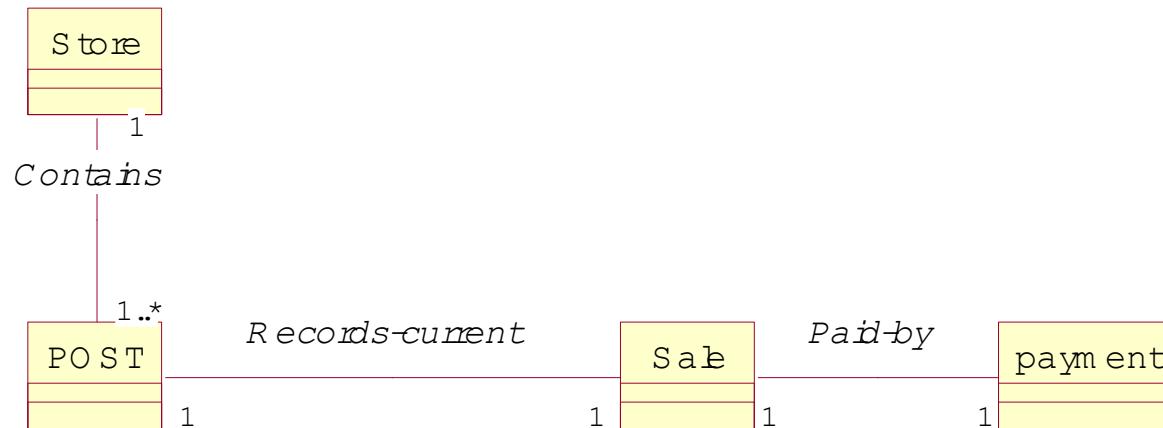
- **Multiplicity**: indicates the number of objects of one class that may be related to a single object of an associated class.
- Can be one of the following types
 - 1 to 1, 1 to 0..*, 1 to 1..*, 1 to n, 1 to 1..n
- The multiplicity value communicates how many instances can be validly associated with 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.

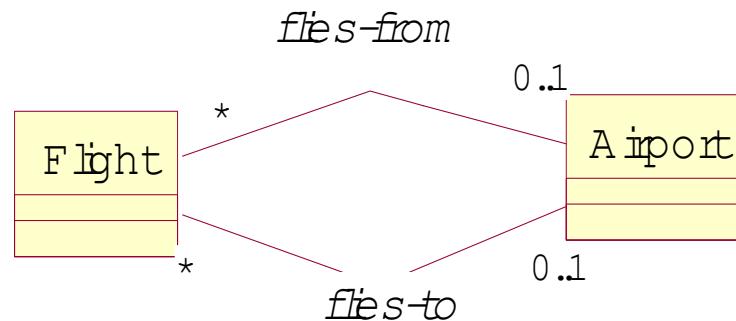
Associations - Contd.



Associations are generally read left to right, top to bottom

Associations - Contd.

Multiple associations between two types

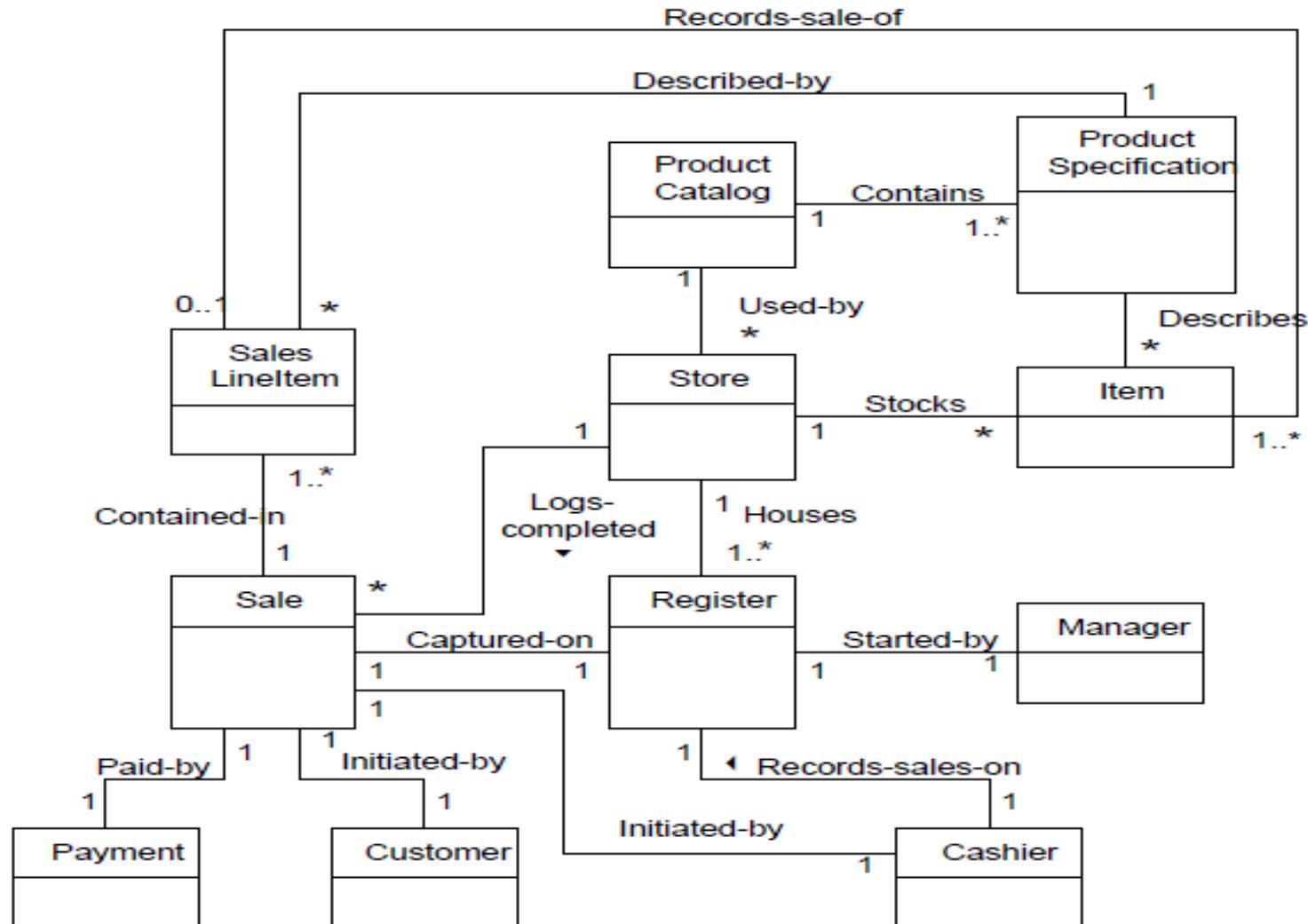


During analysis phase, an association is *not* a statement about data flows, instance variables, or object connections in the software solution, it is a statement that a relationship is meaningful in a purely conceptual sense.

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]

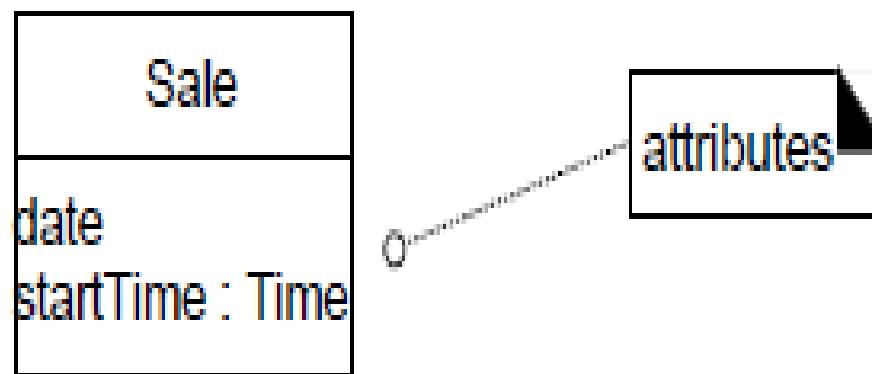
- **Attribute** - is a logical data value of an object.
It is a named property of a class describing values held by each object of the class
- **Attribute Type**: A specification of the external behavior and/or the implementation of the attribute
- Include the following attributes in a domain model
 - 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

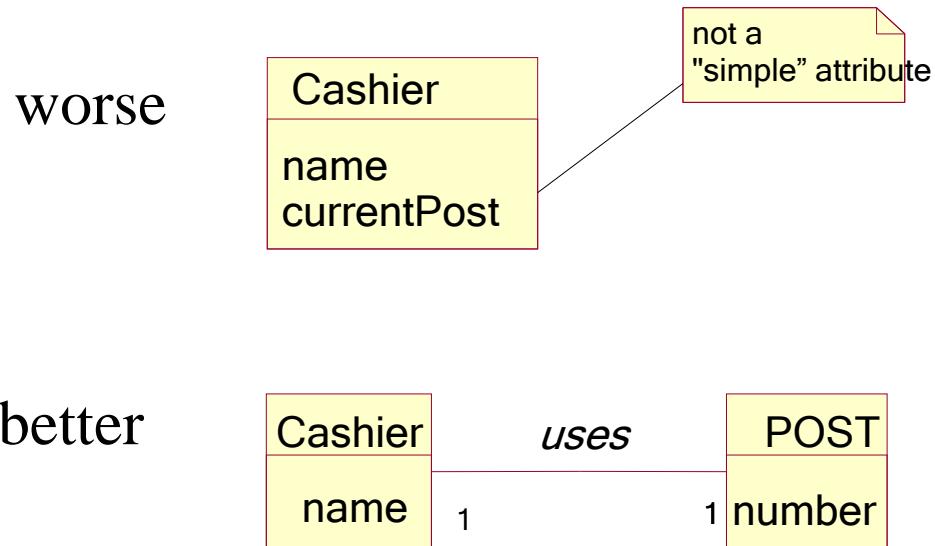
Class
Attributes



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)

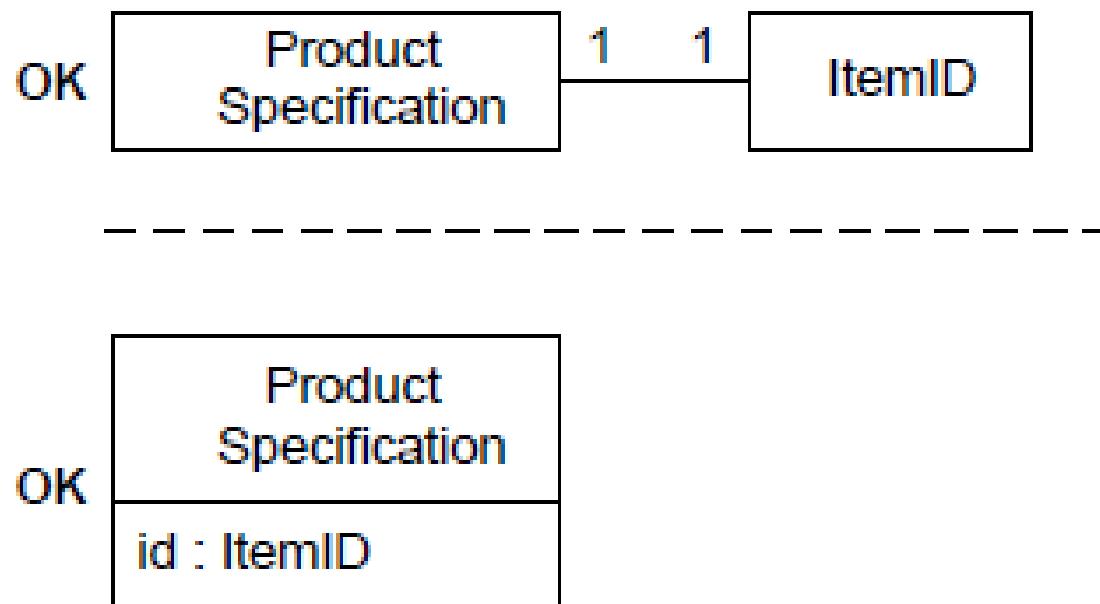
Non-primitive Data Type Classes

- 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 check-sum 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

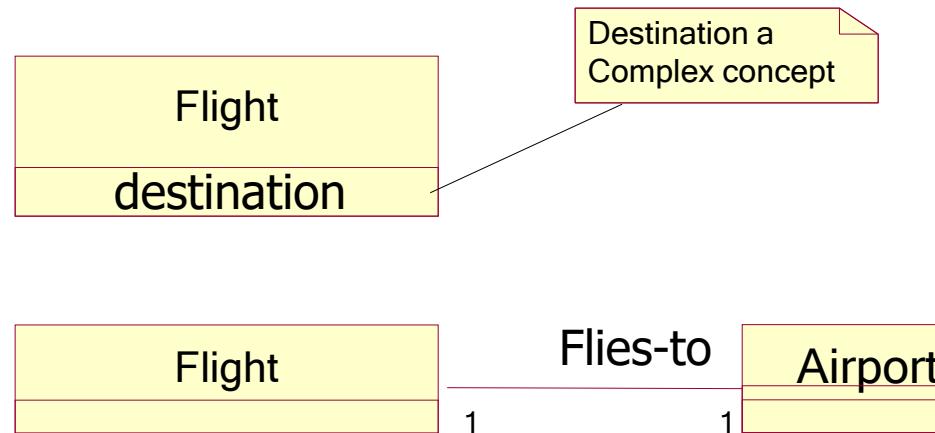


Non-primitive Data Type Classes

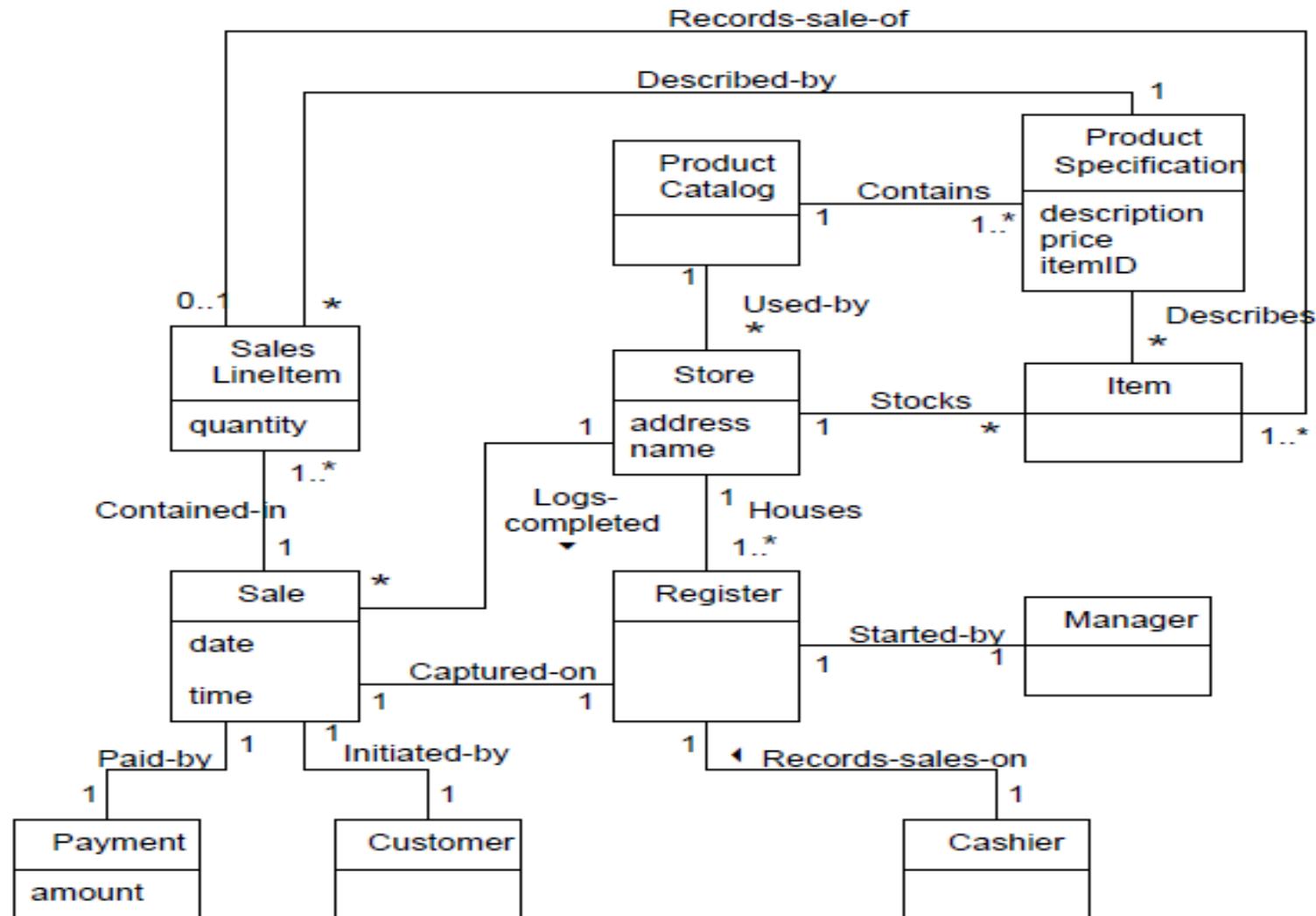
- The *price and amount attributes should be non-primitive Quantity or Money classes* because they are quantities in a unit of currency.
- The *address attribute should be a non-primitive Address class* because it has separate sections.
- The classes *ItemID, Address, and Quantity are data types (unique 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**.

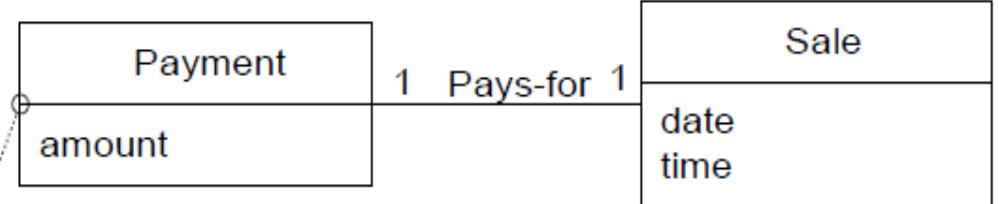
Domain Model v/s Design Model

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.

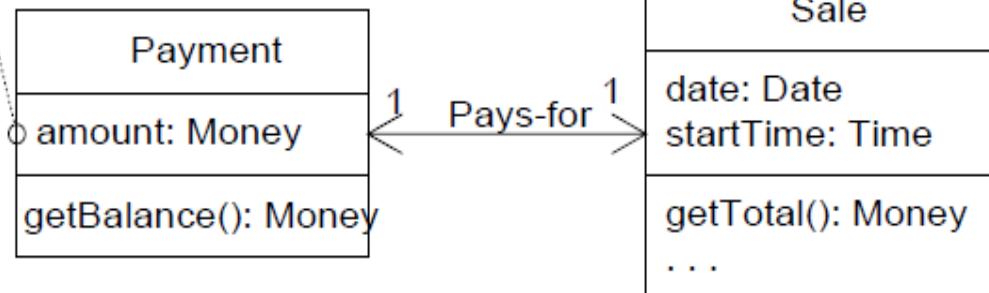
This reduces the representational gap.

This is one of the big ideas in object technology.

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



inspires
objects
and
names in



UP Design Model

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



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.7

Today's Agenda

- System Contracts
- Object Design
 - Interaction Diagrams
 - Sequence Diagrams
 - Collaboration Diagrams

Operation Contracts

System Sequence diagram [1]

Use **CASES** suggest how actors interact with the software system

- An actor generates events to a system, requesting some operation in response
- Example - when a cashier enters an item's UPC, the cashier requests the POST system to record that item purchase. That request event initiates an operation upon the system
- Desirable to isolate and illustrate the operations that an actor requests to a system

System Sequence Diagram [2]

- SSD Shows for a particular scenario of a use case, the events that external actors generate, their order and inter-system events
- A scenario of a use case is a particular instance or realized path
- Should be done for a typical course of events of the use case (usually for the most interesting ones)

System Events, Operations

- **System event** - external input event generated by an actor
- **System operation** - operation of the system that executes in response

Recording System operations

Set of all required systems operations is determined by identifying the system events.

- Examples: `enterItem(UPC,quantity);`
`endSale();`
`makePayment(amount)`
- UML notation -

```
Operation(arg:ArgType=defaultvalue,,,:ReturnType(s))
```

Objectives

- Create contracts for system operations.

Why Contracts

- Use cases are the primary mechanism in the UP to describe system behavior, and are usually sufficient.
- However, sometimes a more detailed description of system behavior has value.
- Contracts for operations can help define system behavior.
- Describes what happens without explaining how.

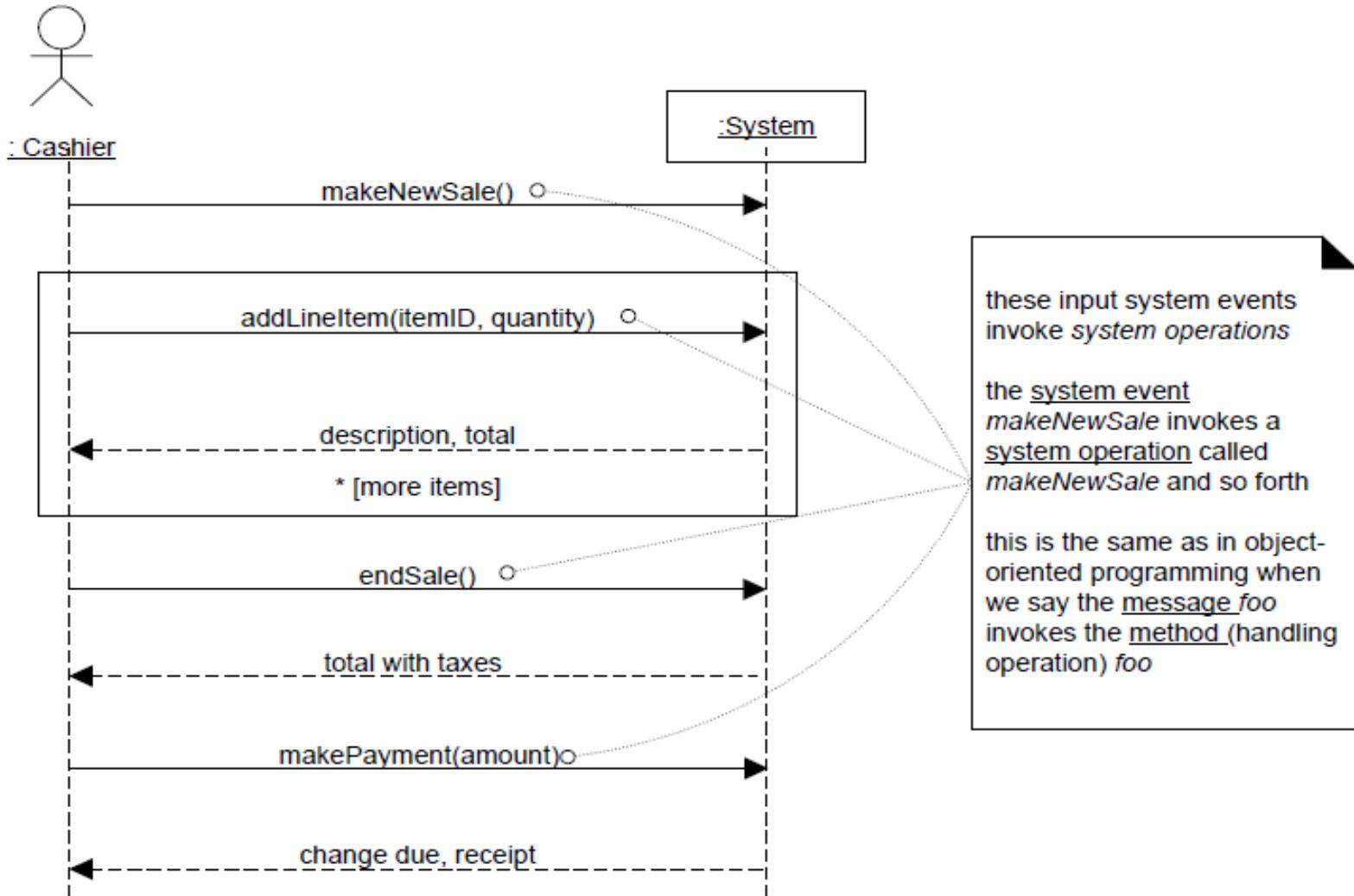
Definition

Contracts describe detailed system behavior in terms of state changes to objects in the Domain Model, after a system operation has executed.

System Operations and the System Interface

- Contracts may be defined for **system operations** – operations that the system as a black box offers in its public interface to handle incoming system events. System operations can be identified by discovering these system events.
- The entire set of system operations, across all use cases, defines the public **system interface**, viewing the system as a single component or class.

System operations handle input system events

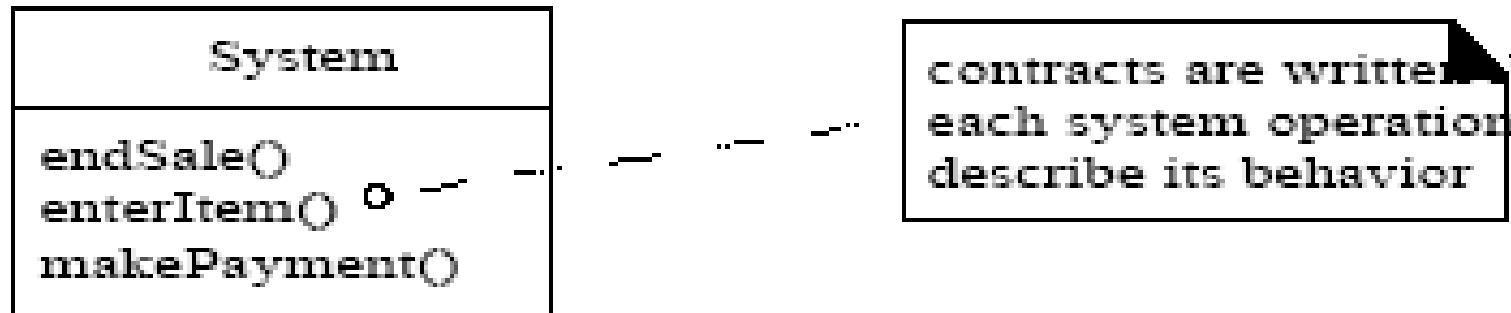


System Contracts

- Define responsibilities within the system.
- Operational contracts take the form of *preconditions* and *post-conditions*
 - Each module tests pre-conditions, then runs, then tests post-conditions before returning to the caller

System Contracts

- Operational contracts state what an operation commits to achieving.
- Emphasizes *what* will happen, not how
- Describes changes in the state of the system
 - Semantics



Contract Sections

Operation:	Name Of operation, and parameters.
Cross References:	(optional) Use cases this can occur within.
Preconditions:	Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation.
Postconditions:	-The state of objects in the Domain Model after completion of the operation.

Example Contract: enterItem

Contract C02: enterItem

Operation: `enterItem(itemID : ItemID,
quantity : integer)`

Cross References: Use Cases: Process Sale

Preconditions: There is a Sale Underway.

Postconditions:

- A SalesLineItem instance *sli* was created (instance creation)
- sli* was associated with the current Sale (association formed)
- sli.quantity* became *quantity* (attribute modification)
- sli* was associated with a ProductSpecification, based on *itemID* match (association formed)

Postconditions

- The postconditions describe changes in the state of objects in the Domain Model. Domain Model state changes include ***instances created, associations formed or broken, and attributes changed.***
- Postconditions are not actions to be performed during the operation; rather, they are declarations about the Domain Model objects that are true when the operation has finished.
- postconditions support fine-grained detail and specificity in declaring what the outcome of the operation must be.

How to Make a Contract

- Identify operations from the system sequence diagram
- For each system operation, construct a contract
- For system operations that are complex and perhaps subtle in their own results, or which are not clear in the use case, construct a contract.
- The post-conditions describe changes in the state of objects in the Domain Model.
- To describe post-conditions, use the following categories:
 - Instance creation and deletion
 - Attribute modification
 - Associations formed and broken

enterItem Postconditions

- ***Instance Creation and Deletion:***

After the *itemID* and *quantity* of an *item* have been entered, what new object should have been created? A

SalesLineItem. Thus:

- ❖ A *SalesLineItem* instance *sli* was created (instance creation).
- ❖ Note the naming of the instance. This name will simplify references to the new instance in other post-condition statements.

enterItem Postconditions

- ***Attribute Modification:***

After the `itemID` and `quantity` of an item have been entered by the cashier, what attributes of new or existing objects should have been modified? The *quantity* of the `SalesLineItem` should have become equal to the *quantity* parameter. Thus:

- ❖ `sli.quantity` became `quantity` (attribute modification).

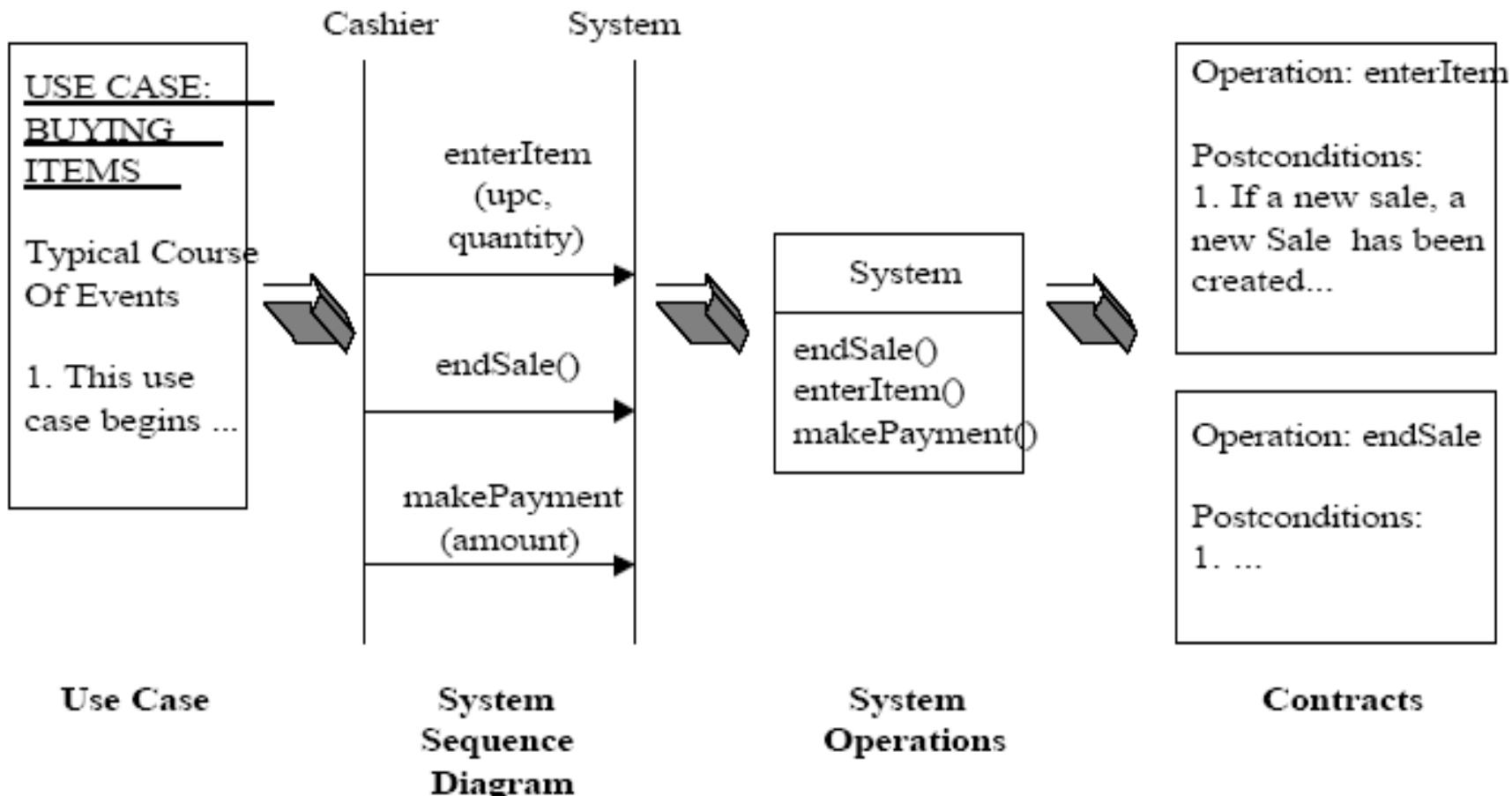
enterItem Postconditions

- ***Associations Formed and Broken:***

After the *itemID* and *quantity* of an *item* have been entered by the cashier, what associations between new or existing objects should have been formed or broken? The new *SalesLineItem* should have been related to its *Sale*, and related to its *ProductSpecification*. Thus:

- ❖ . *sli* was associated with the current *Sale* (association formed).
- ❖ . *sli* was associated with a *ProductSpecification*, based on *ItemID* match (association formed).

POST Contract Process



Writing Contracts Leads to Domain Model Updates

- New conceptual classes, attributes, or associations in the Domain Model are often discovered during contract writing.
- Enhance the Domain Model as you make new discoveries while thinking through the operation contracts.

Contracts vs. Use Cases

- The use cases are the main repository of requirements for the project. They may provide most or all of the detail necessary to know what to do in the design.
- If the details and complexity of required state changes are awkward to capture in use cases, then write operation contracts.
- If, just based on the use cases and through ongoing collaboration with a subject matter expert, the developers can comfortably understand what to do, then avoid writing contracts.

Advice on Writing Contracts

- To make contracts:
 1. Identify system operations from the SSDs.
 2. For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.
 3. To describe the postconditions, use the following categories:
 - ❖ instance creation and deletion
 - ❖ attribute modification
 - ❖ associations formed and broken

Advice on Writing Contracts

- Fill in the Operation section first
 - Post-conditions section next
 - Pre-conditions section last!
- Express post-conditions in the past tense, to emphasize they are declarations about a state change in the past.
 - (better) A SalesLineItem was created.
 - (worse) Create a SalesLineItem.

Advice on Writing Contracts

- Establish a memory between existing objects or those newly created by defining the forming of an association. For example, it is not enough that a new *SalesLineItem* instance is created when the *enterItem* operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with *Sale*. Don't forget to include all the associations formed and broken.

Example Contract

- **makeNewSale:**

Operation: makenewsale()

Cross Reference: Use Cases: Process Sale

Pre-condition: None

Post-condition:

- A Sale instance s was created (instance creation).
- s was associated with the Register (association formed).
- Attributes of s were initialized.

On a project, all these particular postconditions are so obvious from the use case that the makeNewSale contract should probably not be written.

Example Contract

- **endSale:**

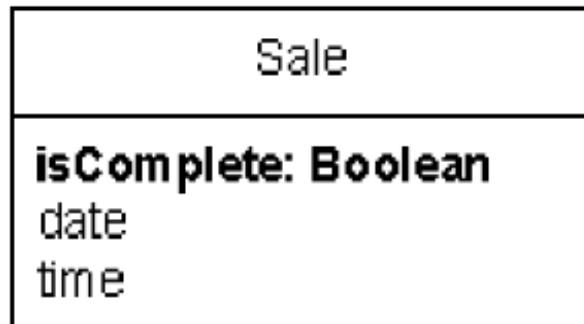
Operation: endsale()

Cross Reference: Use Cases: Process Sale

Pre-condition: There is a sale underway

Post-condition: - *Sale.isComplete became true modification).*

Changes in Domain Model:



Example Contract

- **makePayment:**

Operation: makePayment(amount: Money)

Cross Reference: Use Cases: Process Sale

Pre-condition: There is a sale underway

Post-condition: *- A Payment instance p was created creation).*

- $p.amountTendered$ became amount (attribute modification).

- p was associated with the current Sale (association formed).

- The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales)

Summary : Contracts

A **contract** describes detailed system behavior in terms of state changes to objects in the **domain model**.

- A contract is a **system operation**. It is offered in the system's **public interface**.
- **One use case** may require **one or more system operations** (events) to complete a scenario.
- Most contracts will be written during **elaboration phase**, when most use cases are written. Only write contracts for the most difficult to understand or complicated system operations.

Artifacts of Analysis

Analysis Artifact

Use Cases

Conceptual model

System Sequence diagrams

Contracts

Questions Answered

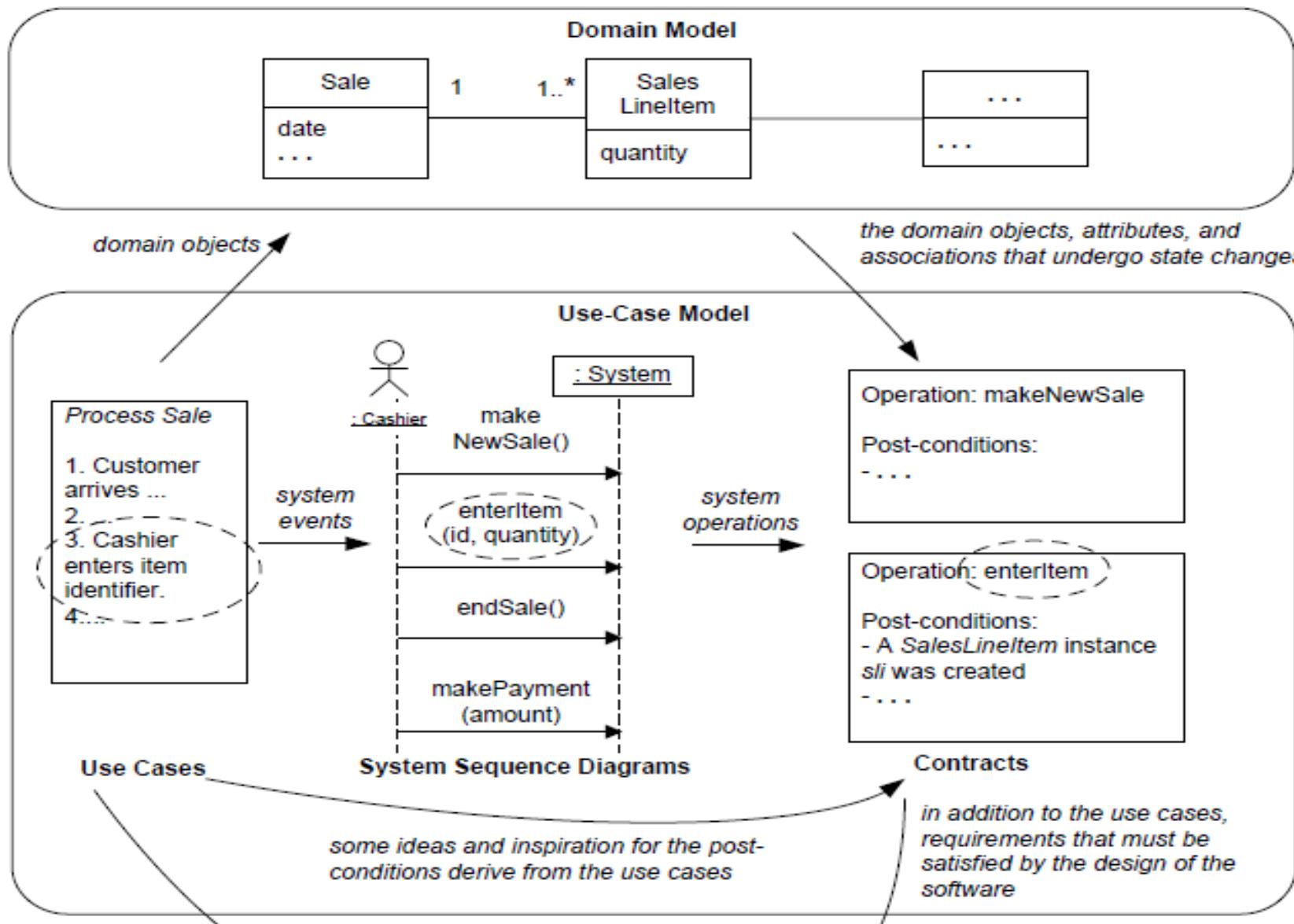
What are the domain processes?

What are the concepts and terms?

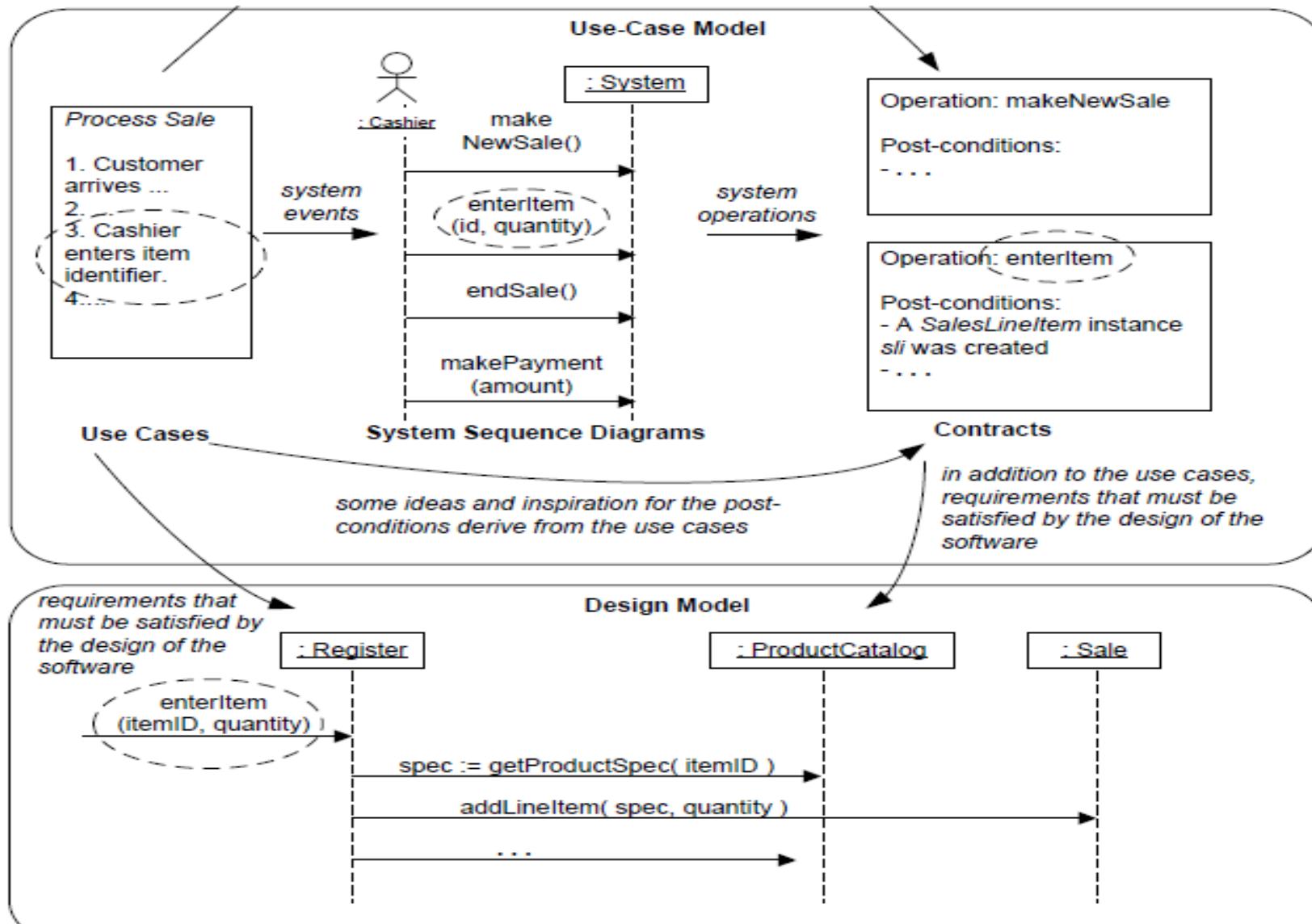
What are system events and operations?

What are the semantics of system operations?

Contract relationship to other artifacts



Contract relationship to other



REQUIREMENTS → DESIGN

- *Till now we have covered requirements analysis.*
The requirements and object-oriented analysis has focused on learning to *do the right thing*. that is, understanding some of the outstanding goals for the Next-Gen POS, and related rules and constraints.
- Now onwards we will start designing:
Design work will stress *do the thing right*; that is, *skillfully designing a solution to satisfy the requirements for this iteration*.

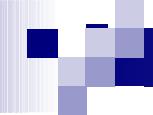
REQUIREMENTS → DESIGN

- In iterative development, a transition from primarily a requirements focus to primarily a design and implementation focus will occur in each iteration.
- It is natural and healthy to discover and change some requirements during the design and implementation work of the *early iterations*.
- During object design, a logical solution based on the object-oriented paradigm is developed. The heart of this solution is the creation of **interaction diagrams**, which illustrate how objects collaborate to fulfill the requirements. We can start designing **class diagram** after or in parallel with **interaction diagram**.

Object Design

Learning Objectives

- To learn notations for representing a design.
- To understand and be able to apply the basic principles of software design.
- To learn techniques and tools for Object-Oriented Design.
 - Interaction diagrams
 - Class diagrams
 - State diagrams



Objectives

- To explain how a software design may be represented as a set of interacting objects that manage their own state and operations
- To introduce various models that describe an object-oriented design
- To introduce design patterns

Models during the Design Phase

Object behavior model

Class model

Interaction Diagrams:

Collaboration diagrams
Sequence diagrams

Static structure
diagrams

Design state model

Contracts for
methods and operations

State diagrams
for classes

Object-oriented Analysis and Design

- Object-oriented analysis
 - What are the domain objects?
 - Described in a domain object model
- Object-oriented design
 - Describing the software solution in terms of collaborating objects, with responsibilities.

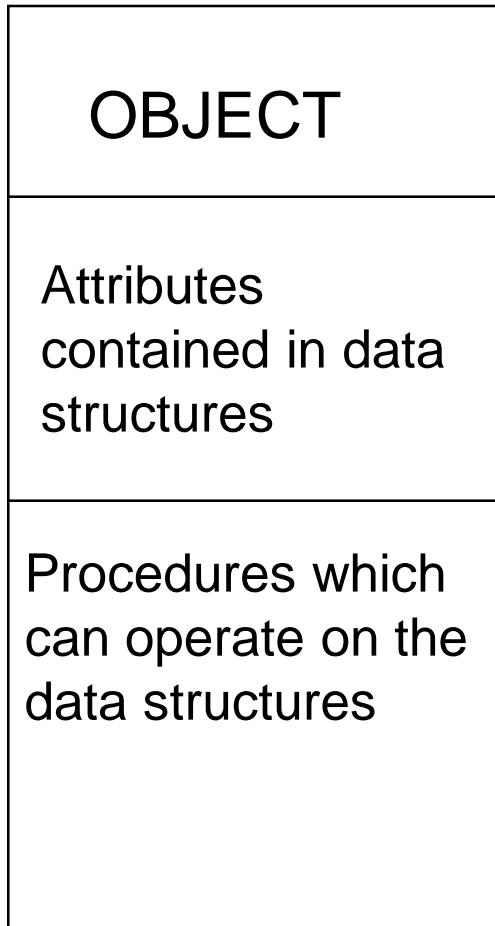
What is Object-Oriented Analysis and Design

- OOA: we find and describe objects or concepts in the problem domain
- OOD: we define how these software objects collaborate to meet the requirements.
 - Attributes and methods.
- OOP: Implementation: we implement the design objects in, say, Java, C++, C#, etc.

Thinking in Terms of Objects and UML

- Object-Oriented Design
 - Emphasizes a conceptual **solution** that fulfills the requirements.
 - Need to define software objects and how they collaborate to fulfill the requirements.
 - For example, in the Library Information System, a *Book* software object may have a *title* attribute and a *getChapter* method.
- Designs are **implemented** in a programming language.
 - In the example, we will have a *Book* class in Java.

The Object



Object Identity

Object State

Object behaviour

Object communication

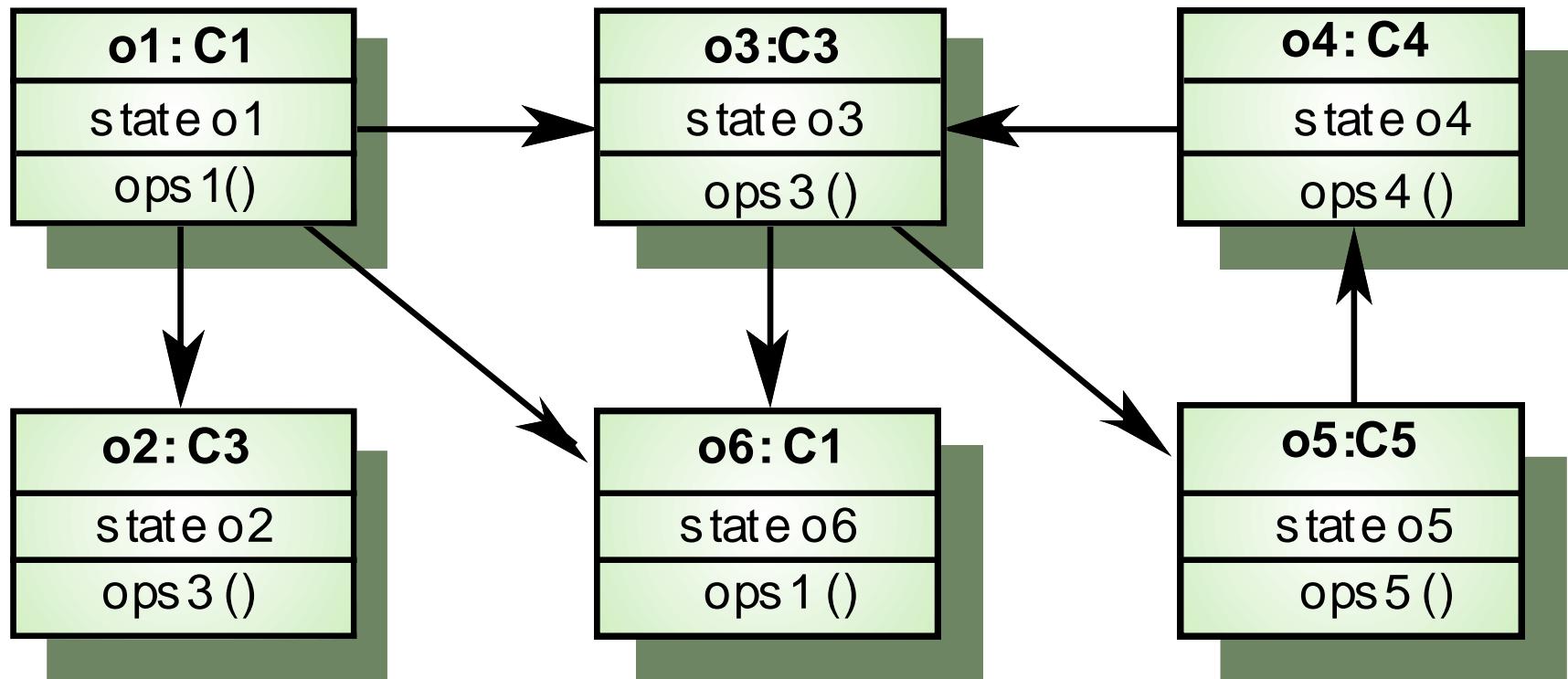
- Conceptually, objects communicate by message passing.
- Messages
 - The name of the service requested by the calling object.
 - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
 - Name = procedure name.
 - Information = parameter list.

Message examples

```
// Call a method associated with a buffer  
// object that returns the next value  
// in the buffer  
v = circularBuffer.Get();
```

```
// Call the method associated with a  
// thermostat object that sets the  
// temperature to be maintained  
thermostat.setTemp(20);
```

Interacting objects



Interactions - Basics

- **Interaction** — behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Realizes use cases
- Static representation of objects working to complete some action.
- Introduce messages between objects and classes.
 - Invocation of an operation.
 - Construction, or destruction, of an object.
- Models flow of control within an operation.
 - Identify how messages are dispatched across time.
 - Focus on the structural relationships among objects involved in an interaction.
- Properties of well-structured interactions
 - Efficient, Simple, Adaptable, Understandable

Interactions – Concrete vs. Prototypical Objects

- Objects within an interaction can be:
 - **Concrete** — something from the real world.
 - **Prototypical** — representative instance of something from the real world.
 - Collaborations use strictly prototypical things.
 - Prototypical instances of interfaces and abstract types are valid.
 - Can be identified from class diagrams.

Interaction Diagrams

- Interaction Diagrams are models that describe how groups of objects collaborate in some behavior.
- Interaction Diagrams provide a thoughtful, cohesive, common starting point for inspiration during programming
- Patterns, principles, and idioms can be applied to improve the quality of the Interaction Diagrams

On to Object Design

- During object design, a logical solution based on the object-oriented paradigm is developed. The heart of this solution is the creation of **interaction diagrams** which illustrates how objects collaborate to fulfill the requirements.
- After or in parallel with —drawing interaction diagrams, **class diagrams** can be drawn.

Introduction

- Why do objects exist?
 - To perform an activity to help fulfill a system's purpose
- Interaction Diagrams are used to model system dynamics
 - How do objects change state?
 - How do objects interact (message passing)?

Collaboration & Sequence Diagrams

- An Interaction Diagram is a generalization of two specialized UML diagram types
 - Collaboration Diagrams: Illustrate object interactions organized around the objects and their links to each other
 - Sequence Diagrams: Illustrate object interactions arranged in time sequence

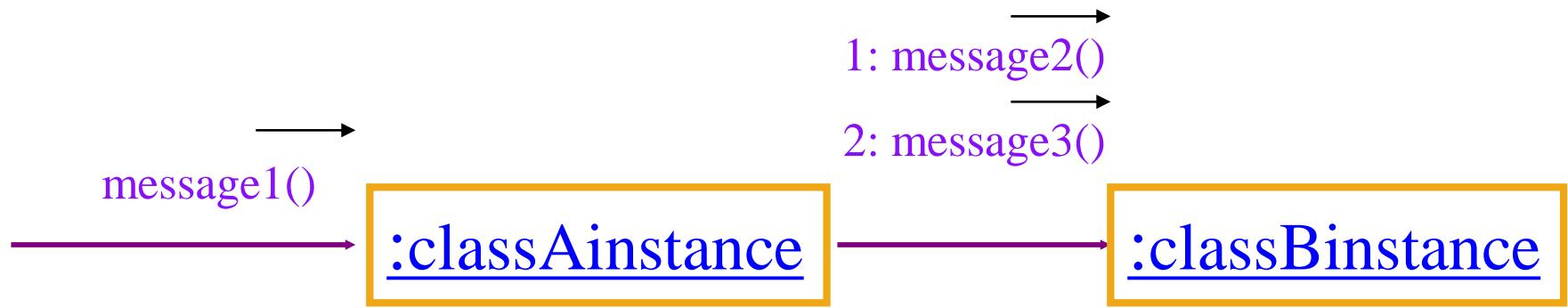
Collaboration & Sequence Diagrams (2)

- Both diagram types are semantically equivalent, however, they may not show the same information
 - Collaboration Diagrams emphasize the structural organization of objects, while Sequence Diagrams emphasize the time ordering of messages
 - Collaboration Diagrams explicitly show object linkages, while links are implied in Sequence Diagrams

Sequence vs. Collaboration Diagrams

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

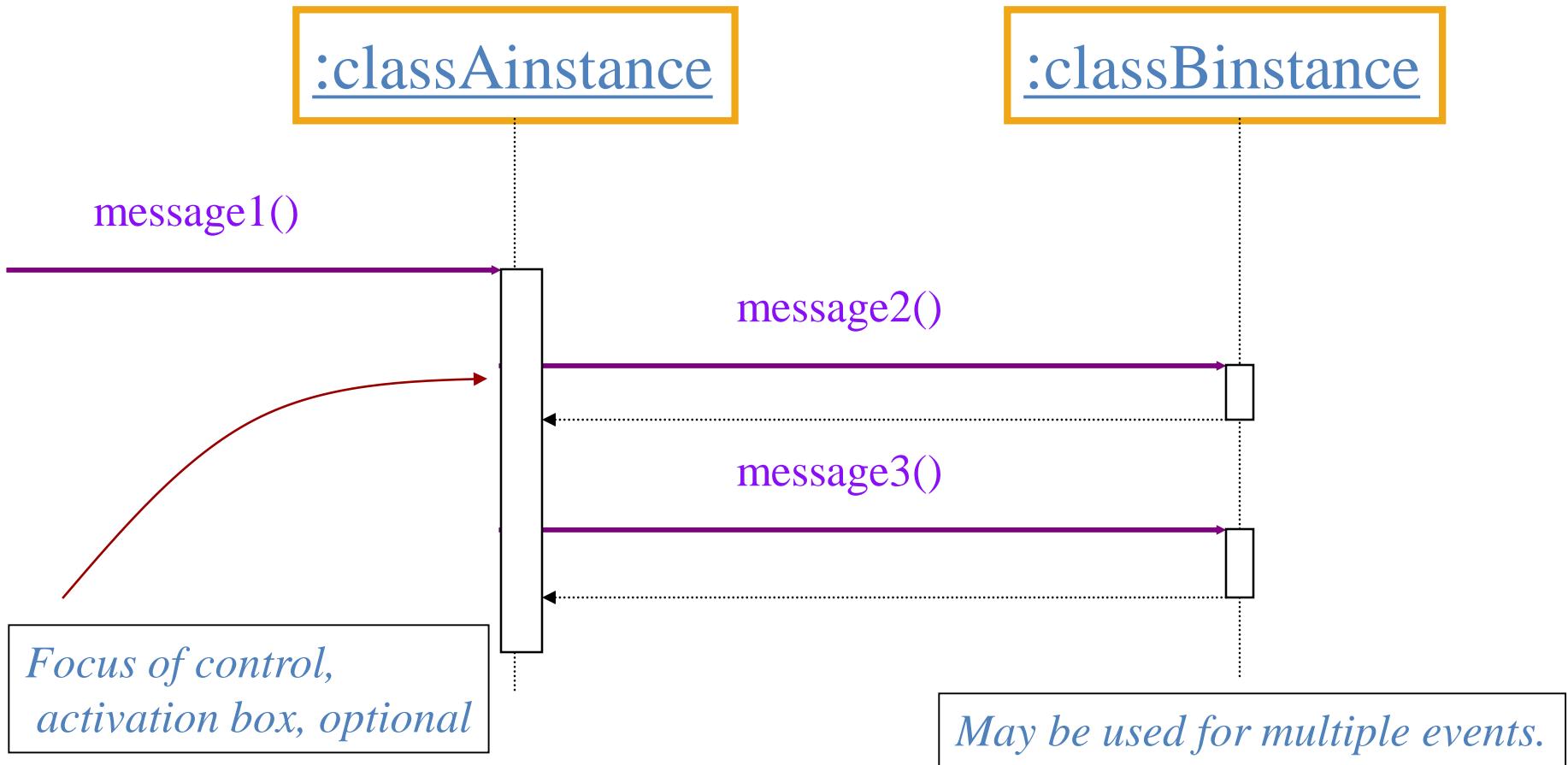
A collaboration diagram



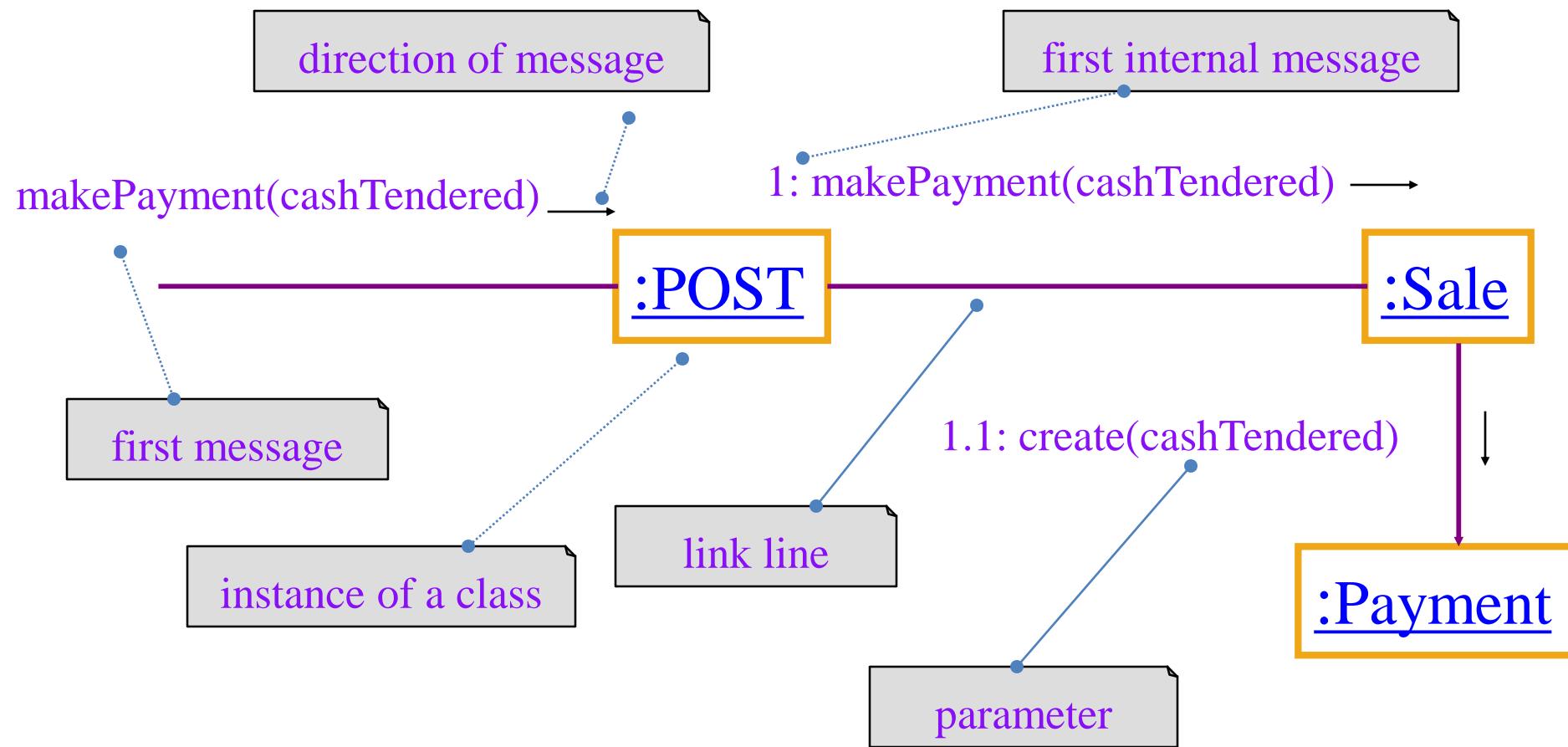
`message1()` is sent to an instance of class A.
`message2()` and `message3()` are sent, in that order, by an instance of class A to an instance of class B.

One diagram for each system event.

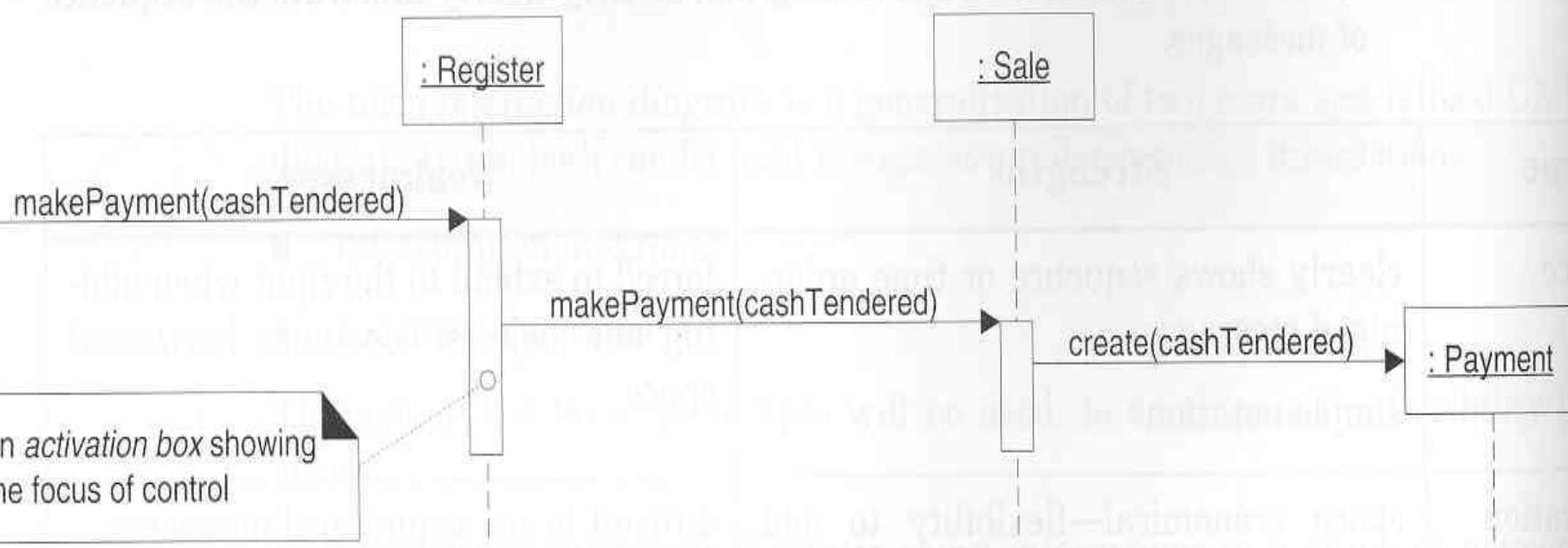
A sequence diagram



Collaboration diagram: makepayment()



Example SD: makePayment



[Larman, 2002]

Interaction Diagrams

Interaction diagrams

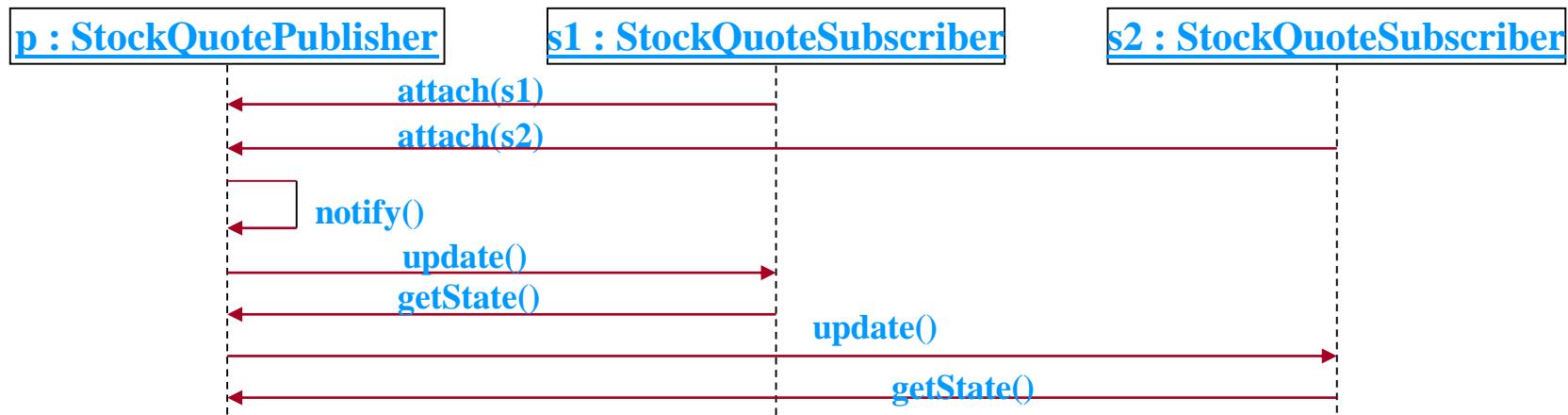
- ❖ *model the dynamic aspects of a system.*
 - ❖ *show the interaction of any kind of instance in any view of a system's architecture (classes, interfaces, components, and nodes).*
 - ❖ *model the system as a whole.*
 - ❖ *are attached to use cases to model a scenario.*
- **Sequence diagram**
 - Emphasizes **time ordering** of messages.
 - Depicts the lifeline of objects.
 - **Collaboration diagram**
 - Emphasizes **structural organization**.
 - Potentially easier to model complex interactions.

Modeling Flow of Control

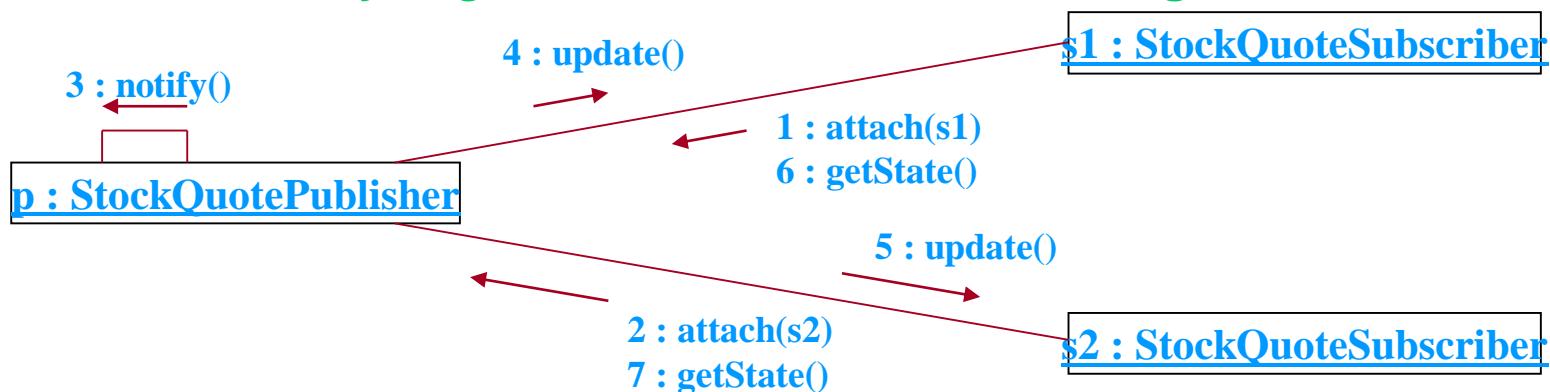
- Set the context for the interaction.
- Set the stage by identifying participating objects.
- Use in two (2) ways:
 1. Model flows of control by **time ordering**:
Specify the messages that pass from object to object in time order.
 2. Model flows of control by **organization**:
If emphasizing the structural organization, identify related links.

Flow of Control by Time vs. by Organization

Flow of Control by *Time* -> Sequence Diagram



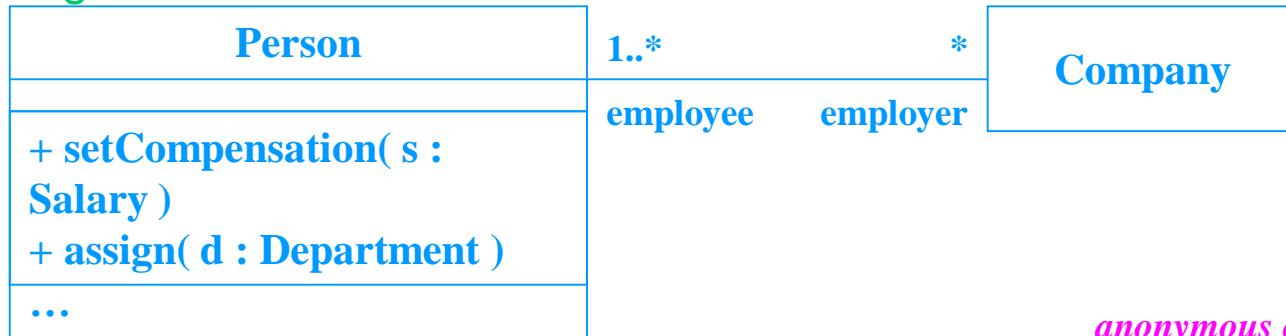
Flow of Control by *Organization* -> Collaboration Diagram



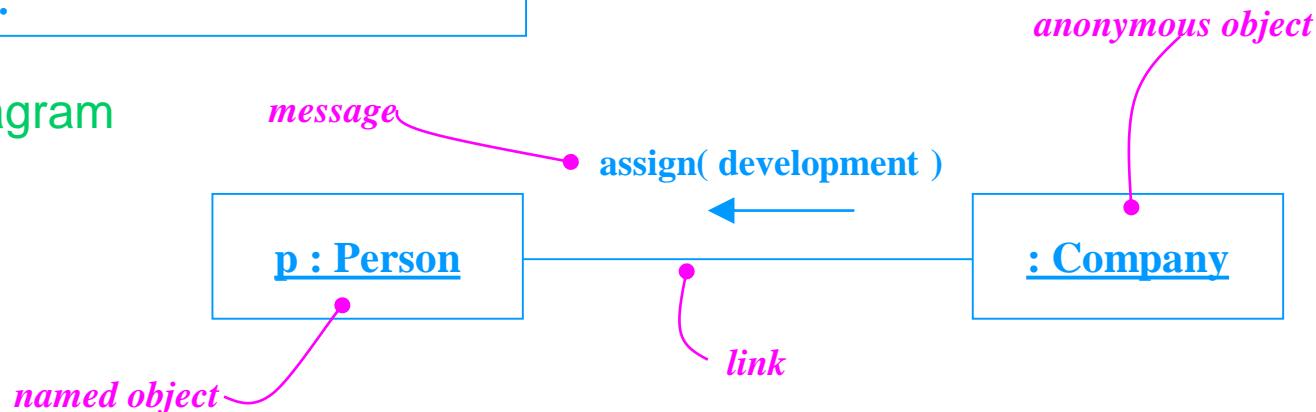
Interactions - Links

- Semantic connection among objects.
- An instance of an association or dependency.
- Provide a *passage for messages*.

Class Diagram

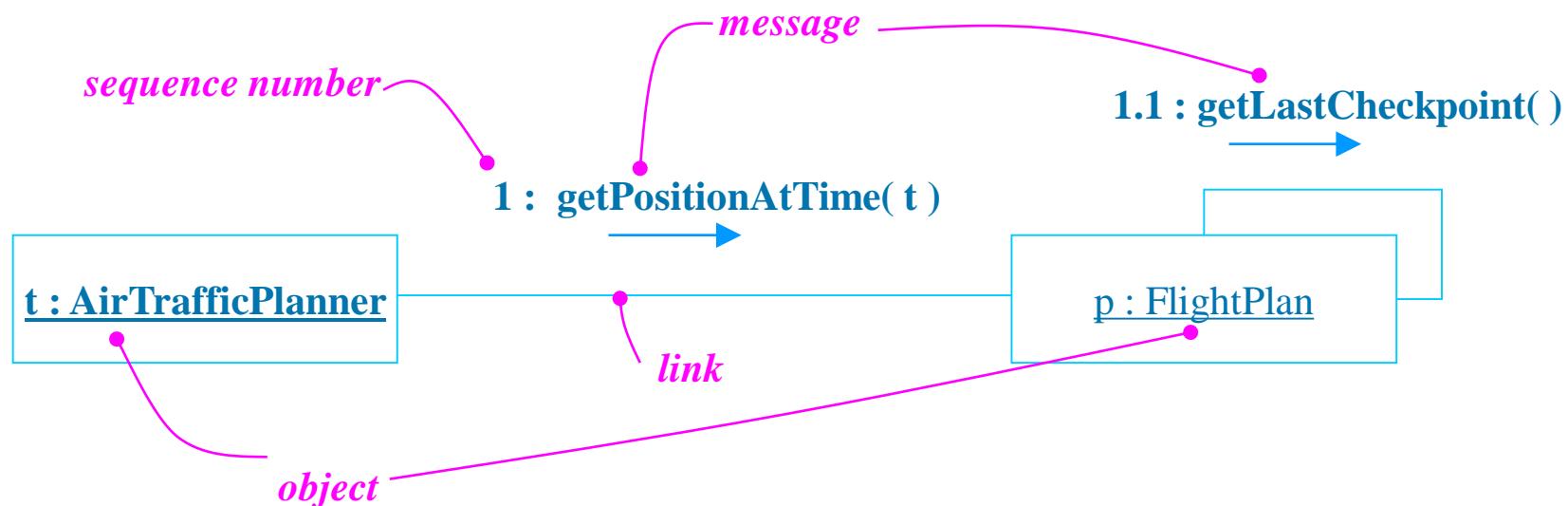


Object Diagram



Interactions - Messages

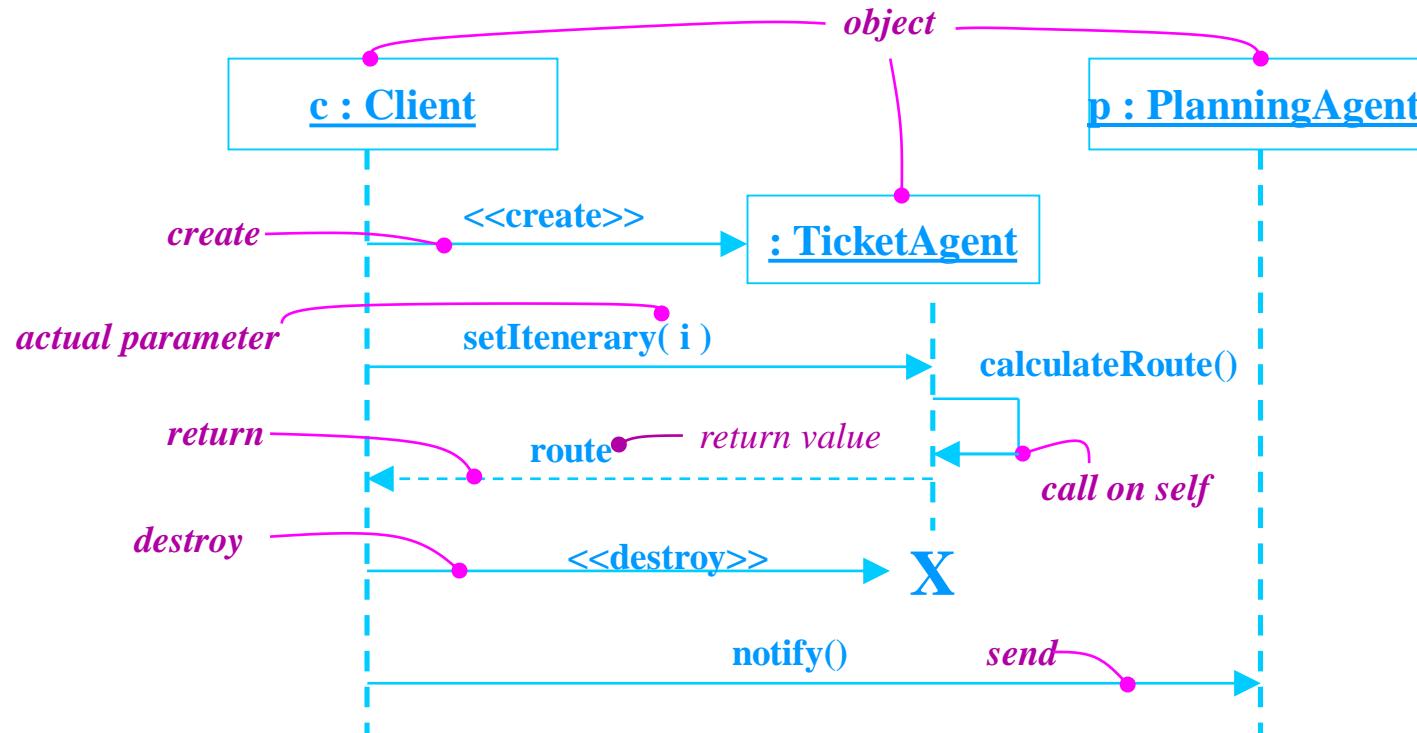
- **Message** — specification of a communication between objects that conveys information with the expectation that activity will ensue.
- Directed line near link.
- Name of operation must be defined on target.



Interactions - Modeling Actions

- **Call*** — invokes an operation on an object.
- **Return** — returns a value to the caller.
- **Send*** — sends a signal to an object.
- **Create** — creates an object.
- **Destroy** — destroys an object.

*Essentially the same, but send implies an event notification or control flow between objects



Interactions - Sequencing

- A stream of messages where actions are delegated forms a sequence.
- The beginning of each sequence is rooted in some process or thread.
- Each thread defines a distinct flow of control.
- Messages are ordered in sequence by time.
 - Represented by a sequence number followed by a colon.

Interactions - Procedural Sequencing vs. Flat Sequencing

Procedural Sequencing

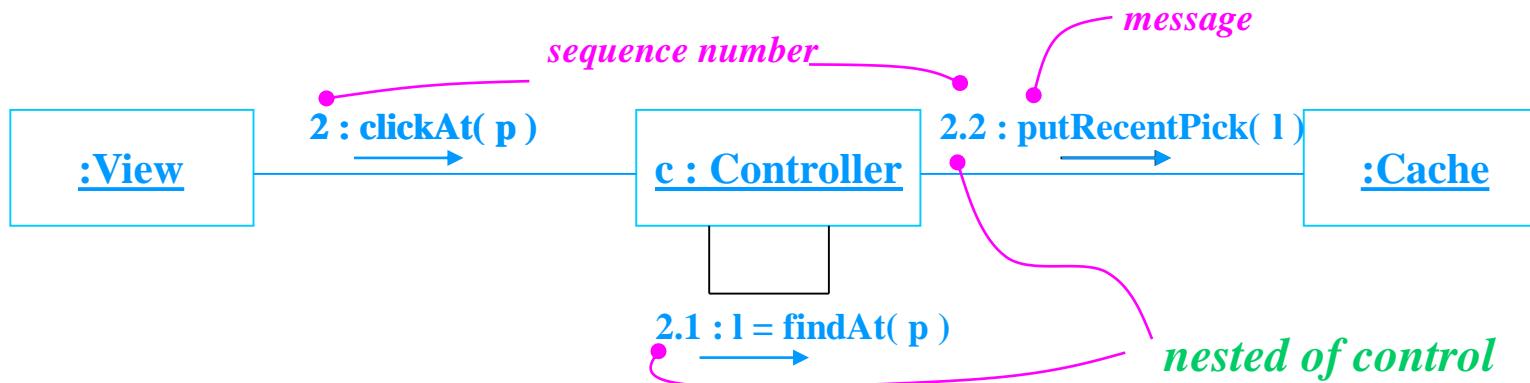
- Most common.
- Each message within the same operation is numbered sequentially.
- **Nested** messages are prefixed with the sequence number of the invoking operation.
- Rendered with filled **solid** arrow.

Flat Sequencing

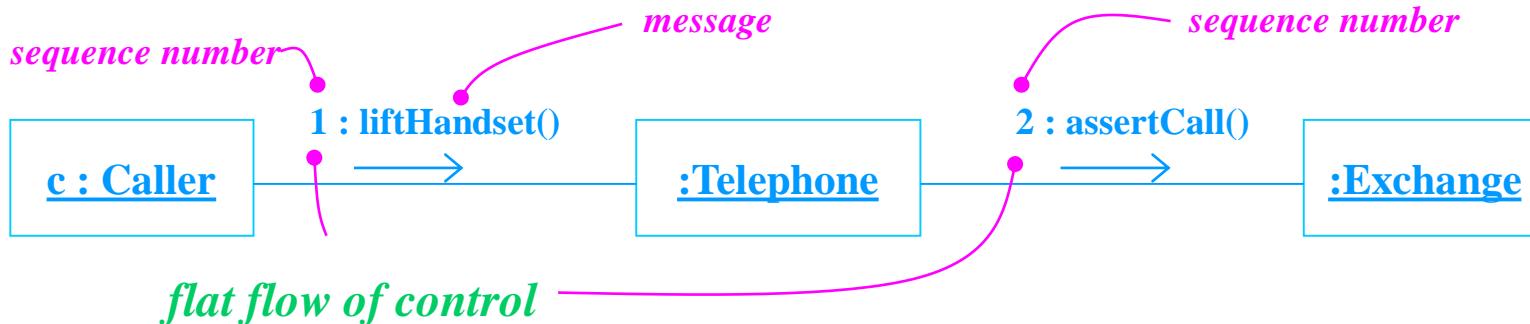
- Infrequent
Not recommended for most situations.
- Each message is numbered sequentially in order of timing.
- Shows progression of control.
- Rendered with **stick** arrowhead.

Interactions - Procedural Sequencing vs. Flat Sequencing

- Procedural Sequencing

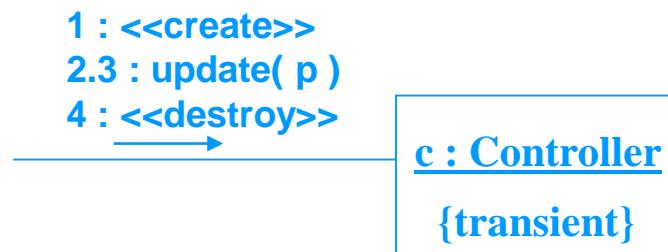


- Flat Sequencing



Interactions - Thread Identification, Creation & Destruction

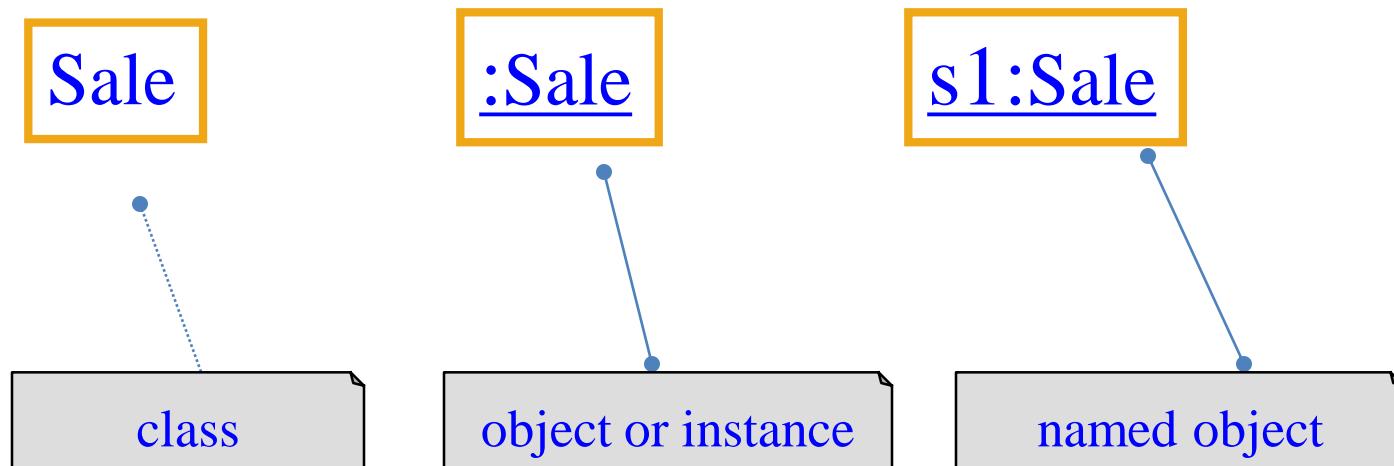
- Message sequence number prefixed with thread identifier.
E.g., *Th2 : openDoor (3)*
the operation openDoor is dispatched, with the actual argument 3, as the 2nd message in the sequence rooted by the process or thread named Th.
- Helps distinguish multiple threads of control.
- Consider coloring messages to match threads.
- Adorn with constraints.
 - new
 - destroyed
 - transient



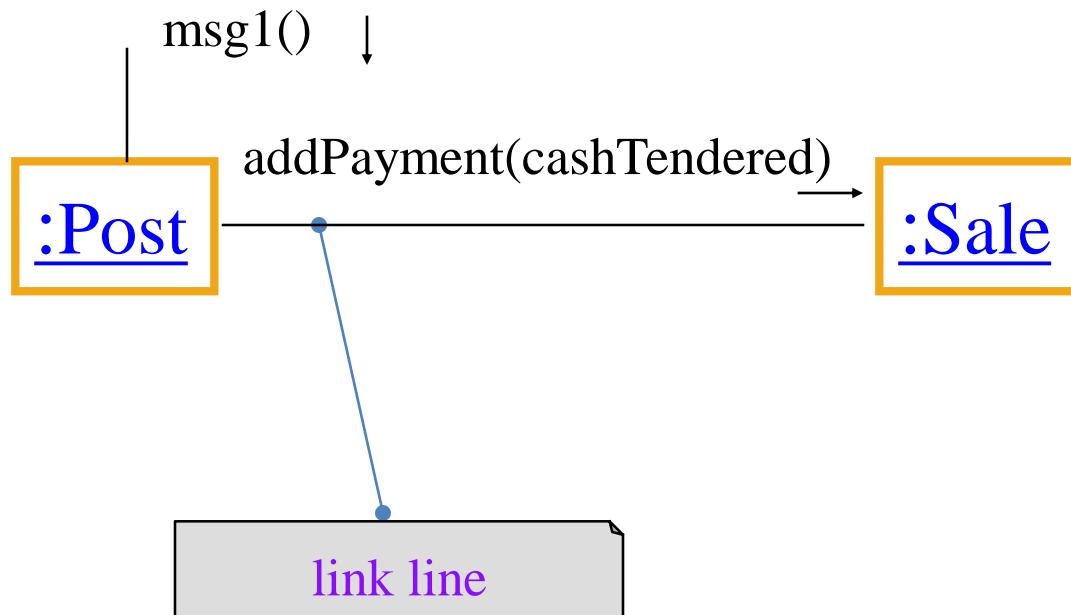
Making collaboration diagrams

- One diagram for each system operation in the current development cycle, e.g. for *makepayment()*.
- If the diagram gets complex, split into smaller diagrams.
- Design a system of interacting objects to fulfill the tasks....*this is the tough part of design!*

Classes and Objects

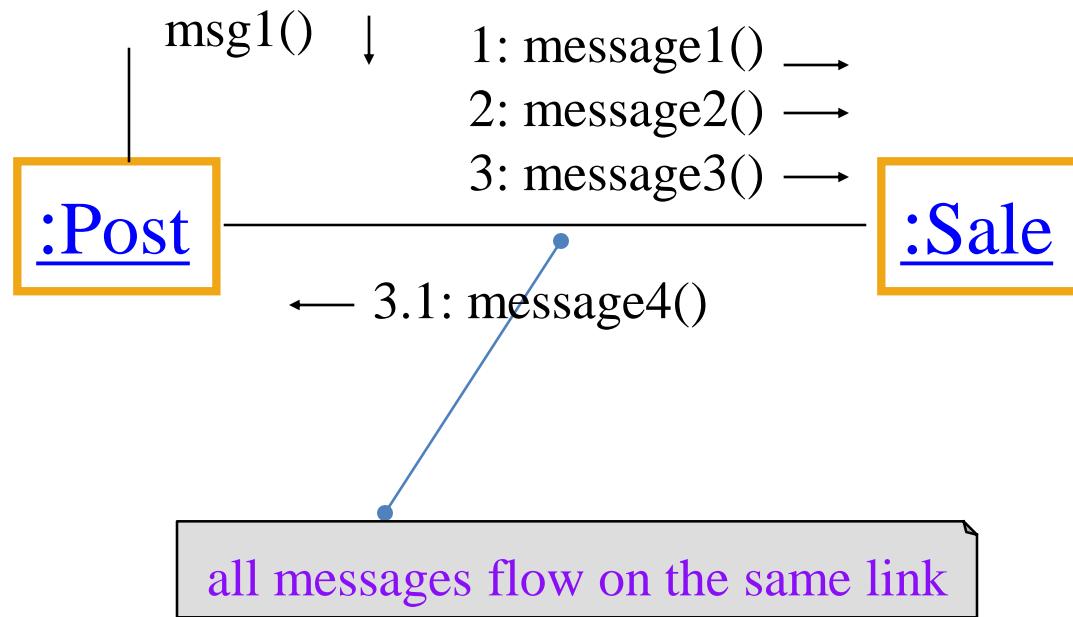


Links

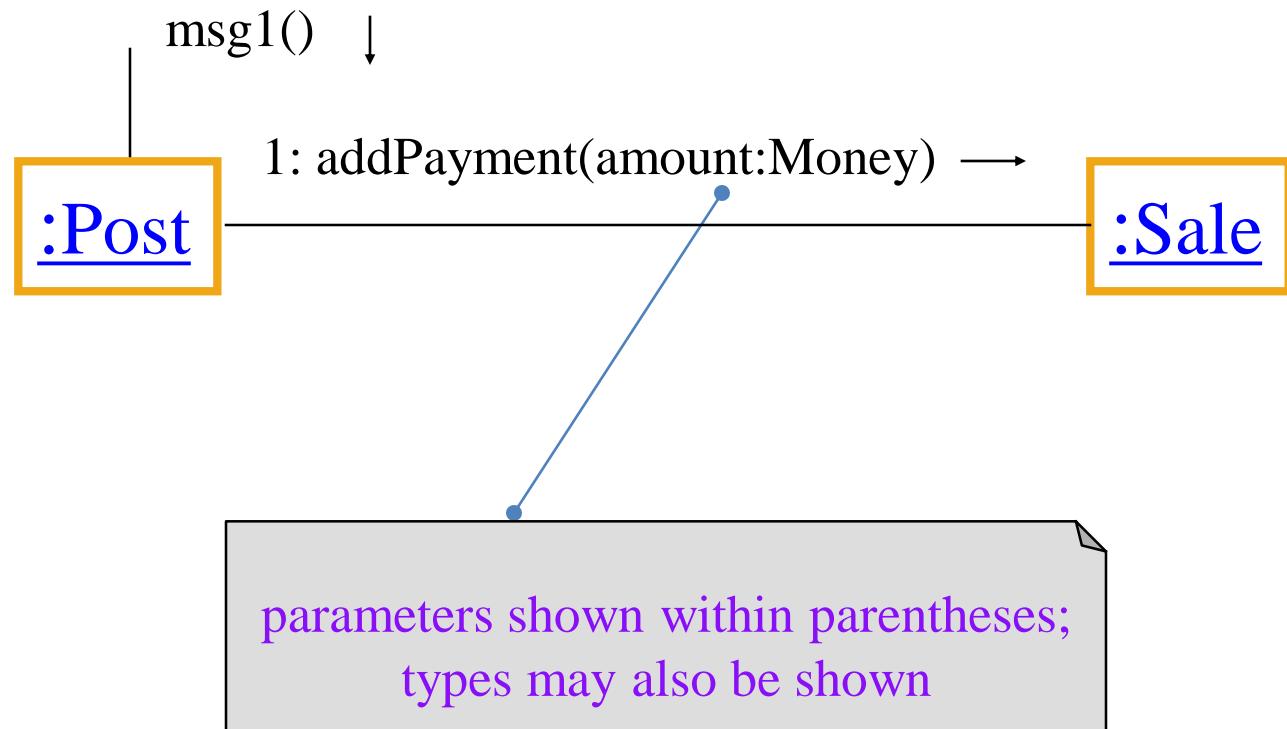


Link - connection path between two objects (an instance of an association)

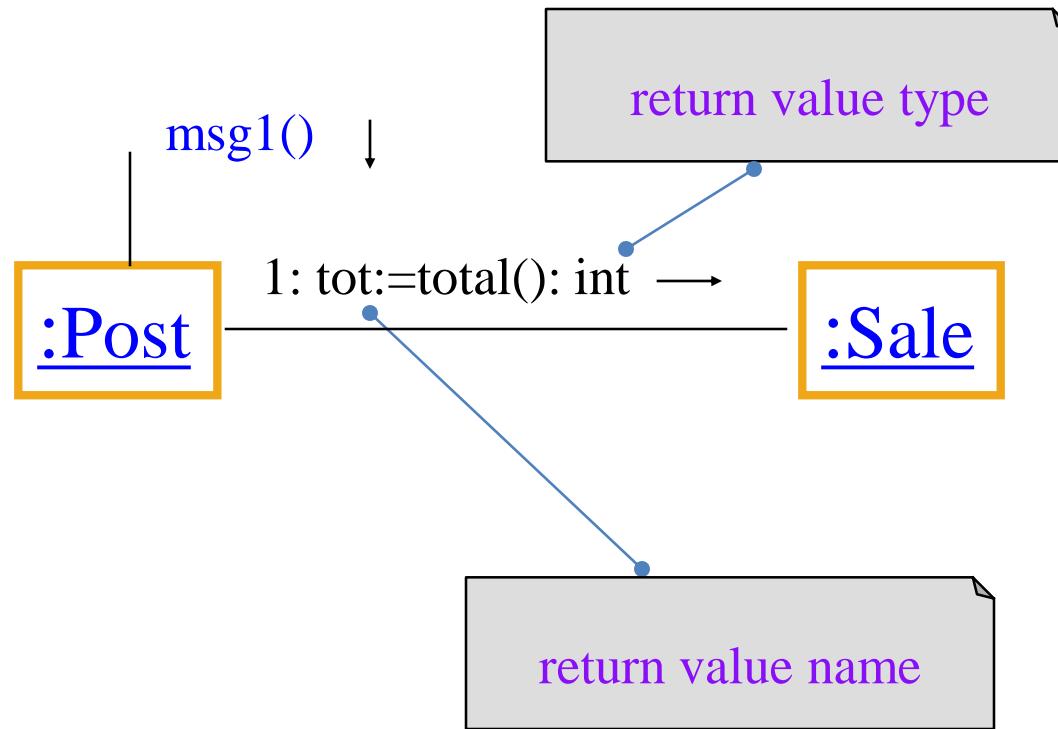
Messages



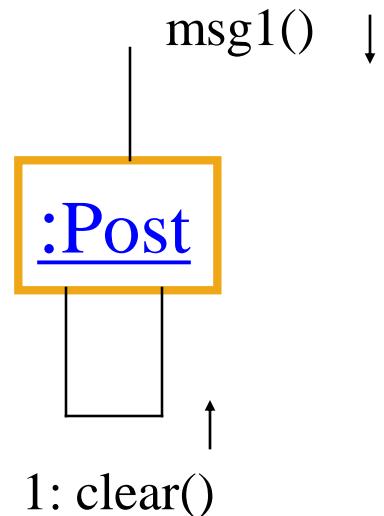
Parameters



Return values



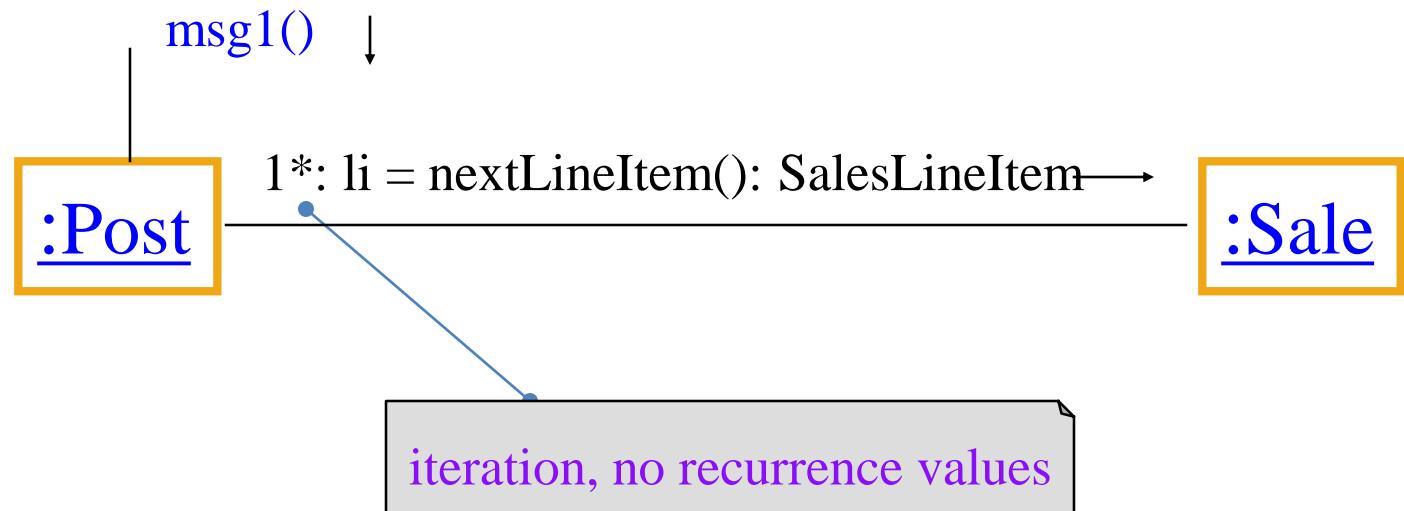
Messages to “self” or “this”



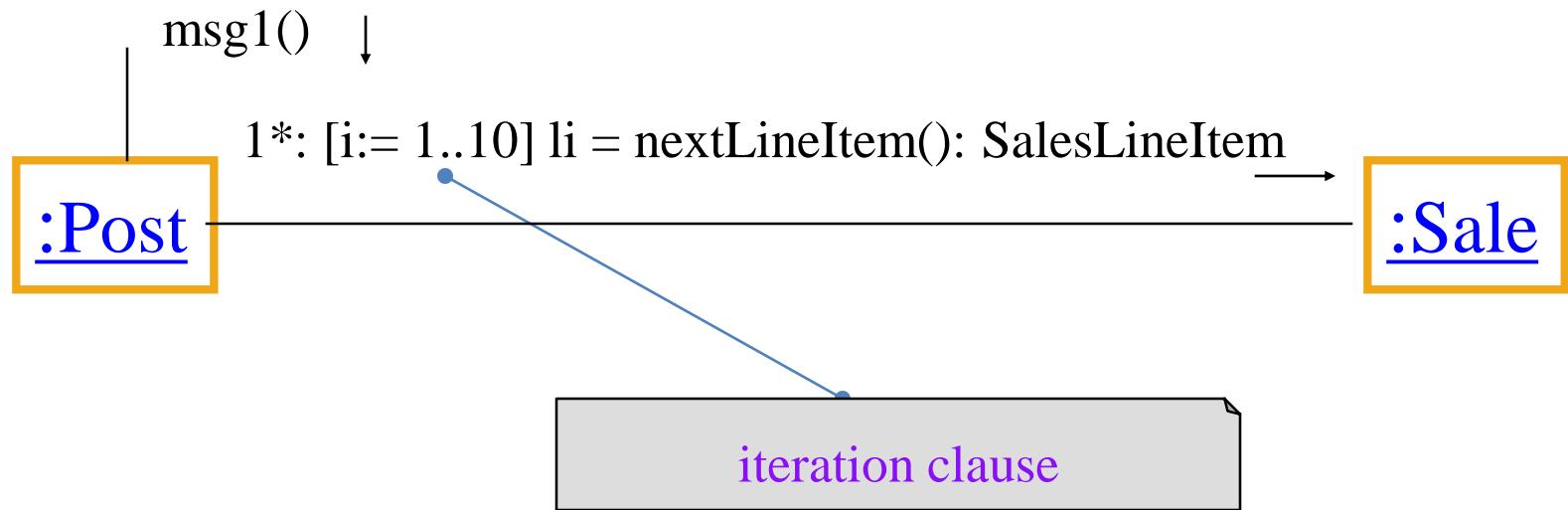
Iteration

Iteration (Looping)

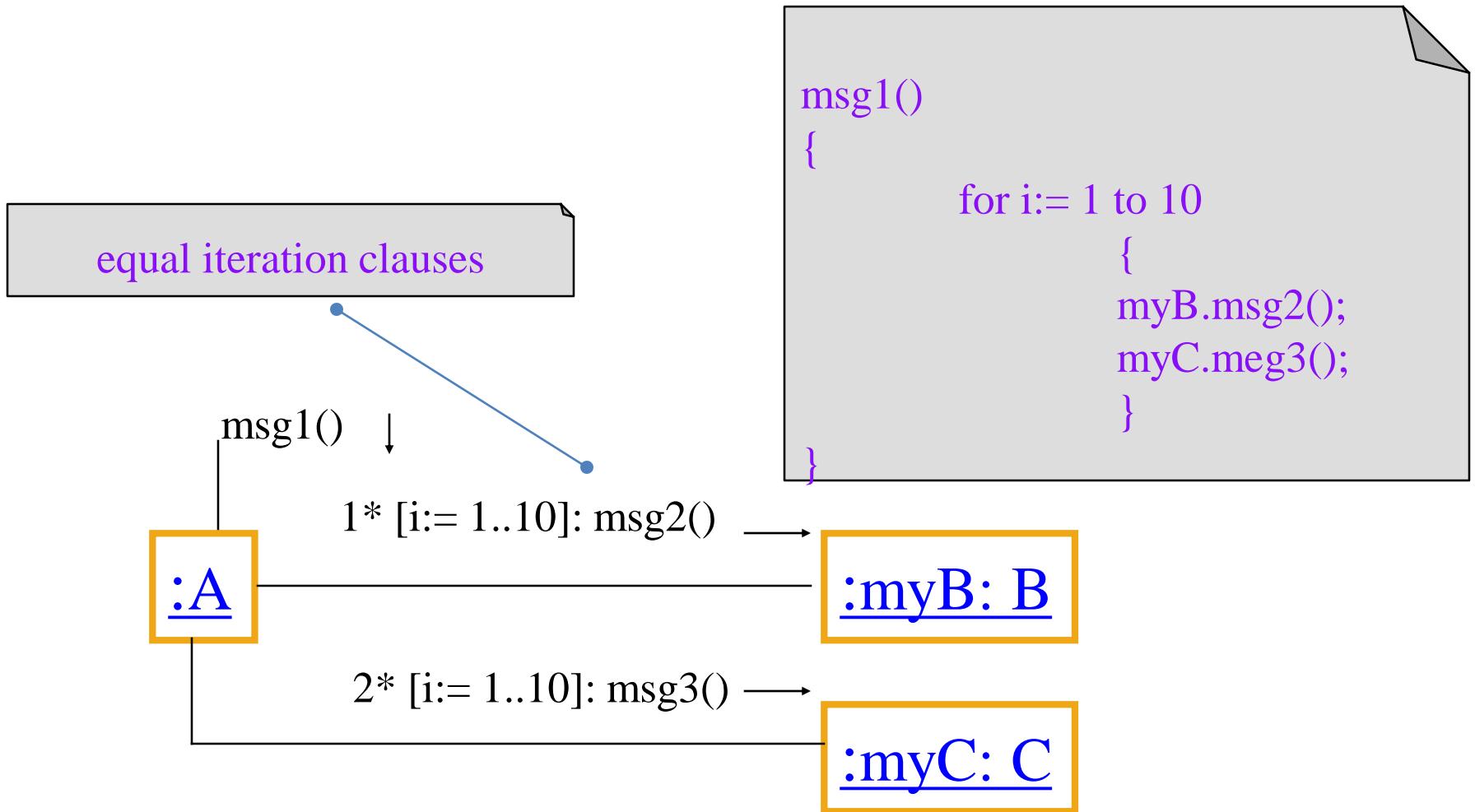
- Seq. Number $^* [i := 1..N]$: message()
- “ * ” is required; [...] clause is optional



Iteration with recurrence



Iteration clause: Multiple messages



Languages and *create*

Language	Meaning of <i>create</i>)
C++	Automatic allocation, or <i>new()</i> operator followed by constructor call.
Java	<i>new()</i> operator followed by a constructor call.

Message sequence numbering

first message not numbered

msg1()

:classA

second

1: msg2()

:classB

2: msg4()

:classC

fourth

third (nested)

1.1: msg3()

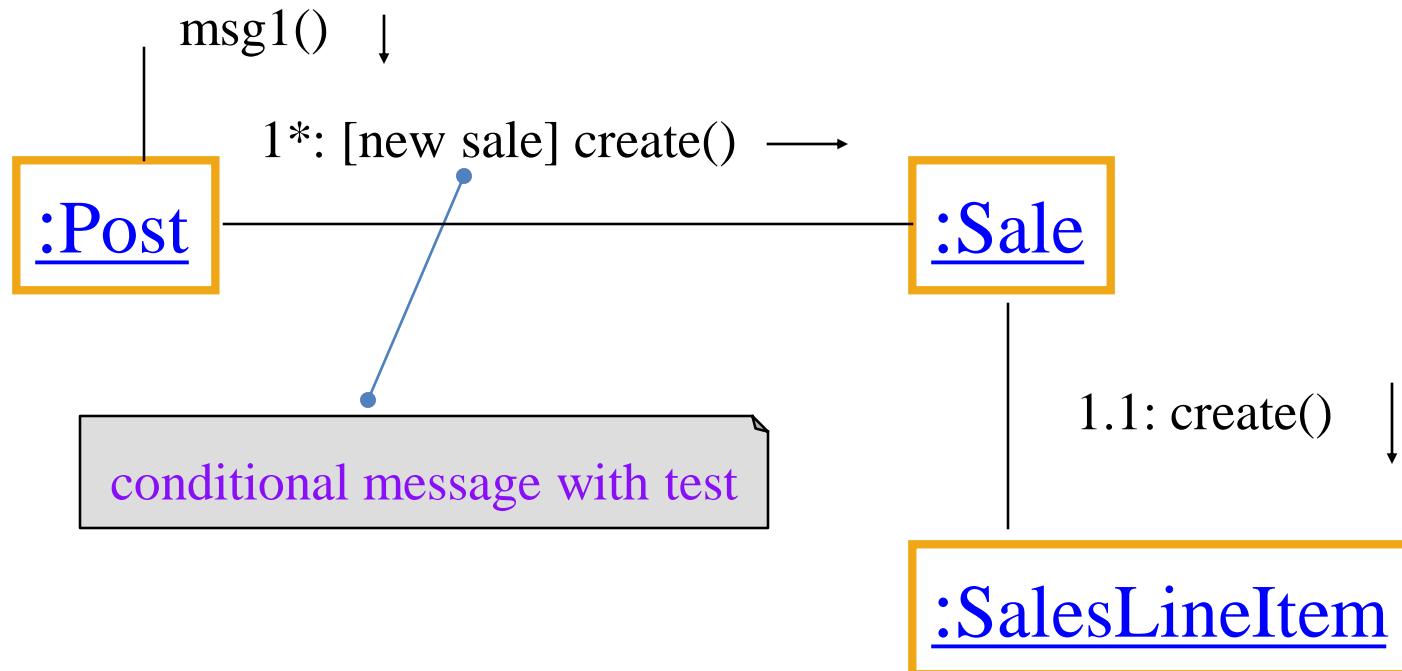
2.1: msg5()

fifth (nested)

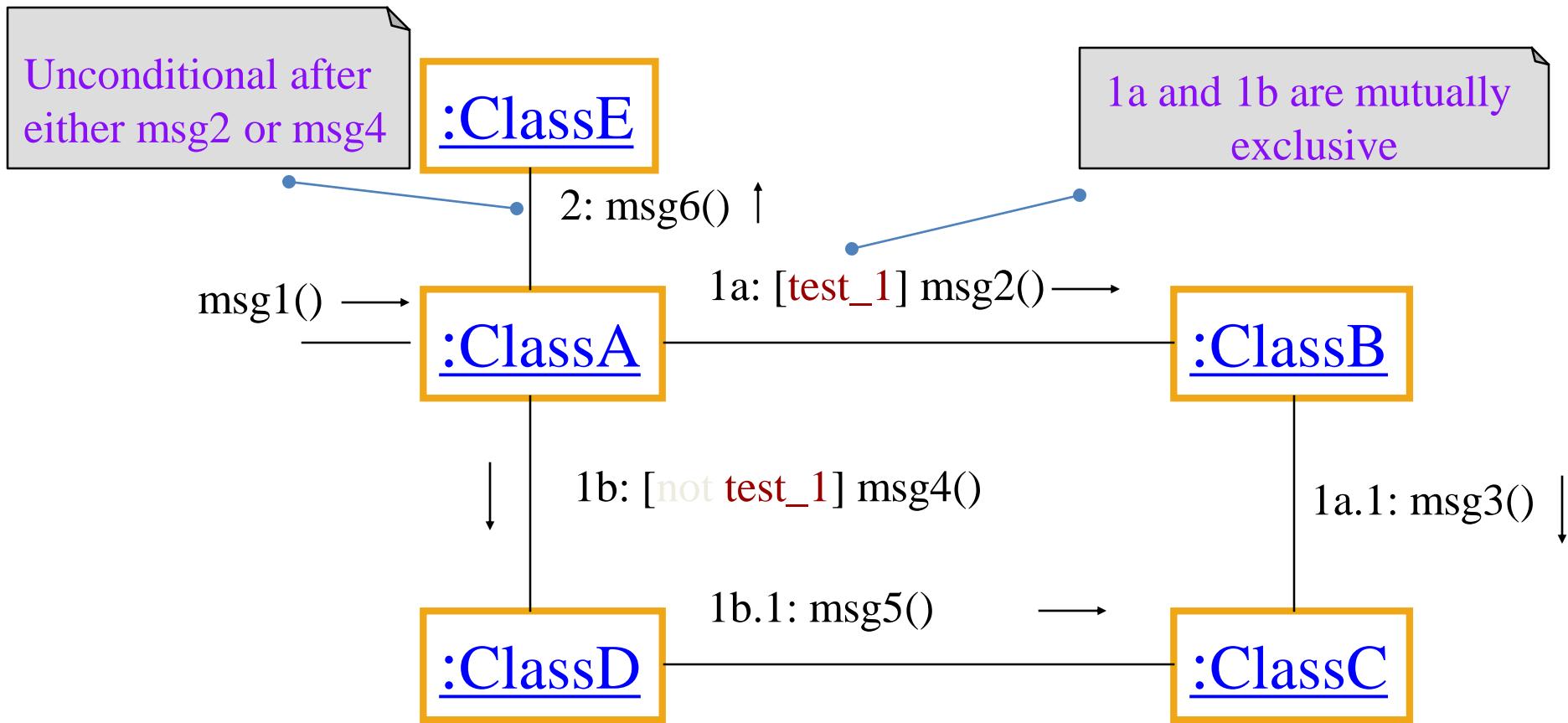
Conditional message

Conditional Message

- Seq. Number [*variable* = *value*] : message()
- Message is sent only if clause evaluates to *true*



Mutually exclusive conditional messages



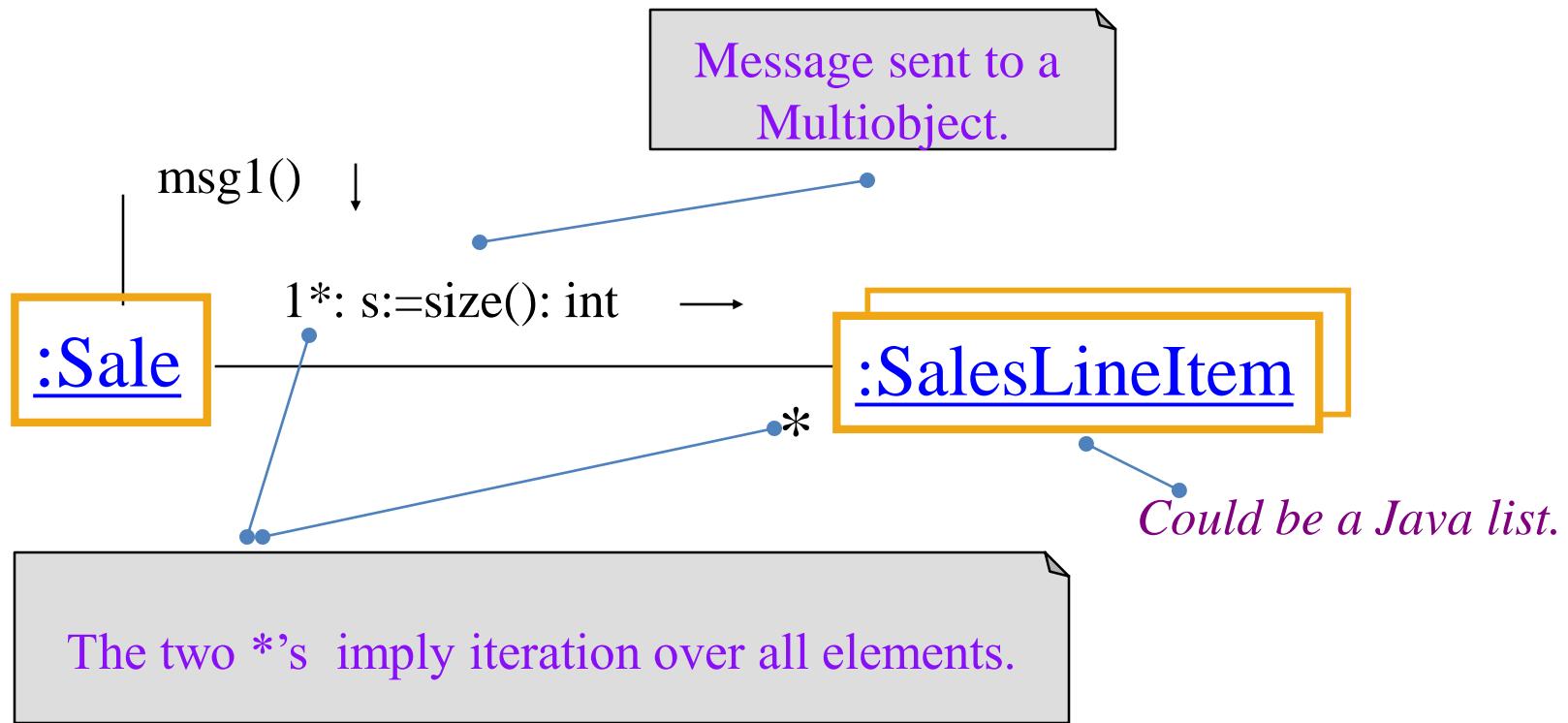
Collections

[sales:Sale](#)

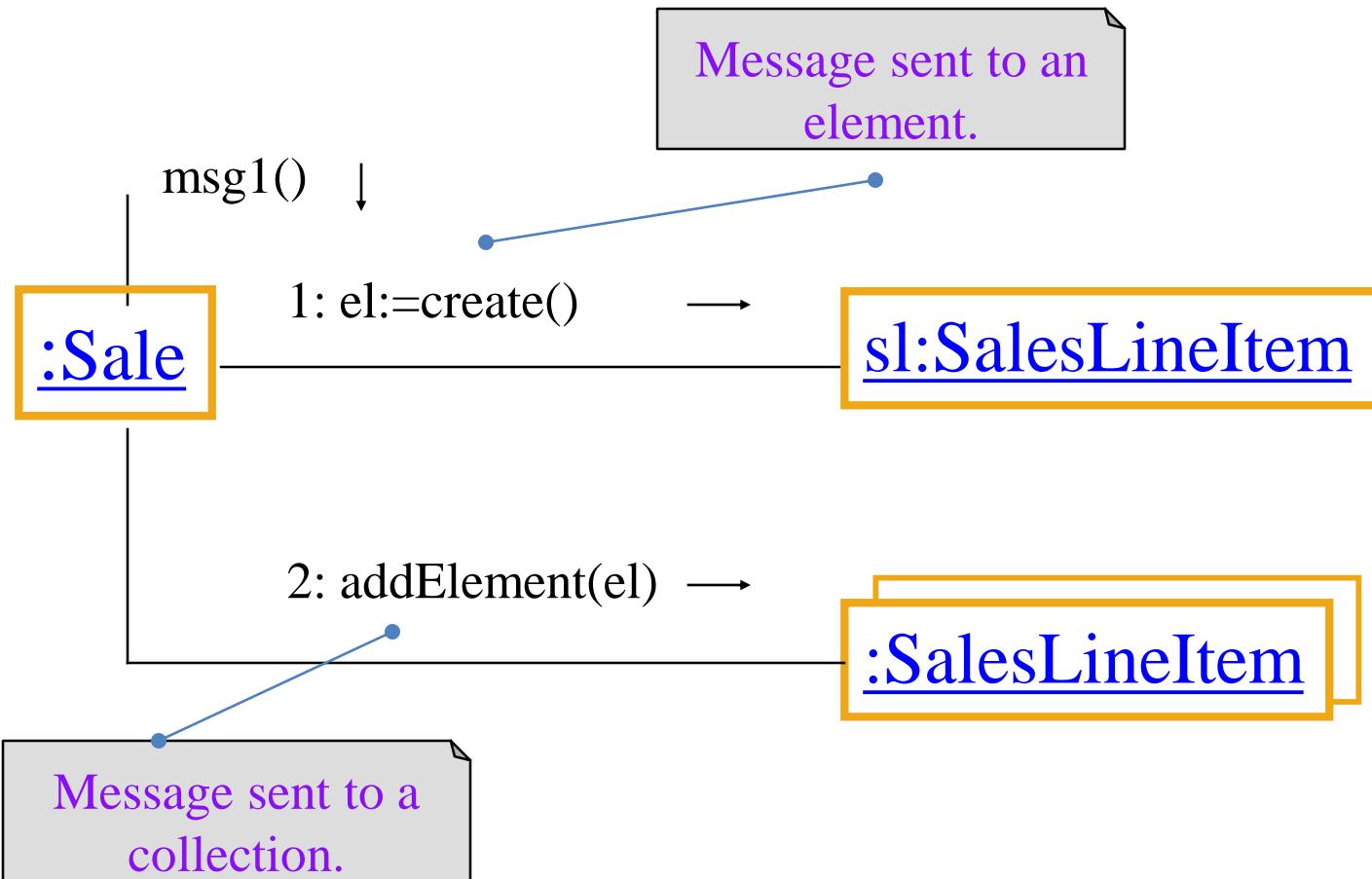


A multiobject or a collection of instances,
e.g. a List of Sales.

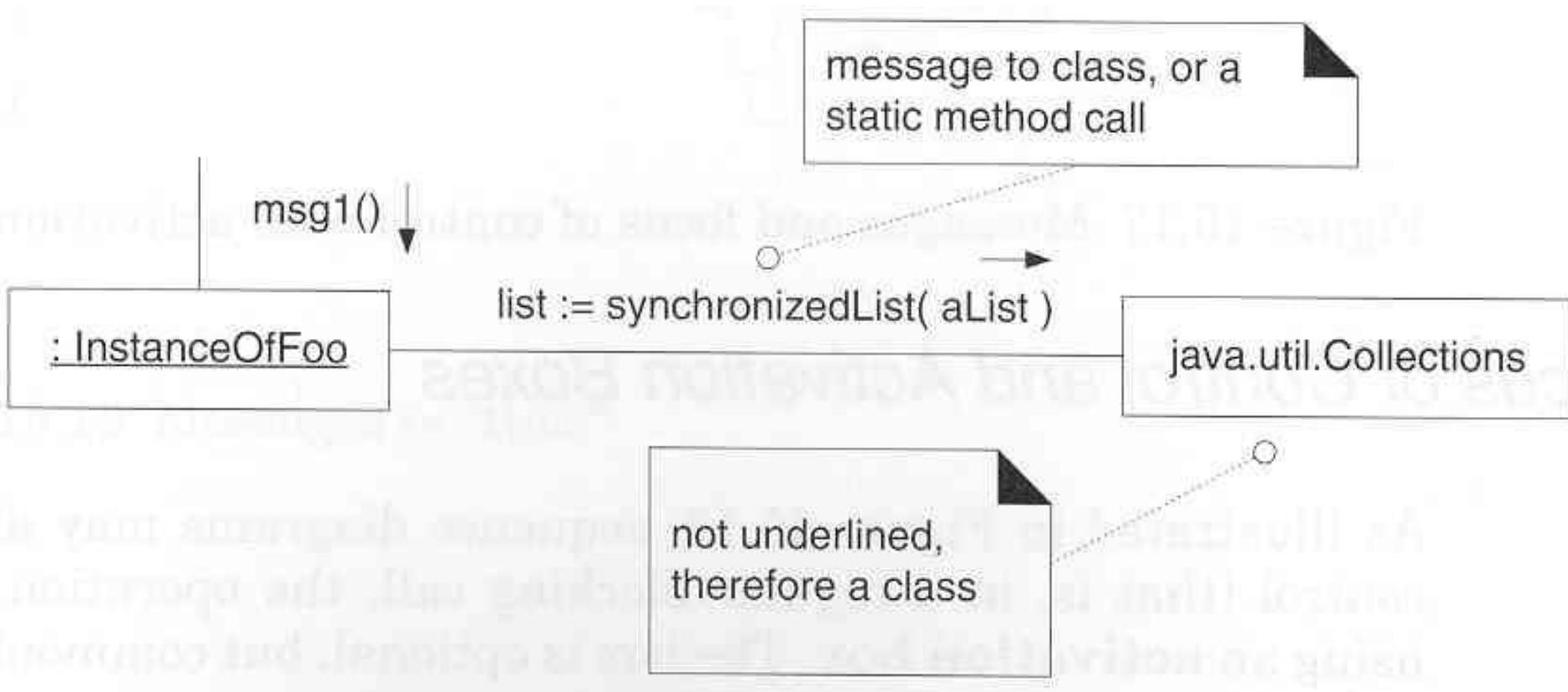
Messages to Multiobjects



Messages to multiobjects and an element [1]



CDs: Messages to a Class



[Larman, 2002]

Design Guidelines

- OO system consists of *interacting objects*.
- How does one determine the *assignment of responsibilities* to various objects?
- There is *no unique assignment* and hence “good” and “poor” designs, “beautiful” and “ugly” designs, “efficient” and “inefficient” designs.
- Design guidelines assist in the derivation of a good design.

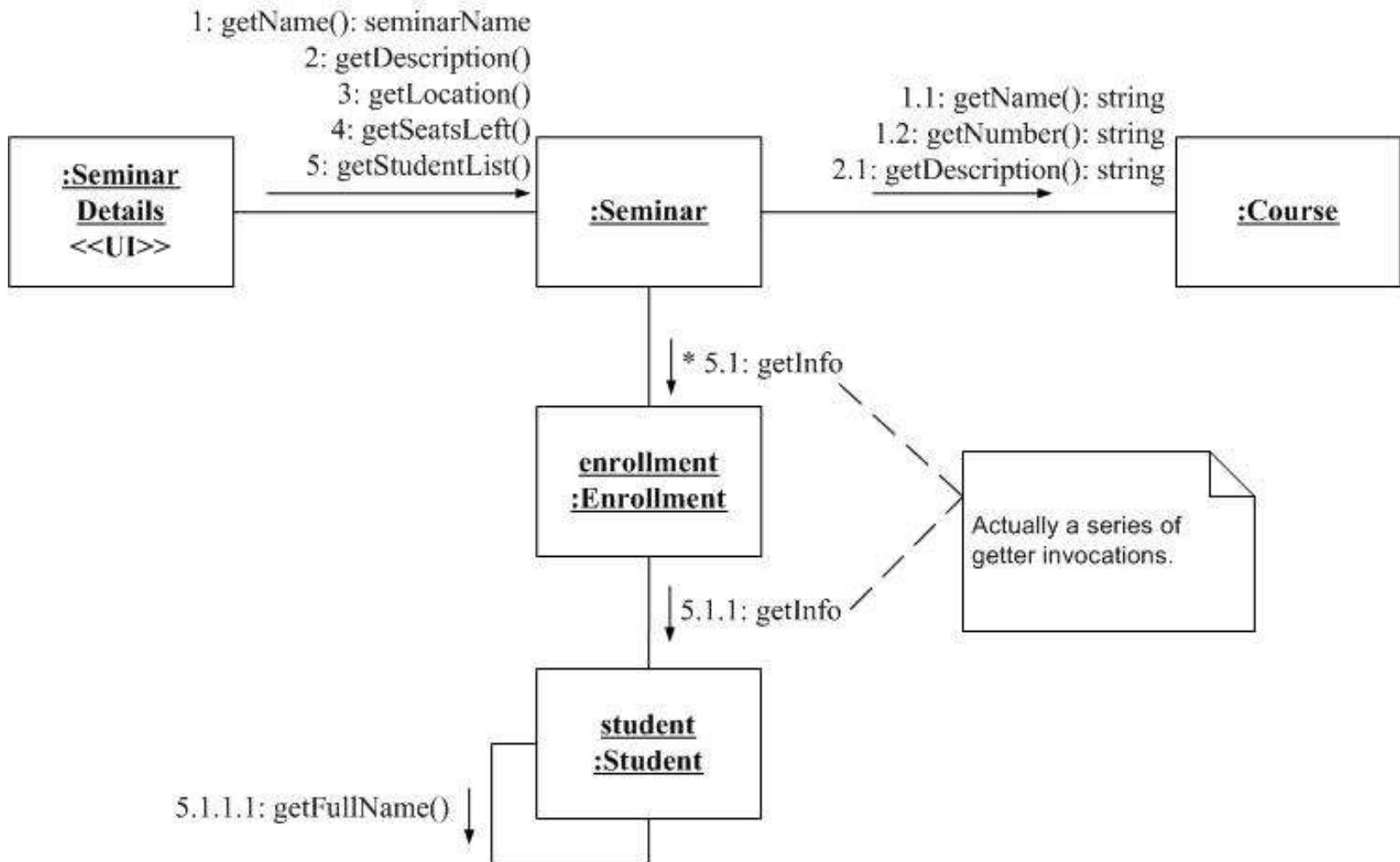
Sequence Diagrams

- Correspond to one scenario within a Use Case
- Model a single operation within a System over time
- Identify the objects involved with each scenario
- Identify the passed messages and actions that occur during a scenario
- Identify the required response of each action

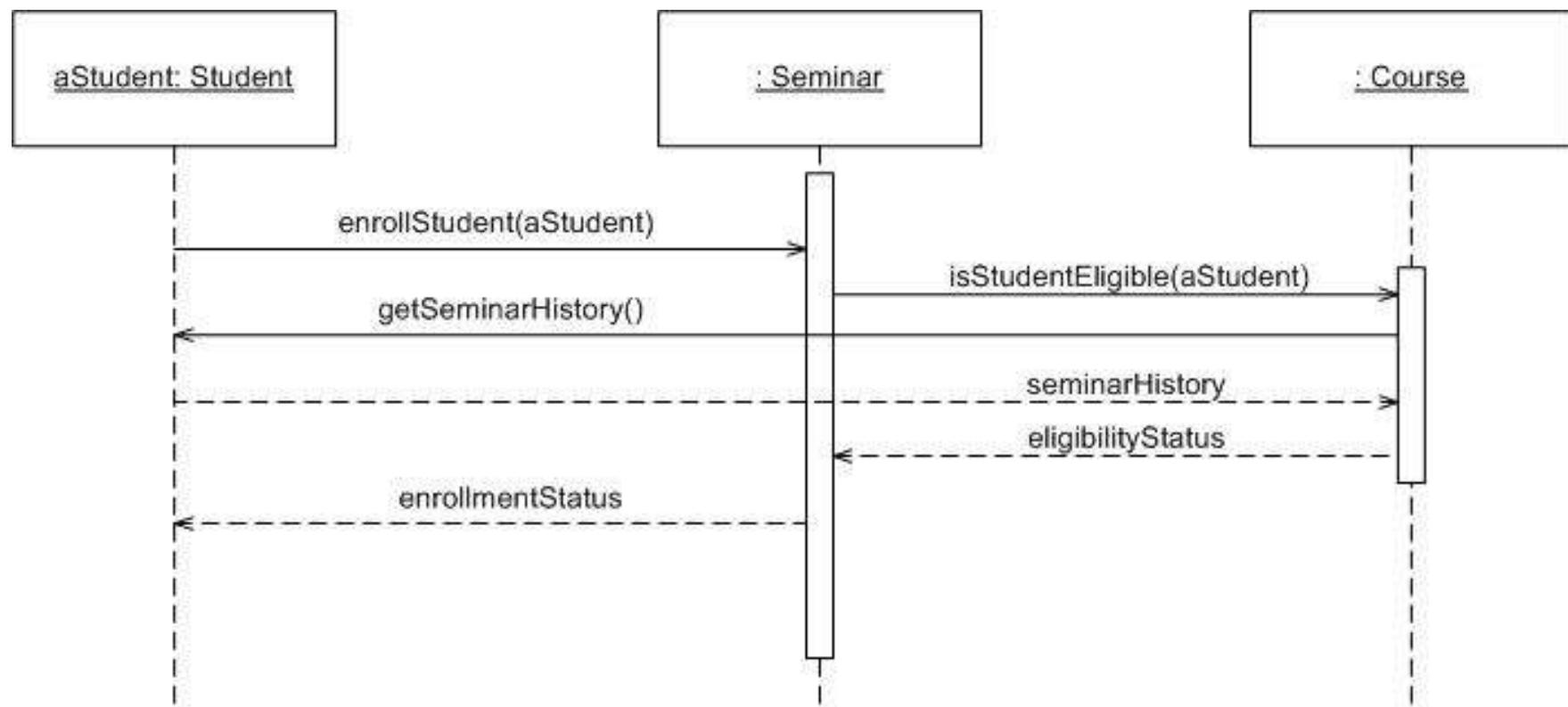
Notes: Collaboration Diagrams

- It is legal to use a collaboration diagram for each system event message.
- Each message between objects is represented with a message expression and small arrow indicating the direction. Many messages may flow along this link. A sequence number may be added to show the sequential order of messages, for example:
1:msg2()
- The order and nesting of subsequent messages is shown with a legal numbering scheme. Nesting is denoted by pre-pending the incoming message number to the outgoing message number.
For example: Message 1. and the nested message 1.1

Communication Diagram



Sequence Diagram



Sequence diagram exercise: Poker use case, *Start New Game Round*

The scenario begins when the player chooses to start a new round in the UI. The UI asks whether any new players want to join the round; if so, the new players are added using the UI.

All players' hands are emptied into the deck, which is then shuffled. The player left of the dealer supplies an ante bet of the proper amount. Next each player is dealt a hand of two cards from the deck in a round-robin fashion; one card to each player, then the second card.

If the player left of the dealer doesn't have enough money to ante, he/she is removed from the game, and the next player supplies the ante. If that player also cannot afford the ante, this cycle continues until such a player is found or all players are removed.

Messages

A **message** is a request from an object asking a second object to carry out a behavior (an operation) belonging to the second object.

A message specifies:

- the **identity** of the object to which it is sent,
- the **name of the operation**, and possibly,
- the **parameters** of the message.

Responsibilities

The principal task of object-oriented program design is to **assign responsibilities to classes**.

A **responsibility** is an obligation of an object to other objects.

Responsibilities

(continued)

An object may be responsible for knowing:

- What it knows – **its attributes**
- Who it knows – the **objects associated** with it
- What it knows how to do – the **operations** it can perform

Responsibilities

(continued)

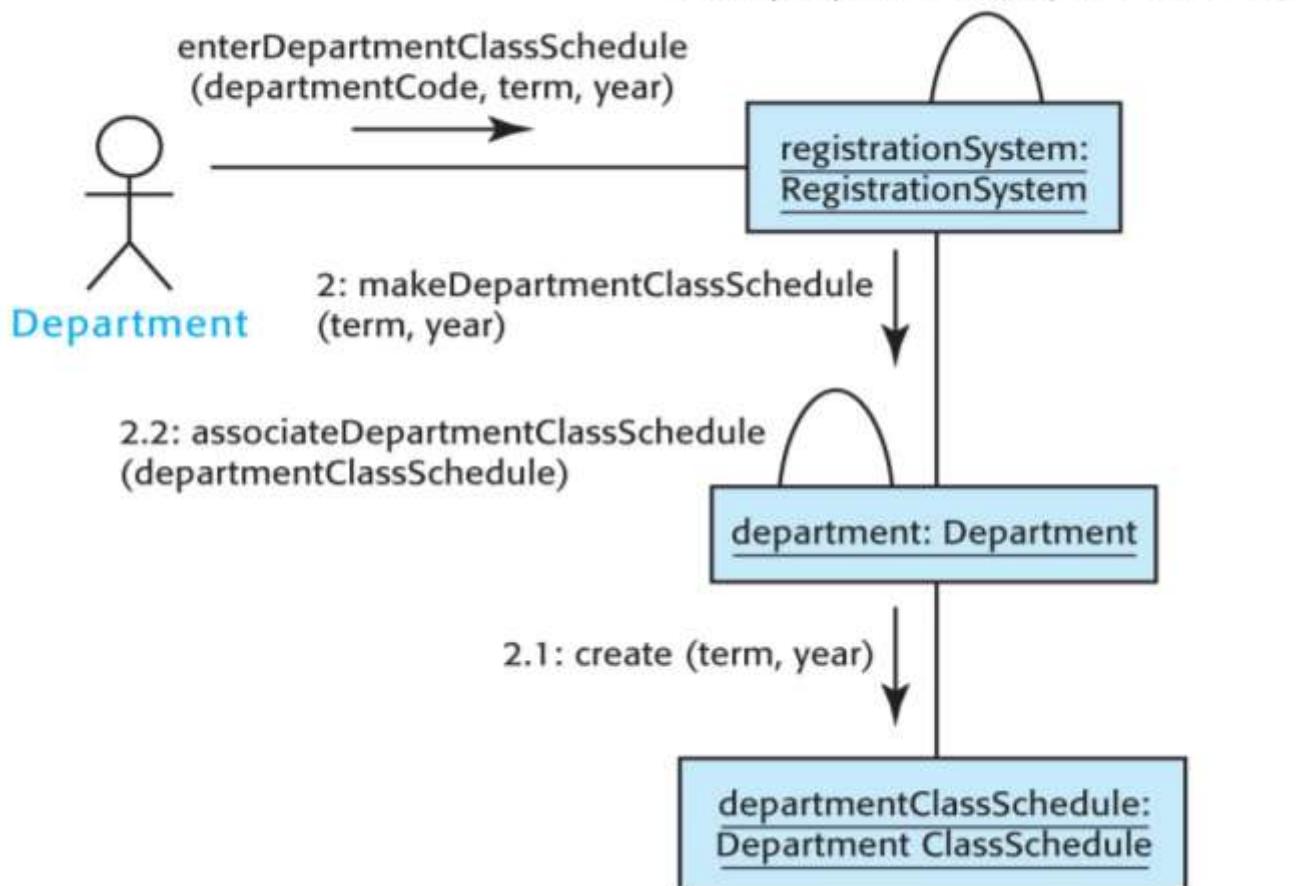
An object may also be responsible for:

- Doing something itself
- Requesting services from other objects
- Controlling and coordinating the activities of other objects

Collaboration Diagrams

(continued)

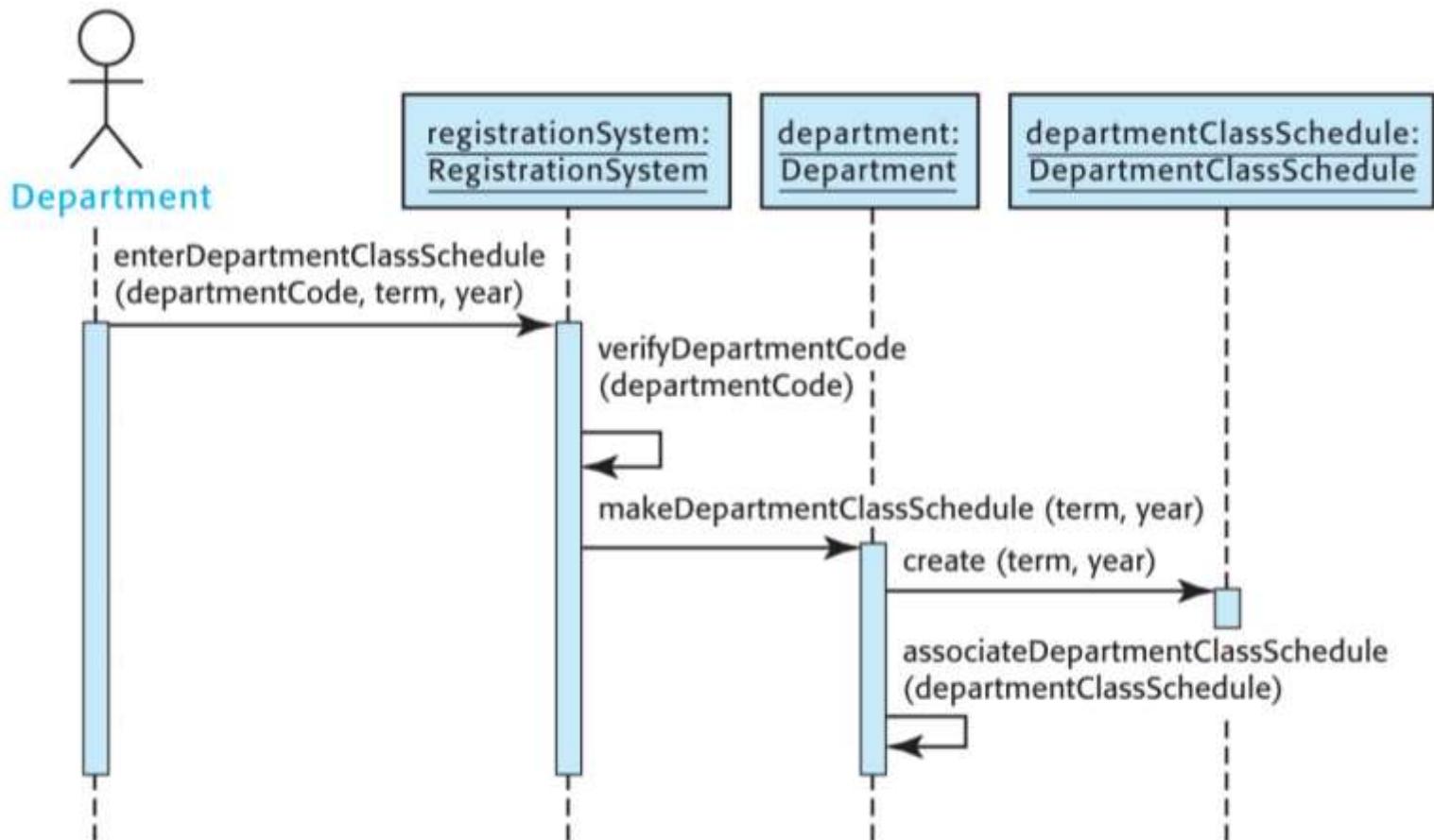
FIGURE 8.5



Sequence Diagrams

(continued)

FIGURE 8.6





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.8

Today's Agenda

- GRASP Patterns

Definitions and names

- Alexander: “A *pattern* is a recurring *solution* to a standard *problem*, in a *context*.”
- Larman: “In OO design, a *pattern* is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs.”
- How is Larman’s definition similar to Alexander’s?
- How are these definitions significantly different?

Naming Patterns—important!

- Patterns have suggestive names:
 - Arched Columns Pattern, Easy Toddler Dress Pattern, etc.
- Why is naming a pattern or principle helpful?
 - It supports chunking and incorporating that concept into our understanding and memory
 - It facilitates communication

Star and Plume Quilt



The GRASP Concept

- **GRASP** stands for
 - General
 - Responsibility
 - Assignment
 - Software
 - Patterns
- The mnemonic idea is, successful OOA/D requires “grasping” the GRASP principles.

These are best considered as design guidelines and principles.
These could also be considered as “thought patterns” useful in software design.

GRASP Concept (cont.)

General **R**esponsibility **A**ssignment **S**oftware **P**atterns

- General = Abstract; **widely applicable**
- Responsibility = **Obligations**, duties, contract
- Assignment = **Giving a responsibility to a module**
- Software = **Computer code**
- Patterns = Regularities, templates, **abstraction**

GRASP : Designing Objects With Responsibilities

- GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational,explainable way.
- Approach to understanding and using design principles is based on patterns of assigning responsibilities.

GRASP Responsibilities

- Responsibilities – obligations or contract (of an object)
- Responsibilities (obligations) have 2 types
 - What the object is responsible for **doing**
 - What are some “doing”-type responsibilities?
 - What the object is responsible for **knowing**
 - What are some “knowing”-type responsibilities?

GRASP: “Doing”-Type Responsibilities

- Perform a directly useful action
 - Change some data
 - (Assign, calculate, create an object,...)
- Initiate actions in other objects
- Control/coordinate other objects
 - They might change some data

GRASP: “Knowing”-Type Responsibilities

- Understand member data (private encapsulated)
- Understand what you (an object) can do yourself
 - Calculate something
 - Create other objects
- Understand related objects

Object Responsibilities

- You can see where this is going:

Design the objects, including

- Their data
- Their methods
- Their interactions with (messages to) other objects

GRASP: Software Patterns

- **Software Pattern:** a general solution strategy for a general kind of problem
 - Why design every system from scratch?
 - Instead, categorize the problems involved
 - Then, apply the standard solution strategies
- Patterns should have names
 - ...to make them easier to think/talk about
- Patterns should come with
 - Suggestions for how to apply it
 - Explanation of its trade-offs

Pattern Example

- **Pattern Name :** Information Expert
- **Solution:** Assign a responsibility to the class that has the information needed to fulfill it.
- **Problem it Solves:** What is a basic principle by which to assign responsibilities to objects?

A Few GRASP Patterns

- Perhaps the most important are:
 - Information Expert pattern
 - Creator pattern
 - Controller pattern
 - High Cohesion pattern
 - Low Coupling pattern

Guidelines and principles

- We first consider three useful guidelines:

Expert

Creator

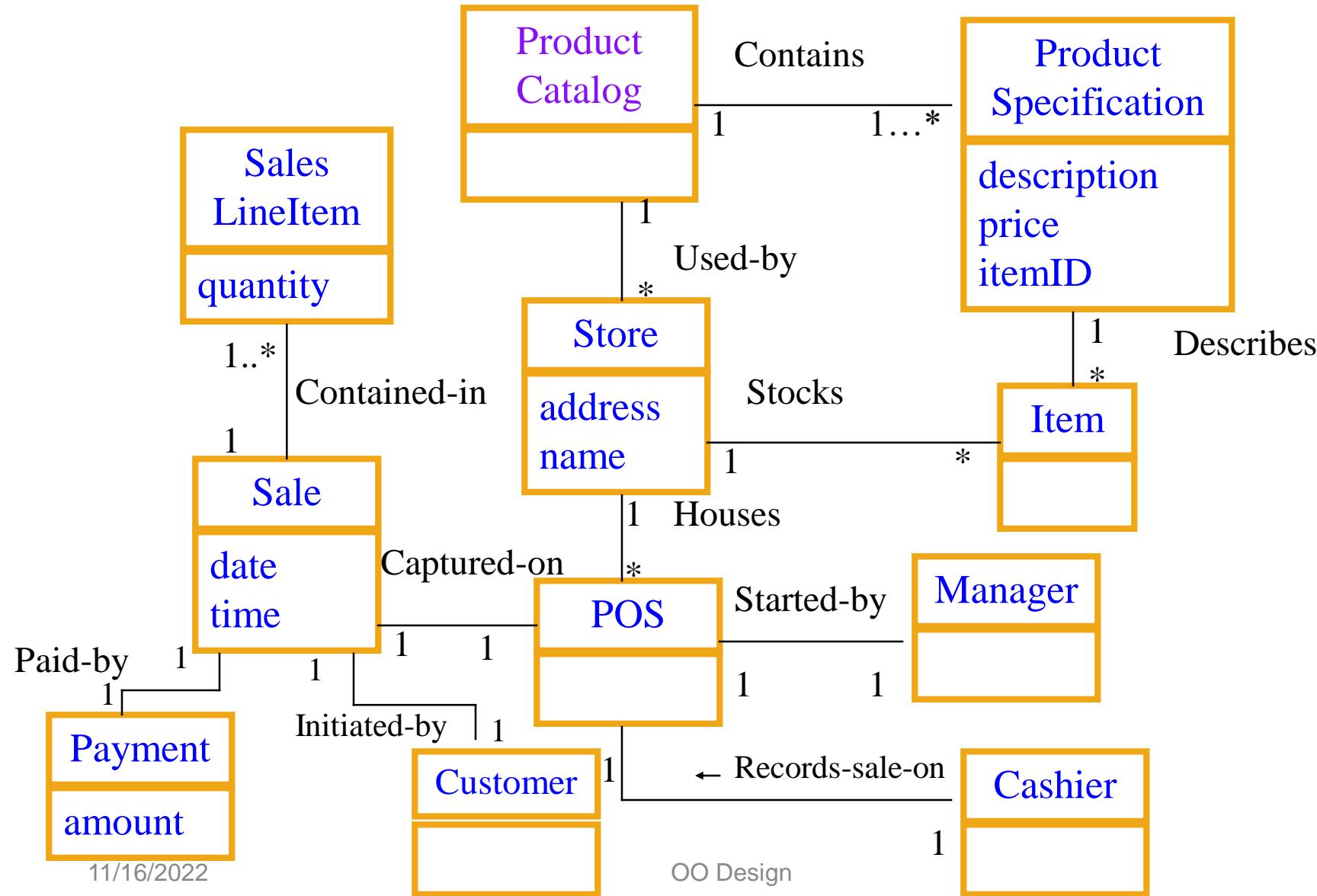
Controller

and two widely applicable principles:

High cohesion

Low coupling

POS: Partial Domain Model



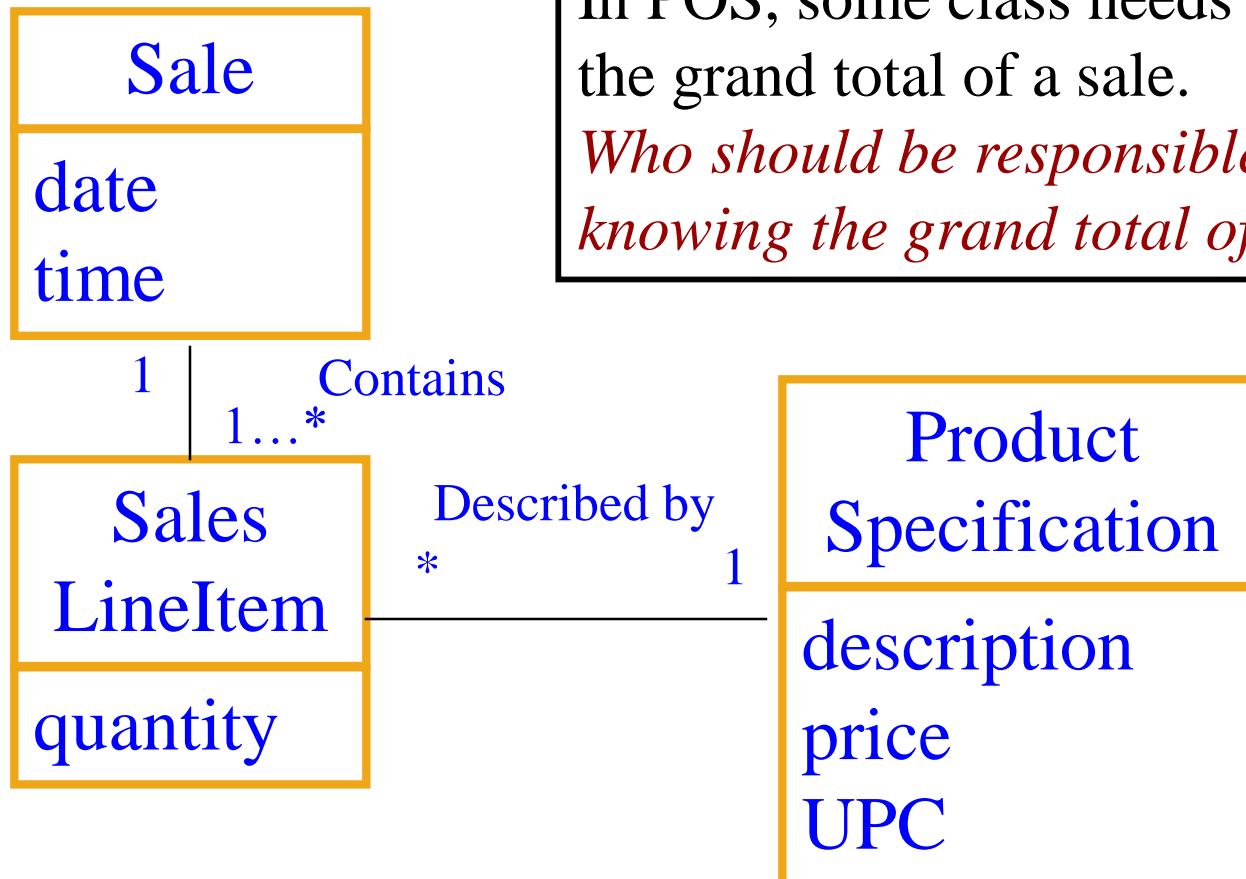
Expert

- Question:
How does one assign responsibilities ?

Answer:

Assign a responsibility to an information expert, i.e. to a class that has the information needed to fulfill that responsibility.

Expert: Example [1]



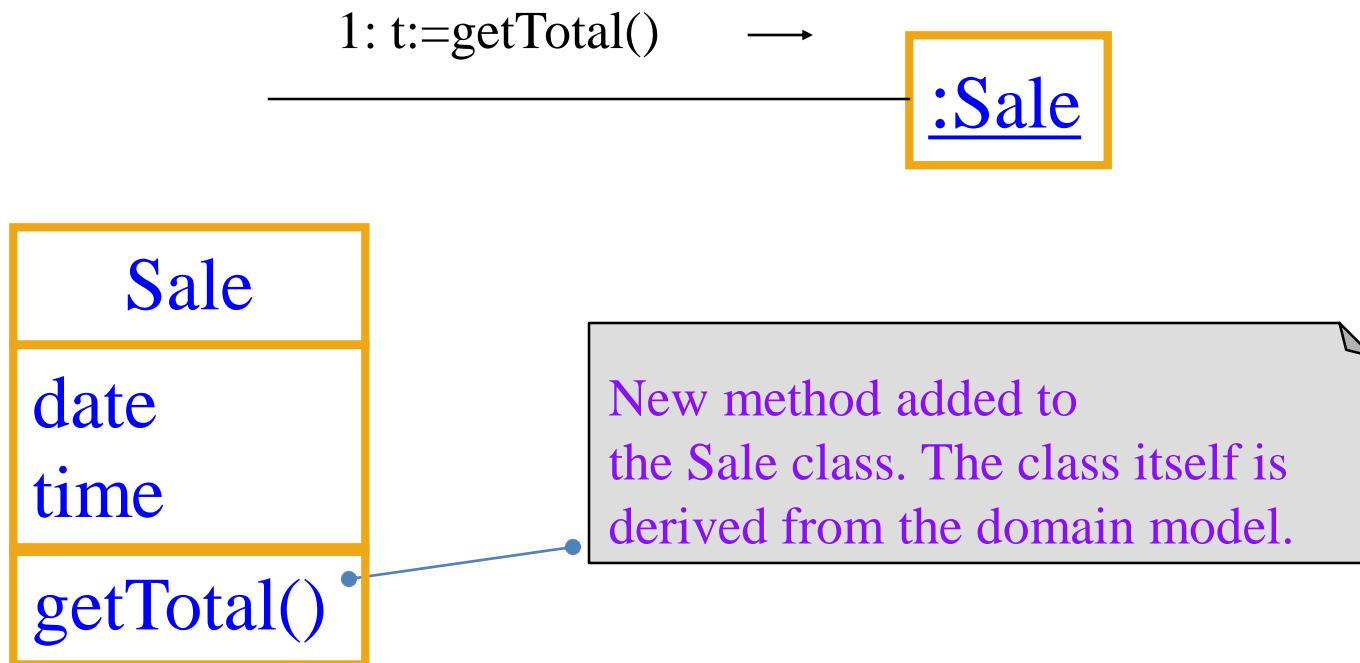
In POS, some class needs to know the grand total of a sale.

Who should be responsible for knowing the grand total of a sale ?

Expert: Example [2]

- From the model, identify the class that contains the information needed to obtain the *grand total*.
- Information needed to obtain the grand total:
 - Knowledge of all *SaleItems*
 - Sum of their subtotals
- Only a *Sale* object possesses this knowledge.
- Sale being the *information expert*, we assign this responsibility to *Sale*.

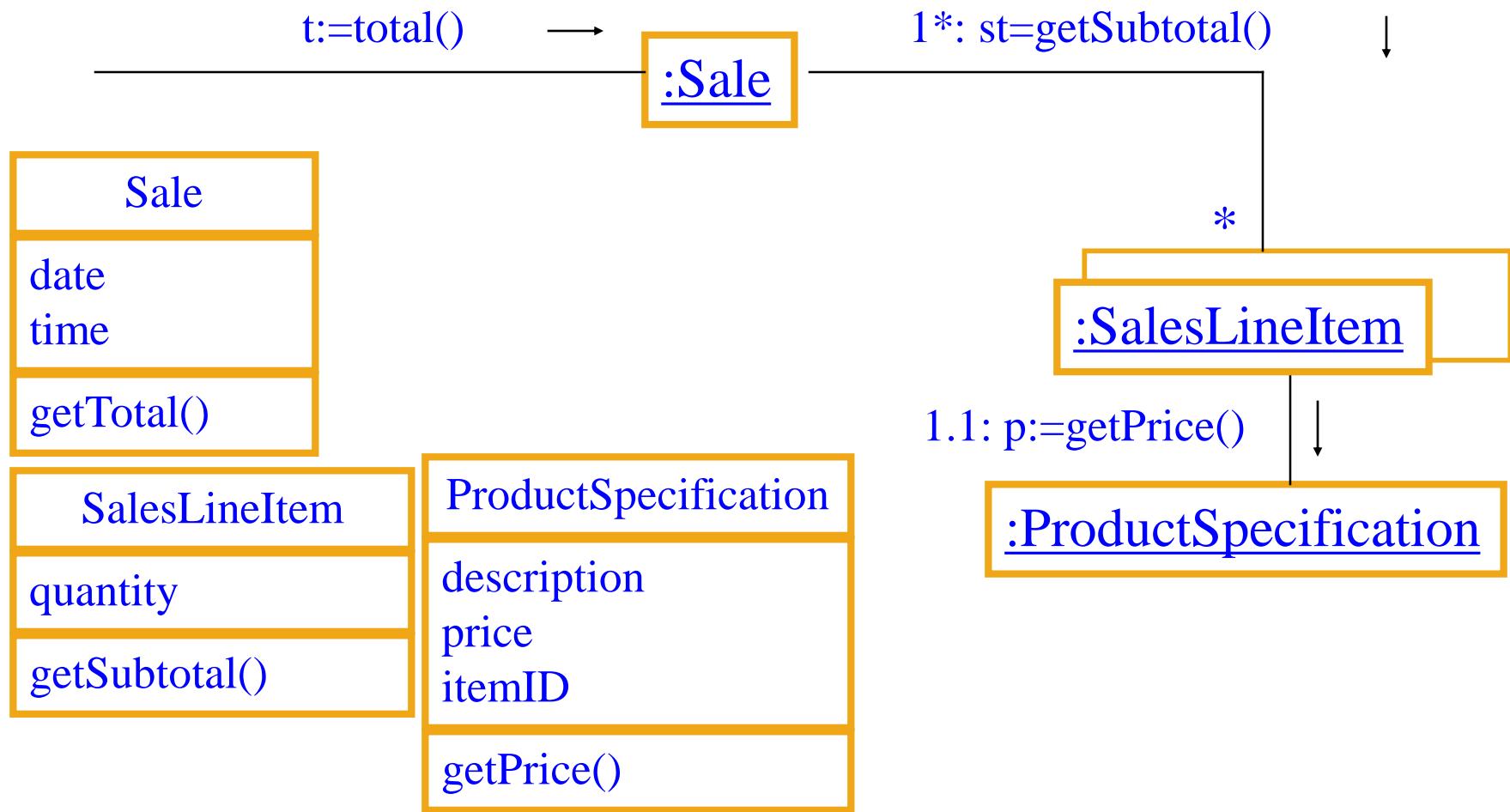
Partial collaboration diagram [1]



Expert: Example [3]

- What information is needed to determine *subtotal* ?
- We need:
 - Quantity of each *SalesLineItem* and its price.
 - Quantity is available with *SalesLineItem* and price with *ProductSpecification*.
- Hence *SalesLineItem* is assigned the responsibility to compute the *subtotal*.

Partial collaboration diagram [2]



Summary: Example [4]

Class	Responsibility
Sale	Knows sale total
SalesLineItem	Knows line item subtotal.
ProductSpecification	Knows the price of a product

Expert: Discussion

- Expert guideline used often.

Fulfillment of a responsibility often requires interaction amongst several objects (3 in our example). There are many *semi-experts* who collaborate in performing a task.

Use of the Expert guideline allows us to retain encapsulation of information.

It often leads to “lightweight” classes collaborating to fulfill a responsibility.

Expert: Disadvantages

- On some occasions the Expert guideline might not suggest a desirable solution.
- *Example: Who should save **Sale** in a database ?*
- As all the information to be saved is in the **Sale** class, it should be responsible for saving the information.
- This implies that the **Sale** class must know about handling databases. Adding database related methods to sale class will make it in-cohesive.
- It also violates the “separation of concerns” principle.

Expert: Benefits

- Objects use their own information to fulfill tasks, hence encapsulation is maintained.
- This leads to low coupling.
- Behavior is distributed across cohesive and lightweight classes.

Expert - Conclusions

- In the real world, items don't tell you their price; line items don't tell you their total
 - But in O-O world, they do!
 - This principle is called “Animation” or the “Do it Myself” principle
- Also works in the workplace (real world)
 - Who puts together the profit/loss statement?
 - Ans: the person in accounting with all the data

Creator [1]

- **Question:**

Who should be responsible for creating an instance of a class ?

Answer:

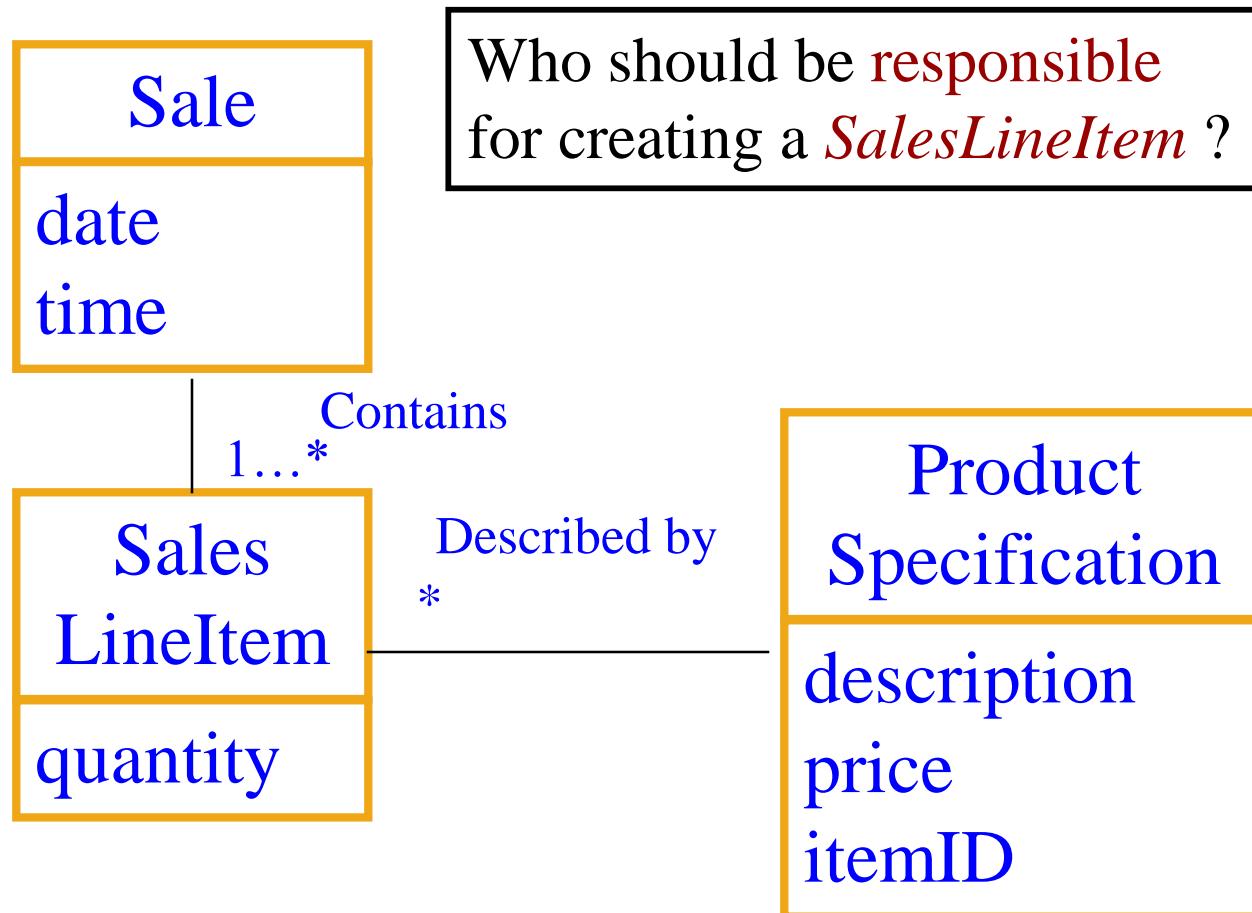
Assign to **B** the responsibility to create an object of class **A** if the following is true:

Creator [2]

- B aggregates objects of type A.
- B contains objects of type A.
- B records instances of objects of type A.
- B closely uses objects of type A.
- B has the data passed to A when A is created.

Implication: B is an expert in the creation of A.

Creator: Example [1]



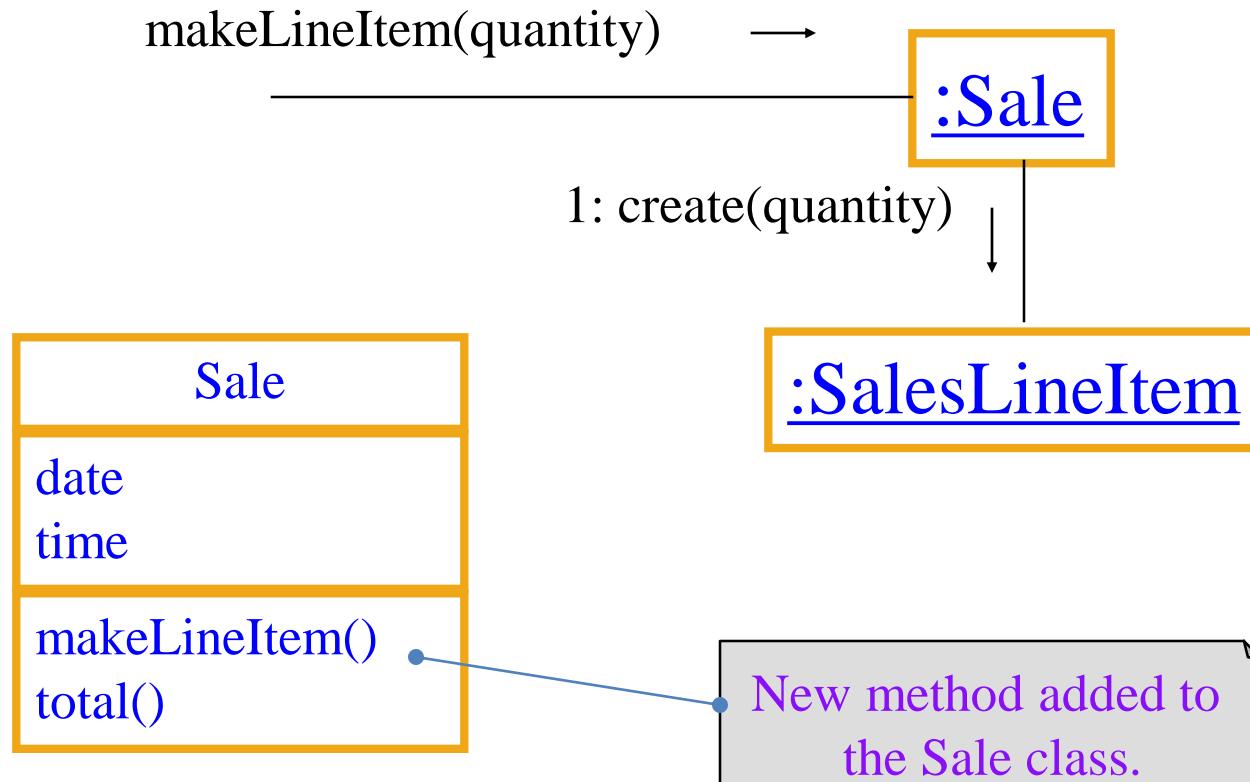
Creator: Example [2]

- *Sale* contains many *SalesLineItem* objects.

This implies that *Sale* is a good candidate for creating a *SalesLineItem*.

This leads to the following collaboration diagram.

Partial collaboration diagram



Creator: Discussion

- **Creator** guides the assignment of responsibility of creating objects. Assigned well, the design can support low coupling increased clarity, encapsulation, and reusability.
- **Creator** suggests that the enclosing container or aggregate class is a good candidate for the creation of the object contained.

Controller [1]

- Question:
 - Who should be responsible for handling system events ?

Recall that a system event is a high level event, an external event, generated by a system actor.

Answer:

Assign a responsibility to handle a system event to a *controller*.

Controller [2]

- A controller is a *non-user interface* object that handles system events. Here is a list of controllers:

Façade controller: Represents the overall system, device, or business.

Use case controller: Represents an artificial handler of all events of a use case.

Role Controller: Represents an animated thing in the domain that would perform the work.

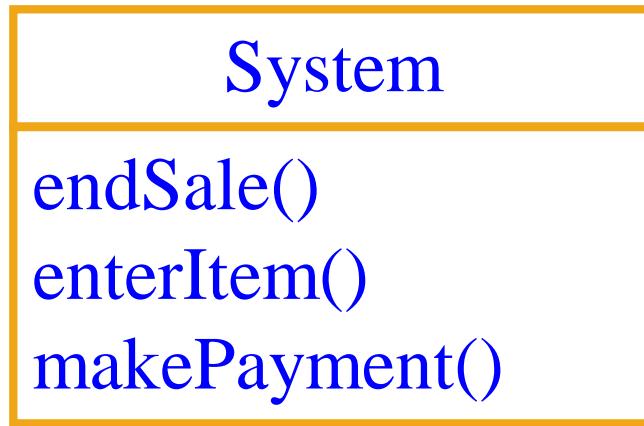
Controller [3]

- Window, applet, view, document do not qualify as controllers. They typically receive system events and delegate them to a controller.

System event examples:

- *Pressing the “end of sale” button.*
- *Request Spell Check.*
- *Request Engine Start.*

System operations



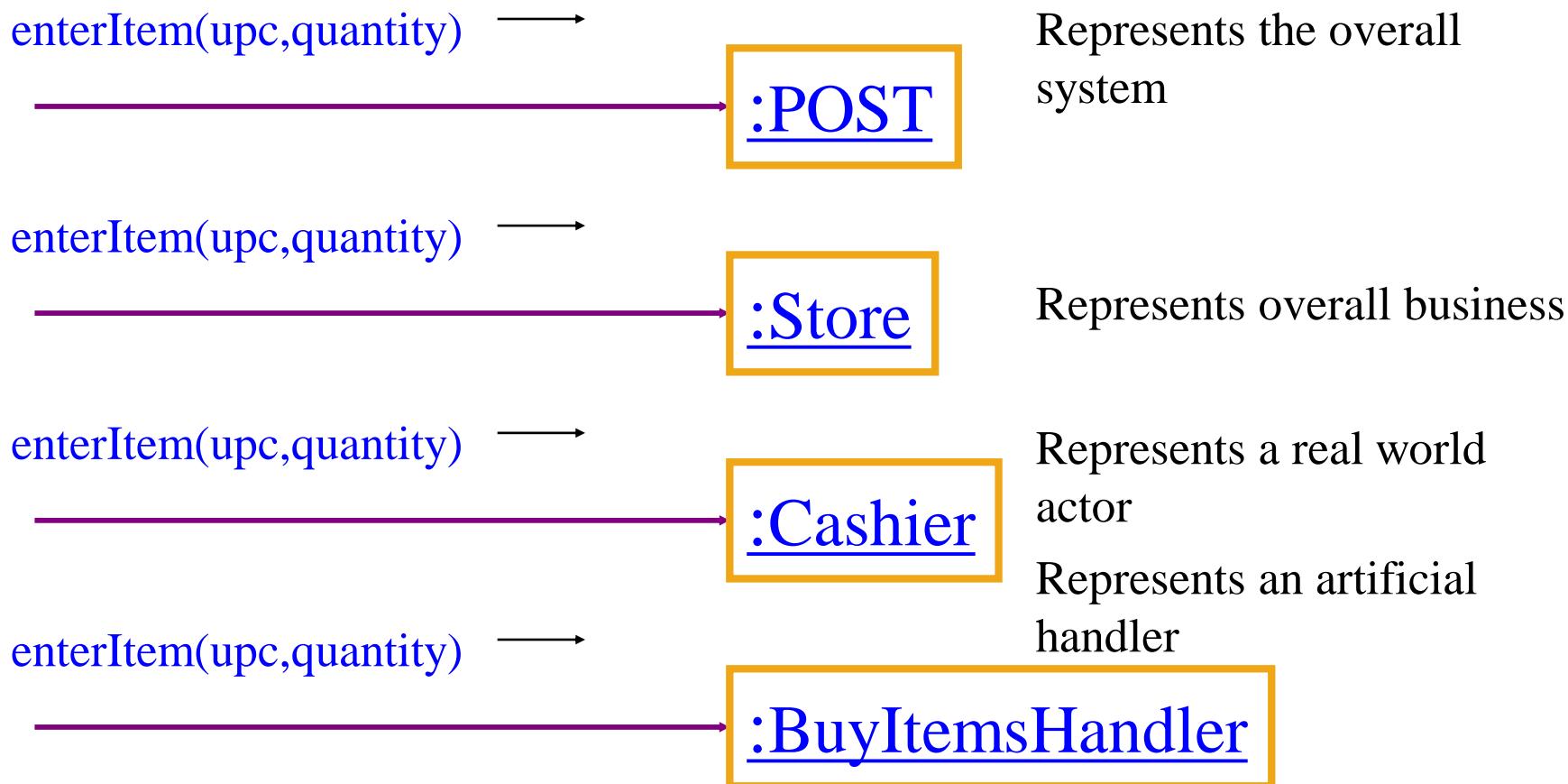
During system behavior analysis, system operations are assigned to the class *System*. It does not imply that the class named *System* performs these during design. Instead, during design, a controller class is assigned to perform these operations.

Controller: Example (1)

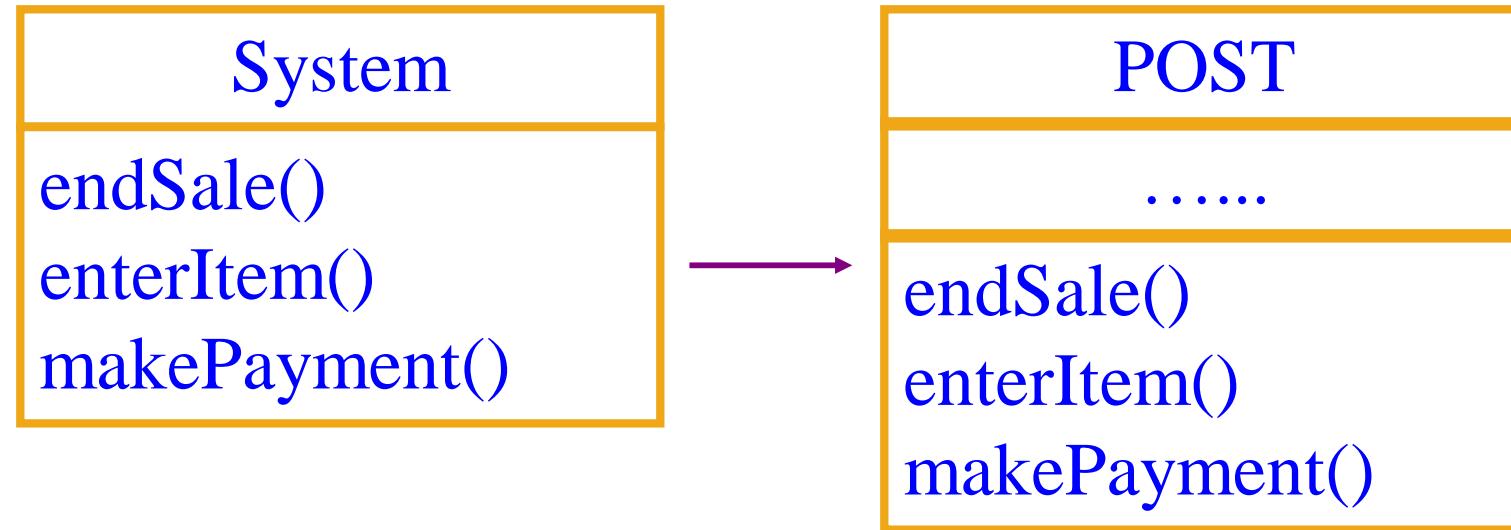
Which object should be responsible for handling the enterItem() system event message ?



Controller: Example [1]



Controller: Example [2]



System operations discovered during analysis

Allocation of system operations during design

During design the system operations, identified during analysis, are assigned to one or more of controller classes.

Controller: Discussion [1]

- Controllers must be chosen to handle *incoming events*.
- Use the *same controller class* for events of *one use case*. This allows maintenance of the state of a use case.
- Do not assign “*too much*” responsibility to a controller. A controller should delegate to other objects work that needs to be done while coordinating an activity.

Controller: Discussion [2]

- **Façade controllers** are suitable when there are only a “few” system events.
- They are also useful when it is not possible to redirect system events to other controllers as, for example, in a message processing system.
- Selecting a ***use case controller*** leads to a different controller for each use case. This controller is not a domain object. For example, for “Buy items” use case one might construct a “BuyItemsHandler” class.

Controller: Discussion [3]

- When an existing controller becomes too large, one may choose a use case controller. This will likely help in maintaining low coupling and high cohesion.

Corollary: External interfacing objects should not have the responsibility for handling system events. Instead, these should be handled in the domain layer objects as opposed to being handled in application layer objects.

Bloated controller

- Signs of a bloated controller:

There is only one controller receiving *all* system events.

The controller performs all tasks itself without delegating any task to other objects.

A controller has many attributes and maintains significant information about the domain.

A controller duplicates information found in other objects.

Avoiding bloated controller

- Add more controllers. If necessary, use role-controllers and use-case controllers.
- An airline reservation system may contain the following controllers:

Role controllers

ReservationAgent

Scheduler

FareAnalyst

Use case controllers

MakeAReservationHandler

ManageSchedulesHandler

ManageFaresHandler

Presentation (Interface) Layer [1]

- Avoid using presentation layer to handle system events.

Use of domain layer objects to handle system events is preferred.

Example:

Assume that POS application has a window that displays sale information and captures cashier's operations. Suppose that a Java applet displays the window. Let us see how the presentation and domain layers can be coupled.

Sample GUI for Point of Sale Terminal

Object Store - X

UPC	1	Quantity	5
Price	2	Description	6
Total	3	Balance	7
Cash	4		

8 Enter Item 9 End Sale 10 Make Payment

Sample course of events

- | Actor | System |
|---|--|
| 1 Customer arrives at the POST checkout with items to purchase. | |
| 2 For each item, the Cashier enters the UPC in 1 of Window 1. The quantity of this item is optionally entered in 5. | 3 Adds information on the item to the current sales transaction. Description and price of the item are displayed in 2 and 6 of Window 1. |

Sample course of events(2)

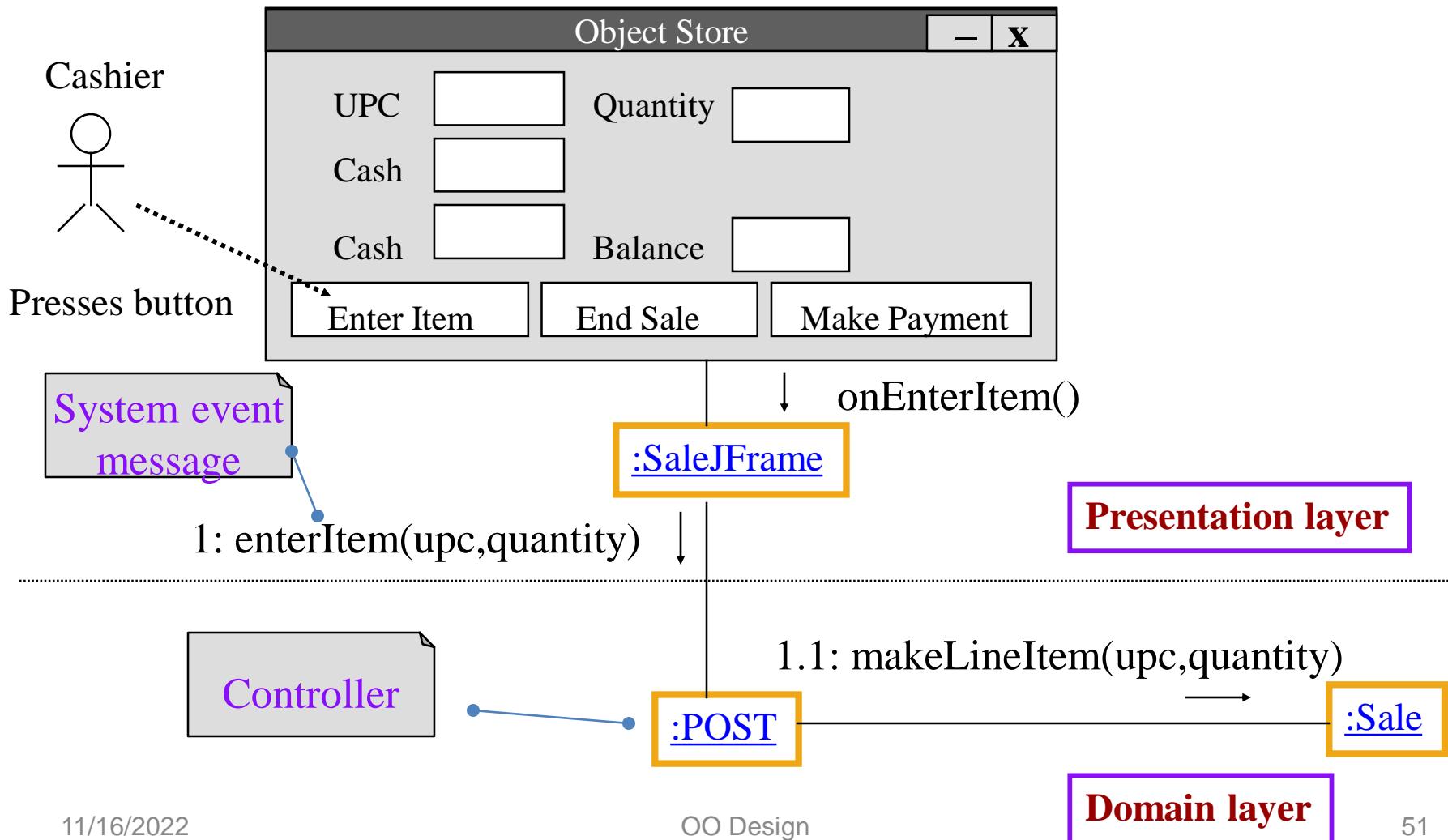
Actor

System

- 4 Completion of item entry is indicated by the Cashier to POST by pressing widget 9.
- 5 Computes and displays the sale total in 3.

6

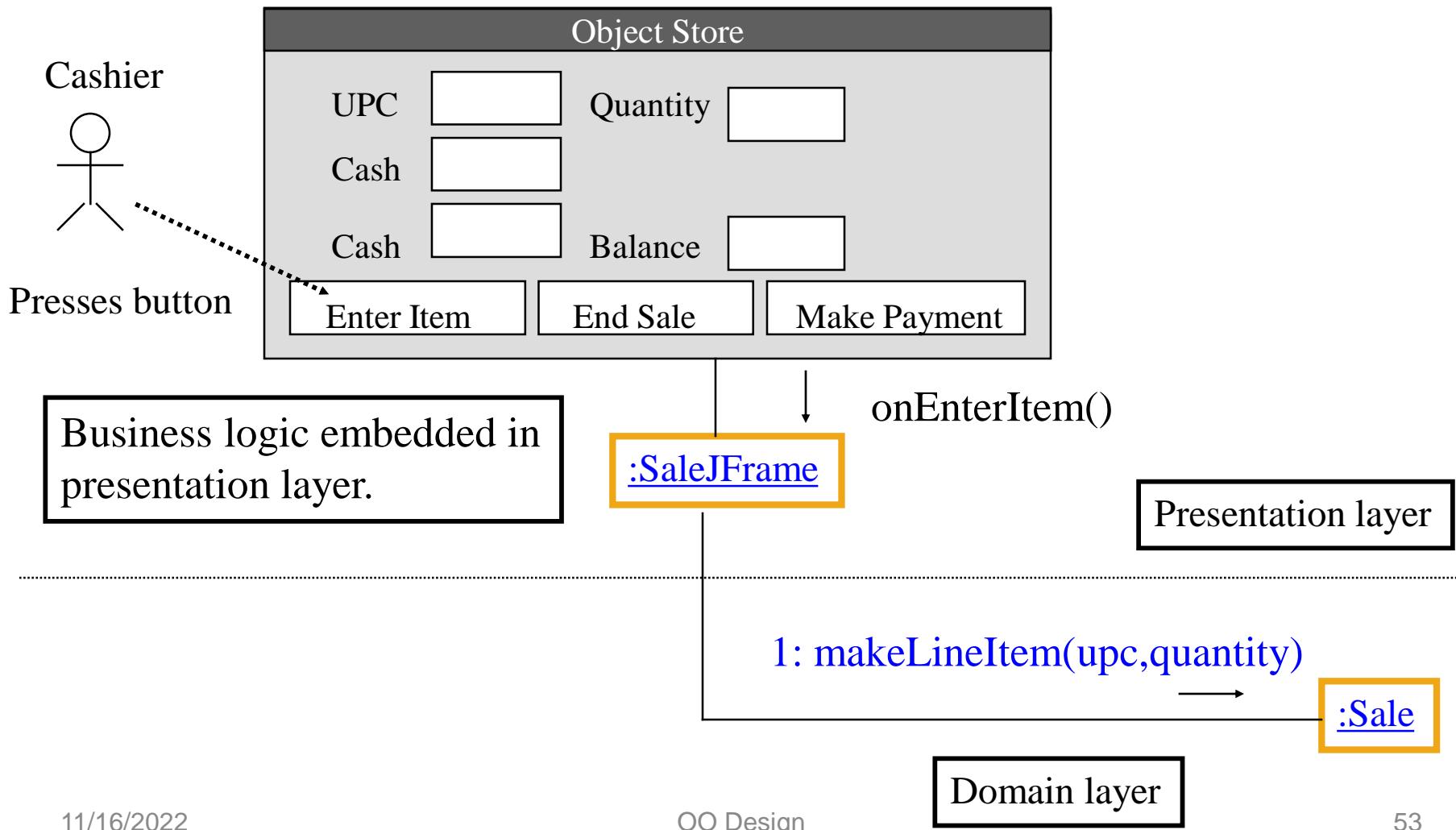
Presentation layer [2]



Presentation layer [3]

- *SaleJFrame* passes on the *enterItem()* message to the domain layer.
- It *does not get involved* in the processing of this message.
- Thus, business processing logic is in a domain object, *not* in the presentation object.
- Presentation layer objects have lower opportunity for re-use due to their coupling with the environment.

Presentation layer: undesirable coupling [4]



Presentation layer [5]

- Some applications do not have a user interface. They receive messages from an external system. For example, LDAP (Light Weight Directory Protocol) might receive a message from a CGI script.

In these cases the messages is encoded in a “stream” or as a CORBA message.

Presentation layer [6]

- In applications with windows, the window might choose who the controller will be.

Different windows may collaborate with different controllers.

However, in a message handling application, the *Command* design pattern is useful.

Low coupling

- How to achieve low *dependency* and increased reuse?

A class with high coupling is undesirable because:

- Changes in related classes force local changes.
- This class is harder to understand in isolation.
- This class is harder to reuse because its use requires the inclusion of all classes it is dependent upon.

The GRASP Pattern : Low Coupling

- Assign a responsibility so that the coupling remains low
- Recall coupling is how strongly one class has knowledge of another class
- The best example of this is not to have POST talking to every class
- POST should not have knowledge of any more classes then its has to

Low coupling: Example (1)

Consider the following partial conceptual model:

Payment

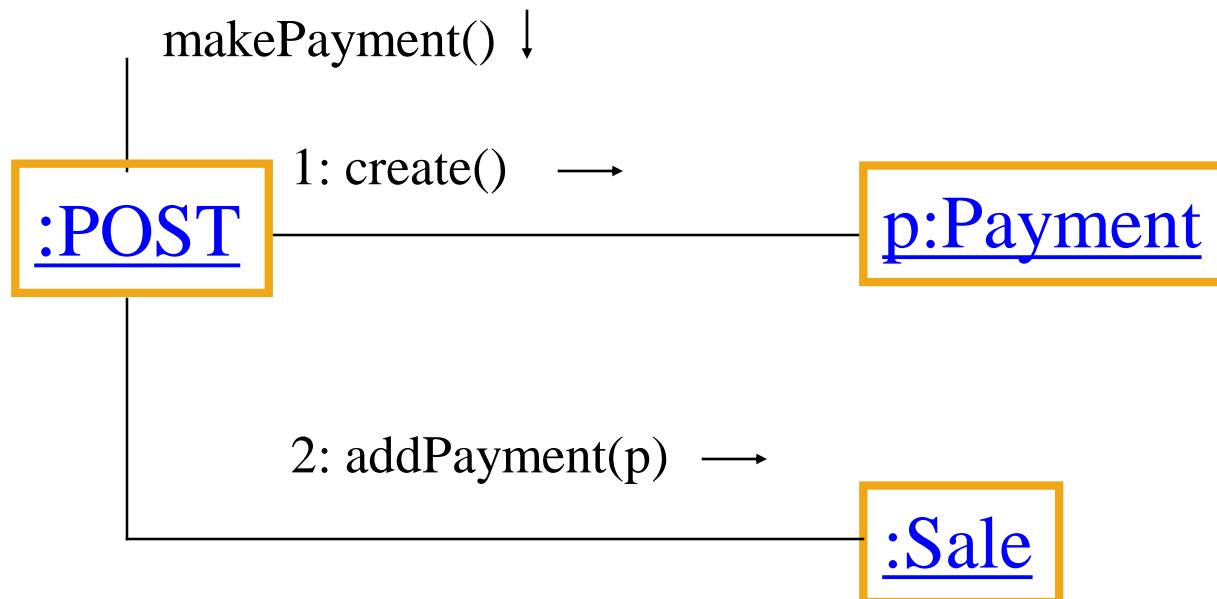
POST

Sale

What class should be responsible for creating an instance of *Payment* ?

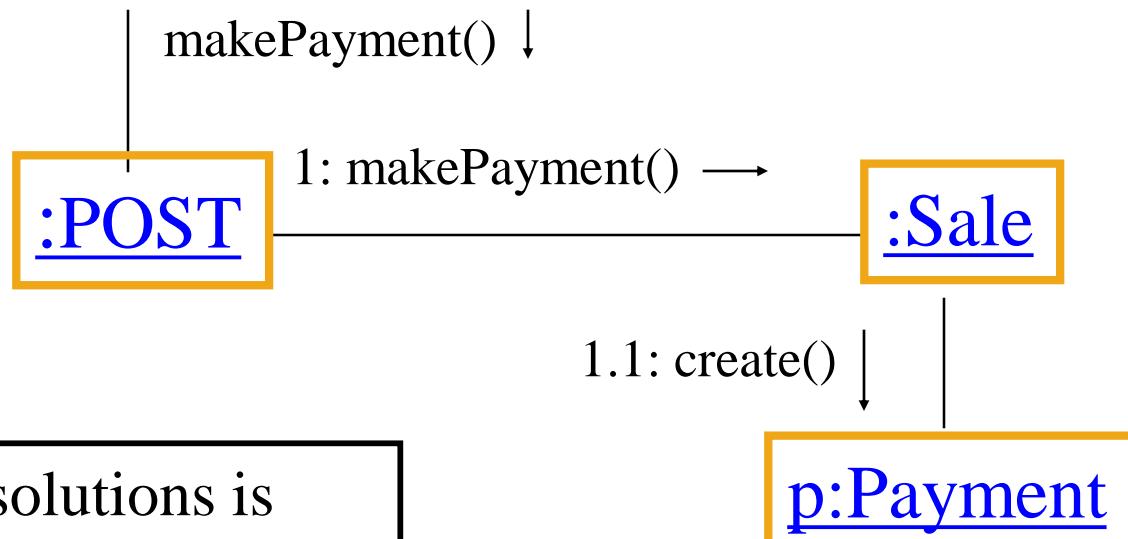
Low coupling: Example [1]

One solution is to let *POST* create an instance of *Payment* and pass a reference this *Payment* to *Sale*. This solution is suggested as *POST* records a *Payment*.



Low coupling: Example [3]

Another solution is to let *Sale* create an instance of *Payment*.



Which of the two solutions is preferable from the point of view of coupling?

Low coupling: Discussion [1]

- Encourages assigning a responsibility to keep coupling low.
- Supports design of relatively independent, hence more reusable, classes.
- Reusable classes have a positive effect on productivity, i.e. may lead to higher productivity.
- However, the quest for low coupling to achieve reusability in a future (mythical!) project may lead to increased project cost.

Low coupling: Discussion [2]

- It is important for the designer to assess the current level of coupling and attempt to reduce it if it is likely to cause problems.

It is important to note that a moderate degree of coupling between classes is normal. After all an OO application is a collection of *communicating objects*!

High cohesion

- Question:
 - How to keep design complexity manageable.

Answer:

Assign responsibilities while maintaining high cohesion.

The GRASP Pattern : High Cohesion

- Assign a responsibility so that all the responsibilities in a class are related
- Recall cohesion is how strongly related the responsibilities are
- The best example of this is not choosing POST to create Payment
- Sale has all the similar responsibilities that create its attributes

High cohesion: Example [1]

- Creation of a *Payment* can be assigned to *POST* as it records a payment in real world. This is suggested by the guidelines associated with *Creator*.

This is acceptable. However, if this logic is applied in several other similar situations, more and more work is likely to be assigned to one class.

This might lead to an in-cohesive class.

High cohesion: Example [2]

- For example, if there are 50 system operations, and *POST* does some work related to each, the *POST* will be a large in-cohesive class.
- In contrast, a design that lets *Sale* create *Payment*, supports higher cohesion in *POST*.
- This design supports both high cohesion and low coupling and hence is preferred over the first one.

Four More GRASP Patterns

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Polymorphism

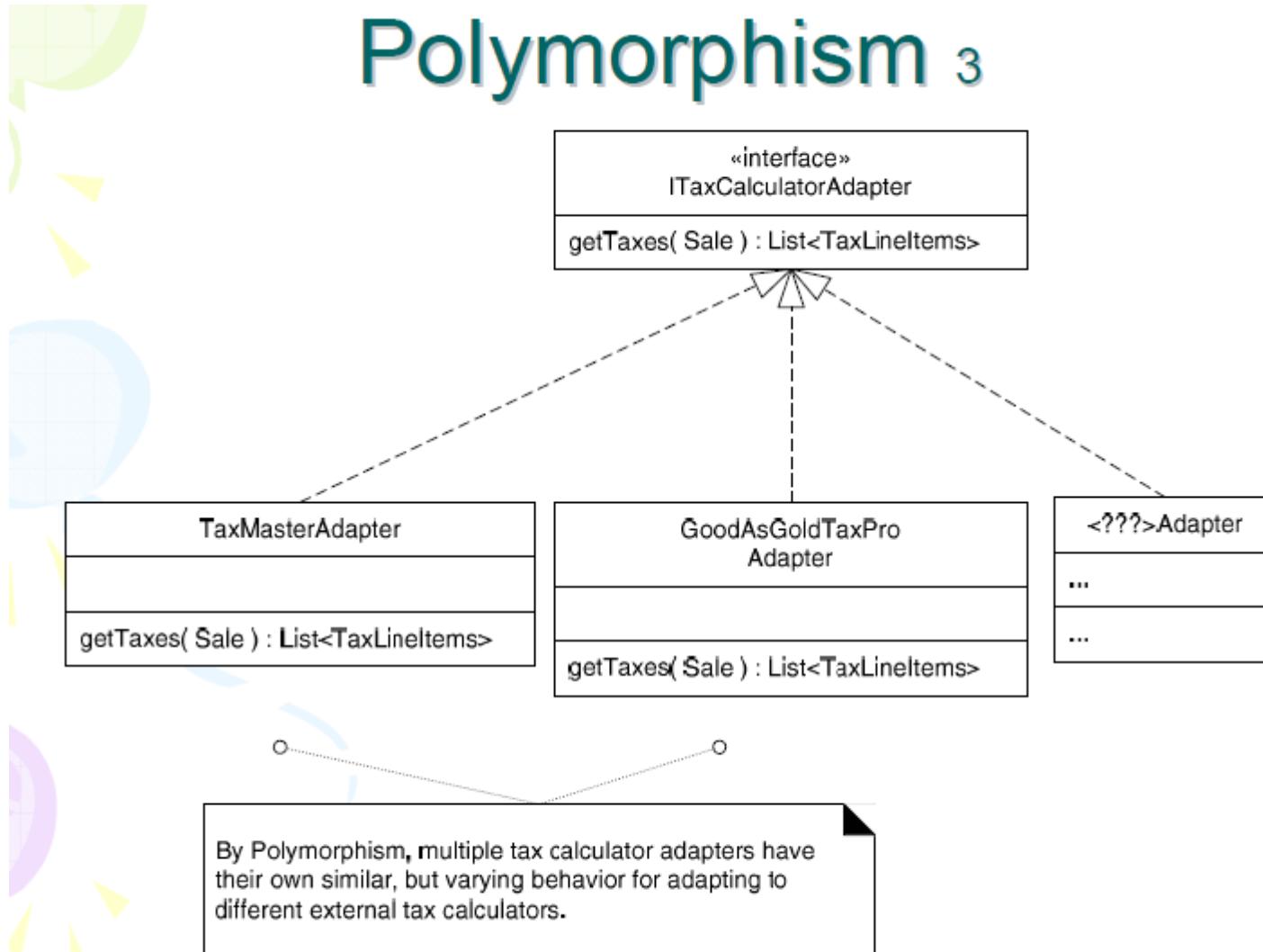
- Problem:
 - How to handle alternatives based on type. Pluggable software components -- how can you replace one server component with another without affecting the client?
- Solution:
 - When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies. In this context, polymorphism means giving the same name to similar or related services

Polymorphism

- One way to handle type-based alternatives is with conditionals: if...else or switch...case statements
- For example, the sqrt function has polymorphic variants for float and double
- (how does it really work?)
- (other examples?)

Polymorphism – Pluggable Components

- Tax calculator uses a standard interface, the TaxCalculatorAdapter, to call any of the actual calculators.



Pure Fabrication

- What object should have responsibility when you don't want to violate High Cohesion and Low Coupling or other goals, but solutions offered by Expert (for example) aren't appropriate?
- Having classes that represent only domain-layer concepts leads to problems.

Pure Fabrication

- Assign a highly cohesive set of responsibilities to a convenience class that does not represent a domain object, but which supports high cohesion, low coupling, and reuse.
- Called “fabrication” because it is “made up,” not immediately obvious

Pure Fabrication

- Database operations are often put in a convenience class. Saving a *Sale* object might, by Expert, belong in the *Sale* class
- Using a “fabricated” class increases the cohesion in *Sale* and reduces the coupling
- The idea of “persistent storage” is not a domain concept

Object Design

- By representational decomposition
- By behavioral decomposition
- Most objects represent things in the problem domain, and so are derived by the former
- Sometimes it is useful to group methods by a behavior or algorithm, even if the resulting class doesn't have a real-world representation

Object Design

- *TableOfContents* would represent an actual table of contents.
- *TableOfContentsGenerator* is a pure fabrication class that creates tables of contents.

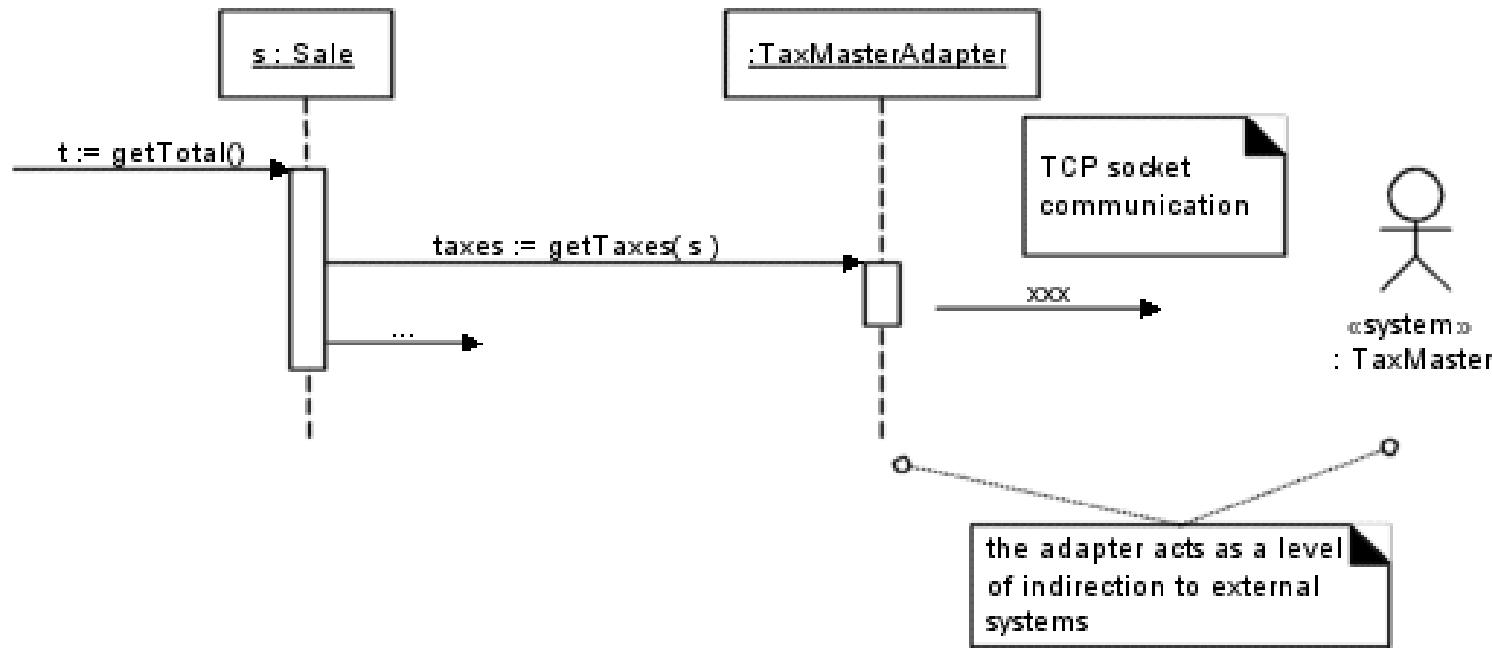
Contraindications

- This can be overused. Information Expert is often a better choice, since it has the information. Use with caution.

Indirection Pattern

- Problem is how to de-couple objects so that low coupling is supported and the chance of reuse is increased? A related issue is to avoid writing special-purpose code too high up in your application.
- Solution is to create an intermediate object that “talks” to both sides.

- TaxCalculatorAdapter – acts as intermediaries to external tax calculators



Indirection

- Example is the TaxCalculatorAdapter. These provide a consistent interface to disparate inner objects and hide the variations
- “Most problems in computer science can be solved by adding another layer of indirection.”
- “Many performance problems can be solved by removing another layer of indirection.”

Protected Variations

- Problem : How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements.
- Solution: identify points of predicted variation or instability; assign responsibilities to create a stable interface around them

Protected Variations

- Example –NextGen POS: The prior **external tax calculator problem** and its solution with Polymorphism illustrate Protected Variations
 - The **point of instability** or variation is the **different interfaces** or APIs of **external tax calculators**
 - By adding a level of indirection, an interface, and using polymorphism with various ITaxCalculatorAdapter implementations, **protection** within the system **from variations in external APIs is achieved.**
 - **Internal objects collaborate with a stable interface**; the various adapter implementations hide the variations to the external systems.

Protected Variations

- Many other patterns and concepts in software design derive from this, including data encapsulation, polymorphism, data-driven designs, interfaces, virtual machines, etc.

Data-Driven Designs

- These include techniques such as reading codes, values, class file paths, class names and so on from an external source to **“parameterize” a system at run time**
- Also includes style sheets, metadata, reading window layouts, etc.

Protected Variations example

- The tax calculator problem illustrates this. The point of instability is the different interfaces of different calculators
- This pattern protects against variations in external APIs

Service Lookup

- Includes techniques such as using naming services (like JNDI (Java Naming and Directory Interface) in Java)
- Protects clients from variations in the location of services
- Special case of data-driven design

Interpreter-Driven Designs

- Include rule interpreters that execute rules read from an external source, script or language interpreters that read and run programs, virtual machines, constraint logic engines, etc.
- Allows changing the behavior of a system via external logic
- SQL stored functions; Excel formulas

Reflective or Meta-Level Designs

- Getting metadata from an external source.
- Special case of data-driven design

Uniform Access

- Syntactic construct so that both a method and field access are expressed the same way
- For example `aCircle.radius` may invoke a `radius()` method or simply refer to the `radius` field.

Standard Languages

- Stick with standards

Liskov Substitution Principle

- “What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T

Liskov Substitution Principle

- Translating: software that refers to a type T should work properly with any subclass of T

Don't Talk to Strangers

- Original version of Protected Variations. A method should only send messages to:
 - The *this* object (self)
 - A parameter of the method
 - An attribute of *this*
 - An element of a collection which is an attribute of *this*
 - An object created within the method.

Possible Problems with PV

- Overgeneralization: trying to protect against future variations by writing code that can be extended, when these variations will never happen

Information Hiding

- Private variables
- Hide information about the design from other modules, at the points of difficulty or likely change. (David Parnas)

Open-Closed Principle

- Modules should be both open (for extension) and closed (to modification in ways that affect clients.)
- OCP includes all software components, including methods, classes, subsystems, applications, etc.

What's Next

- “After identifying your requirements and creating a domain model, add methods to the appropriate classes and define the messaging between the objects to fulfill the requirements.”

The Critical Tool

- Not UML
- A mind well educated in design principles

Object Design

- What has been done? Prior activities
- How do things relate?
- How much design modeling to do, and how?
- What is the output?

“Process” Inputs

- Use Cases—the most architecturally significant, high business value
- Programming experiments to find showstopper technical problems
- Use case text defines visible behavior that objects must support
- Sequence diagrams
- Operation contracts

Activities

- Start coding, with test-first development
- Start some UML modeling for object design
- Or start with another modeling technique such as CRC cards
- Models are to understand and communicate, not to document

Outputs

- UML interaction, class, and package diagrams
- UI sketches and prototypes
- Database models
- Report sketches and prototypes



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.9

Today's Agenda

- Object Oriented Design
- GRASP Patterns

Four More GRASP Patterns

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Polymorphism

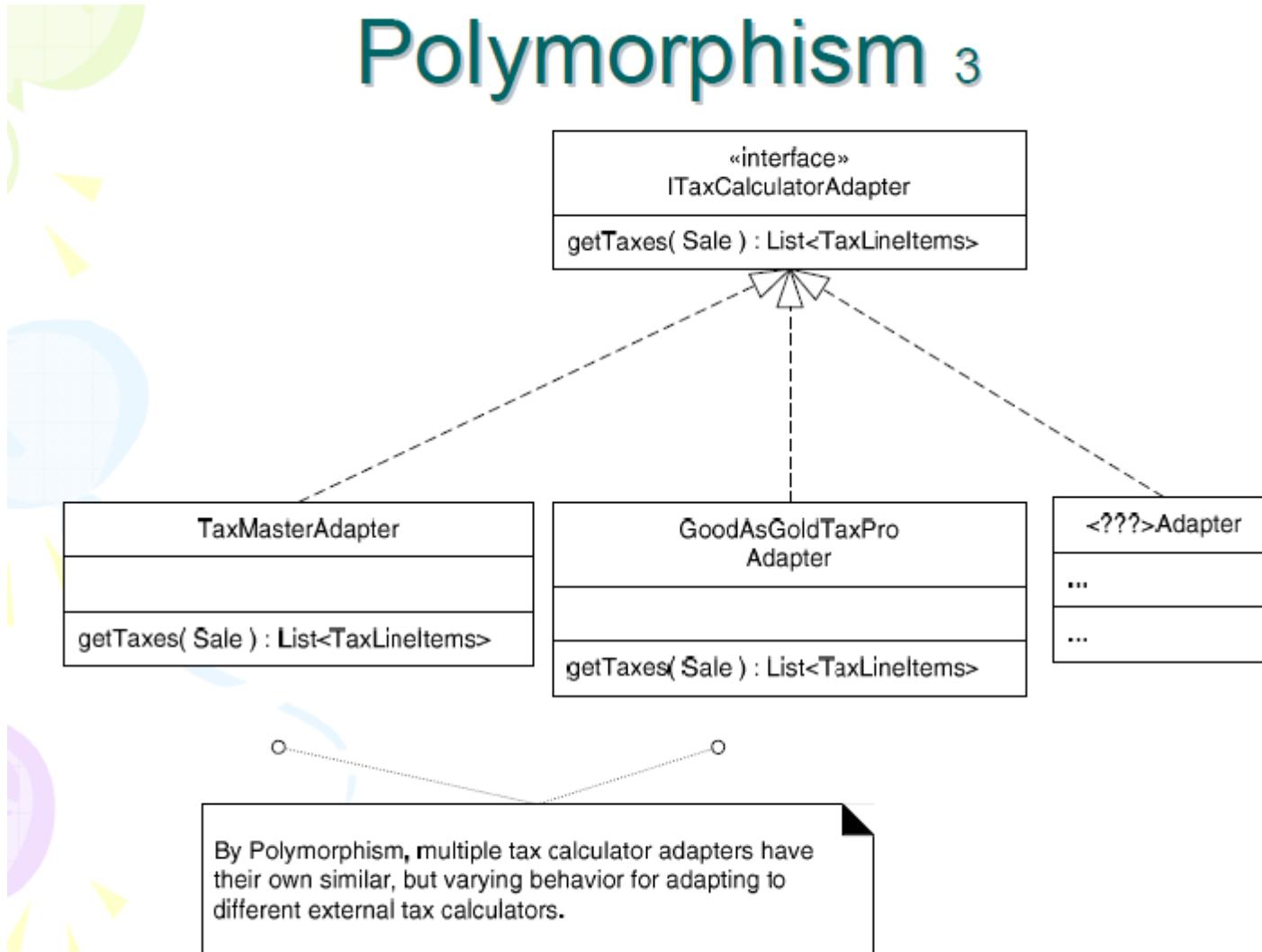
- Problem:
 - How to handle alternatives based on type. Pluggable software components -- how can you replace one server component with another without affecting the client?
- Solution:
 - When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies. In this context, polymorphism means giving the same name to similar or related services

Polymorphism

- One way to handle type-based alternatives is with conditionals: if...else or switch...case statements
- For example, the sqrt function has polymorphic variants for float and double
- (how does it really work?)
- (other examples?)

Polymorphism – Pluggable Components

- Tax calculator uses a standard interface, the TaxCalculatorAdapter, to call any of the actual calculators.



Pure Fabrication

- What object should have responsibility when you don't want to violate High Cohesion and Low Coupling or other goals, but solutions offered by Expert (for example) aren't appropriate?
- Having classes that represent only domain-layer concepts leads to problems.

Pure Fabrication

- Assign a highly cohesive set of responsibilities to a convenience class that does not represent a domain object, but which supports high cohesion, low coupling, and reuse.
- Called “fabrication” because it is “made up,” not immediately obvious

Pure Fabrication

- Database operations are often put in a convenience class. Saving a *Sale* object might, by Expert, belong in the *Sale* class
- Using a “fabricated” class increases the cohesion in *Sale* and reduces the coupling
- The idea of “persistent storage” is not a domain concept

Object Design

- By representational decomposition
- By behavioral decomposition
- Most objects represent things in the problem domain, and so are derived by the former
- Sometimes it is useful to group methods by a behavior or algorithm, even if the resulting class doesn't have a real-world representation

Object Design

- *TableOfContents* would represent an actual table of contents.
- *TableOfContentsGenerator* is a pure fabrication class that creates tables of contents.

Contraindications

- This can be overused. Information Expert is often a better choice, since it has the information. Use with caution.

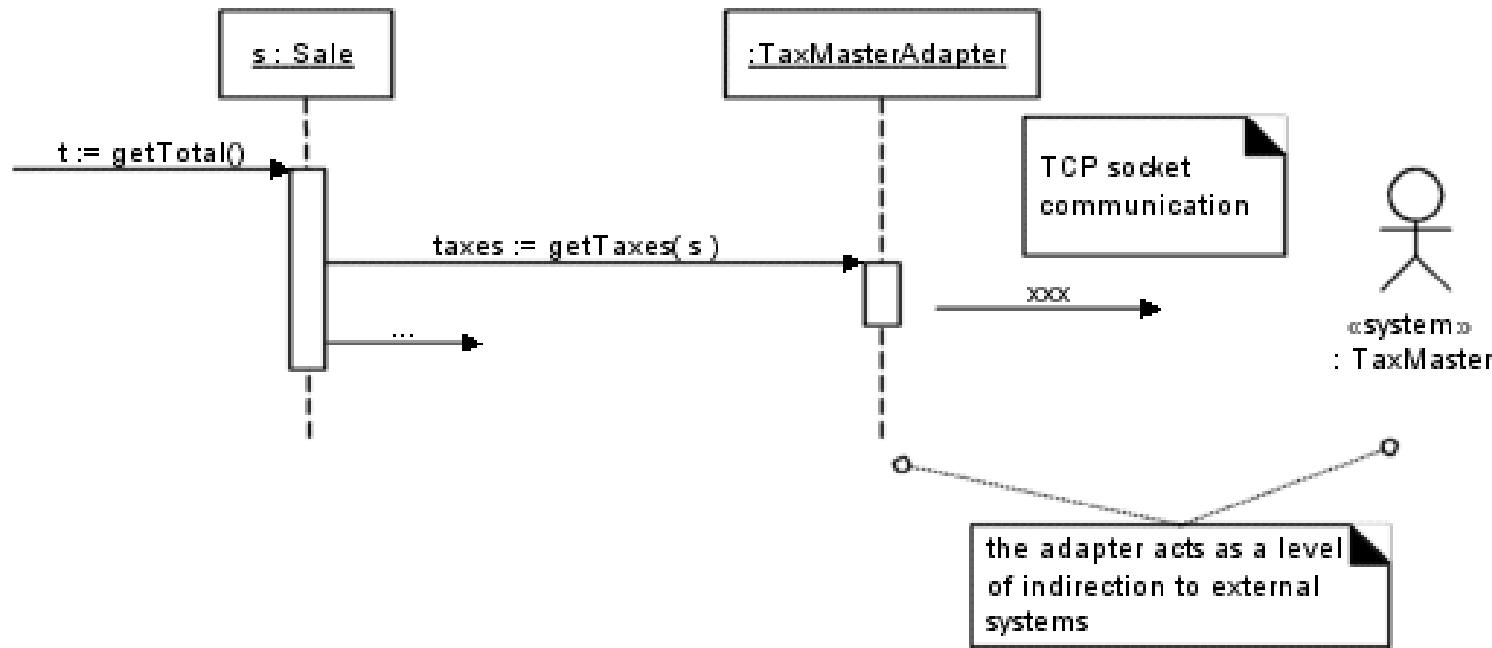
Indirection Pattern

- Problem is how to de-couple objects so that low coupling is supported and the chance of reuse is increased? A related issue is to avoid writing special-purpose code too high up in your application.
- Solution is to create an intermediate object that “talks” to both sides.

Indirection

- Example is the TaxCalculatorAdapter. These provide a consistent interface to disparate inner objects and hide the variations
- “Most problems in computer science can be solved by adding another layer of indirection.”
- “Many performance problems can be solved by removing another layer of indirection.”

- TaxCalculatorAdapter – acts as intermediaries to external tax calculators



Protected Variations

- Problem : How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements.
- Solution: identify points of predicted variation or instability; assign responsibilities to create a stable interface around them

Protected Variations

- Example –NextGen POS: The prior **external tax calculator problem** and its solution with Polymorphism illustrate Protected Variations
 - The **point of instability** or variation is the **different interfaces** or APIs of **external tax calculators**
 - By adding a level of indirection, an interface, and using polymorphism with various ITaxCalculatorAdapter implementations, **protection** within the system **from variations in external APIs is achieved.**
 - **Internal objects collaborate with a stable interface**; the various adapter implementations hide the variations to the external systems.

Protected Variations

- Many other patterns and concepts in software design derive from this, including data encapsulation, polymorphism, data-driven designs, interfaces, virtual machines, etc.

Data-Driven Designs

- These include techniques such as reading codes, values, class file paths, class names and so on from an external source to **“parameterize” a system at run time**
- Also includes style sheets, metadata, reading window layouts, etc.

Protected Variations example

- The tax calculator problem illustrates this. The point of instability is the different interfaces of different calculators
- This pattern protects against variations in external APIs

Service Lookup

- Includes techniques such as using naming services (like JNDI (Java Naming and Directory Interface) in Java)
- Protects clients from variations in the location of services
- Special case of data-driven design

Interpreter-Driven Designs

- Include **rule interpreters that execute rules read from an external source**, script or language interpreters that read and run programs, virtual machines, constraint logic engines, etc.
- Allows changing the behavior of a system via external logic
- SQL stored functions; Excel formulas

Reflective or Meta-Level Designs

- Getting metadata from an external source.
- Special case of data-driven design

Uniform Access

- Syntactic construct so that both a method and field access are expressed the same way
- For example `aCircle.radius` may invoke a `radius()` method or simply refer to the `radius` field.

Standard Languages

- Stick with standards such as SQL provide protection against a proliferation of varying languages.

Liskov Substitution Principle

- “What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T

Liskov Substitution Principle

- Translating: software that refers to a type T should work properly with any subclass of T

Don't Talk to Strangers

- Original version of Protected Variations. A method should only send messages to:
 - The *this* object (self)
 - A parameter of the method
 - An attribute of *this*
 - An element of a collection which is an attribute of *this*
 - An object created within the method.

Possible Problems with PV

- Overgeneralization: trying to protect against future variations by writing code that can be extended, when these variations will never happen

Information Hiding

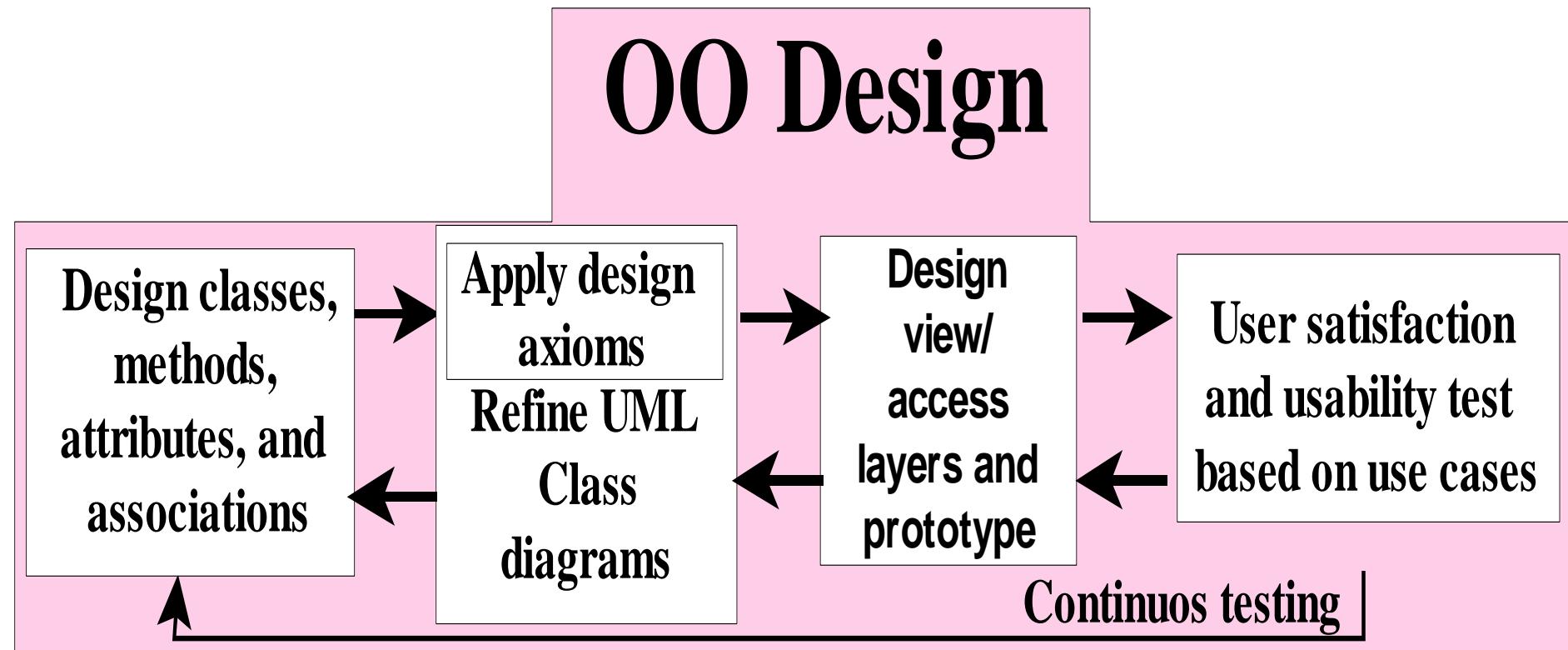
- Private variables
- Hide information about the design from other modules, at the points of difficulty or likely change. (David Parnas)

Open-Closed Principle

- Modules should be both open (for extension) and closed (to modification in ways that affect clients.)
- OCP includes all software components, including methods, classes, subsystems, applications, etc.

OO Design Rules

Object-Oriented Design Process in the Unified Approach



OO Design Axioms

- An axiom = is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.
- A theorem = is a proposition that may not be self-evident but can be proven from accepted axioms. Therefore, is equivalent to a law or principle.
- A theorem is valid if its referent axioms & deductive steps are valid.
- A corollary = is a proposition that follows from an axiom or another proposition that has been proven

Suh's design axioms to OOD :

- Axiom 1 : *The independence axiom.*
Maintain the independence of components
- Axiom 2 : *The information axiom.* Minimize the information content of the design.

Axiom 1 → states that, during the design process, as we go from requirement and use-case to a system component, each component must satisfy that requirement, without affecting other requirements

- Axiom 2 → concerned with simplicity.

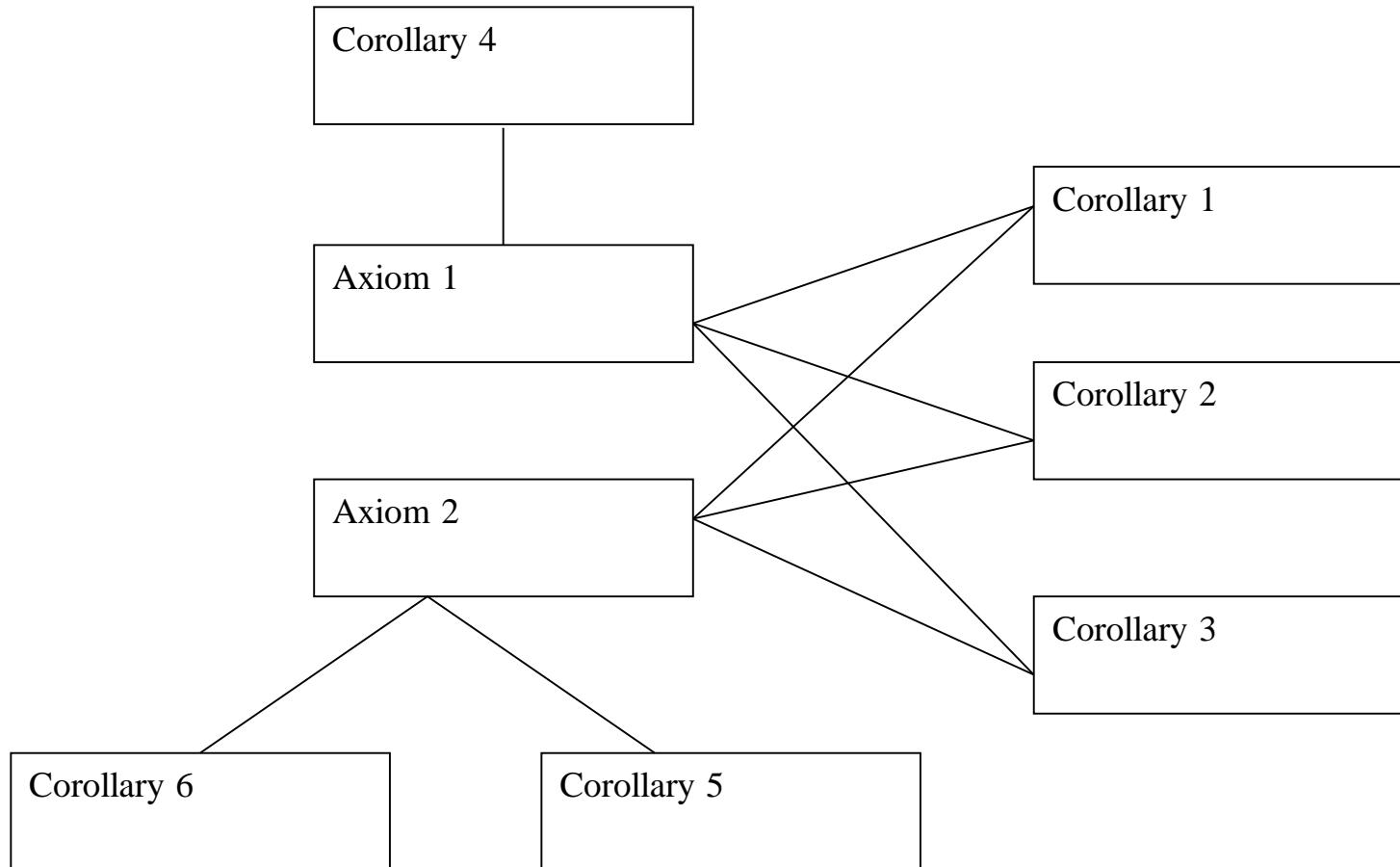
Occam's razor rule of simplicity in OO terms

- *The best designs usually involve the least complex code but not necessarily the fewest number of classes or methods. Minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application. In an object-oriented system, the best way to minimize complexity is to use inheritance and the system's built-in classes and to add as little as possible to what already is there.*

Corollaries

- May be called **Design rules**, and all are derived from the two basic axioms :
- The origin of corollaries as shown in figure 2. Corollaries 1,2 and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 & 6 are from axiom 2.

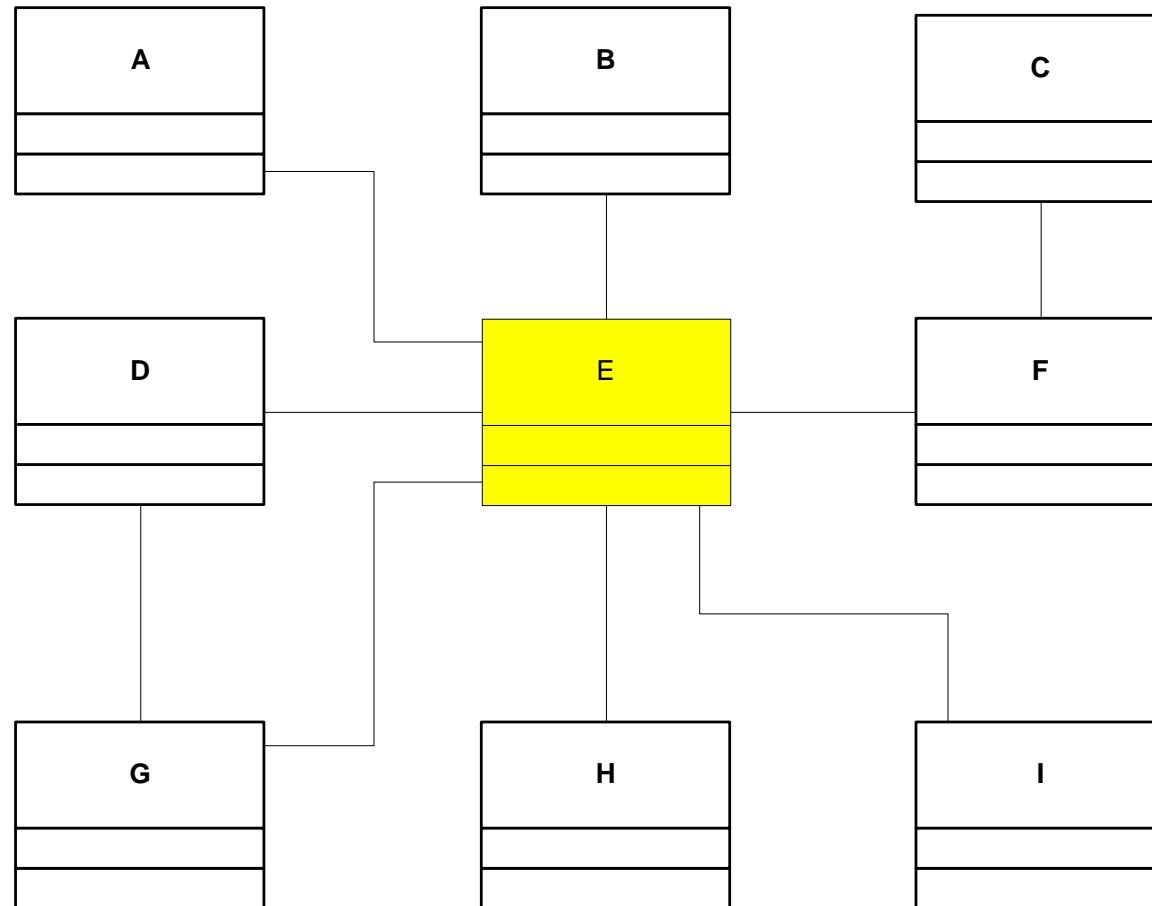
Origin of corollaries



Corollary 1 : Uncoupled design with less information content.

- *Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects*
- *Main goal → maximize objects cohesiveness among objects & sw components → to improve coupling*
- *Strong coupling among objects complicates a system, since the class is harder to understand or highly interrelated with other classes.*
- *Degree or strength of coupling between two components is measured by the amount & complexity of information transmitted between them*

Tightly Coupled Object



- OO design has 2 types of coupling : *Interaction coupling and Inheritance coupling*
- *Interaction coupling* → the amount & complexity of messages between components.
 - Desirable to have a little interaction.
 - Minimize the number of messages sent & received by an object

Types of Coupling

Degree of coupling	Name	Description
Very High	Content Coupling	Connection involves direct reference to attributes or methods of another object
High	Common Coupling	Connection involves two objects accessing a 'global data space', for both to read & write
Medium	Control Coupling	Connection involves explicit control of the processing logic of one object by another
Low	Stamp coupling	Connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure
Very low	Data coupling	Connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. (this is the goal of an architectural design)

- *Inheritance coupling* → coupling between super- and subclasses
 - A subclass is coupled to its superclass in terms of attributes & methods
 - High inheritance coupling is desirable
 - Each specialization class should not inherit lots of unrelated & unneeded methods & attributes

Cohesion

- *Need to consider interaction within a single object or sw component* → Cohesion
 - Cohesion → reflects the ‘single-purpose ness’ of an object (see corollaries 2 & 3)
 - Method cohesion → a method should carry only one function.
 - A method carries multiple functions is undesirable

Corollary 2 : Single purpose

- *Each class must have a purpose & clearly defined*
- *Each method must provide only one service*

Corollary 3 : Large number of simple classes.

- *Keeping the classes simple allows reusability*
- *A class that easily can be understood and reused (or inherited) contributes to the overall system*
- *Complex & poorly designed class usually cannot be reused*
- *Guideline → The smaller are your classes, the better are your chances of reusing them in other projects. Large & complex classes are too specialized to be reused*
- *The emphasis OOD places on encapsulation, modularization, and polymorphism suggests reuse rather than building anew*
- *Primary benefit of sw reusability → Higher productivity*

Corollary 4 : Strong mapping.

- *There must be a strong association between the analysis's object and design's object*
- *OOA and OOD are based on the same model*
- *As the model progresses from analysis to implementation, more detailed is added*

Corollary 5 : Standardization.

- *Promote standardization by designing interchangeable components and reusing existing classes or components*
- *The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications*

Corollary 6 : Design with inheritance.

- *Common behavior (methods) must be moved to super classes.*
- *The super class-subclass structure must make logical sense*

OO Design Axioms

1. *The independence axiom. Maintain the independence of components.*
 2. *The information axiom. Minimize the complexity* (information content) of the design.
- Design rules (axioms):
 - Design highly cohesive objects that require low coupling (1, 2).
 - Each class should have a single purpose (1, 2).
 - A large number of simpler classes enhances reusability (1, 2).
 - Map strongly from objects in the analysis to objects in the design (1).
 - Promote standardization by reusing classes and building to standard interfaces (2).
 - Design for inheritance—move common behavior to superclasses (2).

Restructuring the Design

- Factoring
 - Separate aspects of a method or class into a new method or class
- Normalization
 - Identifies classes missing from the design
- Challenge inheritance relationships to ensure they only support a generalization/specialization semantics

Optimizing the Design

- Review access paths
- Review attributes of each class
- Review direct and indirect fan-out
- Consider execution order of statements in often-used methods
- Avoid recomputation by creating derived attributes and triggers

Five Rules For Identifying Bad Design

1. If it looks messy then it's probably a bad design
2. If it is too complex then it's probably a bad design
3. If it is too big then it's probably a bad design
4. If people don't like it then it's probably a bad design
5. If it doesn't work then it's probably a bad design

Avoiding Design Pitfalls

- Keep a careful eye on the class design and make sure that an object's role remains well defined
- If an object loses focus, you need to modify the design
 - single purpose

Avoiding Design Pitfalls

- Move some functions into new classes that the object would use
 - uncoupled design with less information content
- Break up the class into two or more classes
 - large number of simple classes

Class Diagrams

Class Diagrams

- In the system design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects
- With detailed modeling, the classes of the conceptual design are often split in a number of subclasses
- In order to further describe the behavior of systems, these diagrams can be complemented by state diagram or UML state machine
- Also instead of class diagrams, Object role modeling can be used if you just want to model the classes and their relationships

Class Diagrams

- The main building block in object oriented modeling
- They are used both for general conceptual modeling of the systematics of the application, and for detailed modeling translating the models into programming code
- The classes in a diagram represent both the main objects and/or interactions in the application and the objects to be programmed
- In the diagram these classes are represented with boxes which contain three parts

Class notation

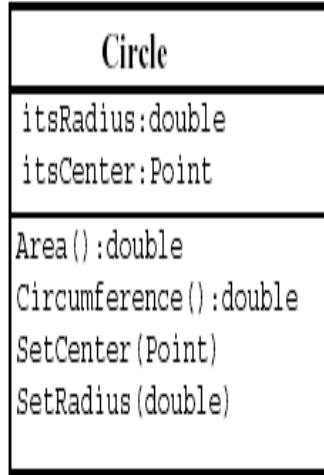
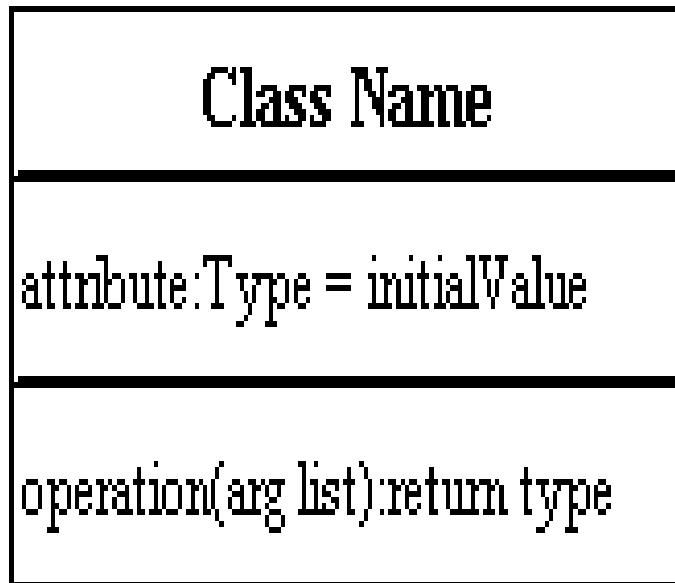


A class diagram exhibits classes and their associations used in an application.

Contrast this with a conceptual model which shows domain concepts and their associations.

Note: you can include visibility attributes: + for public, – for private, # for protected, ~ package (default)

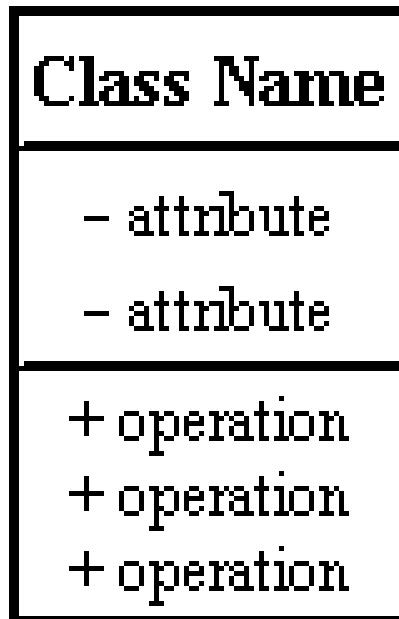
Classes



- Classes are the building blocks in object-oriented programming. Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded, and capitalized), list the attributes in the second partition, and write operations into the third.

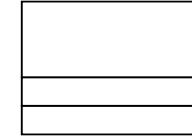
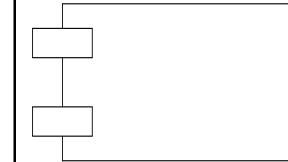
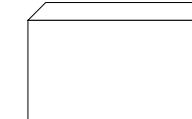
Visibility

- + *public*
- *private*
- # *protected*



- Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.

Structural Modeling: Core Elements

Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
interface	a named set of operations that characterize the behavior of an element.	
component	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
node	a run-time physical object that represents a computational resource.	

Structural Modeling: Core Elements

(cont'd)

Construct	Description	Syntax
constraint ¹	a semantic condition or restriction.	

¹ An extension mechanism useful for specifying structural elements.

Structural Modeling:

Core Relationships

Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances.	_____
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	
generalization	a taxonomic relationship between a more general and a more specific element.	
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	

Structural Modeling:

Core Relationships (cont'd)

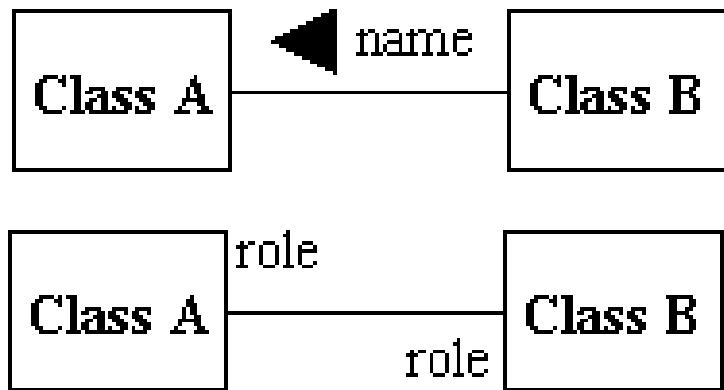
Construct	Description	Syntax
realization	a relationship between a specification and its implementation.	-----→

Relationships

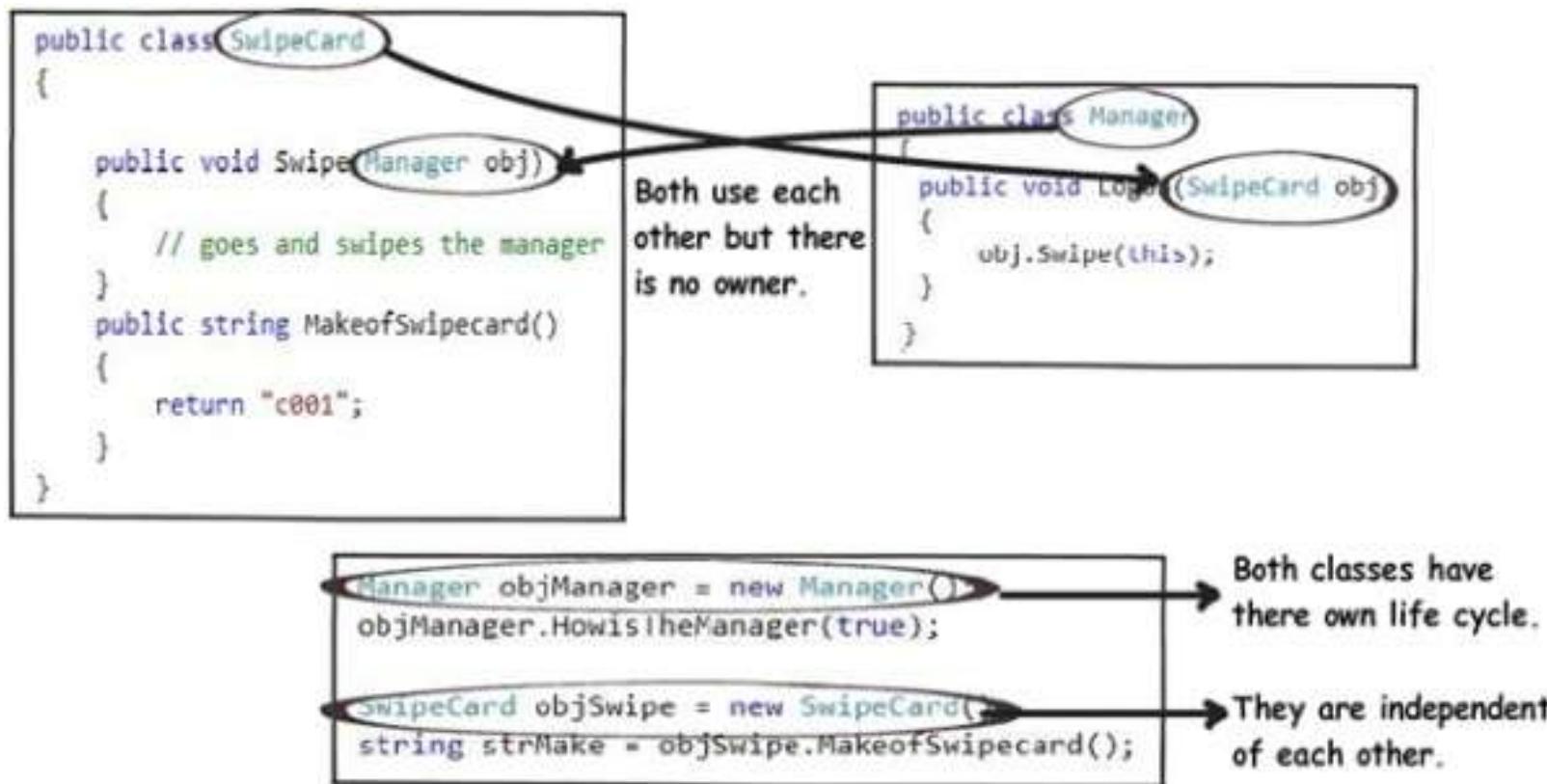
- **Association** : An *Association* is a bi-directional connection between classes (e.g. the "hires/works for" relationship between an "employee" class and a "boss" class). It is represented by a solid line. This line can be qualified with the name of relationship, and can also feature multiplicity rules (eg. one-to-one, one-to-many, many-to-many) for the relationship.



Associations



- Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other.
Note: It's uncommon to name both the association and the class roles.



Association

Multiplicity (Cardinality)

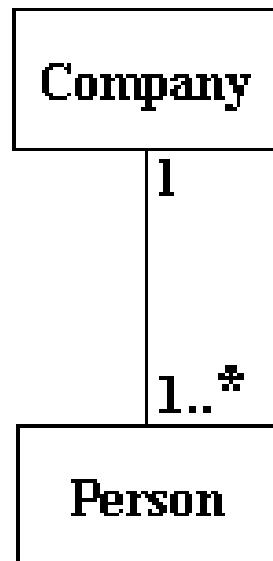
1 *no more than one*

0..1 *zero or one*

***** *many*

0..* *zero or many*

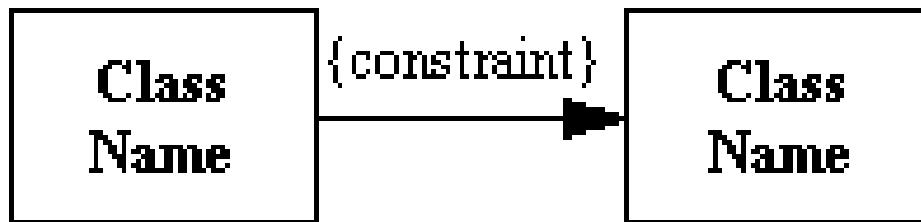
1..* *one or many*



- Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to *one* instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.

Constraint

- Place constraints inside curly braces {}.



Simple Constraint

Relationships

- **Aggregation:** Aggregations indicate a whole-part relationship, and are known as "has-a" relationships. An aggregation is represented as a line connecting the related classes with a diamond next to the class representing the whole.



```
public class Manager
{
    // Aggregation relation
    public List<Worker> Workers = new List<Worker>();
    .....
}
```

Worker is the child object and manager is the parent object.

```
public class Worker
{
    public string WorkerName = "";
}
```

Worker class cannot belong to other parent object.

```
Worker obj = new Wor
    ^
    +-- Worker
```

But worker object can have his own life time.

Aggregation

Relationships

- **Composition:** If a class cannot exist by itself, and instead must be a member of another class, then that class has a Composition relationship with the containing class. A Composition relationship is indicated by a line with a filled diamond.



```
public class Manager Managers Salary will increase if
{
    public double Salary;
    public void HowisTheManager(bool Good)...
}
```

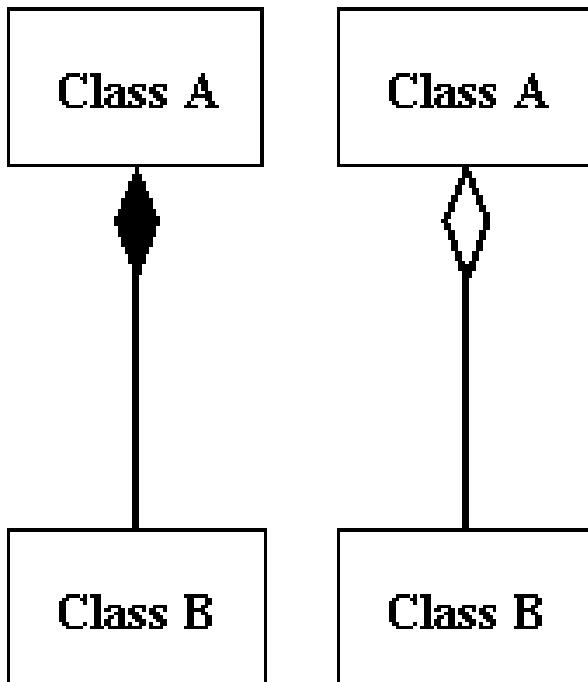
```
public class Project
{
    public bool IsSuccess...
    public Project(Manager obj)...
        Project will successful if manager
        is good.
}
```

```
Project obj = new Project(
    Project.Project(Manager obj))
```

Project and manager depend
on each other.

Composition

Composition and Aggregation



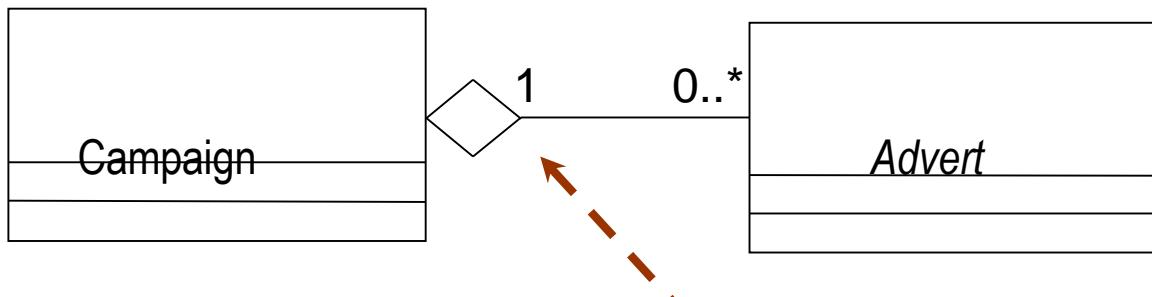
- Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.

Component-based Development

- The contribution of object-orientation:
 - Encapsulation of internal details makes it easier to use components in systems for which they were not designed
 - Generalization hierarchies make it easier to create new specialized classes when they are needed
 - Composition and aggregation structures can be used to encapsulate components

Composition and Aggregation

- Special types of association, both sometimes called whole–part
- A campaign is made up of adverts:



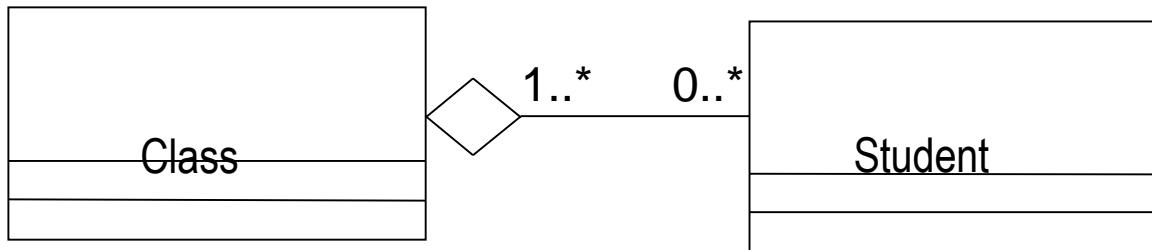
*Unfilled diamond
signifies aggregation*

Composition and Aggregation

- Aggregation is essentially any whole–part relationship
- Semantics can be very imprecise
- Composition is ‘stronger’:
 - Each part may belong to only one whole at a time
 - When the whole is destroyed, so are all its parts

Composition and Aggregation

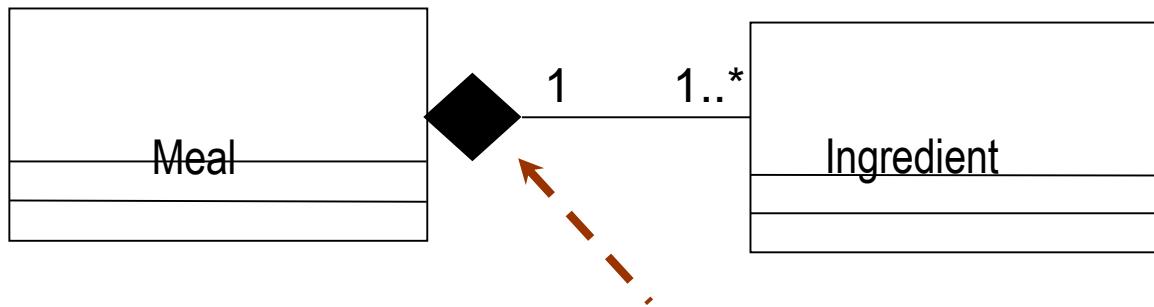
- An everyday example



- - Students could be in several classes
 - If class is cancelled, students are not destroyed!

Composition and Aggregation

- Another everyday example



Filled diamond signifies composition

- This is (probably) composition
 - Ingredient is in only one meal at a time
 - If you drop your dinner on the floor, you probably lose the ingredients too

Adding Structure

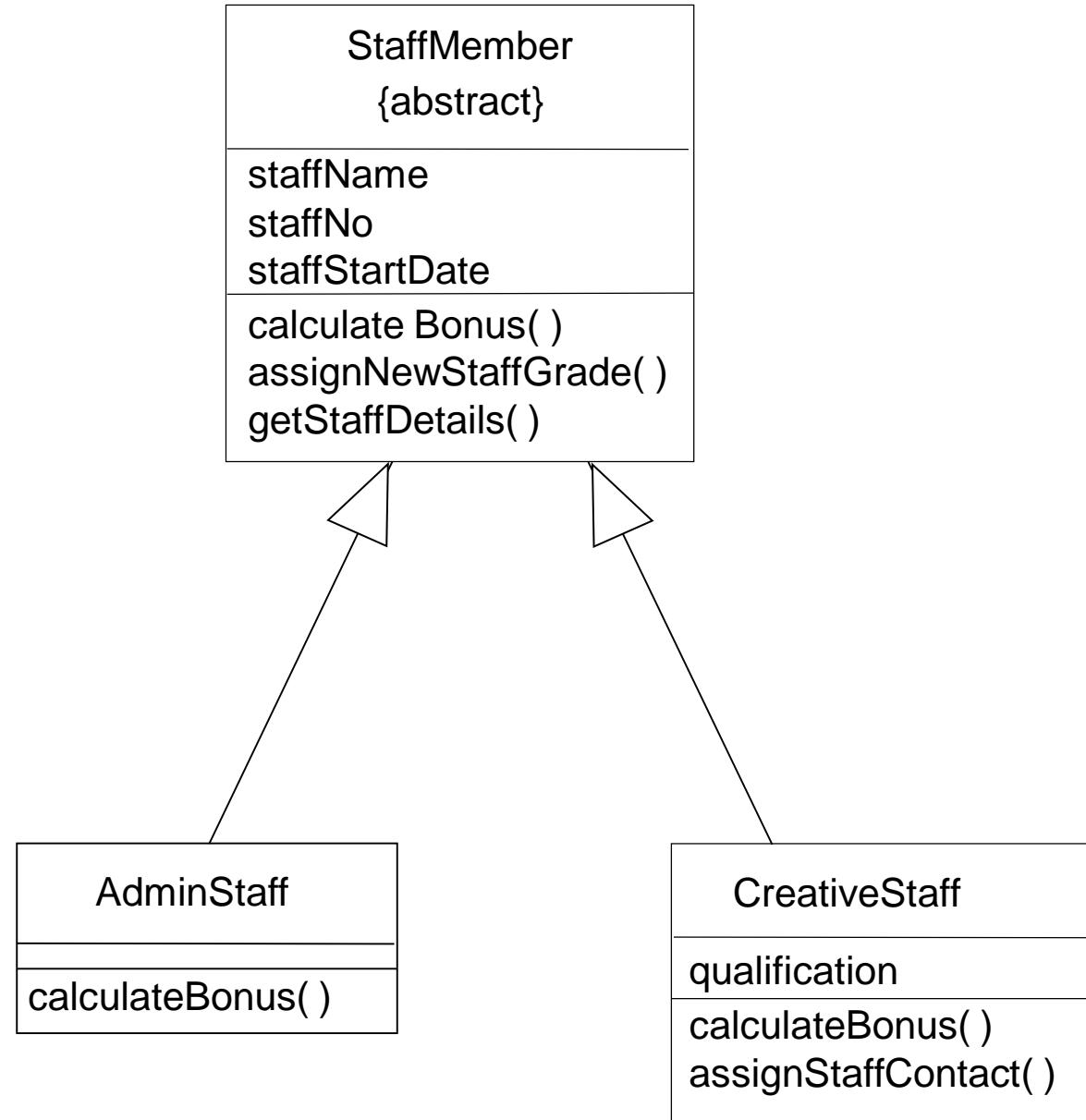
- Add generalization structures when
 - Two classes are similar in most details, but differ in some respects
 - May differ
 - In behaviour (operations or methods)
 - In data (attributes)
 - In associations with other classes

Adding Structure

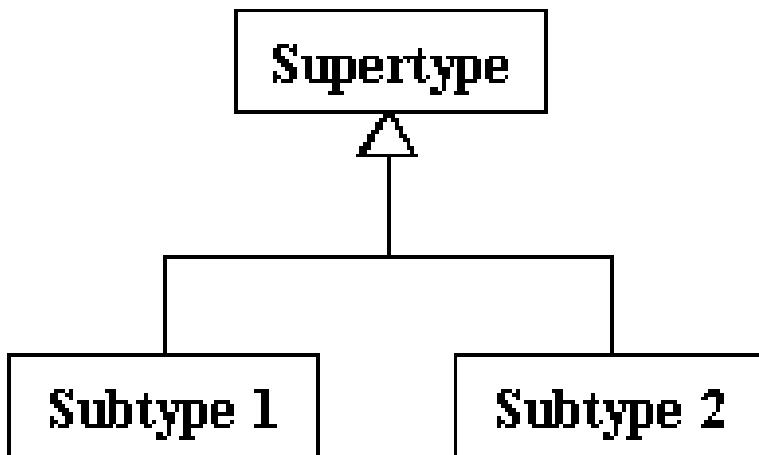
- Two types of staff:

Creative	<p>Have qualifications recorded</p> <p>Can be client contact for campaign</p> <p>Bonus based on campaigns they have worked on</p>
Admin	<p>Qualifications are not recorded</p> <p>Not associated with campaigns</p> <p>Bonus not based on campaign profits</p>

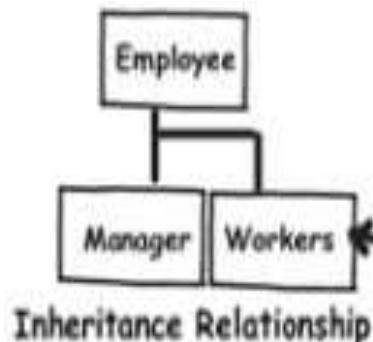
Adding Structure



Generalization

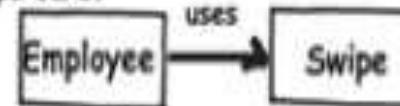


- Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



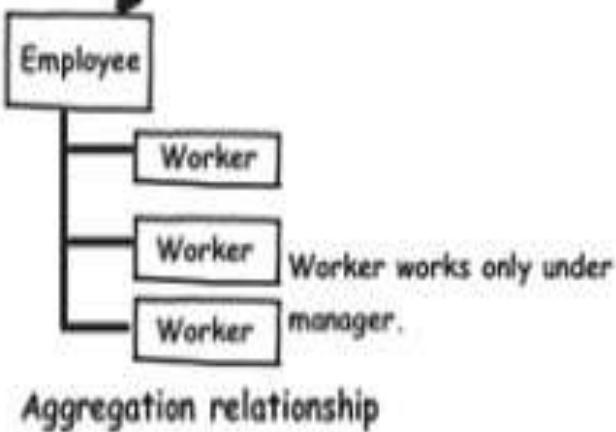
Inheritance Relationship

Swipe card can stay without manager and manager can have a new swipe card.



Association Relationship

1. Manager is a type of employee in the company.
2. He has a swipe card by which he enters the company premises
3. He has many workers who work under him.
4. He has the responsibility of ensuring project success.



Aggregation relationship

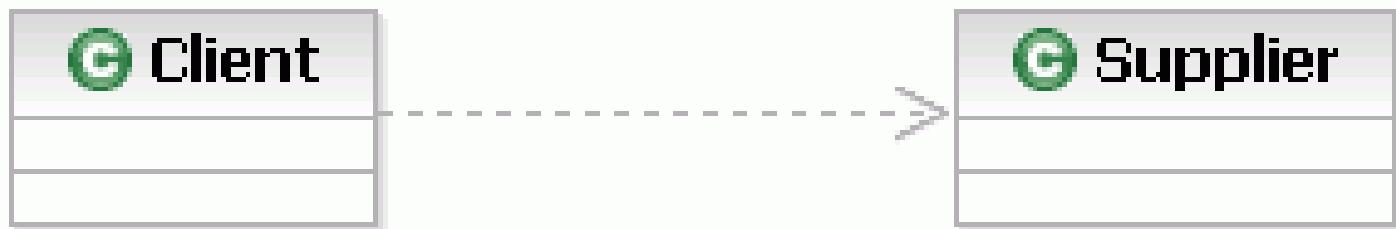


Managers salary will be incremented if project is successful. A project to be successful needs a good manager.

Composition Relationship

Relationships

- **Dependency:** When a class uses another class, perhaps as a member variable or a parameter, and so "depends" on that class, a Dependency relationship is formed. A **Dependency** is a relationship where the client does not have semantic knowledge of the supplier. It can be used, for example to show the relationship between an "encoder" class and a "generic Algorithm" class that is inserted in runtime to help it encode a stream. A dependency is shown as a dashed line pointing from the client to the supplier.



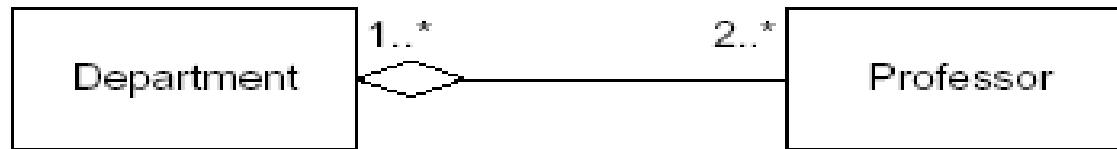
Relationships

Relationships

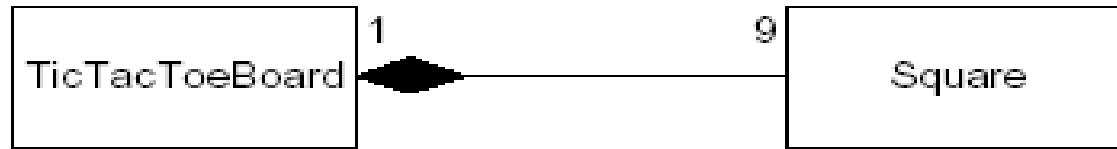
Association



Aggregation



Composition



Multiplicities

Exact: number

Range: use .. between pair of numbers

Arbitrary: *

Statechart diagrams

Statechart diagrams

- Describe the dynamic behavior of an individual object as a finite state machine.
- How to model object life cycles using statecharts
- How to develop statechart diagrams from interaction diagrams
- How to model concurrent behaviour in an object

UML Statechart Diagram

- A statechart diagram (also called a *state diagram*) shows the sequence of states that an object goes through during its life in response to outside stimuli and messages.

State

- 'A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action or waits for some event....

Conceptually, an object remains in a state for an interval of time. However, the semantics allow for modelling "flow-through" states which are instantaneous as well as transitions that are not instantaneous.'

(OMG, 2001)

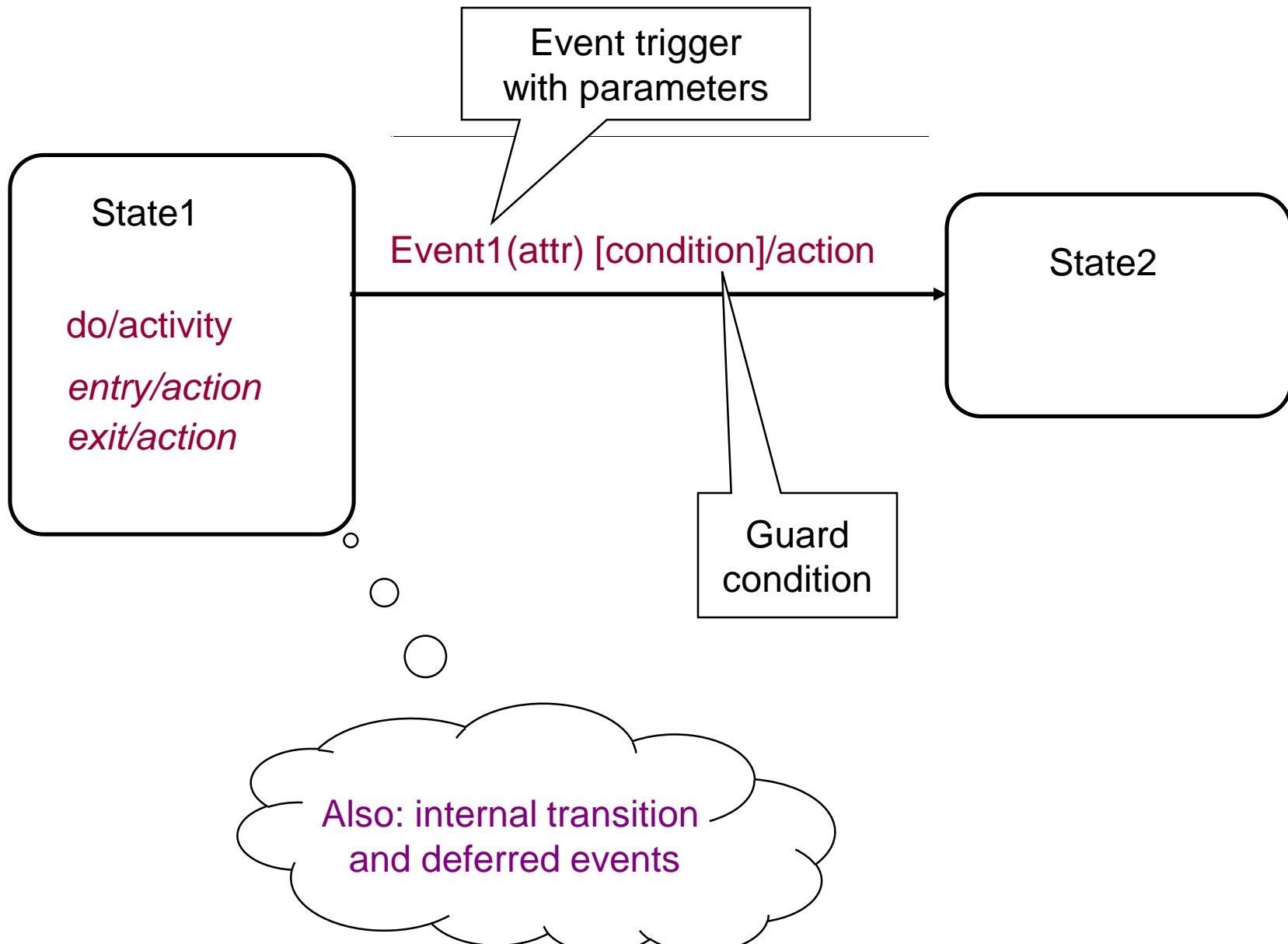
State Transition Diagrams

State transition diagrams are a useful tool for constructing the individual classes. Specifically, they aid in two important ways in “fleshing out” the structure of the class:

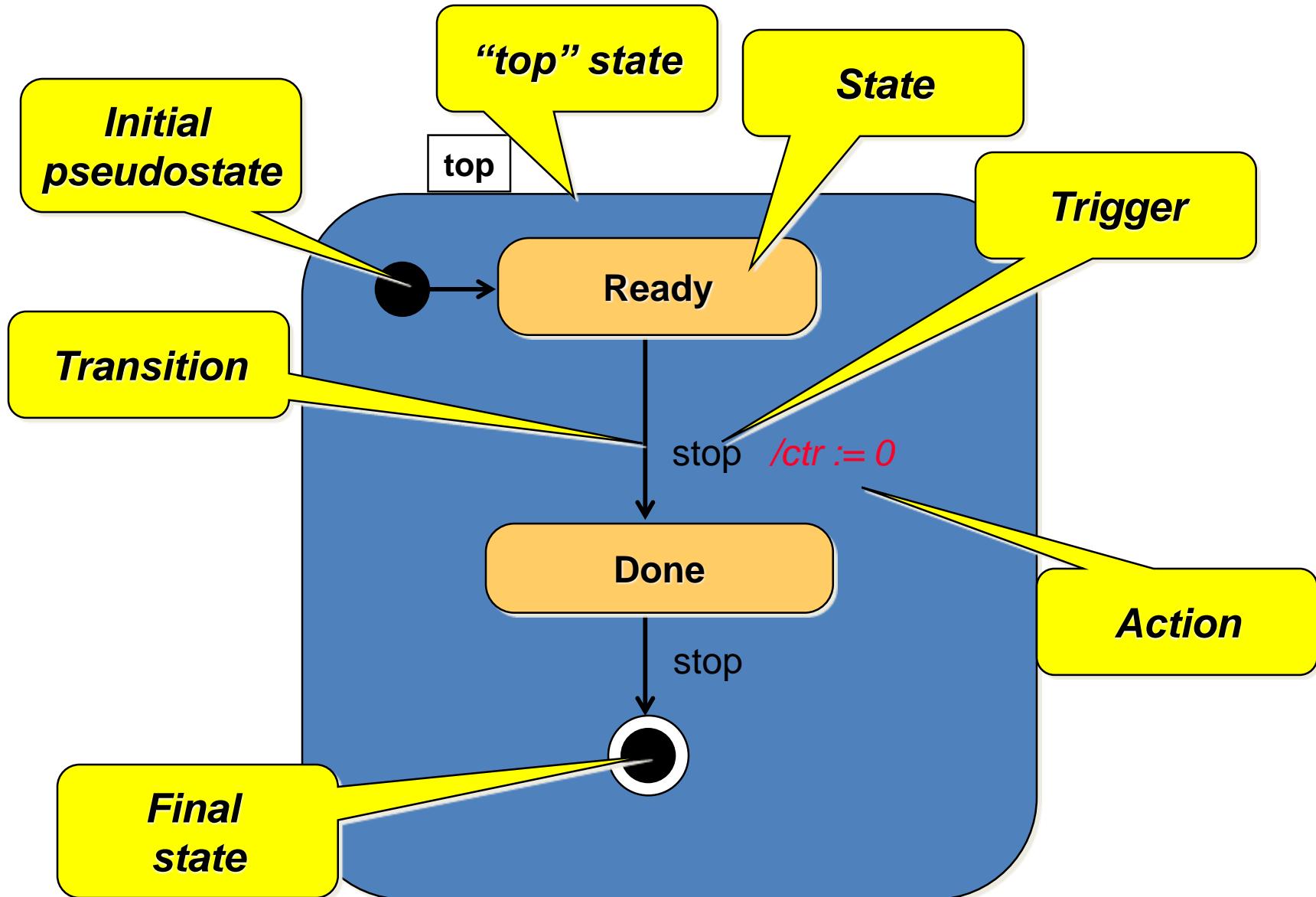
1. method development -- State transition diagrams provide the “blueprints” for developing the algorithms that implement methods in the class
2. attribute identification – Attributes contain the state information needed for regulating the behaviors of the instances of the class

When constructing state transition diagrams, take care to ensure that the post-conditions stipulated in the contracts are enforced.

UML Statechart Diagram Notation



Basic UML Statechart Diagram



Types of Event

- A *change event* occurs when a condition becomes true
- A *call event* occurs when an object receives a call to one of its operations either from another object or from itself
- A *signal event* occurs when an object receives a signal (an asynchronous communication)
- An *elapsed-time event* is caused by the passage of a designated period of time after a specified event (frequently the entry to the current state)

UML - Statechart Diagram Example

Event

Initial state

State

button1&2Pressed

button2Pressed

Blink
Hours

Increment
Hours

Transition

button1Pressed

button1&2Pressed

button2Pressed

Blink
Minutes

Increment
Minutes

button1Pressed

Stop
Blinking

Final state

Final state

Blink
Seconds

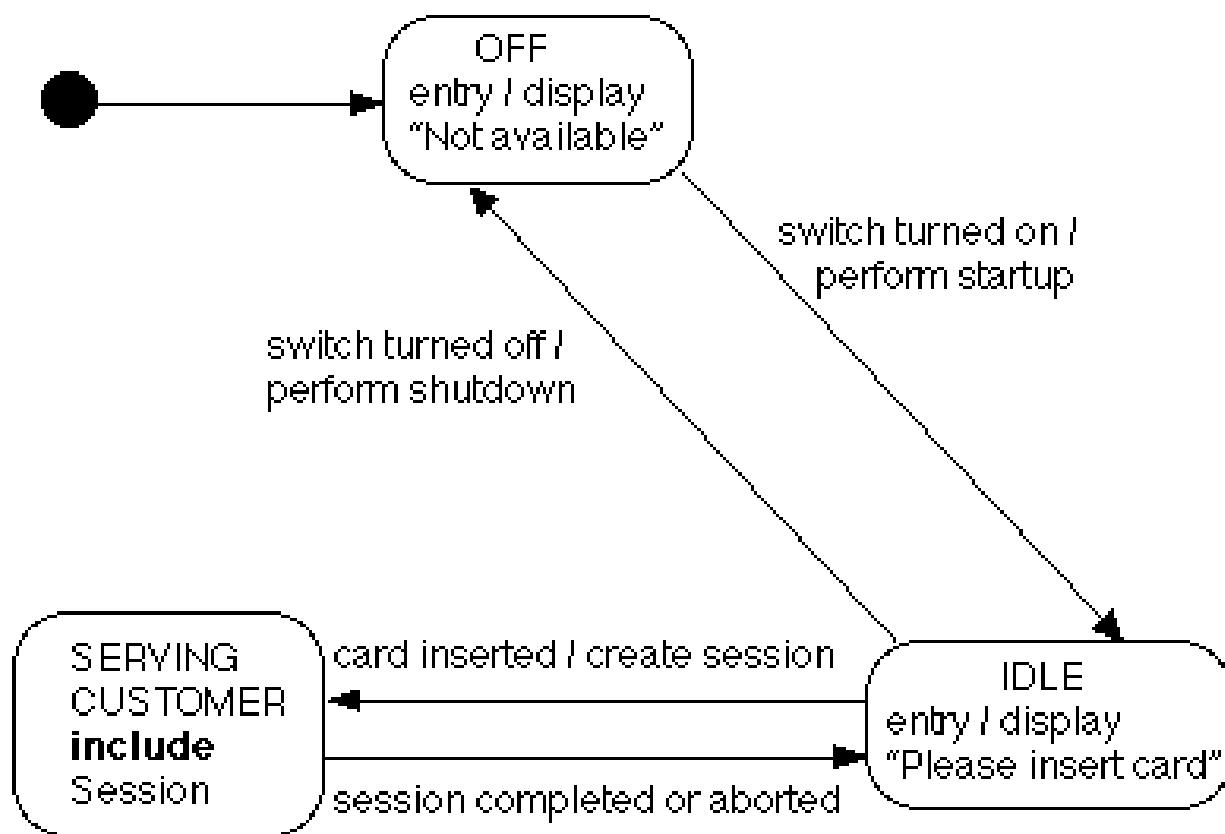
Increment
Seconds

button1&2Pressed

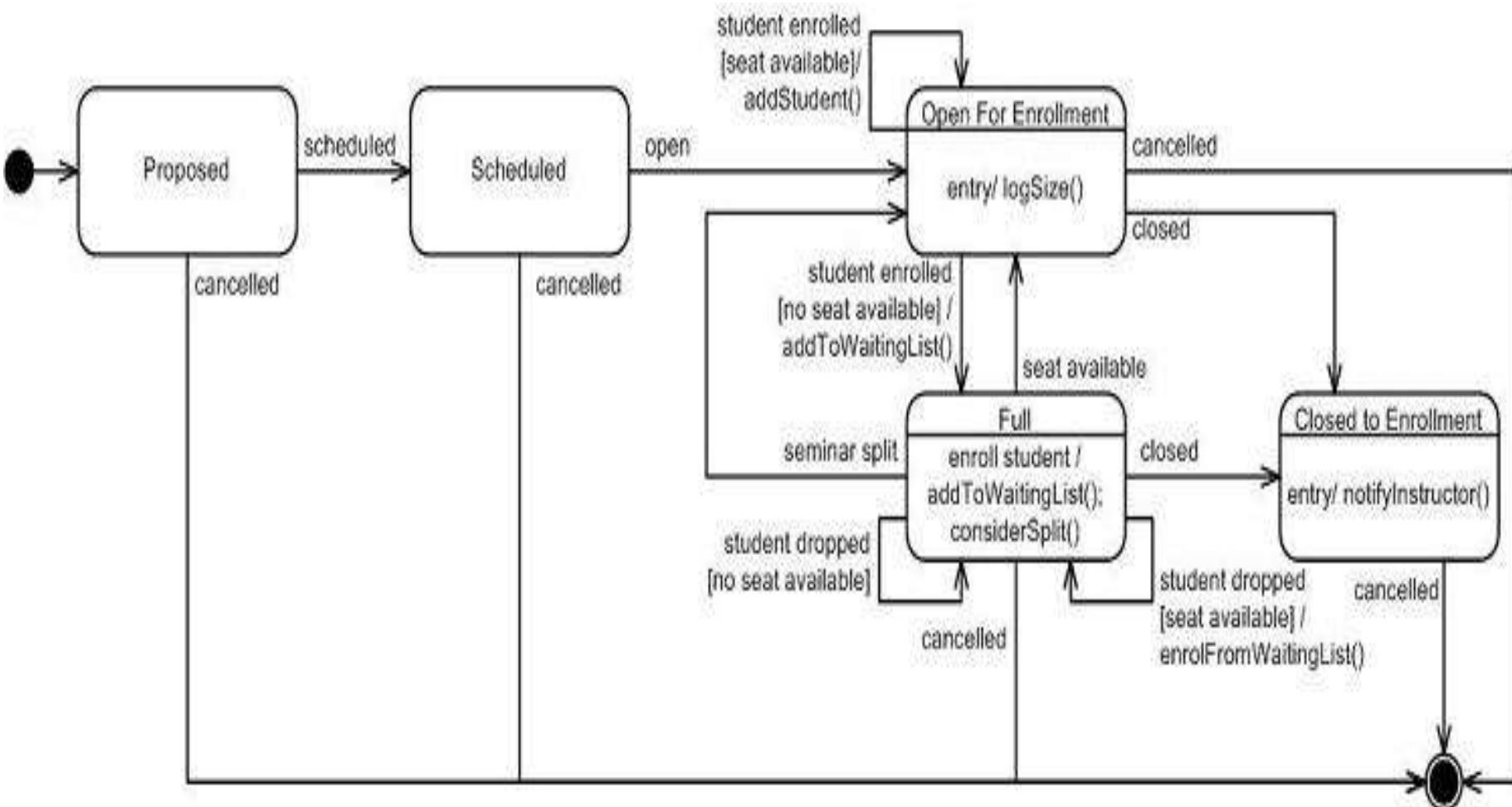
Final state

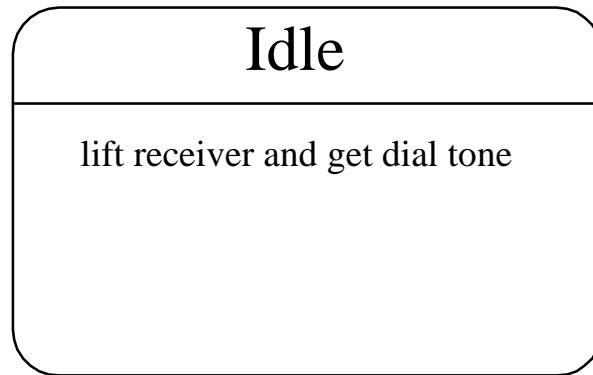
Statechart diagrams for objects with interesting dynamic behaviour - represent **behaviour** as **states** and **transitions**

State-Chart for Overall ATM (includes System Startup and System Shutdown Use Cases)

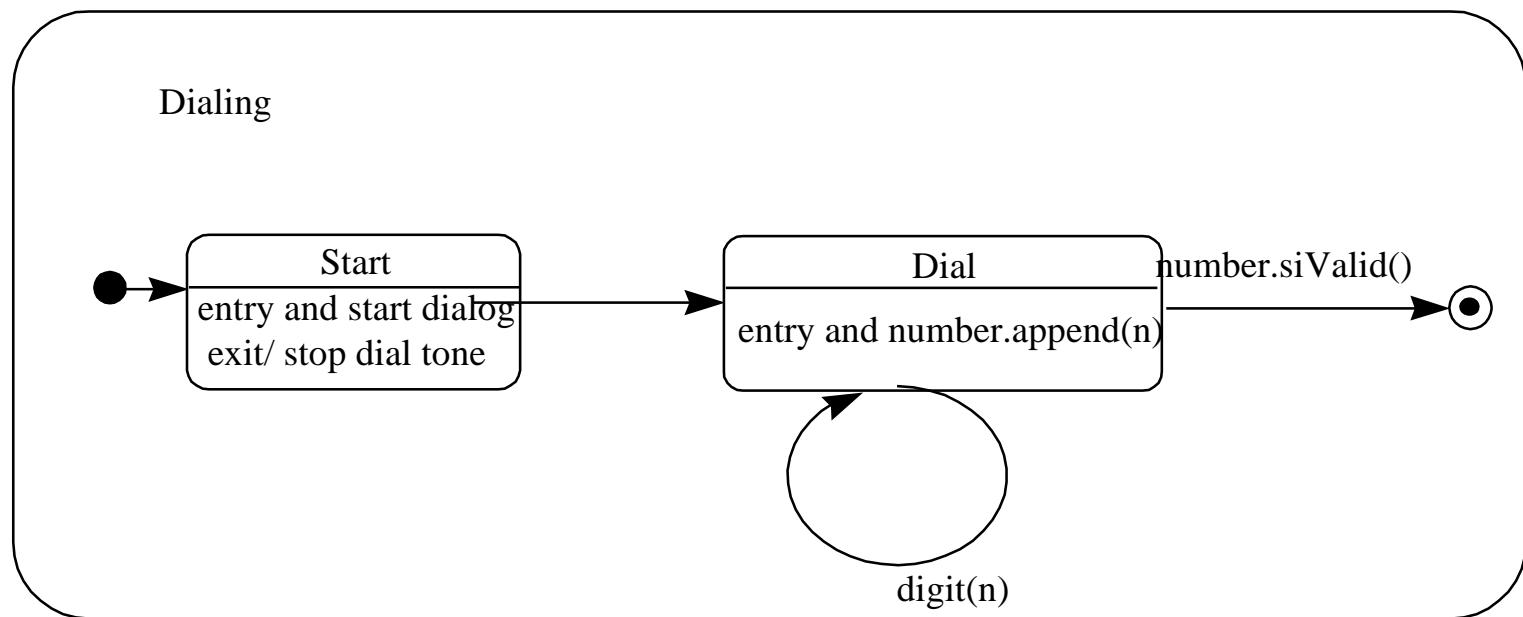


A seminar class during registration





State

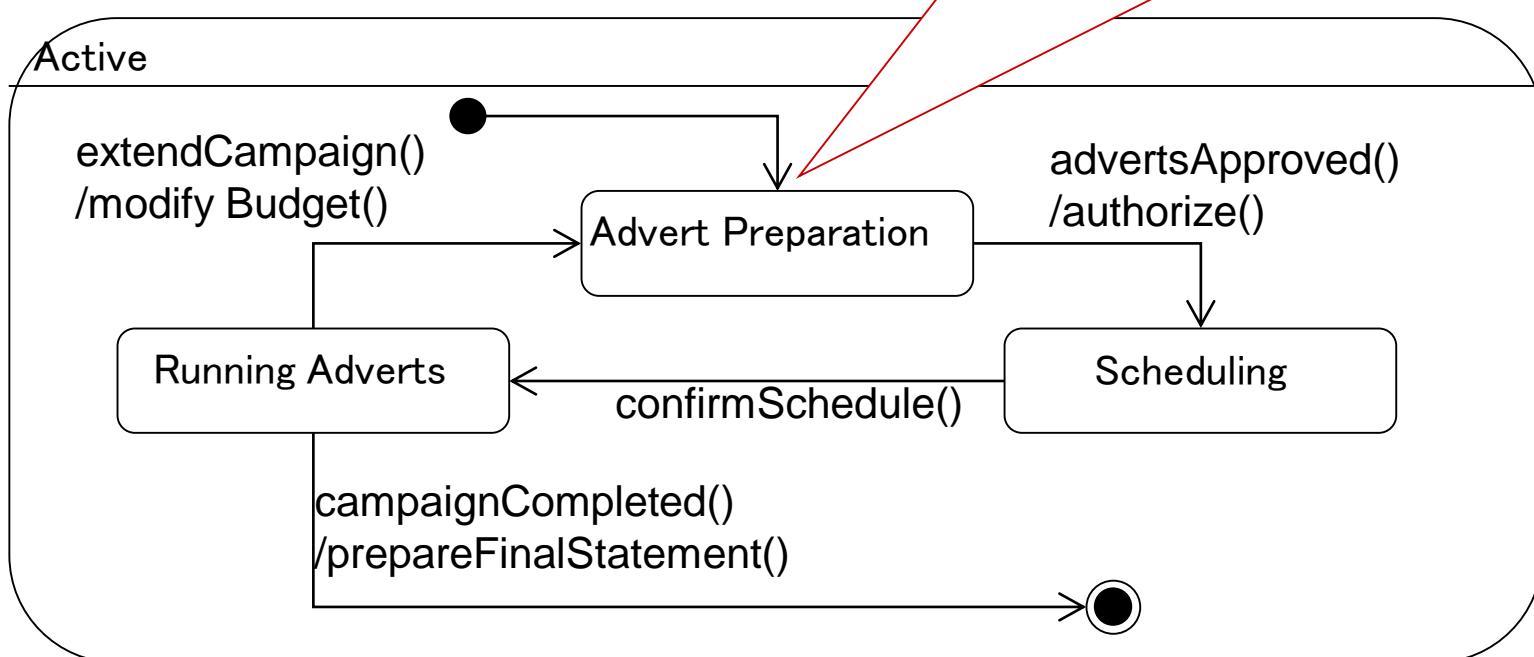


Substates

Nested Substates

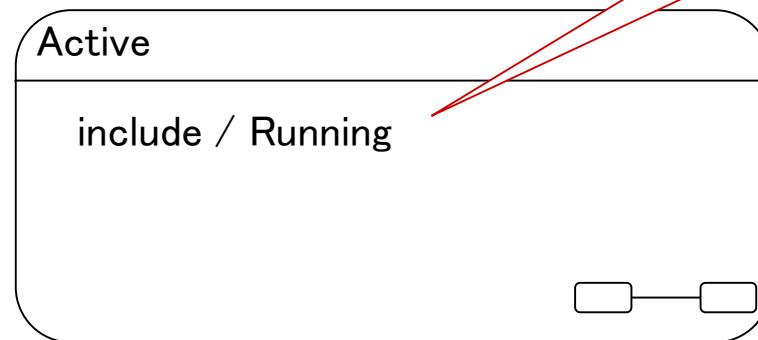
The Active state of Campaign showing nested substates

The transition from the initial pseudostate symbol should not be labelled with an event but may be labelled with an action, though it is not required in this example



Nested States

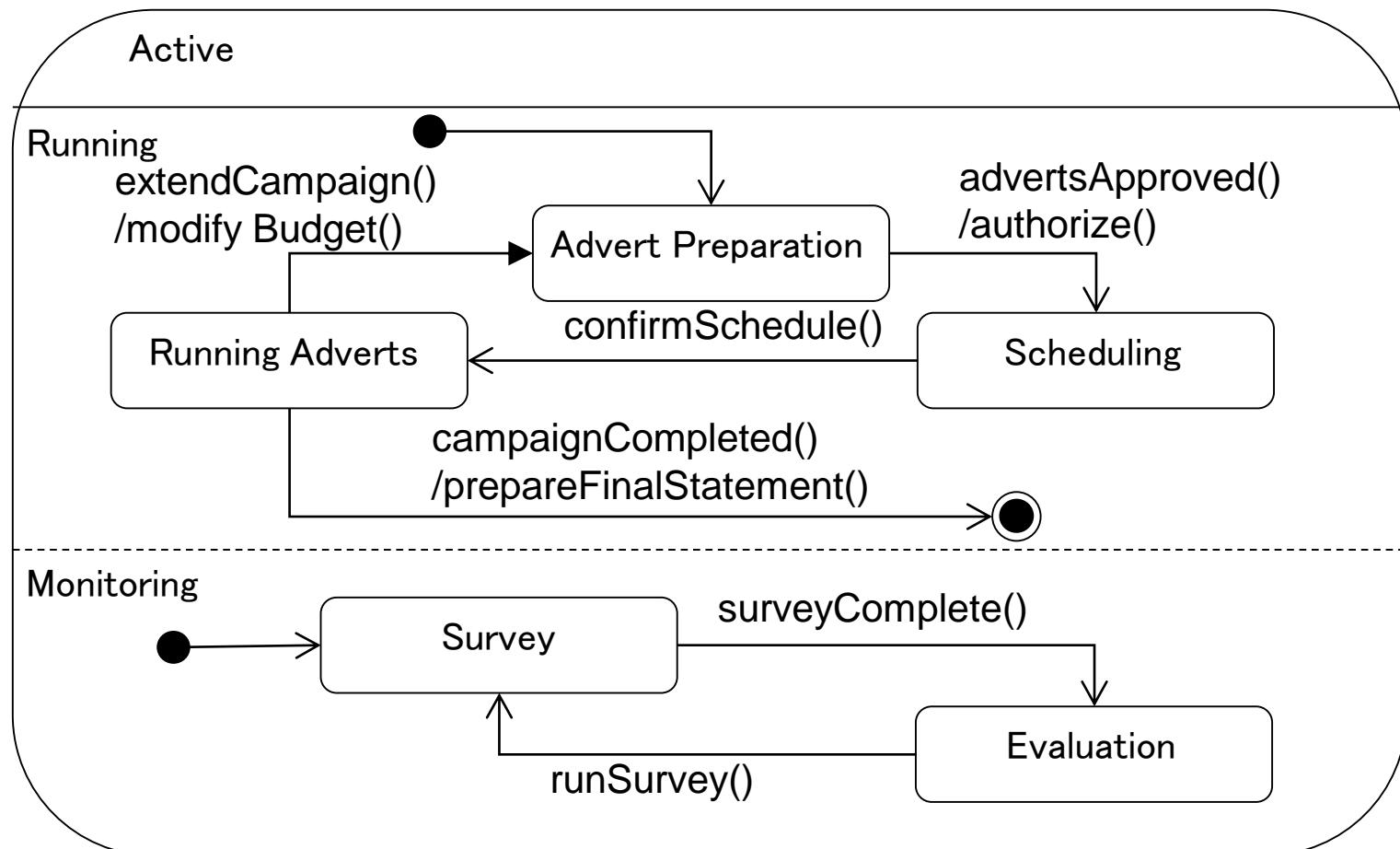
The Active state of Campaign with the detail hidden



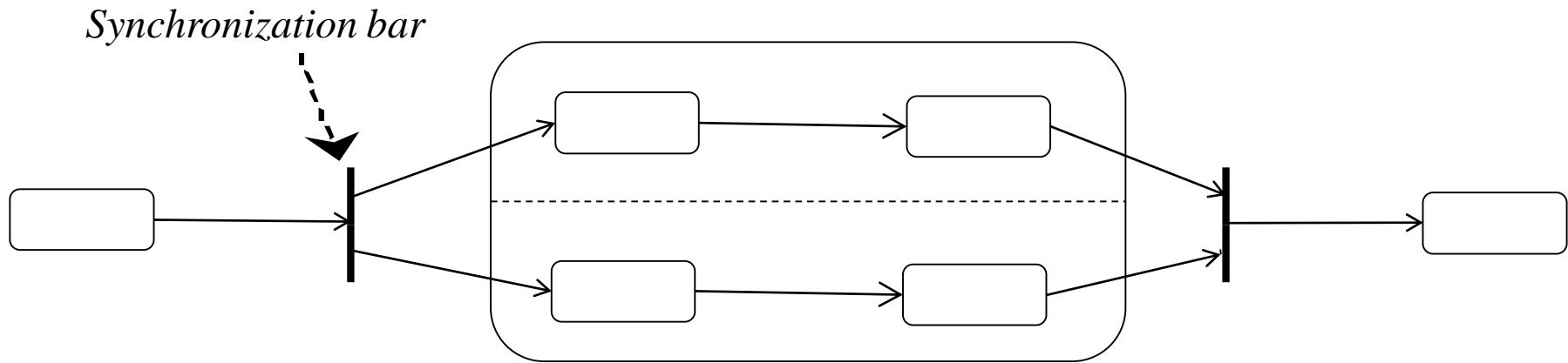
The submachine
Running is
referenced using the
include statement

Hidden
decomposition
indicator icon

The Active State with Concurrent Substates



Synchronized Concurrent Threads.



- Explicitly showing how an event triggering a transition to a state with nested concurrent states causes specific concurrent substates to be entered
- Shows that the composite state is not exited until both concurrent nested statecharts are exited

Problem Statement: Control for a Toy Car

- Power is turned on
 - Car moves forward and car headlight shines
- Power is turned off
 - Car stops and headlight goes out.
- Power is turned on
 - Headlight shines
- Power is turned off
 - Headlight goes out.
- Power is turned on
 - Car runs backward with its headlight shining.

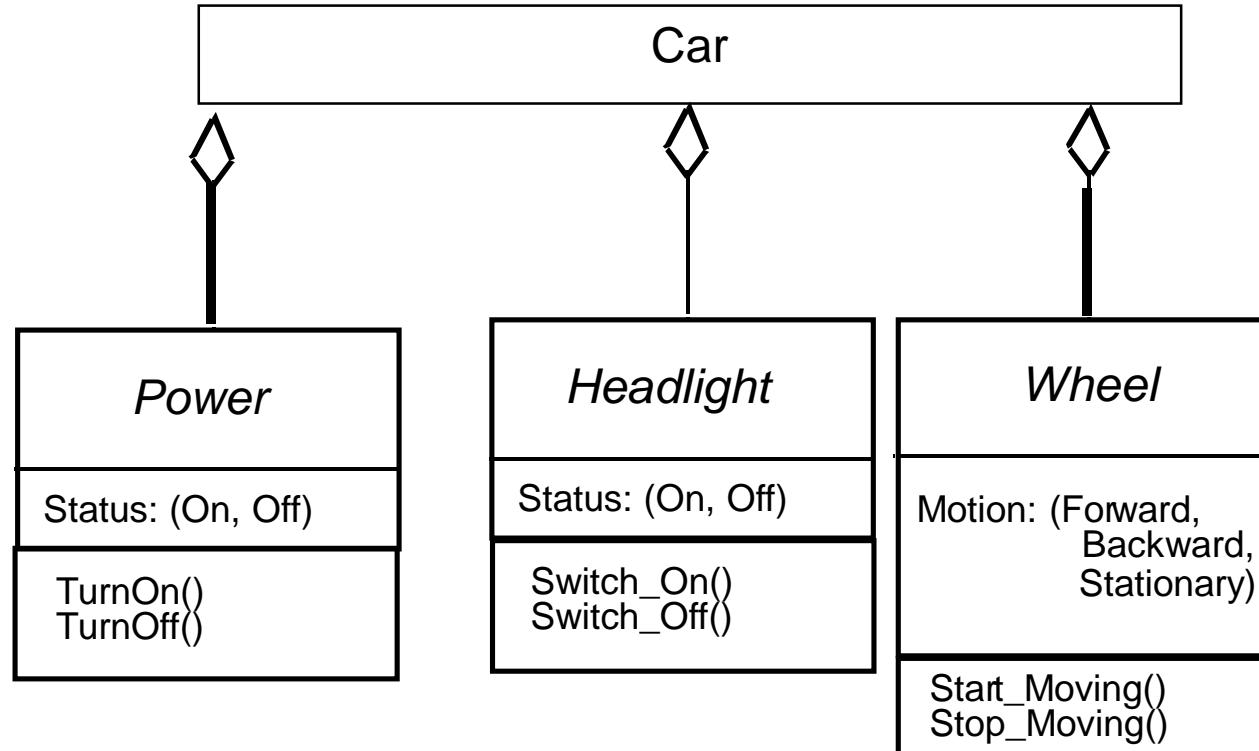
- Power is turned off
 - Car stops and headlight goes out.
- Power is turned on
 - Headlight shines
- Power is turned off
 - Headlight goes out.
- Power is turned on
 - Car runs forward with its headlight shining.

Find the Functional Model: Do Use Case Modeling

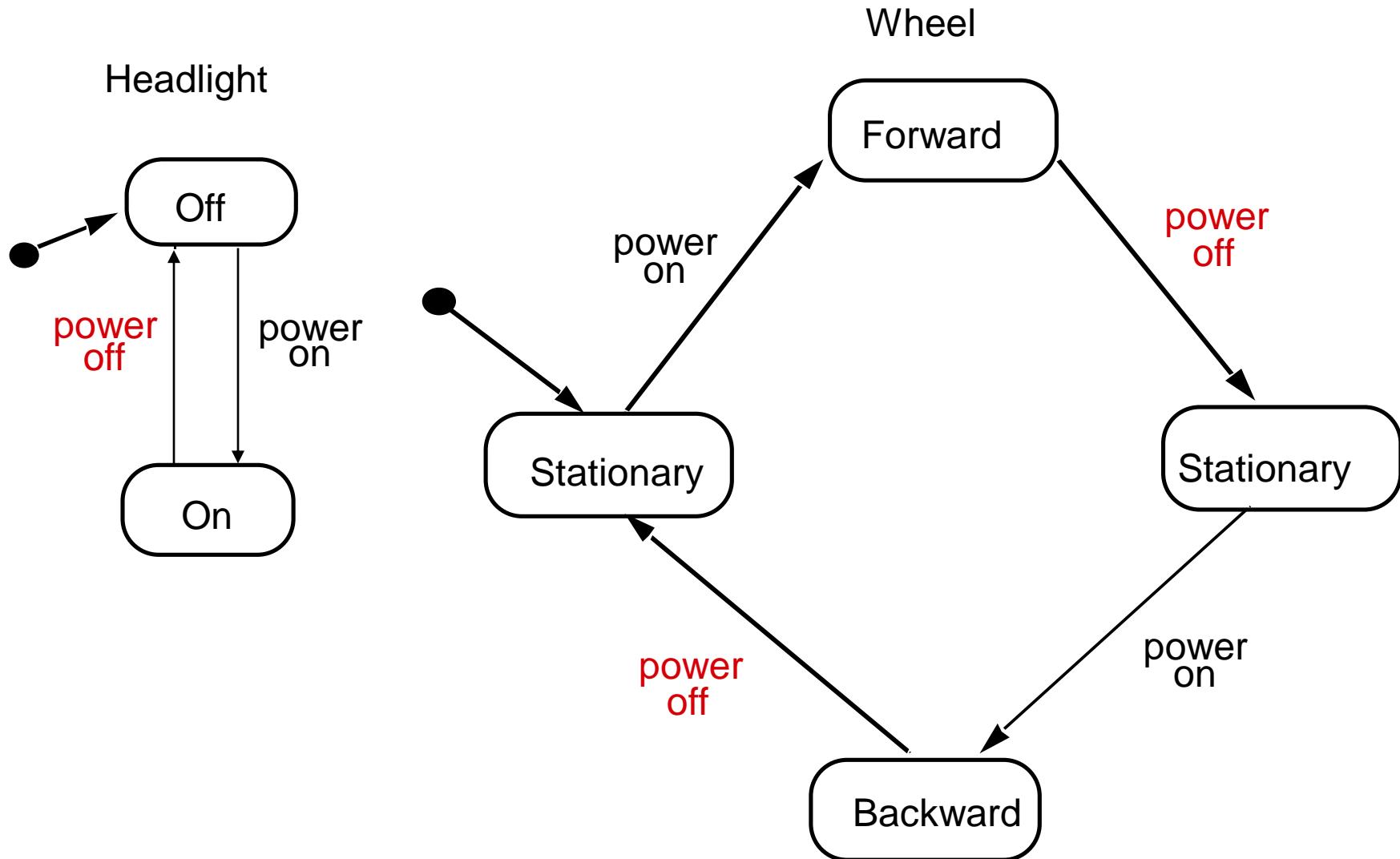
- Use case 1: Move car forward
 - Entry condition: Car is stationary, headlights are off
 - Flow of events
 - Driver turns power on
 - Exit condition:
 - Car runs forward with its headlight shining.
- Use case 2: Stop forward moving car
 - Entry condition: Car moves forward, headlights on
 - Flow of events:
 - Driver turns power off, car stops, headlights go out.
 - Exit condition: Car stationary, headlights are out.

- Use case 3: Move car backward
 - Entry condition: Car is stationary, headlights off
 - Flow of events:
 - Driver turns power on
 - Exit condition: Car moves backward, headlights on
- Use case 4: Stop backward moving car
 - Entry condition: Car moves backward, headlights on
 - Flow of events:
 - Driver turns power off, car stops, headlights go out.
 - Exit condition: Car stationary, headlights are out.

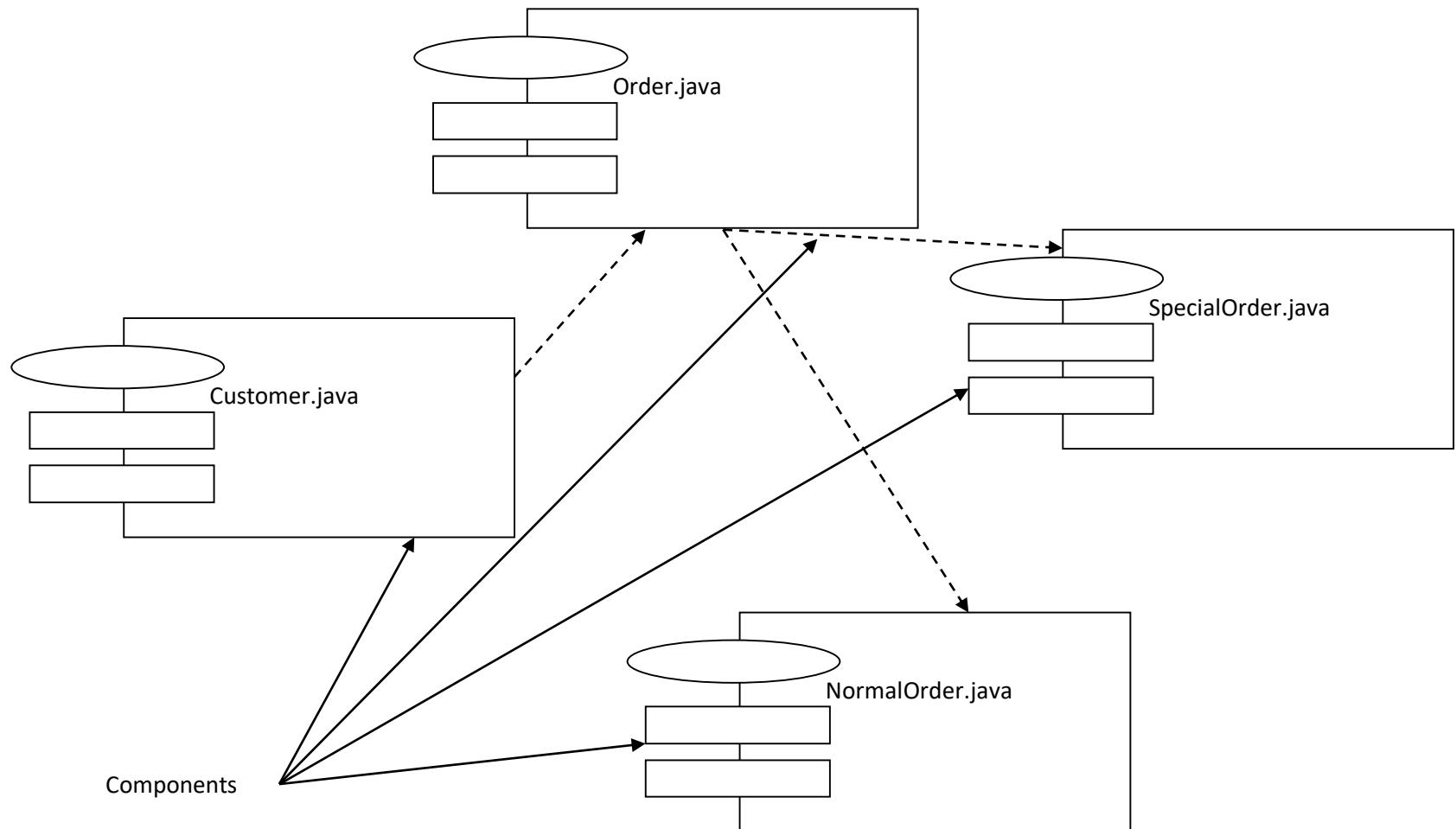
Toy Car: Object Model



Toy Car: Dynamic Model



Component Diagram

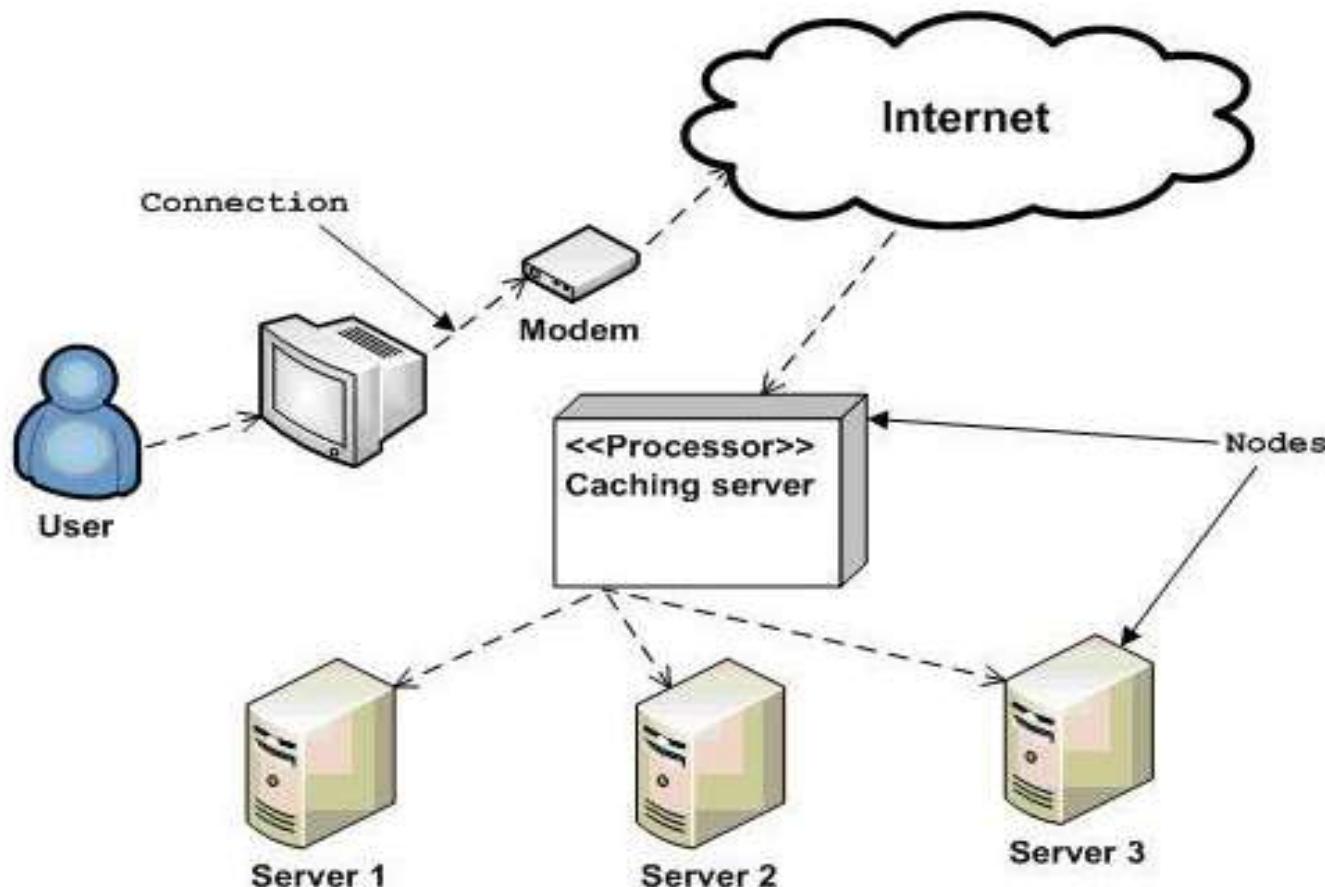


Component Diagram

Used in:

- Model the components of a system.
- Model database schema.
- Model executables of an application.
- Model system's source code

Deployment Diagram

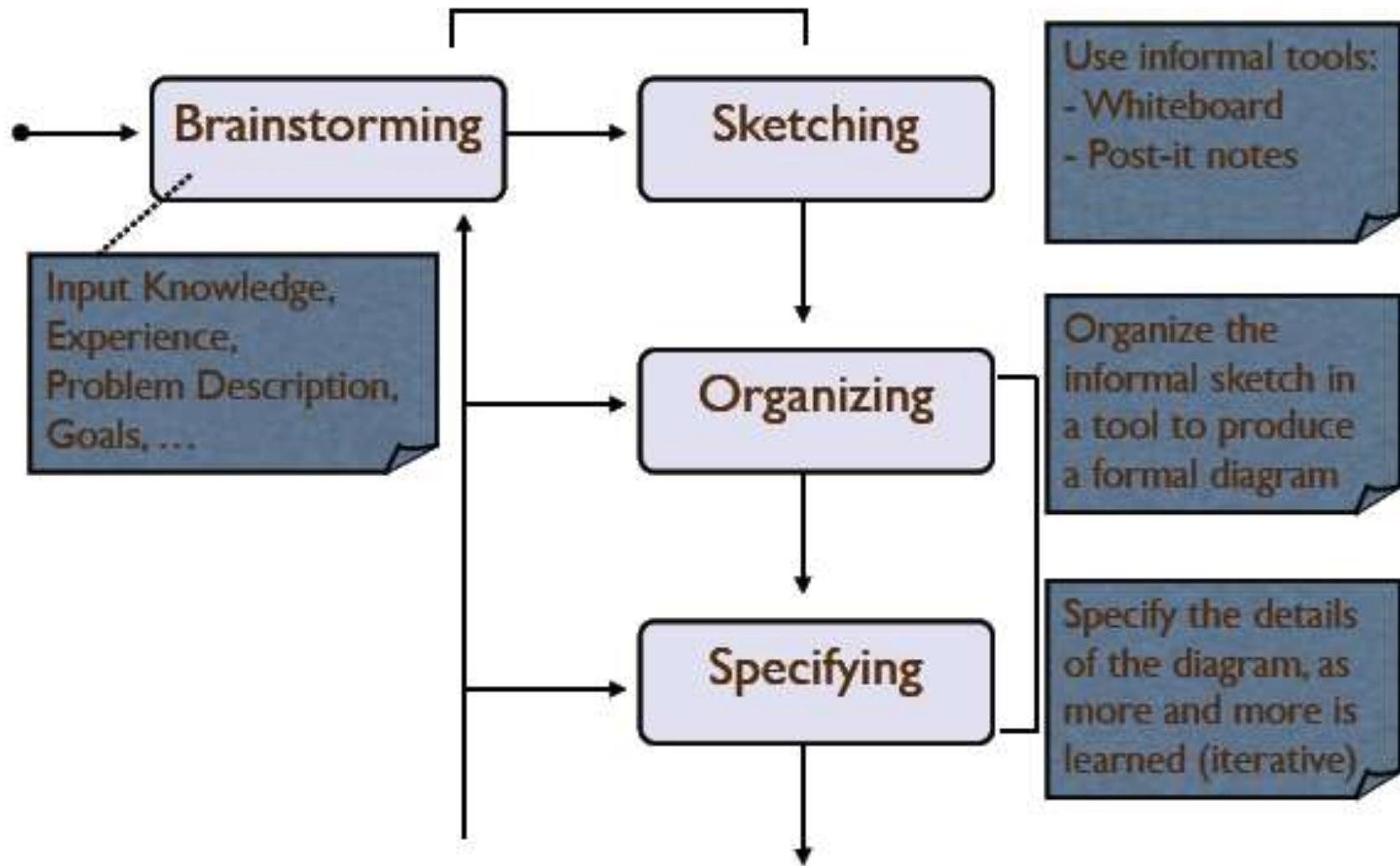


Deployment Diagram

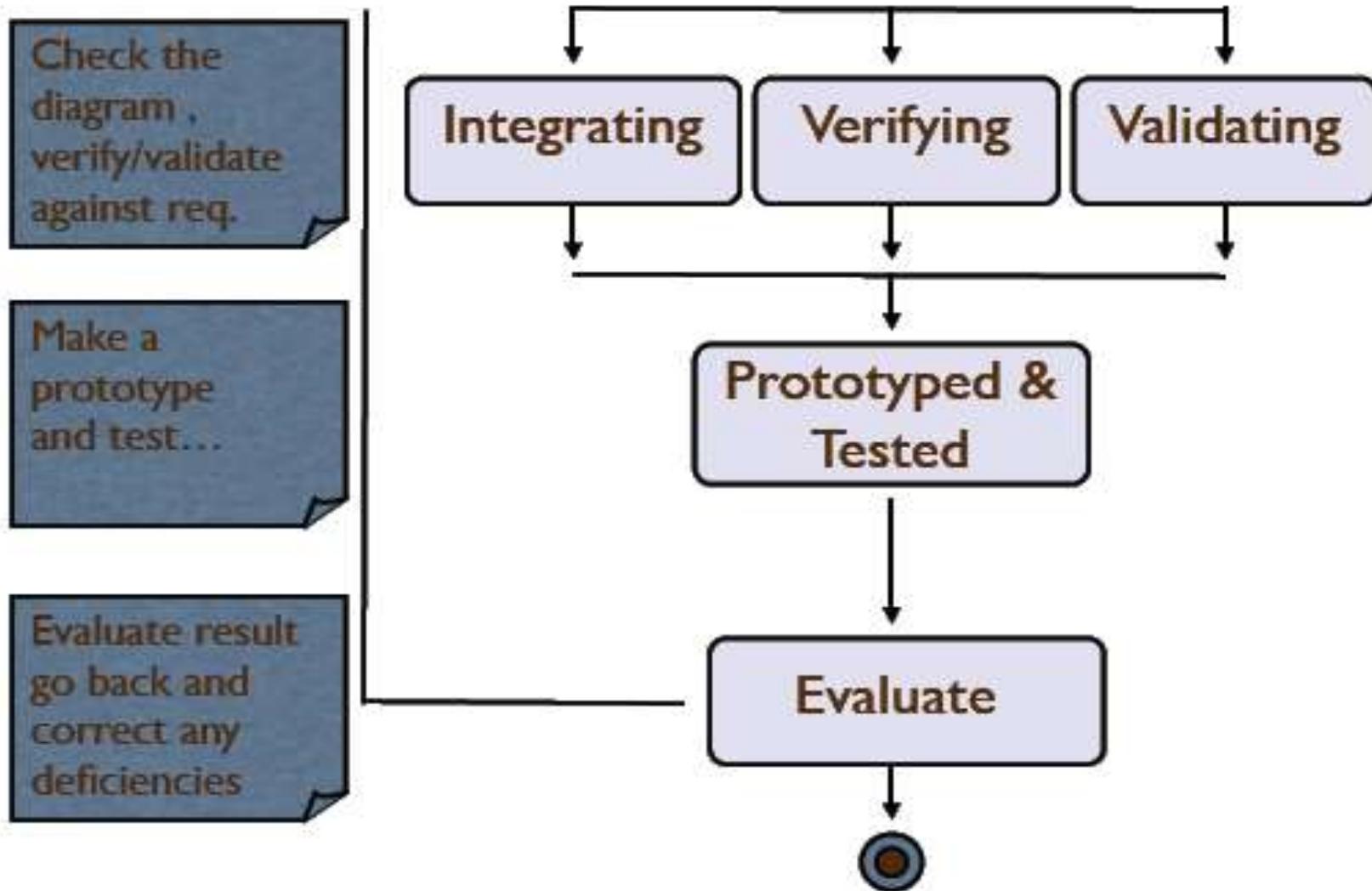
Used in:

- To model the hardware topology of a system.
- To model embedded system.
- To model hardware details for a client/server system.
- To model hardware details of a distributed application.
- Forward and reverse engineering.

A Process for Making Models...



A Process for Making Models...





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.10

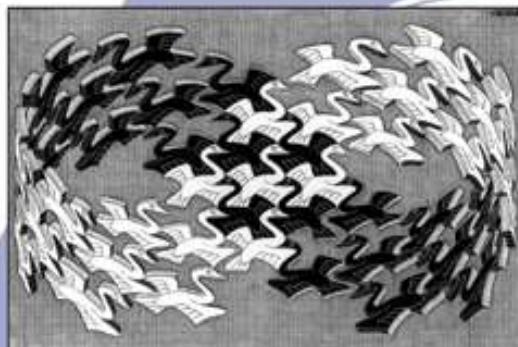
Today's Agenda

- Object Oriented Design
- GoF Patterns

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

1995



The “gang of four” (GoF)

Design Patterns book [catalogs 23 different patterns](#)

- Solutions to different classes of problems, in C++ & Smalltalk
- Problems and solutions are broadly applicable, used by many people over many years
- Patterns suggest opportunities for reuse in analysis, design and programming
- GOF presents each pattern in a [structured format](#)
 - *What do you think of this format? Pros and cons?*

Elements of Design Patterns

- Design patterns have 4 essential elements:
 - Pattern name: increases vocabulary of designers
 - Problem: intent, context, when to apply
 - Solution: UML-like structure, abstract code
 - Consequences: results and tradeoffs



Design Patterns are NOT

- Data structures that can be encoded in classes and reused as *is* (i.e., linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)
- If they are not familiar data structures or complex domain-specific subsystems, *what are they?*
- They are:
 - “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

Three Types of GoF Patterns

- **Creational patterns:**
 - Deal with initializing and configuring objects
- **Structural patterns:**
 - Composition of classes or objects
 - Decouple interface and implementation of classes
- **Behavioral patterns:**
 - Deal with dynamic interactions among societies of objects
 - How they distribute responsibility

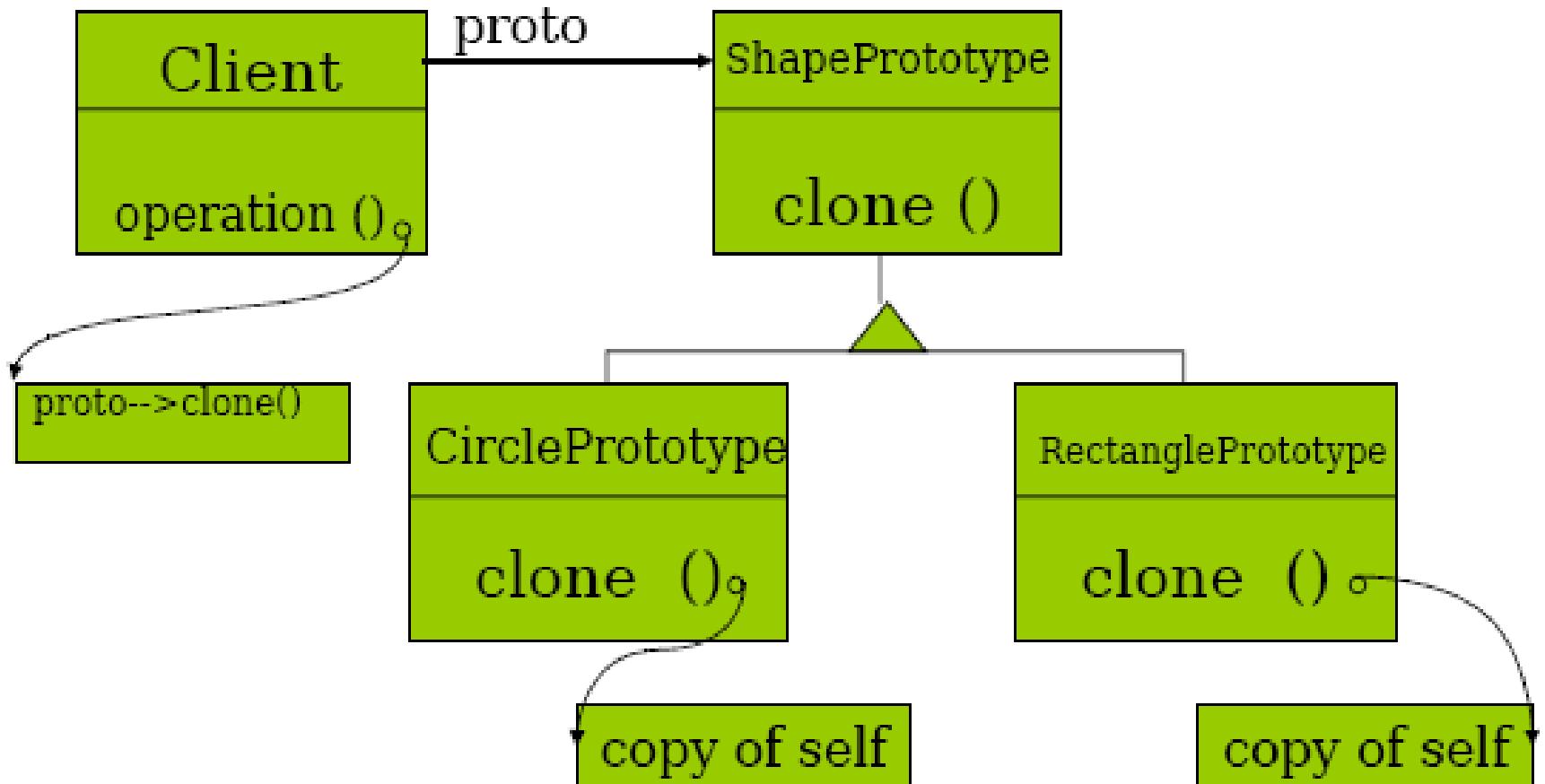
Benefits of Design Patterns

- Improves communication between designers by use of pattern names vs. the details of the patterns.
- Captures experience of solving a type of problem.
- Provide a way of reusing design.
- Provide a mechanism for making designs more reusable.
- Provides a mechanism for systematizing the reuse of things that have been seen before.
- Can be used to teach good design.

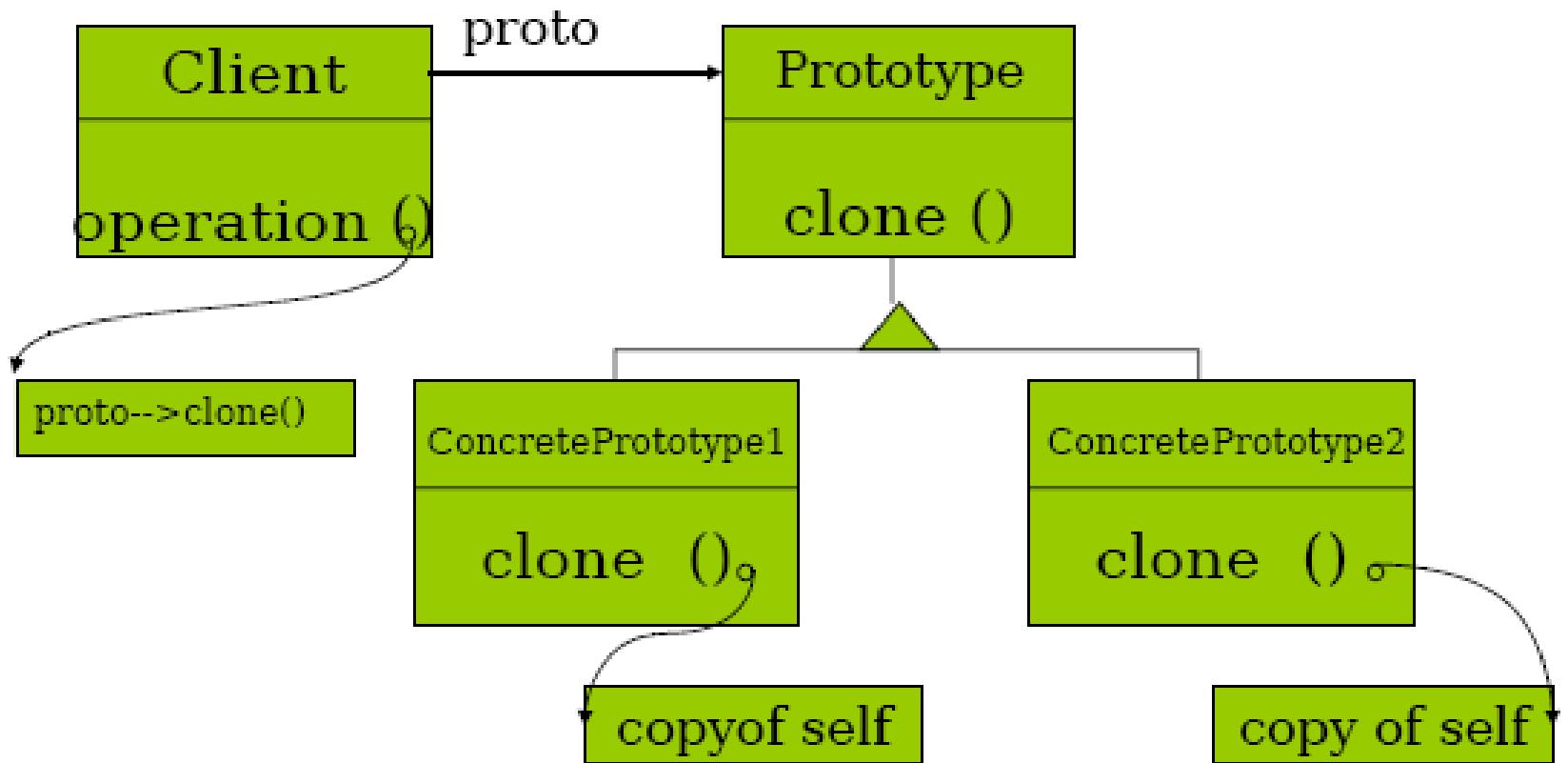
A Problem

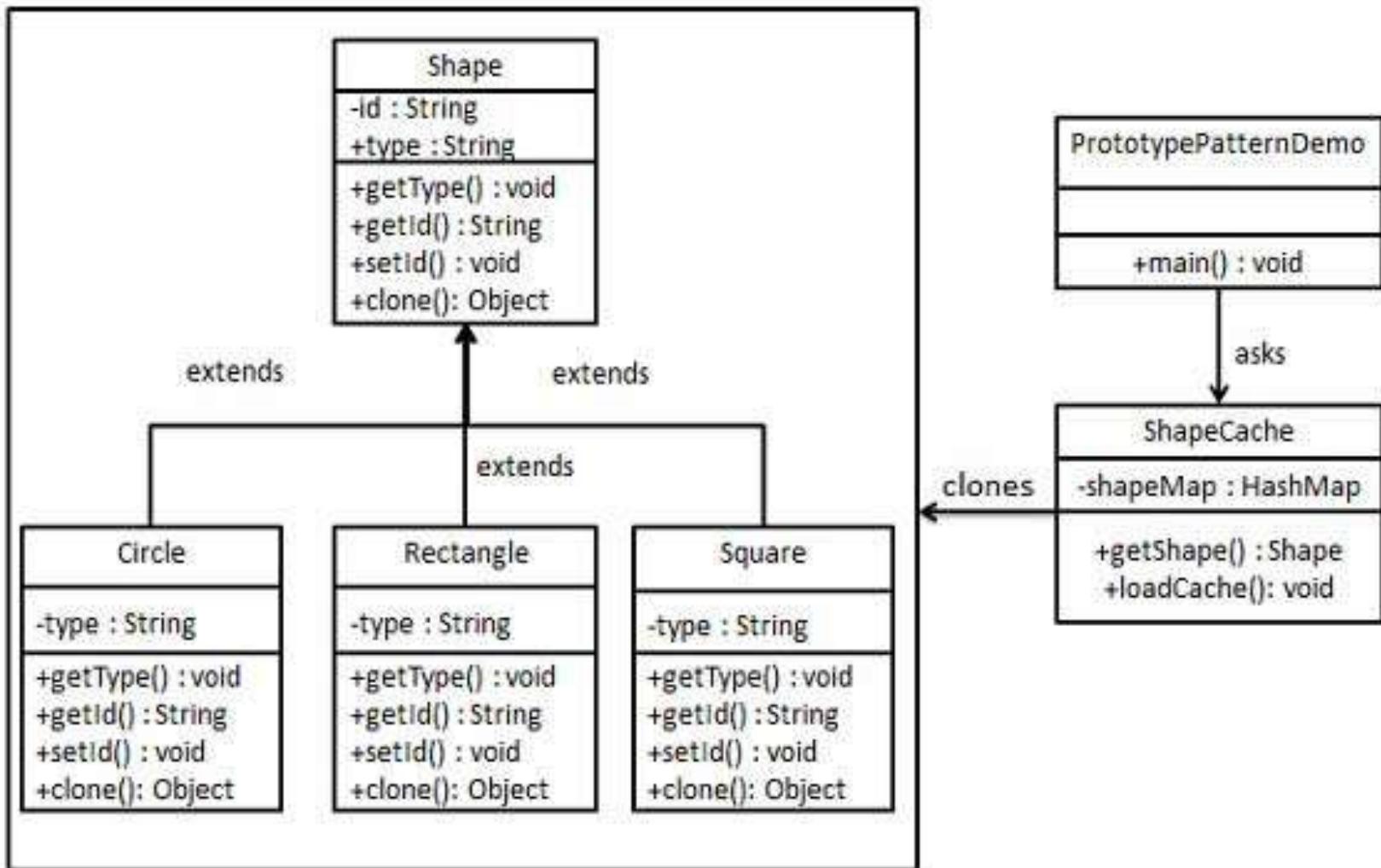
- In a graphical editor, by clicking on an object, one can obtain a copy of the original object. Obtaining a copy of an existing object is a common design problem in on-line compositions.
- We can provide a design solution to solve this problem, and reuse this design whenever similar situation arises.

The Solution



The Design Pattern: Prototype





Describing a Design Pattern

- ✓ Specify the generic problem that is solved
- ✓ Motivate the design pattern solution with the help of an example
- ✓ Provide the structure for the pattern
- ✓ Discuss collaborations between classes
- ✓ Discuss other issues related to the pattern such as trade-offs, implementation techniques etc.

Template provided by Erich Gamma et. al

- *Pattern name, its classification*
- *Intent, Motivation, Applicability*
- *Structure, Participants, Collaborations*
- *Consequences, Implementation, Sample code,*
- *Known uses*
- *Related patterns*

Organization

Patterns:

Behavioral: concerned about ways in which objects interact
Observer

Structural: concerned about the composition of objects and classes
Façade

Creational : concerned about ways to create new objects
Abstract Factory
Factory Method
Singleton

Excellent reference:

Design Patterns book by Erich Gamma, et al., Addison-Wesley, 1994.

Software Design – **Templates**

- What are templates?
 - They provide (generic) solutions – often for more than one instance of a problem.
 - These solutions can be (contextually) reused with minor adaptations.
- Templates are contextual but generic solutions that are re-usable with minor adaptations.

Software Design – Templates

- Observations:
 - There are some differences among templates
 - Problem contexts are different:
 - Low level design (close to coding)
 - Medium level design
 - High level design (close to architecture)
 - Often leads to different granularities!

Software Design – Templates

- Low level templates:
 - Fine grained;
 - Often no additional work required to use.
 - Implementation (say language or OS) specific
 - a.k.a **IDIOMS**
 - Almost always work as advertised!
 - Examples:
 - (Iteration & Arrays),
 - (Sharing & Indirection)
 - (Program Structuring in C)

Software Design – Templates

- Medium level templates:
 - Not so fine-grained;
 - Adaptations often required for specific situation/problem.
 - May lead to minor problems when mis-applied.
 - a.k.a **(DESIGN) PATTERNS**
 - Examples:
 - ([Browser – Composite Pattern](#))
 - ([Set Enumeration – Iterator Pattern](#))

Software Design – Templates

- High level templates:
 - Coarse-grained.
 - Often require lot of customization and implementation to achieve desired result.
 - Often Specific to problem domain.
 - When mis-applied lead to disastrous results!
 - a.k.a **(ARCHITECTURAL) FRAMEWORKS**
 - Examples:
 - **(Document Publishing – Pipe & Filter Architecture)**

Patterns vs. Frameworks

- Frameworks are partially completed software systems that may be targeted at a specified type of application
- However patterns
 - are more abstract and general than frameworks
 - cannot be directly implemented in a particular software environment
 - are more primitive than frameworks

((Software) Design) Patterns

- Definition:
 - Contextual Design solutions of medium granularity that can be re-used in recurring design problems with minimal adaptation.
- Usage:
 - Documentation of (Expert) Knowledge of Design solutions.
- Purpose:
 - Communication and Persistence for Re-use.

((Software) Design) Patterns

- Why study patterns?
 - “to organize implicit knowledge about how people solve recurring problems when they go about building things”
 - **Christopher Alexander**,
(Architect by profession;
Father of modern study of patterns)

Design Patterns [1]

- A **solution** to a **problem** that occurs repeatedly in a variety of contexts.
- Each pattern has a **name**.
- Use of each pattern has **consequences**.

Design Patterns [2]

- Generally at a “higher level” of abstraction.
- *Not* about designs such as **linked lists** or **hash tables**.
- Generally descriptions of **communicating objects** and **classes**.

Key Principles

- Key principles that underlie patterns
 - abstraction
 - encapsulation
 - information hiding
 - modularization
 - separation of concerns

Key Principles

- Coupling and cohesion
- Sufficiency, Completeness and primitiveness
- Separation of policy and implementation
- Separation of interface and implementation
- Single point of reference
- Divide and conquer

Buschmann et al. (1996)

Patterns Documentation

- Name
- Classification
- Motivation
 - Problem Scenario
- Applicability
 - Situations
- Structure
 - Graphical representation
 - Often just the classes in the pattern
- Participants
- Collaborations
 - How participants collaborate to carry out responsibilities?
- Implementation
- Consequences
- Related Patterns

Patterns Classification

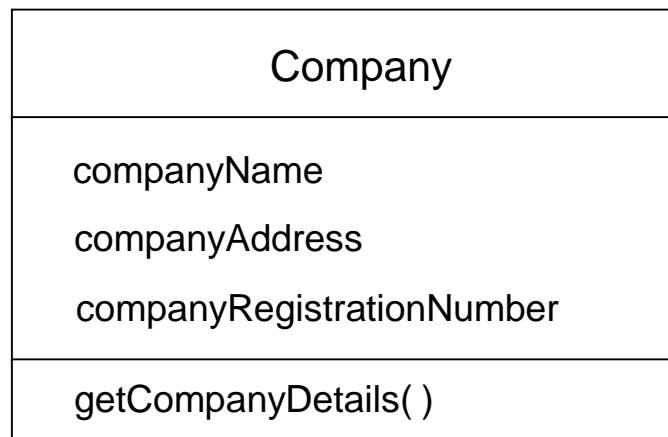
- Basic Classification
 - by Purpose
 - Creational vs. Structural vs. Behavioral
 - by Scope
 - Object vs. Class
- Domain Level Classification:
 - Patterns in Telecom Software
 - Patterns in Distributed Computing

Creational Patterns

- Concerned with the construction of object instances
- Separate the operation of an application from how its objects are created
- Gives the designer considerable flexibility in configuring all aspects of object creation

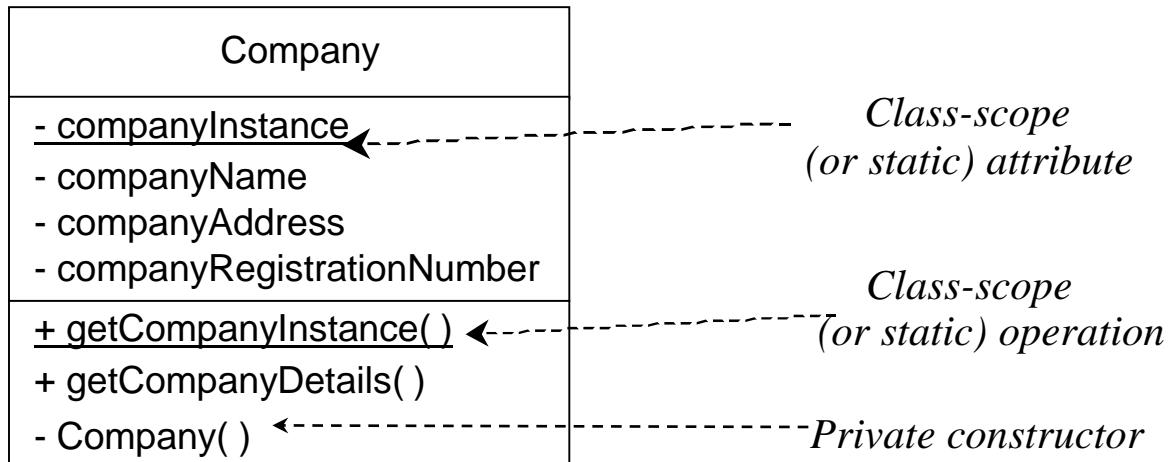
Creational Patterns: Singleton

- How does one ensure that only one instance of the company class is created?



Creational Patterns: Singleton

- Solution—restrict access to the constructor!



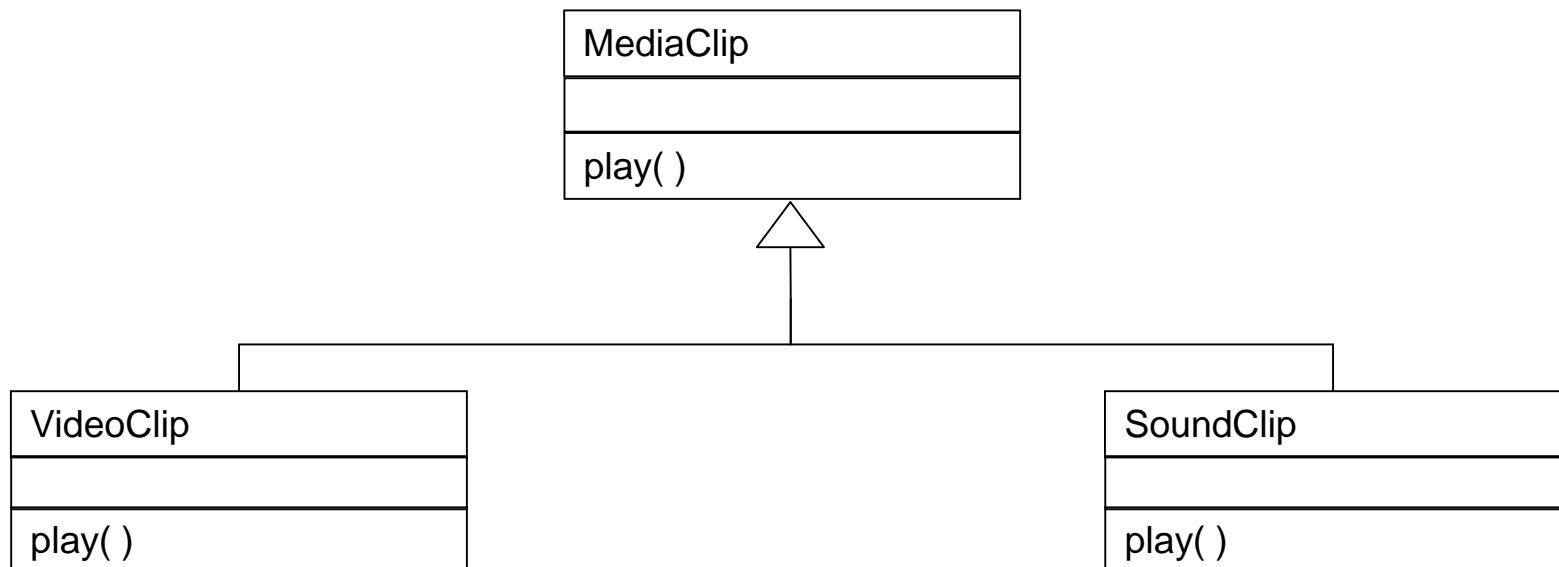
The use of class-scope operations allows global access

Structural Patterns

- Concerned with the way in which classes and objects are organized
- Offer effective ways of using object-oriented constructs such as inheritance, aggregation and composition to satisfy particular requirements

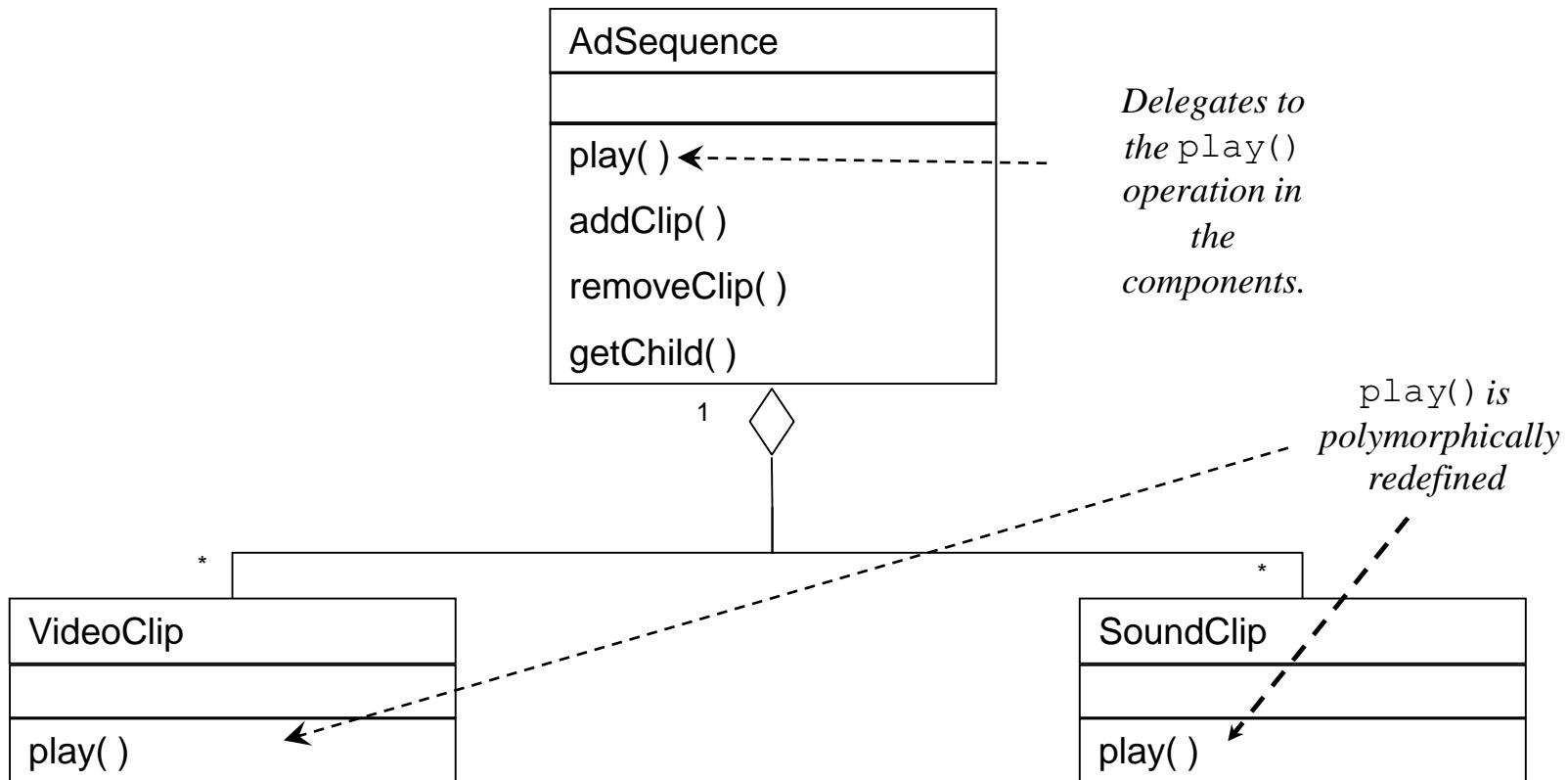
Structural Patterns: Composite

- How can we present the same interface for a media clip whether it is composite or not?

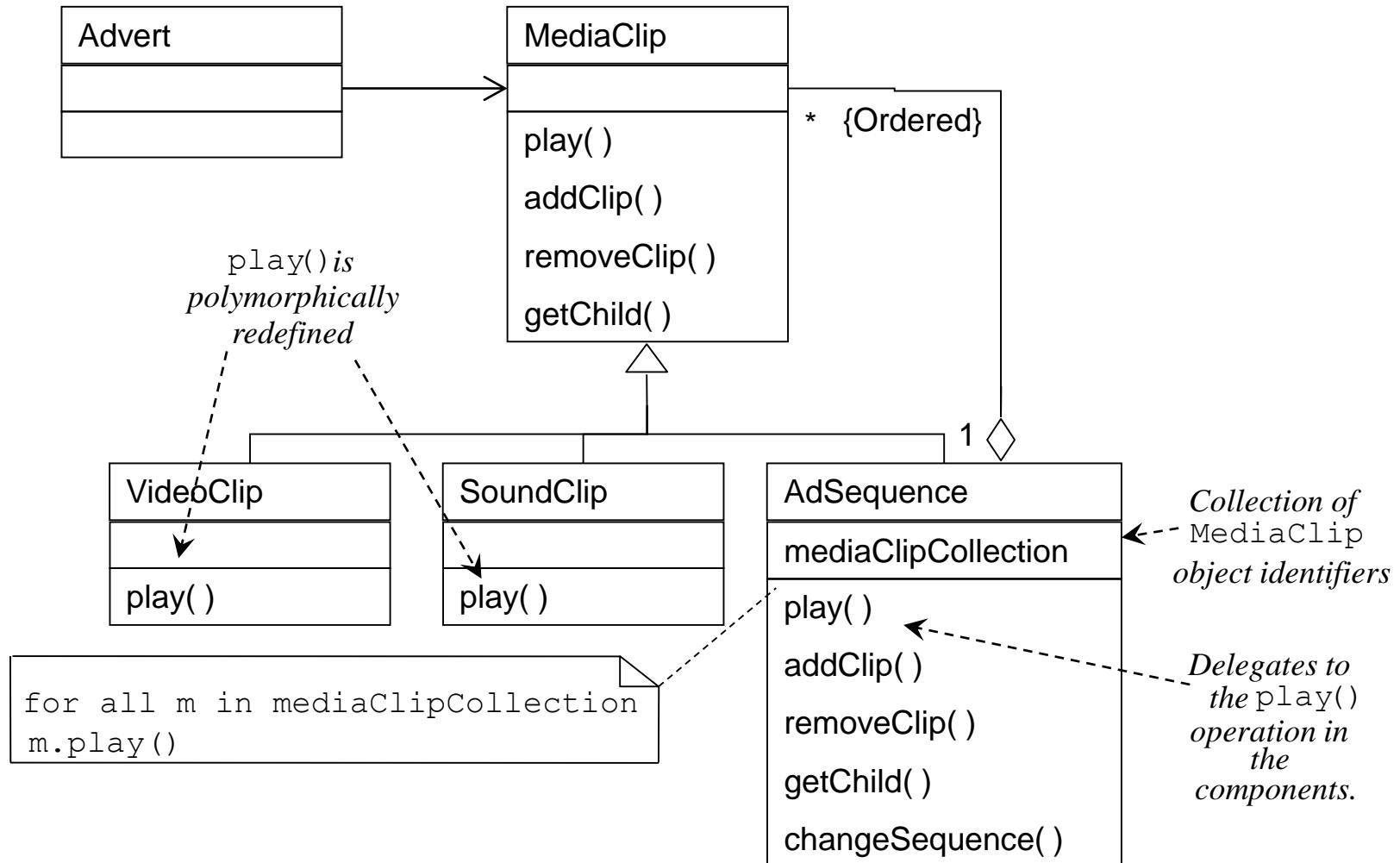


Structural Patterns: Composite

How can we incorporate composite structures?

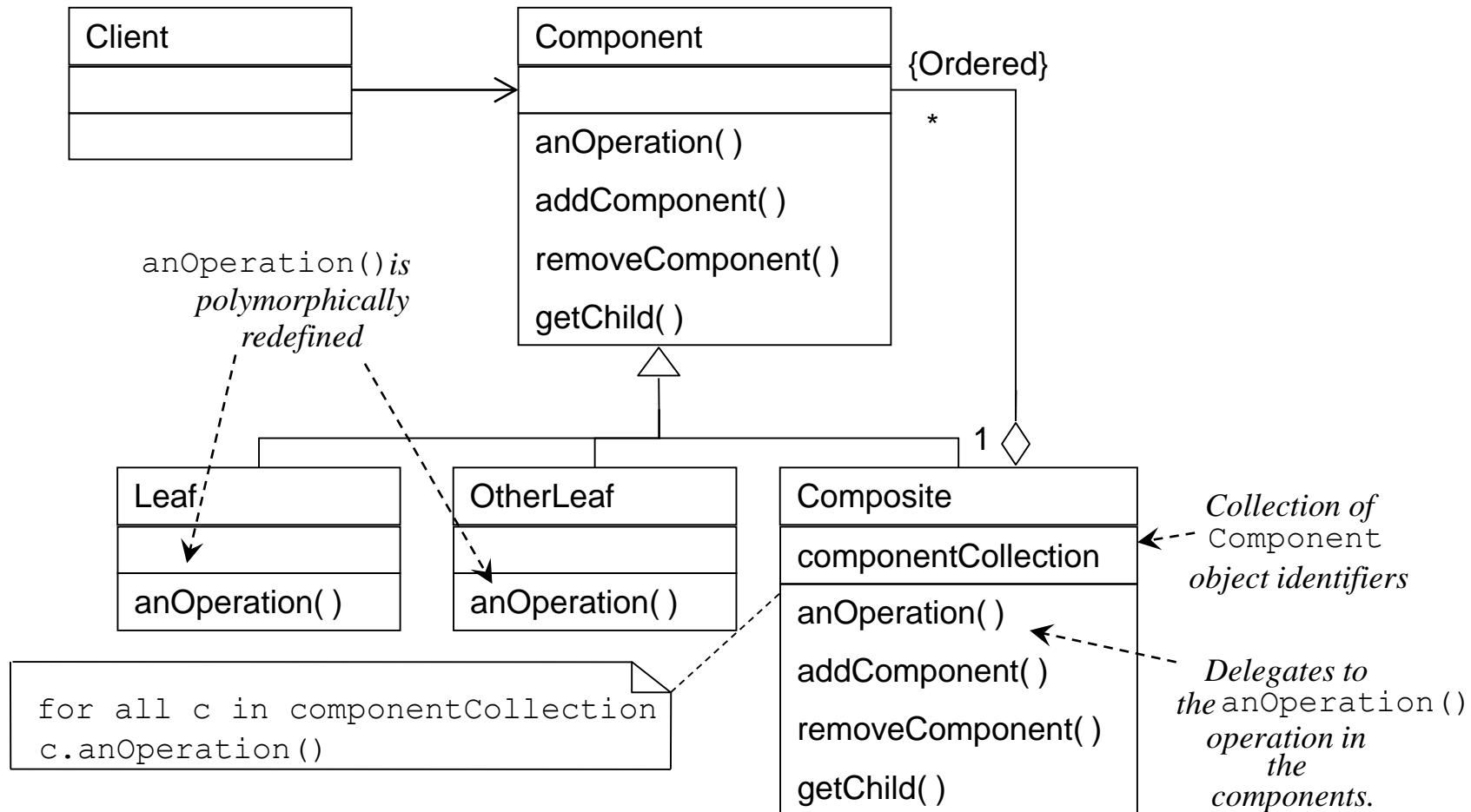


Composite applied to Agate



Composite Pattern

General Form



Behavioural Patterns

- Address the problems that arise when assigning responsibilities to classes and when designing algorithms
- Suggest particular static relationships between objects and classes and also describe how the objects communicate

Patterns

- Software context:
 - Most of the work originated in SmallTalk community.
 - SmallTalk was a programming language:
 - Object Oriented
 - Meant for rapid prototyping
 - Came with (what are known today as) IDE and API
 - IDE elements were in SmallTalk!
 - (Concrete) Precursor to modeling, frameworks and patterns!

Patterns – MVC framework

- SmallTalk's user interface framework
 - Model - refers to data model
 - View – refers to external views or presentation of data.
 - Controller – refers to module relating reactions of view or presentation to changes in data.

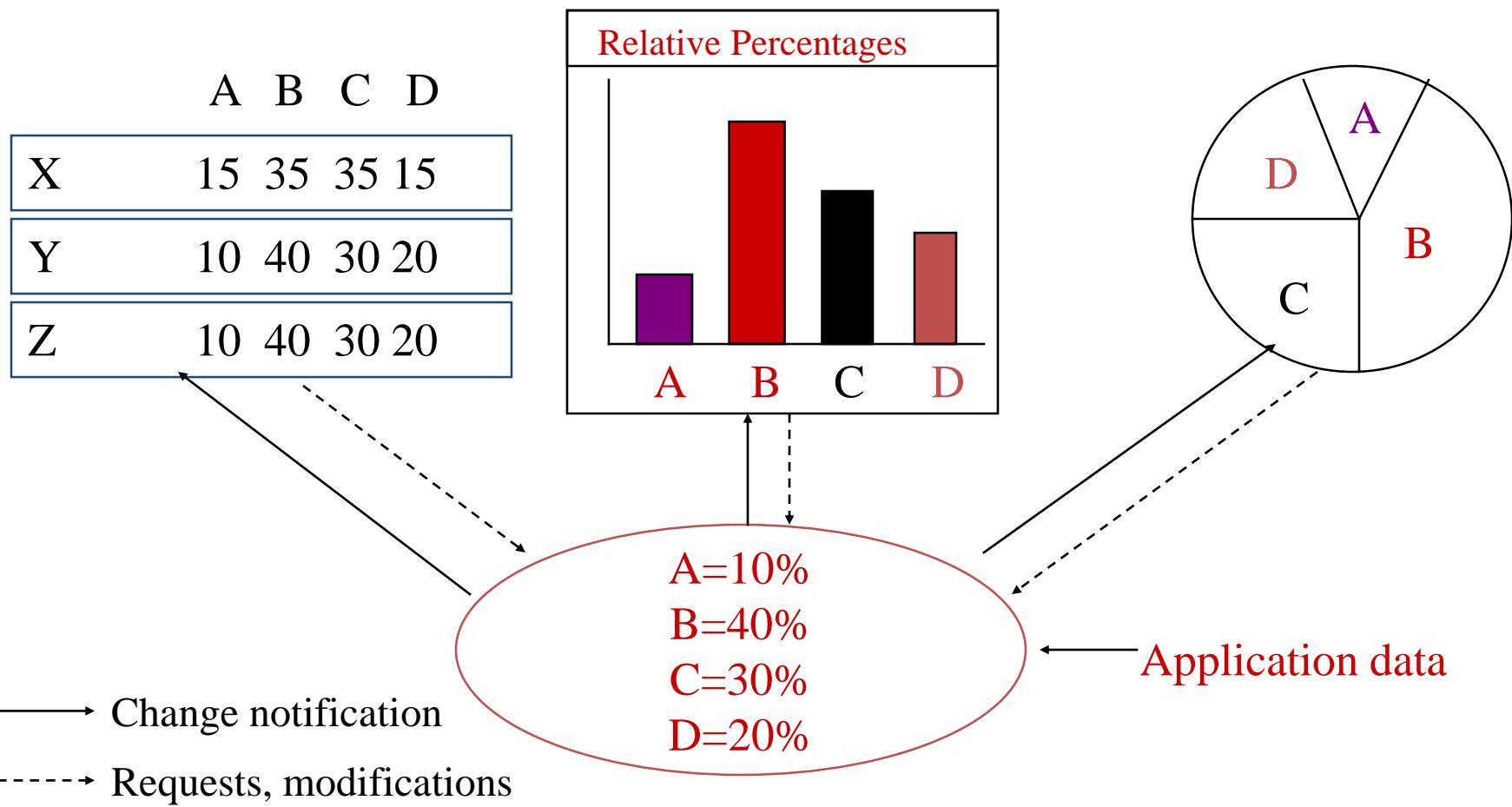
Observer Pattern [1]

- Need to **separate** presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be **reused**.
- **Change** in one view automatically **reflected** in other views. Also, change in the application data is reflected in all views.
- Defines **one-to-many dependency** amongst objects so that when one object changes its state, all its dependents are notified.

Observer Pattern – Classification & Applicability

- A behavioral (object) pattern:
 - Concerns objects and their behavior.
- Applicability
 - Vary and reuse 2 different abstractions independently.
 - Change to one object requires change in (one or more) other objects – whose identity is not necessarily known

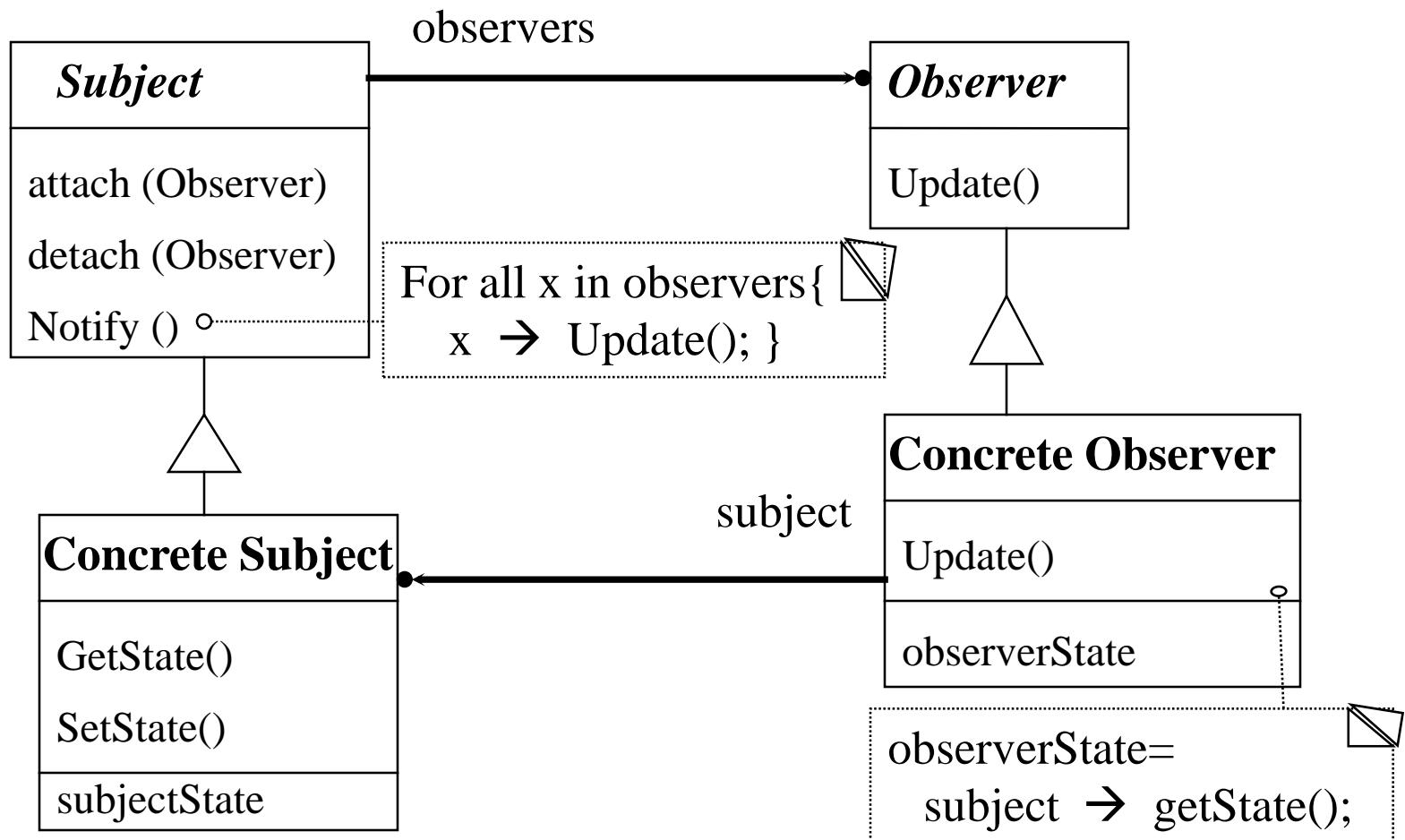
Observer Pattern [2]



Observer Pattern

- Model – View paradigm can be generalized:
 - A view is an observer
 - A model is an subject that is observed.
 - A subject may have any number of observers.
 - Referred to as the observer pattern, also Publish-Subscribe or Dependents

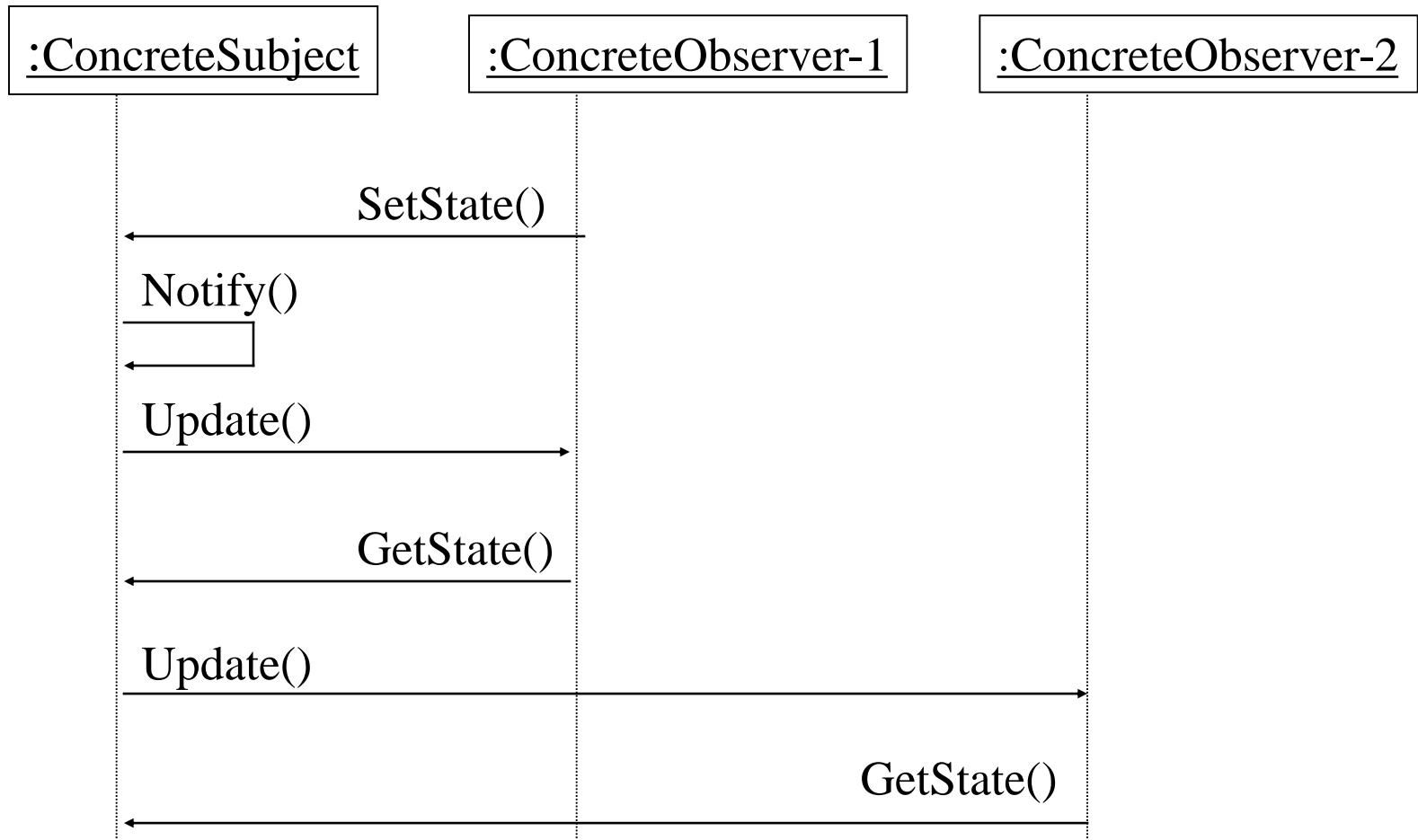
Observer Pattern [3]



Observer Pattern - Participants

- **Subject**
 - Has a list of observers;
 - interfaces for attaching/detaching an observer
- **Observer**
 - An updating interface for objects that gets notified of changes in a subject.
- **ConcreteSubject**
 - Stores state of interest to observers
 - Sends notification when state changes.
- **ConcreteObserver**
 - Implements updating interface.

Class collaboration in Observer



Observer Pattern - Implementation

interface

```
Observer {  
    void update (Observable sub, Object arg)  
}
```

Java terminology for Subject.

```
class Observable {  
    public void addObserver(Observer o) {}  
    public void deleteObserver (Observer o) {}  
    public void notifyObservers(Object arg) {}  
  
    public boolean hasChanged() {}  
    ...  
}
```

Observer Pattern - Implementation

```
public PiChartView implements Observer {  
    void update(Observable sub, Object arg) {  
        // repaint the pi-chart  
    }  
}  
  
class StatsTable extends Observable {  
    public boolean hasChanged() {  
        // override to decide when it is considered changed  
    }  
}
```

A Concrete Observer.

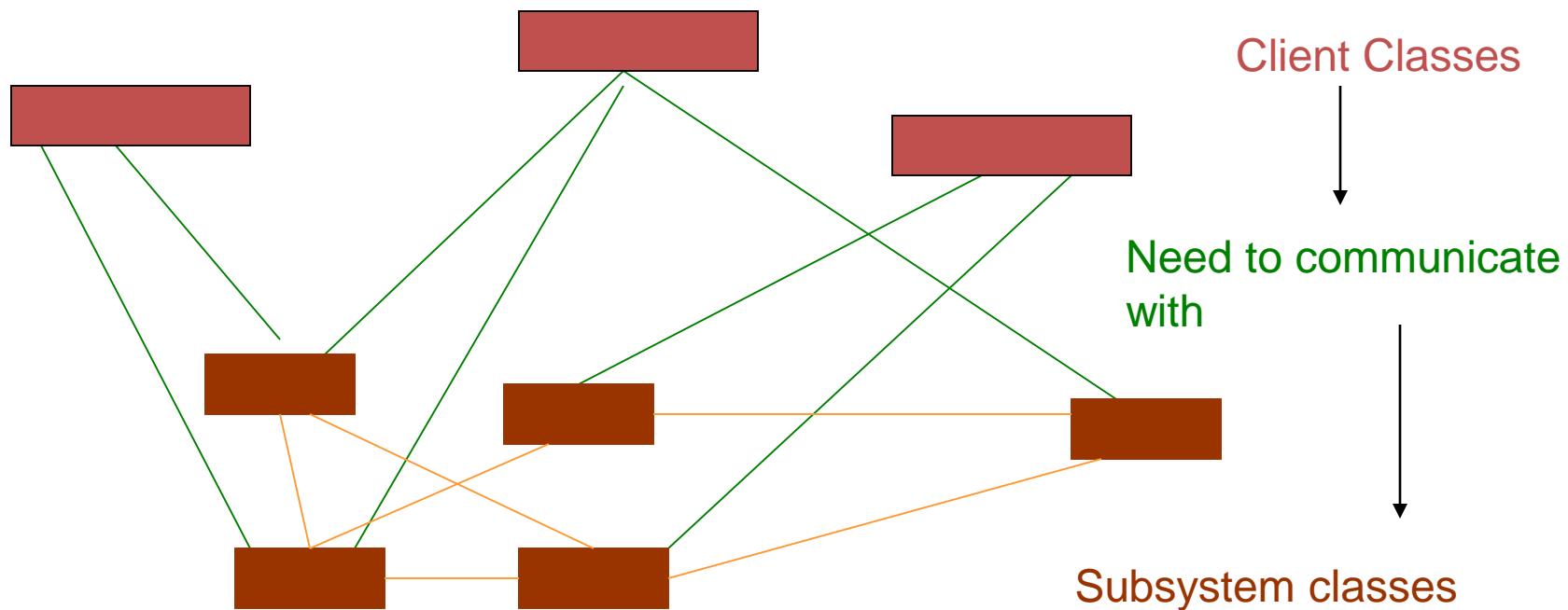
When to use the Observer Pattern?

- *When* an abstraction has **two aspects**: one dependent on the other. Encapsulating these aspects in separate objects allows one to **vary** and **reuse** them independently.
- *When* a change to one object requires changing others and the number of objects to be changed is **not known**.
- *When* an object should be able to notify others **without knowing** who they are. Avoid tight coupling between objects.

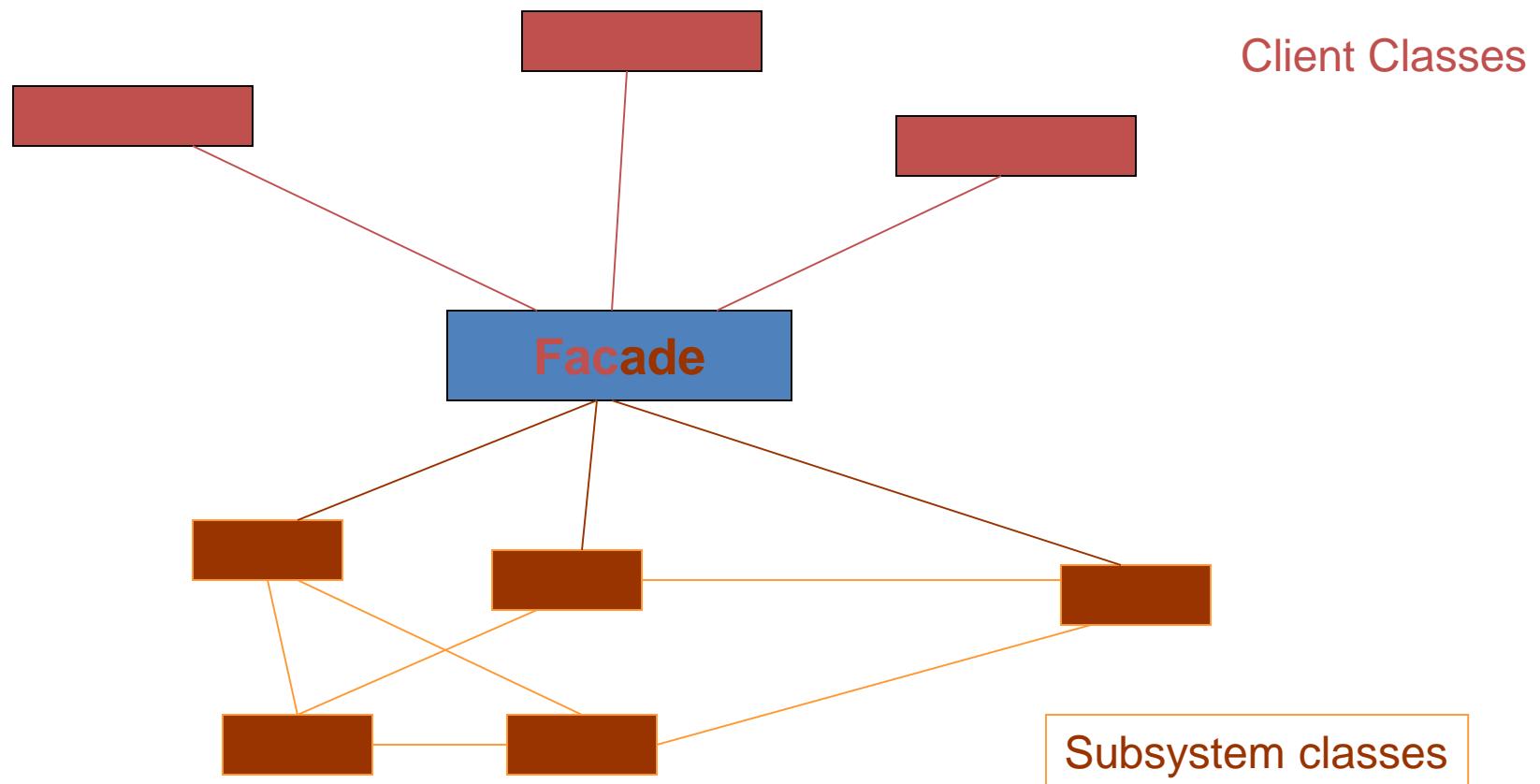
Observer Pattern: Consequences

- *Abstract coupling* between subject and observer. Subject has no knowledge of concrete observer classes.
(What design principle is used?)
- *Support for broadcast communication.* A subject need not specify the receivers; all interested objects receive the notification.
- *Unexpected updates:* Observers need not be concerned about when the updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

Facade Pattern: Problem



Facade Pattern: Solution



Facade Pattern: Why and What?

- Subsystems often get complex as they evolve.
- Need to provide a **simple interface** to many, often small, classes.

But not necessarily to ALL classes of the subsystem.

- Façade provides a simple default view good enough for most clients.
- Facade **decouples** a subsystem from its clients.
- A façade can be a single entry point to each subsystem level. This allows layering.

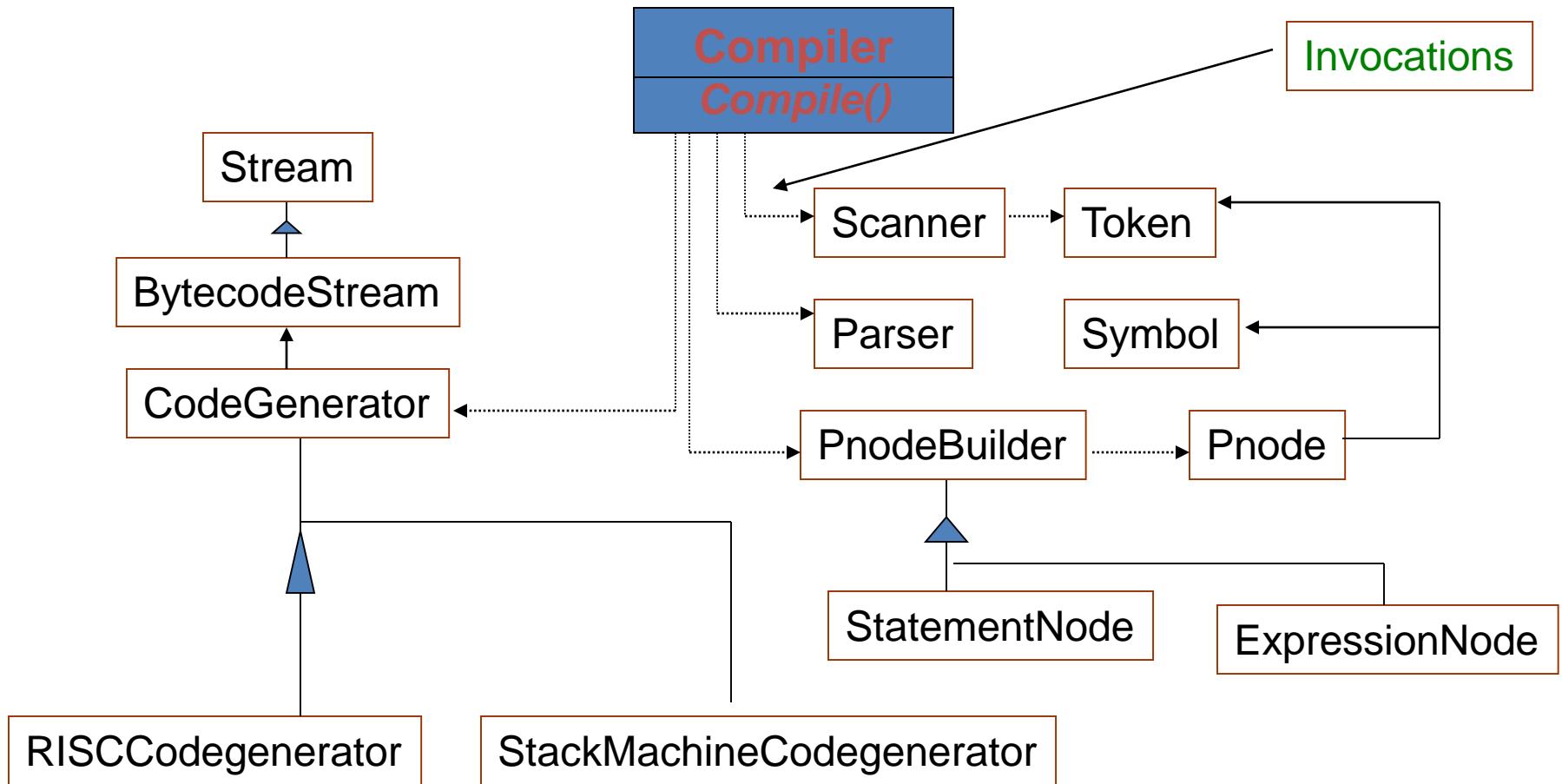
Facade Pattern: Participants and Communication

- Participants: Façade and subsystem classes
- Clients communicate with subsystem classes by sending requests to façade.
- Façade forwards requests to the appropriate subsystem classes.
- Clients do not have direct access to subsystem classes.

Facade Pattern: Benefits

- Shields clients from subsystem classes; reduces the number of objects that clients deal with.
- Promotes weak coupling between subsystem and its clients.
- Helps in layering the system. Helps eliminate circular dependencies.

Example: A compiler



Façade Pattern: Code [1]

```
class Scanner { // Takes a stream of characters and produces a stream of tokens.

public:
    Scanner (istream&);

    virtual Scanner();
    virtual Token& Scan();

private:
    istream& _inputStream;
};
```

Façade Pattern: Code [2]

```
class parser { // Builds a parse tree from tokens using the PNodeBuilder.

public:
    Parser ();
    virtual ~Parser()
    virtual void Parse (Scanner&, PNodeBuilder&);

};
```

Façade Pattern: Code [3]

```
class Pnodebuilder {           // Builds a parse tree incrementally. Parse tree
    public:                   // consists of Pnode objects.

        Pnodebuilder ();

        virtual Pnode* NewVariable ( // Node for a variable.

            Char* variableName
        ) const;

        virtual Pnode* NewAssignment ( // Node for an assignment.

            Pnode* variable, Pnode* expression
        ) const;

    Private:                  // Similarly...more nodes.

        Pnode* _node;

    };
}
```

Façade Pattern: Code [4]

```
class Pnode { // An interface to manipulate the program node and its children.

public:
    // Manipulate program node.

    virtual void GetSourcePosition (int& line, int& index);

    // Manipulate child node.

    virtual void Add (Pnode* );
    virtual void Remove (Pnode* );
    // .....

    virtual void traverse (Codegenerator&); // Traverse tree to generate code.

protected:
    PNode();
};

};
```

Façade Pattern: Code [5]

```
class CodeGenerator {           // Generate bytecode.  
public:  
    // Manipulate program node.  
    virtual void Visit (StatementNode*);  
    virtual void Visit (ExpressionNode*);  
    // ....  
Protected:  
    CodeGenerator (BytecodeStream&);  
    BytecodeStream& _output;  
};
```

Façade Pattern: Code [6]

```
void ExpressionNode::Traverse (CodeGenerator& cg) {
```

```
    cg.Visit (this);
```

```
    ListIterator<Pnode*> i(_children);
```

```
    For (i.First(); !i.IsDone(); i.Next()){
```

```
        i.CurrentItem()→Traverse(cg);
```

```
    };
```

```
};
```

Façade Pattern: Code [7]

```
class Compiler {           // Façade. Offers a simple interface to compile and
public:                   // Generate code.
    Compiler();           ← Could also take a CodeGenerator
    virtual void Compile (istream&, BytecodeStream&);    Parameter for increased generality.
}
void Compiler:: Compile (istream& input, BytecodeStream& output) {
    Scanner scanner (input);
    PnodeBuilder builder;
    Parser parser;
    parser.Parse (scanner, builder);
    RISCCCodeGenerator generator (output);
    Pnode* parseTree = builder.GetRootNode();
    parseTree→Traverse (generator);
}
```

Facade Pattern: Another Example from POS [1]

- Assume that rules are desired to invalidate an action:
 - Suppose that when a new Sale is created, it will be paid by a gift certificate
 - Only one item can be purchased using a gift certificate.
 - Hence, subsequent **enterItem** operations must be invalidated in some cases. (Which ones?)

How does a designer factor out the handling of such rules?

Facade Pattern: Another Example [2]

- Define a “rule engine” subsystem (e.g. **POSRuleEngineFacade**).
- It evaluates a set of rules against an operation and indicates if the rule has invalidated an operation.
- Calls to this façade are placed near the start of the methods that need to be validated.
 - Example: Invoke the façade to check if a new **salesLineItem** created by **makeLineItem** is valid or not. (See page 370 of Larman.)



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.11

Today's Agenda

- Object Oriented Design
- GoF Patterns

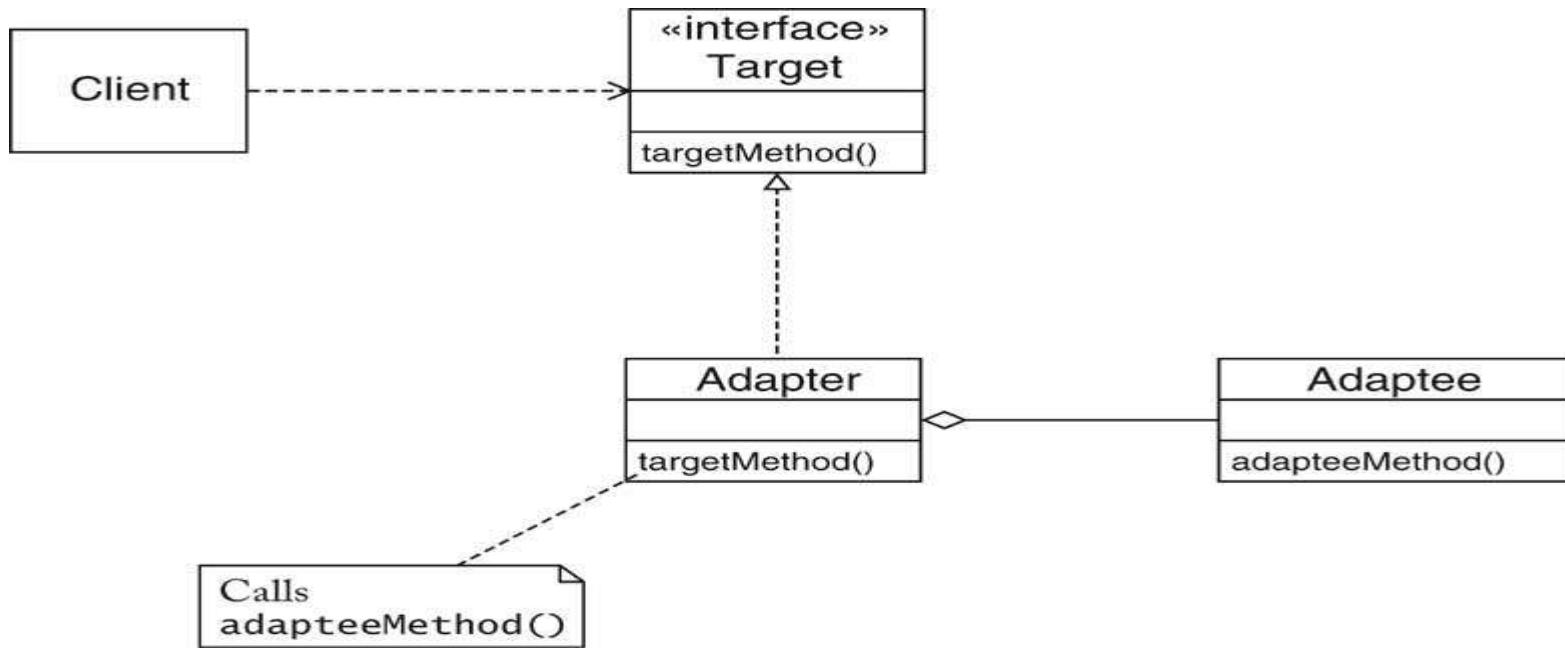
Adapter Pattern

Context

- You want to use an existing class (adaptee) without modifying it.
- The context in which you want to use the class requires target interface that is different from that of the adaptee.
- The target interface and the adaptee interface are conceptually related.

Solution

1. Define an adapter class that implements the target interface.
2. The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods.
3. The client wraps the adaptee into an adapter class object



Name in Design Pattern	Actual Name (Icon->Component)
Adaptee	Icon
Target	JComponent
Adapter	IconAdapter
Client	The class that wants to add icons into a container
targetMethod()	paintComponent(), getPreferredSize()
adapteeMethod()	paintIcon(), getIconWidth(), getIconHeight()

```
public class IconAdapter extends JComponent
{
    private Icon icon; // Make icon a part of Iconadapter
    public IconAdapter(Icon icon)
    {
        this.icon = icon;
    }
    public void paintComponent(Graphics g) // Overriding the method
    {
        icon.paintIcon(this, g, 0, 0); // calling method of icon
    }
    public Dimension getPreferredSize() // Overriding the method
    {
        return new Dimension(icon.getIconWidth(),
            icon.getIconHeight());
    }
}
```

```
public class IconAdapterTester
{
    public static void main(String[] args)
    {
        Icon icon = new CarlIcon(300);
        JComponent component = new IconAdapter(icon);
        JFrame frame = new JFrame();
        frame.add(component, BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Scroll Bars

1. Scroll Bars can be added to components to display more information that can be shown on screen.

2. Adding a Scroll bar

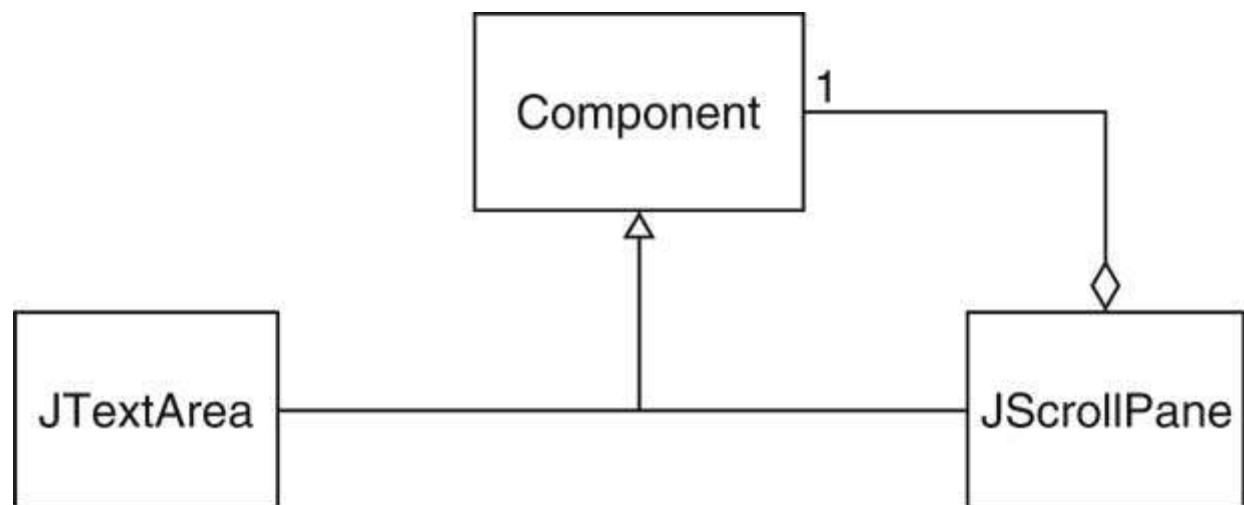
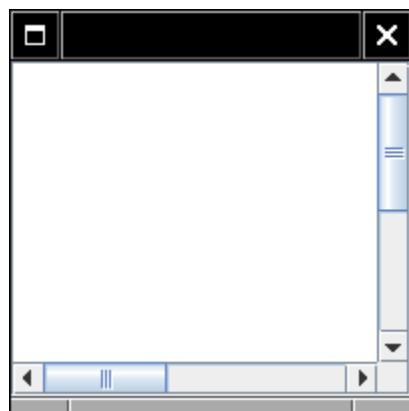
```
JTextArea area = new JTextArea(10,25);
```

```
JScrollPane scroller = new JScrollPane(area);
```

3. Scroll bars adds/enhances functionality to the underlying component.

4. Adding/Enhancing Functionality is called Decoration.

5. JScrollPane Decorates a Component and is again a Component.



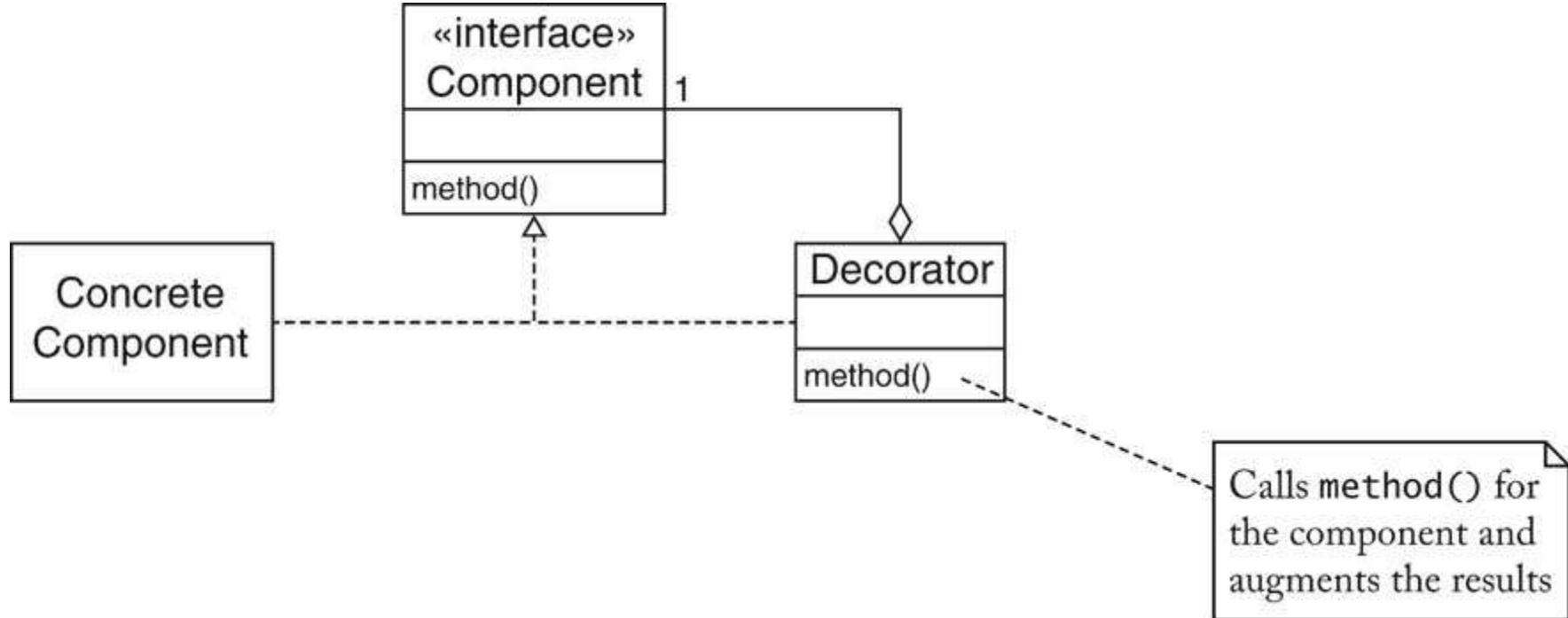
Context

Decorator Pattern

1. Component objects can be decorated (visually or behaviorally enhanced)
2. The decorated object can be used in the same way as the undecorated object
3. The component class does not want to take on the responsibility of the decoration
4. There may be an open-ended set of possible decorations

Solution

1. Define an interface type that is an abstraction for the component
2. Concrete component classes realize this interface type.
3. Decorator classes also realize this interface type.
4. A decorator object manages the component object that it decorates
5. When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.



Name in Design Pattern	Actual Name (scroll bars)
Component	Component
ConcreteComponent	JTextArea
Decorator	JScrollPane
method()	a method of Component (e.g. <code>paint</code>)

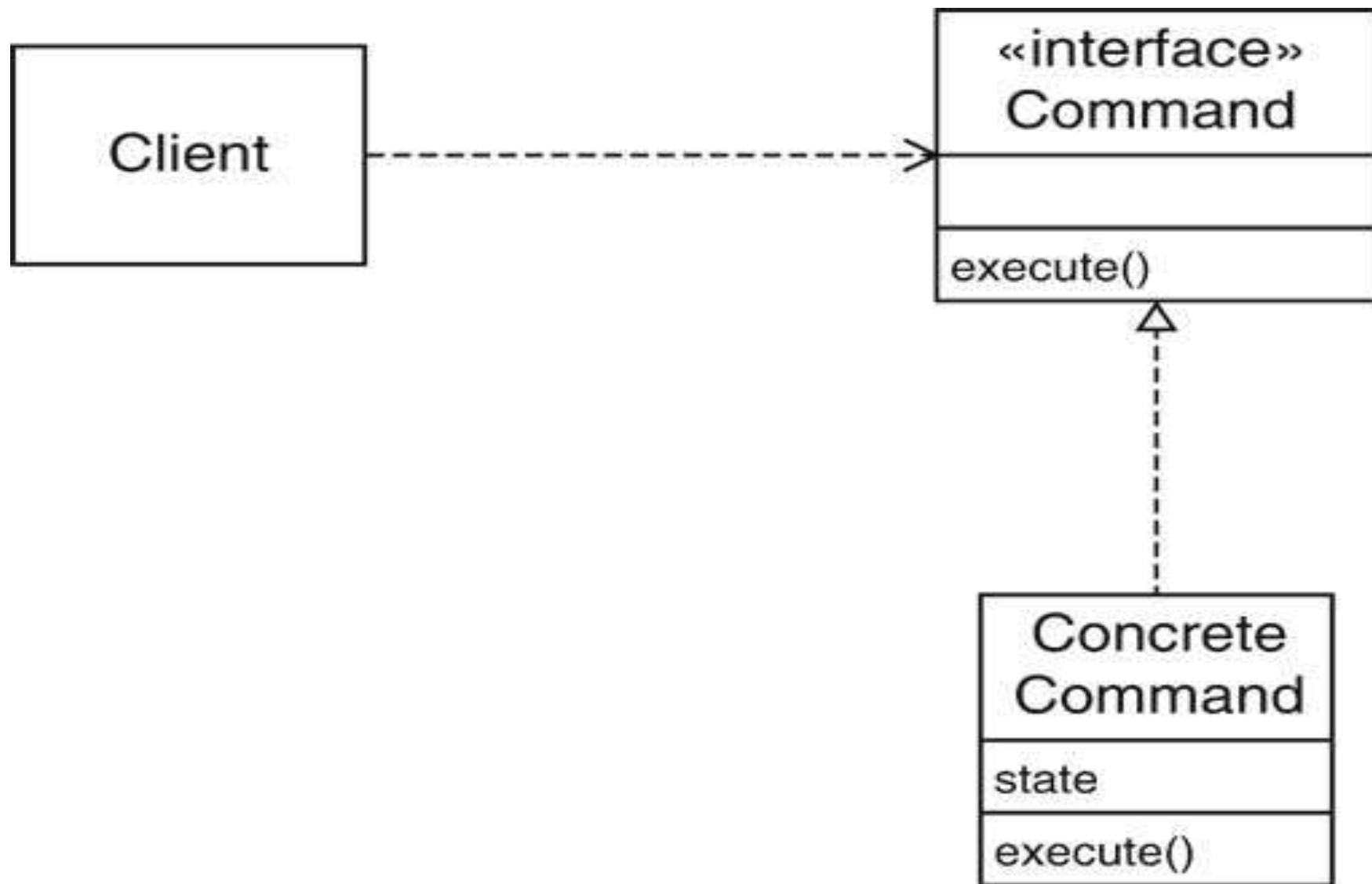
The **COMMAND** Pattern

Context

- You want to implement commands that behave like objects
 - (i) because you need to store additional information with commands
 - (ii) because you want to collect commands

Solution

1. Define a command interface type with a method to execute the command.
2. Supply methods in the command interface type to manipulate the state of command objects.
3. Each concrete command class implements the command interface type.
4. To invoke the command, call the execute method.



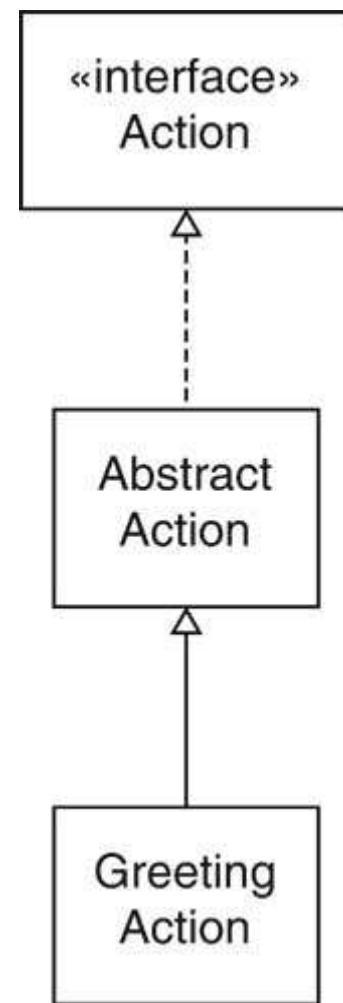
Action Interface

Action interface lets the user to implement commands

Extends ActionListener. Stores Name of the action and an icon

Important Methods :

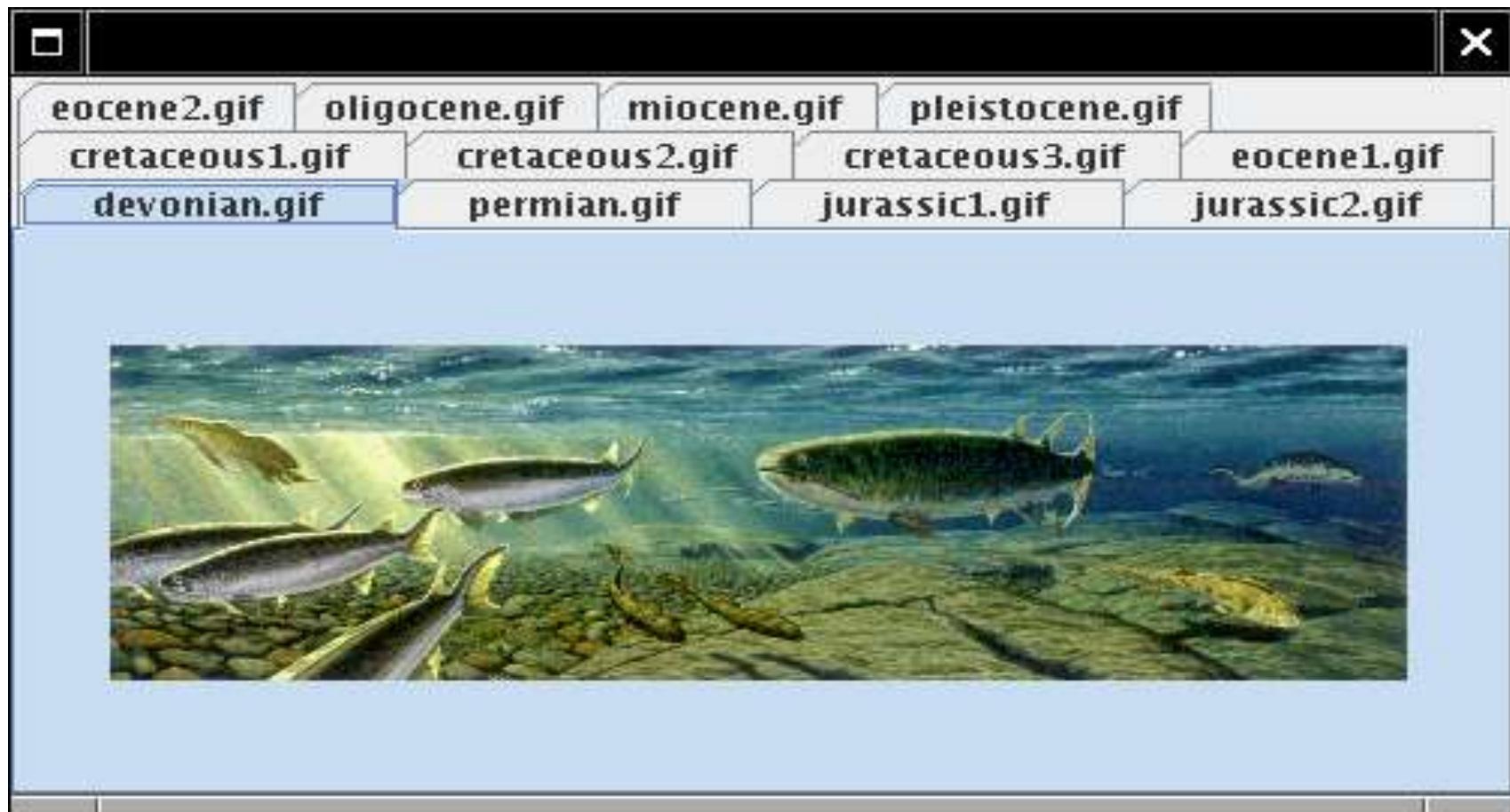
Method Summary	
void	addPropertyChangeListener(PropertyChangeListener listener) Adds a PropertyChange listener.
Object	getValue(String key) Gets one of this object's properties using the associated key.
boolean	isEnabled() Returns the enabled state of the Action.
void	putValue(String key, Object value) Sets one of this object's properties using the associated key.
void	removePropertyChangeListener(PropertyChangeListener listener) Removes a PropertyChange listener.
void	setEnabled(boolean b) Sets the enabled state of the Action.



Proxies

- 1. Proxy: a person who is authorized to act on another person's behalf**
- 2. Example: Delay instantiation of object**
- 3. Expensive to load image**
- 4. Not necessary to load image that user doesn't look at**
- 5. Proxy defers loading until user clicks on tab**

Deferred Image Loading



1. Normally, programmer uses image for label:

```
JLabel label = new JLabel(new ImageIcon(imageName));
```

2. Use proxy instead:

```
JLabel label = new JLabel(new ImageProxy(imageName));
```

3. paintIcon loads image if not previously loaded

```
public void paintIcon(Component c, Graphics g, int x, int y)
{
    if (image == null) image = new ImageIcon(name);
    image.paintIcon(c, g, x, y);
}
```

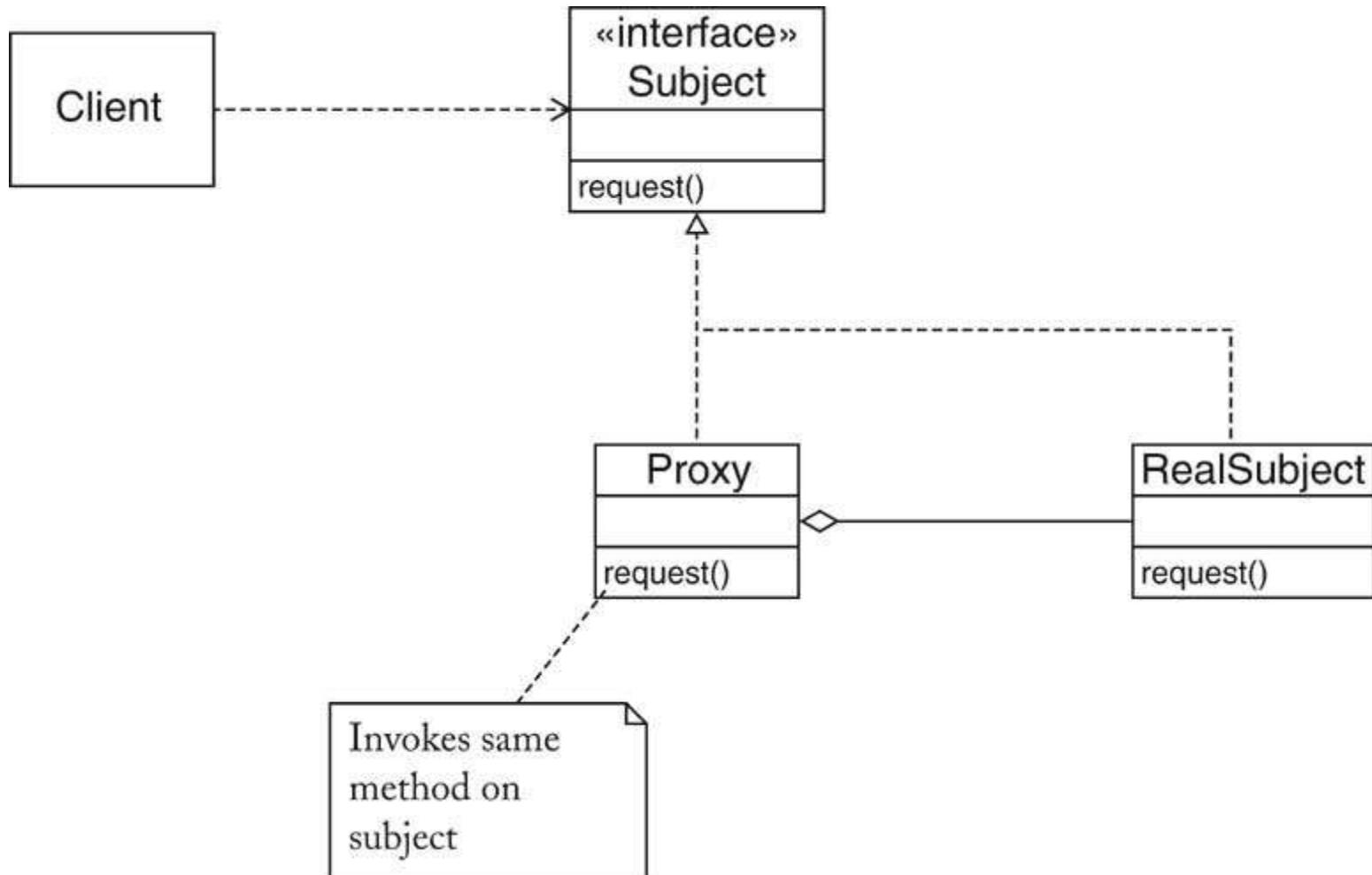
4. Another use for proxies: remote method invocation

Context

1. A class (the real subject) provides a service that is specified by an interface type (the subject type)
2. There is a need to modify the service in order to make it more versatile.
3. Neither the client nor the real subject should be affected by the modification.

Solution

1. Define a proxy class that implements the subject interface type. The proxy holds a reference to the real subject, or otherwise knows how to locate it.
2. The client uses a proxy object.
3. Each proxy method invokes the same method on the real subject and provides the necessary modifications.



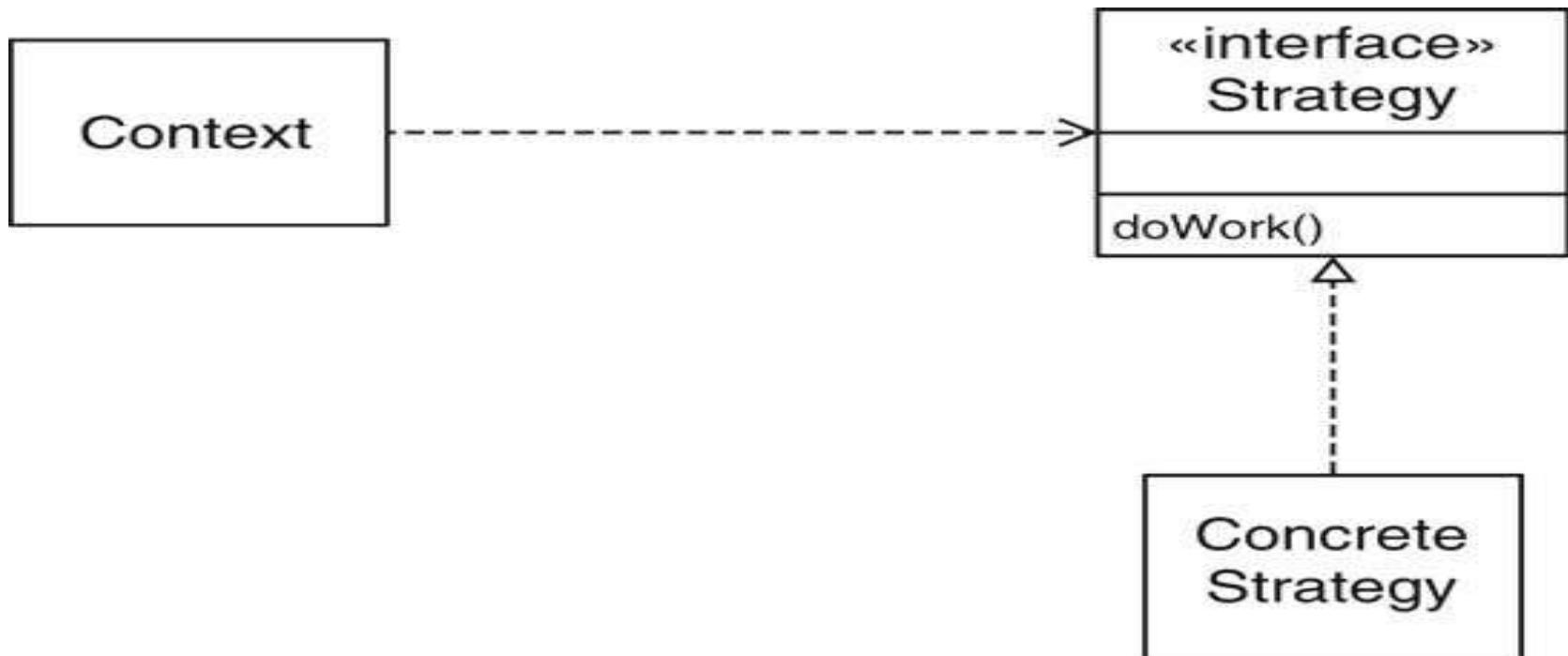
Strategy Pattern

Context:

- A class can benefit from different variants for an algorithm
- Clients sometimes want to replace standard algorithms with custom versions

Solution

- Define an interface type that is an abstraction for the algorithm
- Actual strategy classes realize this interface type.
- Clients can supply strategy objects
- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object



Example : Sorting

Name in Design Pattern	Actual Name (sorting)
Context	Collections
Strategy	Comparator
ConcreteStrategy	a class that implements Comparator
doWork()	compare

Example : Layout Managers

Name in Design Pattern	Actual Name (layout management)
Context	Container
Strategy	LayoutManager
ConcreteStrategy	a layout manager such as BorderLayout
doWork()	a method such as layoutContainer

Some Typical Applications where Strategy can be helpful

- Saving Files in Different Formats
`file.setFormat(<Name of Format>);`
`file.save();`
- Compress Files using different algorithms
- Various Encryption/Decryption Schemes

Factory

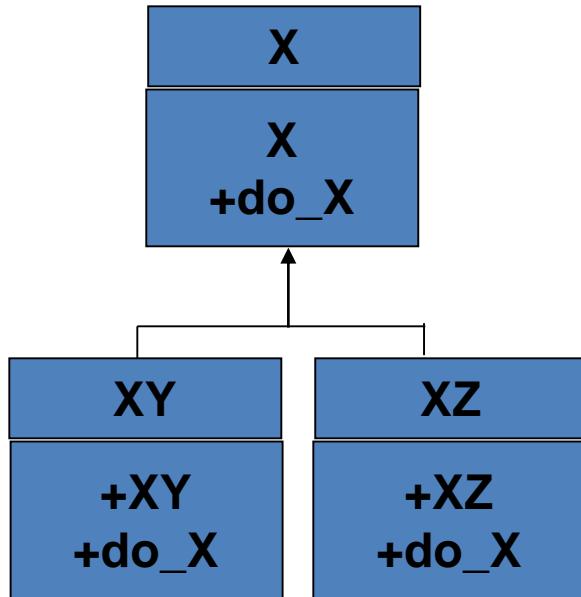
- Context/Problem
 - Who should be responsible for creating objects when there are special considerations, such as complex logic, a desire to separate the creation responsibilities for better cohesion, and so forth
- Solution
 - Create a Pure Fabrication object called a Factory that handles the creation

EXAMPLE

- Let's suppose an application asks for entering the name and sex of a person.
 - If the sex is Male (M)
 - it displays welcome message saying Hello Mr. <Name>
 - if the sex is Female (F)
 - it displays message saying Hello Ms <Name>

Factory Pattern

- Factory Pattern returns an instance of one of several possible classes depending upon the data provided to it.
- Generally all classes that it returns have a common parent class and common methods



Produces different instance of X depending upon the value of arguments

X getClass(int){ };

```
class Namer
{
String last;
String first;
public String getFirst()
{ return first; }
public String getLast()
{ return last; }
}
```

```
class FirstFirst extends Namer
{
FirstFirst(String s)
{
int i = s.lastIndexOf(" ");
if( i > 0)
{
first = s.substring(0,i).trim();
last = s.substring(i+1).trim();
}
else
{
first = "";
last = s;
}
}
```

```
class LastFirst extends Namer
{
    LastFirst(String s)
    {
        int i = s.lastIndexOf(",");
        if( i > 0)
        {
            last = s.substring(0,i).trim();
            first= s.substring(i+1).trim();
        }
        else
        {
            first = "";
            last = s;
        }
    }
}
```

```
class NameFactory
{
    Namer namer;
    Namer getNamer(String s)
    {
        int i = s.indexOf(",");
        if(i > 0) return new LastFirst(s);
        else
            return new FirstFirst(s);
    }
}
```

Advantages of Factory Objects

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

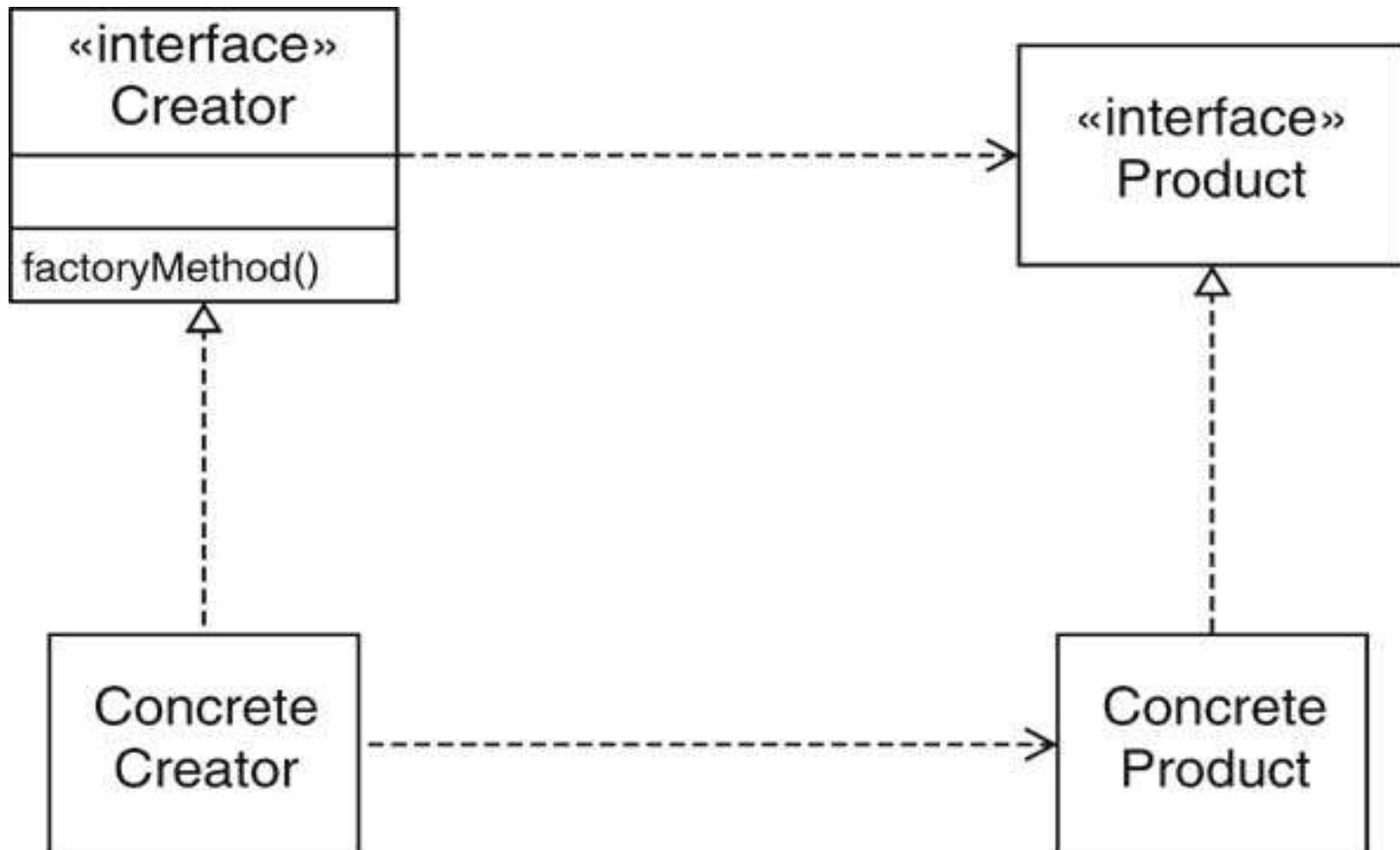
The FACTORY METHOD Pattern

Context

- A type (the creator) creates objects of another type (the product).
- Subclasses of the creator type need to create different kinds of product objects.
- Clients do not need to know the exact type of product objects.

Solution

1. Define a creator type that expresses the commonality of all creators.
2. Define a product type that expresses the commonality of all products.
3. Define a method, called the factory method, in the creator type. The factory method yields a product object.
4. Each concrete creator class implements the factory method so that it returns an object of a concrete product class.



Name in Design Pattern	Actual Name (iterator)
Creator	Collection
ConcreteCreator	A subclass of Collection
factoryMethod()	iterator()
Product	Iterator
ConcreteProduct	A subclass of Iterator (which is often anonymous)

Abstract Factory - Intent

- “Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”
 - provide a simple creational interface for a complex family of classes
 - Client does not have to know any of those details.
 - avoid naming concrete classes
 - Clients use abstract creational interfaces and abstract product interfaces. Concrete classes can be changed without affecting clients.
 - Clients can stay blissfully unaware of implementation details
 - This is critically important!

Motivating Examples

- Provide different “look and feel” without affecting client -- see motif, presentation manager example in text.
- Design neural network layers to contain the technology for their own construction, allowing networks to focus on learning strategies.

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- The *new* operator considered harmful.

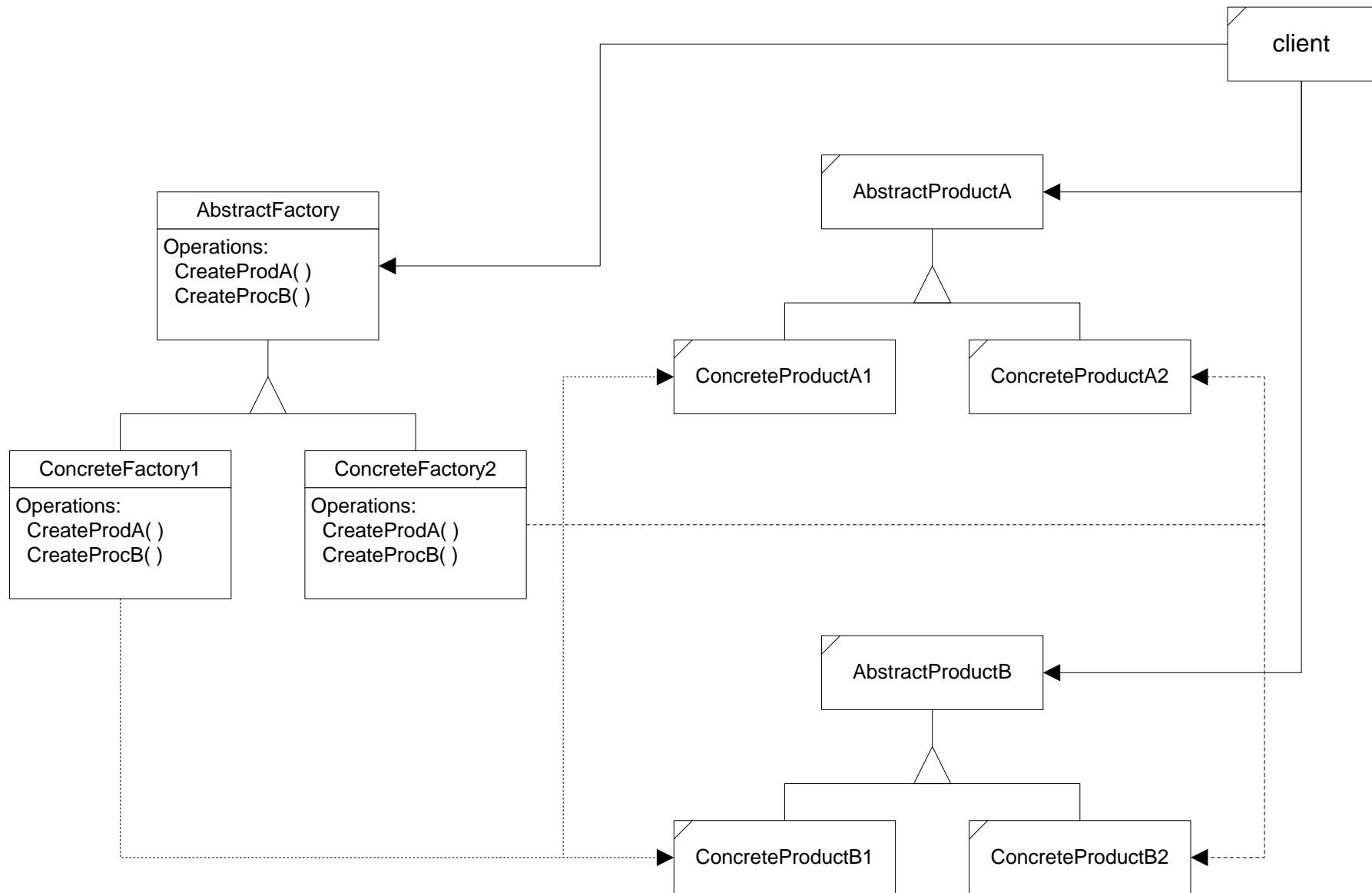
Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Applicability

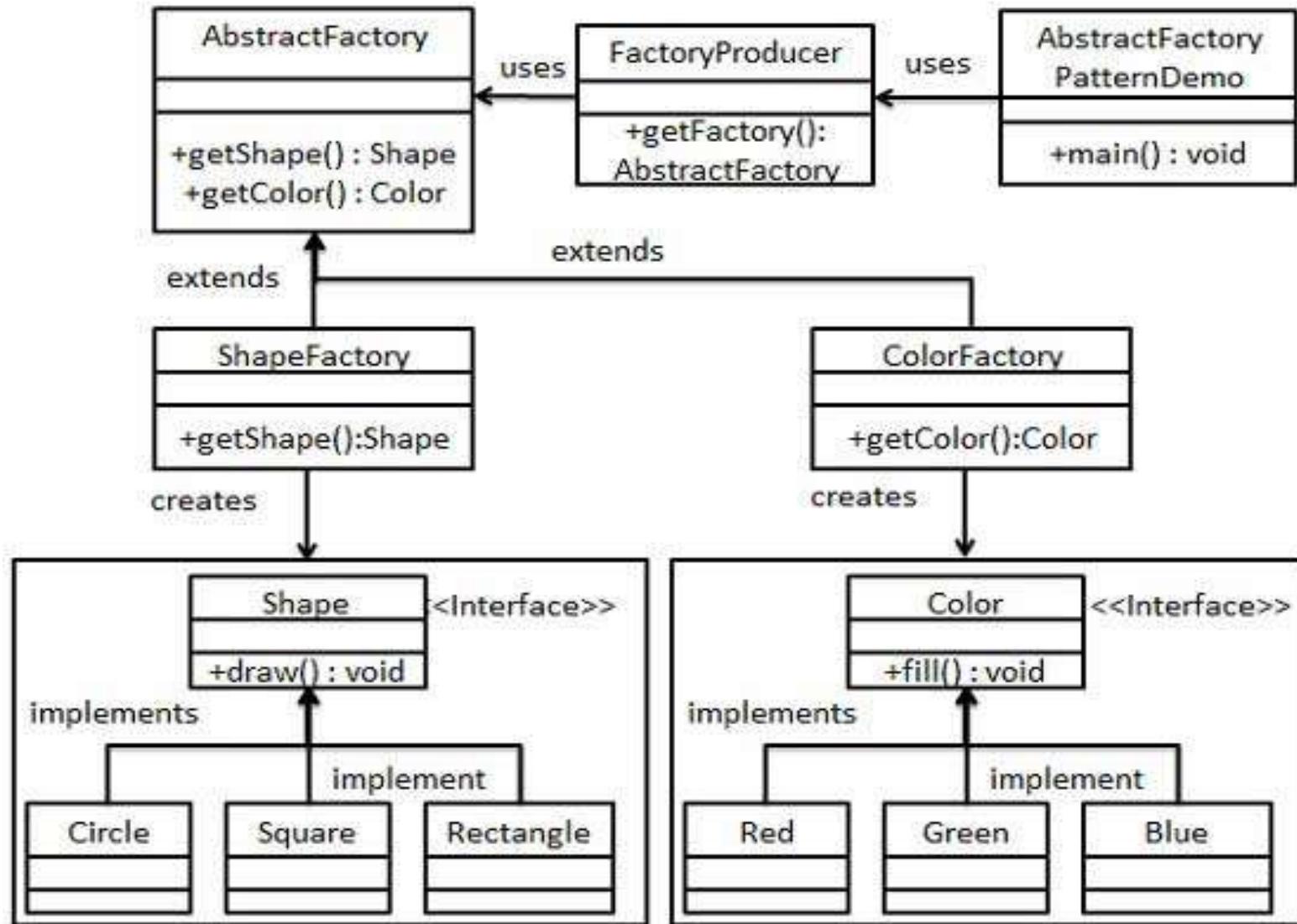
- Use the Abstract Factory Pattern if:
 - clients need to be ignorant of how servers are created, composed, and represented.
 - clients need to operate with one of several families of products
 - a family of products must be used together, not mixed with products of other families, e.g., there are constraints on the way objects are composed.
 - you provide a library and want to show just the interface, not implementation of the library components.
 - Giving customers your product header files may disclose some of your proprietary value.

Abstract Factory Structure



Participants

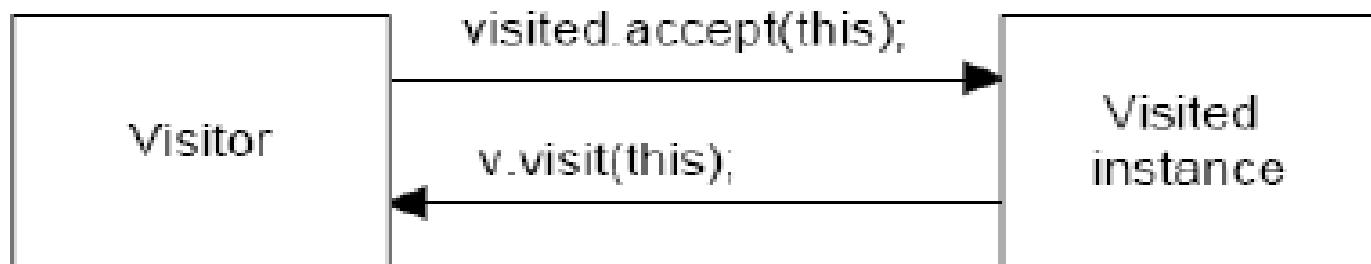
- **AbstractFactory**
 - provide an interface for building product objects
- **ConcreteFactory**
 - implements the creation functionality for a specific product family
- **AbstractProduct**
 - provides an interface for using product objects
- **ConcreteProduct**
 - created by a **ConcreteFactory**, implements the **AbstractProduct** interface for a specific product family
- **Client**
 - uses only abstract interfaces so is independent of the implementation.



Why Visitors?

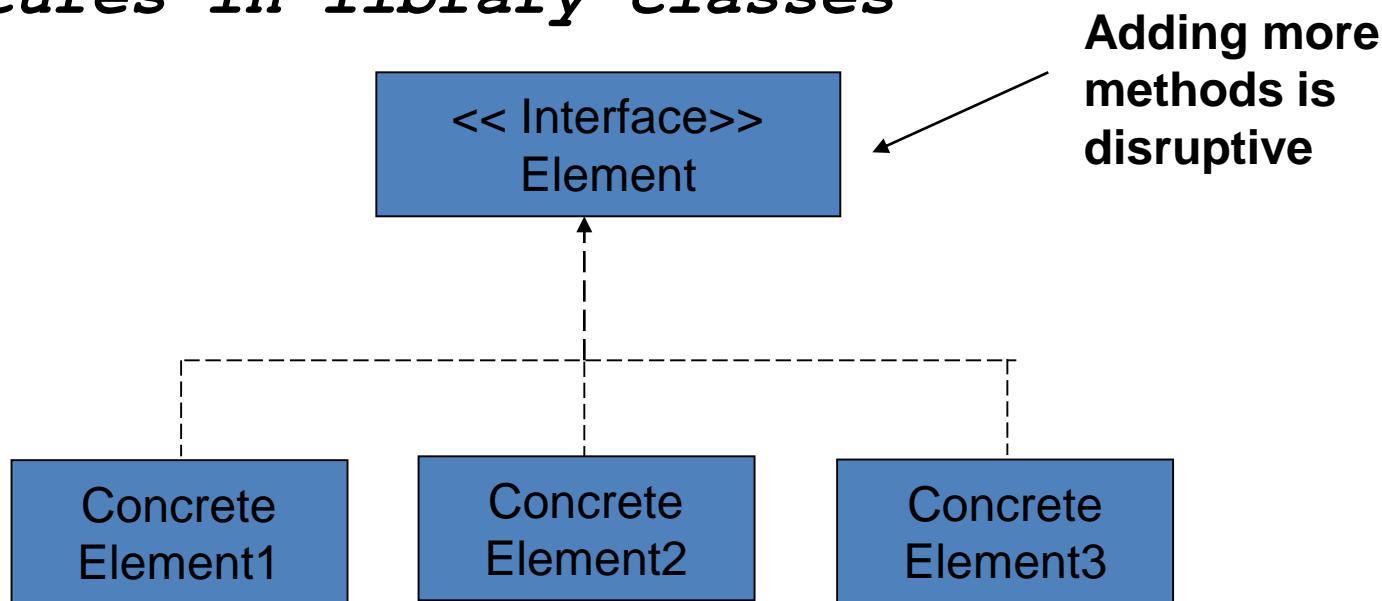
- The Visitor pattern is one among many design patterns aimed at making object-oriented systems more flexible
- The issue addressed by the Visitor pattern is the manipulation of composite objects.
- Without visitors, such manipulation runs into several problems as illustrated by considering an implementation of integer lists, written in Java

- There is only one way that an outside class can gain access to another class
 - by calling its public methods.
- In the Visitor case, visiting each class means that you are calling a method already installed for this purpose, called *accept*.
- The *accept method has one argument: the instance of the visitor, and in return, it calls the visit method of the Visitor. passing itself as an argument.*



Visitor Pattern

- *Provides Mechanism for adding new operations to the already provided interface.*
- *Extensible mechanism for supporting new features in library classes*



- Each accept method takes a visitor as argument.
- The interface Visitor has a header for each of the basic classes.
- The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes.
- The price is that all objects must have an accept method.

- Each Element class support a single method:
- Void accept(Visitor v)
- Visitor interface:

Public interface visitor

```
{  
Void visitElementType1(ElementType1 elmt);  
Void visitElementType2(ElementType2 elmt);  
Void visitElementType3(ElementType3 elmt);  
  
Void visitElementTypen(ElementTypen elmt);}
```

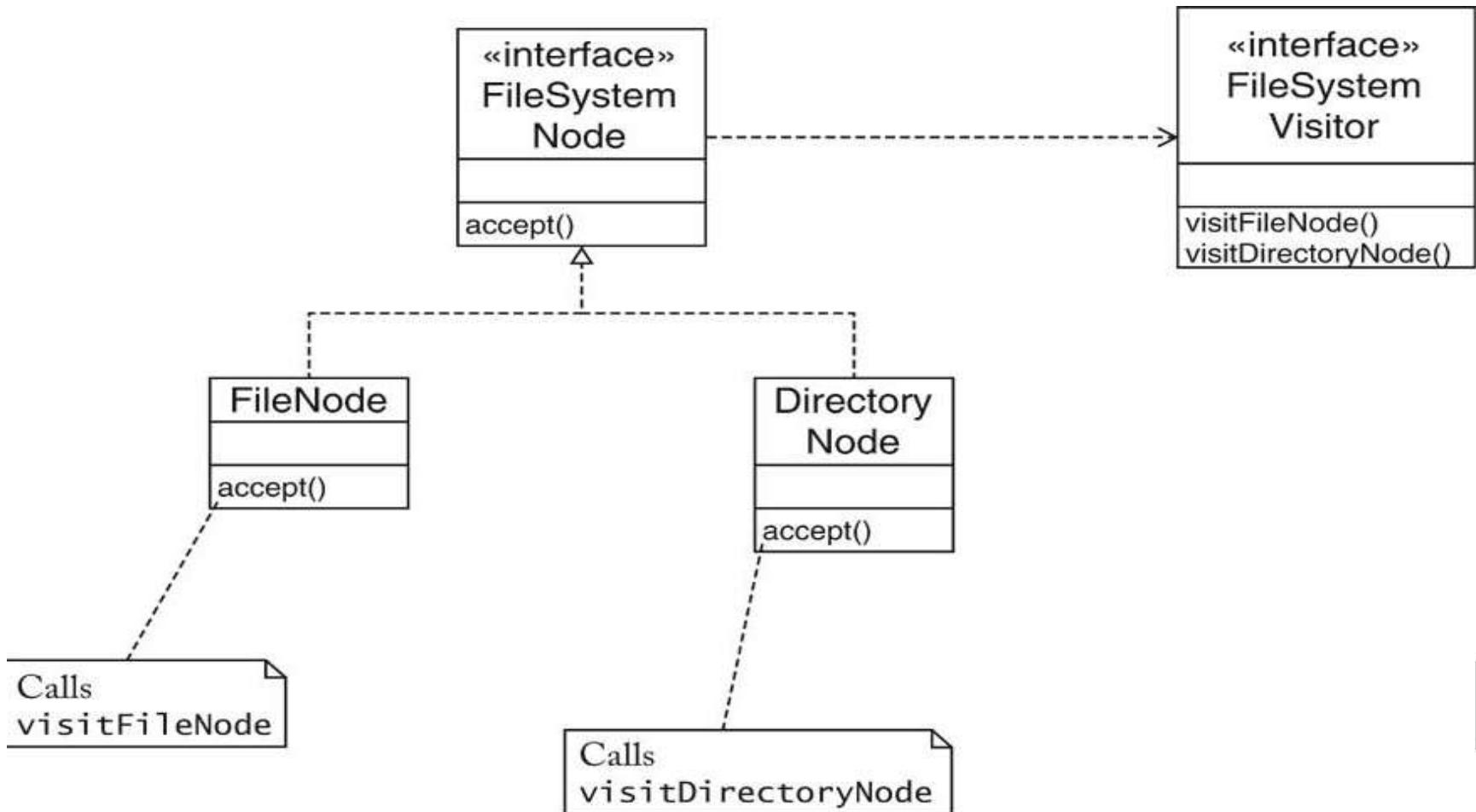
- Public class ElementType1

```
{ v.visitElementType1(this);  
}
```

.....

```
}
```

Example

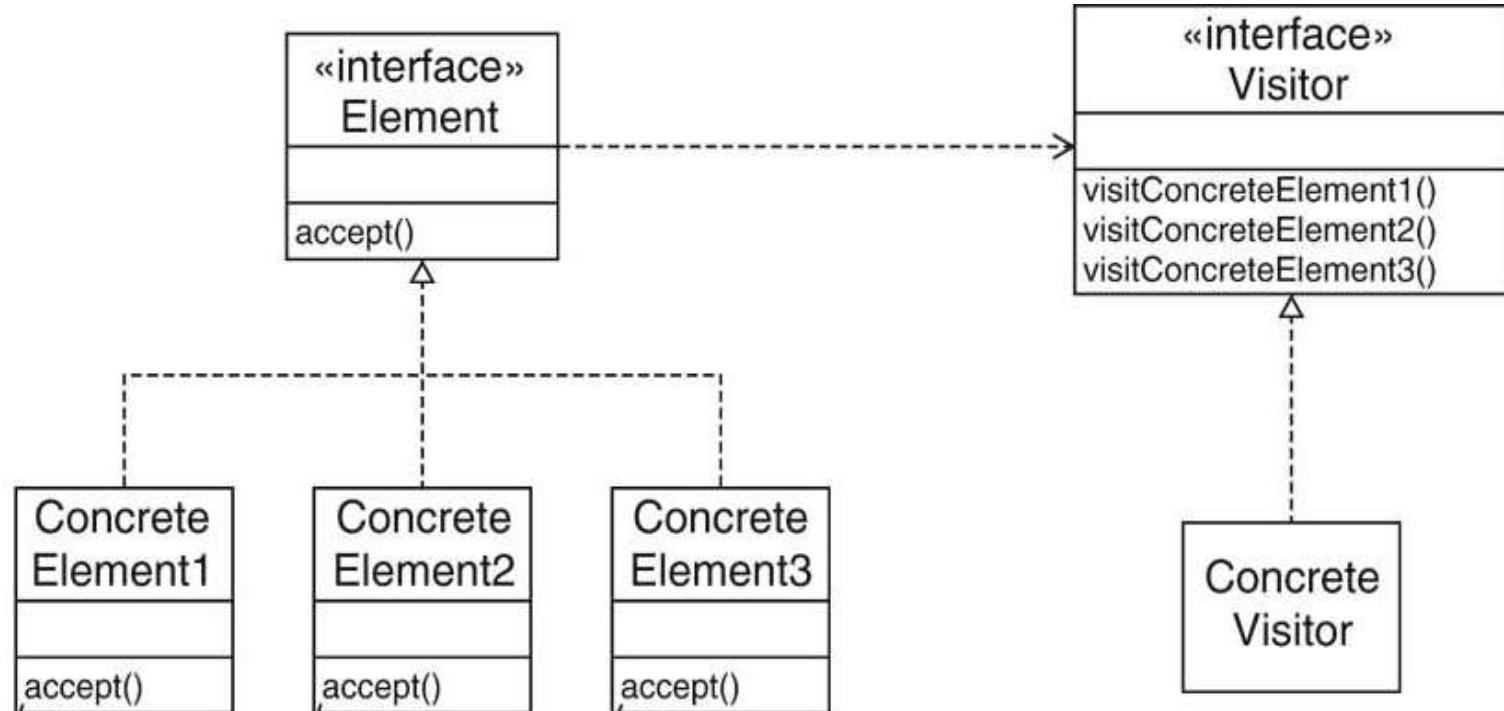


The VISITOR Pattern

- **Context**
1. An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types.
 2. The set of operations should be extensible over time.
 3. The set of element classes is fixed.

Solution

1. Define a visitor interface type that has methods for visiting elements of each of the given types.
2. Each element class defines an accept method that invokes the matching element visitation method on the visitor parameter.
3. To implement an operation, define a class that implements the visitor interface type and supplies the operations action for each element type.



Calls
visitConcreteElement1()

Calls
visitConcreteElement2()

Calls
visitConcreteElement3()

Before Using Patterns

- Before using a pattern to resolve the problem ask
 - Is there a pattern that addresses a similar problem?
 - Does the pattern trigger an alternative solution that may be more acceptable?
 - Is there a simpler solution? Patterns should not be used just for the sake of it

Before Using Patterns

- Is the context of the pattern consistent with that of the problem?
- Are the consequences of using the pattern acceptable?
- Are there constraints imposed by the software environment that would conflict with the use of the pattern?

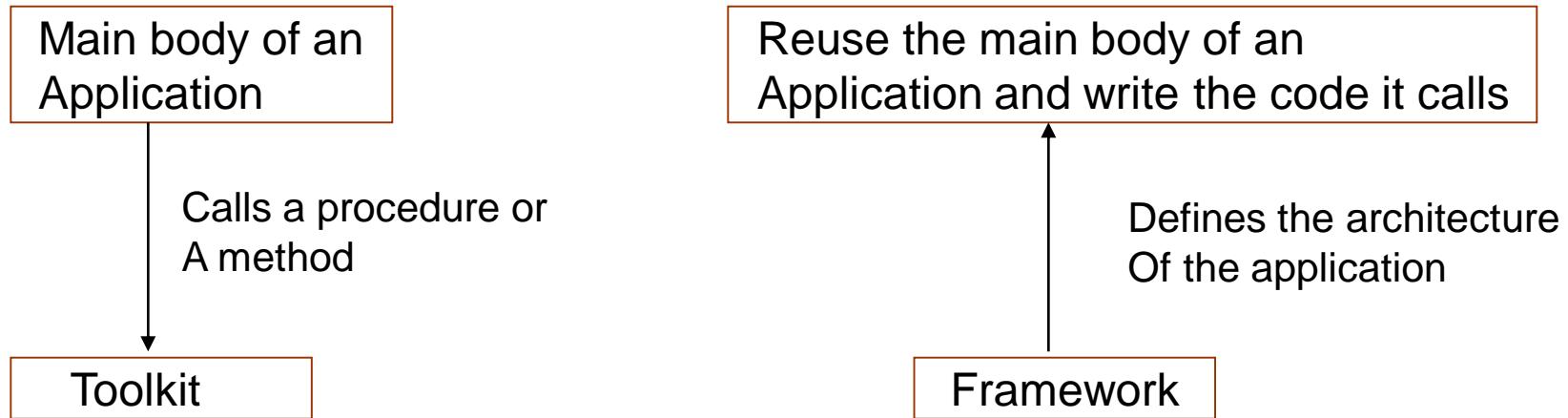
Using Patterns

- After selecting a suitable pattern
 1. Read the pattern to get a complete overview
 2. Study the Structure, Participants and Collaborations of the pattern in detail
 3. Examine the Sample Code to see an example of the pattern in use

Using Patterns

4. Choose names for the pattern's participants (i.e. classes) that are meaningful to the application
5. Define the classes
6. Choose application specific names for the operations
7. Implement operations that perform the responsibilities and collaborations in the pattern

Toolkits and Frameworks



Toolkits: Collection of related and reusable classes
e.g. C++ I/O stream library

Framework: A set of cooperating classes that make up a reusable design for a specific class of applications e.g. drawing, compilers, CAD/CAM etc.

Toolkits and Frameworks

Advantages and disadvantages of using frameworks.

1. When using frameworks, what defines the architecture of the application?
2. What is more difficult to design: Application, toolkit, or frameworks?
3. How do changes in framework effect an application?
4. How do design patterns differ from frameworks?

1. Is this a valid implementation of a singleton? Explain your answer.

```
public class Singleton {  
    private static Singleton s_singleton = new Singleton();  
    private Singleton() {}  
    private static Singleton getInstance() {  
        return s_singleton;  
    }  
}
```

Answer: This class is impossible to access because all the methods, including the constructor are private. The getInstance() method need to be public and static otherwise you have no chance to access the class.

2. Does the following code fragment implement the Factory Method design pattern?

```
public class XMLReaderFactory {  
    // This method returns an instance of a class  
    // that implements the XMLReader interface.  
    // The specific class it creates and returns is  
    // based on a system property.  
    public static XMLReader createXMLReader();  
}  
  
public interface XMLReader {  
    public void setContentHandler(ContentHandler handler);  
    public void parse(InputStream is);  
}
```

Answer: Not really. The XMLReaderFactory it creates a class XMLReader that is independent of the client but it does not define an abstract factory method so it does not follow literally the Factory Method Design Pattern.

- You want to implement a text paragraph. A paragraph is a sequence of lines. Each line is represented by a string. The Paragraph class provides method that appends a line to the end of the paragraph with a format algorithm (e.g., left-align or centered) which can be selected at runtime. It also has to be possible to add new format algorithms to the program without modifying the Paragraph class. Which design pattern could you use?

strategy

Software Metrics

Questions

- How big is the program?
 - Huge!!
- How close are you to finishing?
 - We are almost there!!
- Can you, as a manager, make any useful decisions from such subjective information?
- Need information like, cost, effort, size of project.

Metrics

- Quantifiable measures that could be used to measure characteristics of a software system or the software development process
- Required in all phases
- Required for effective management
- Managers need **quantifiable** information, and not **subjective** information
 - Subjective information goes against the fundamental goal of **engineering**

Why Metrics?

- goal: good software design => low costs
 - small testing effort
 - low maintenance costs
 - many reusable fragments
- What are metrics?
 - Definition of measurable criterions to distinguish “good” from “bad” design
- Use metrics to indicate source of “badness”

Definitions

- **Measure** - Quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process.
 - E.g., Number of errors
- **Measurement** - The act of determining a measure
- **Metric** - A quantitative measure of the degree to which a system, component, or process possesses a given attribute
 - E.g., Number of errors found per person hours expended

Definitions

- **Indicator** – An indicator is a metric or combination of metrics that provide insight into the software process, a software project or the product itself.
- **Direct Metrics:** Immediately measurable attributes (e.g. line of code, execution speed, defects reported)
- **Indirect Metrics:** Aspects that are not immediately quantifiable (e.g. functionality, quantity, reliability)
- **Faults:**
 - Errors: Faults found by the practitioners during software development
 - Defects: Faults found by the customers after release

Why Do We Measure?

- To indicate the quality of the product.
- To assess the productivity of the people who produce the product
- To assess the benefits derived from new software engineering methods and tools
- To form a baseline for estimation
- To help justify requests for new tools or additional training
- Estimate the cost & schedule of future projects
- Forecast future staffing needs
- Anticipate and reduce future maintenance needs

Metrics and Software Quality

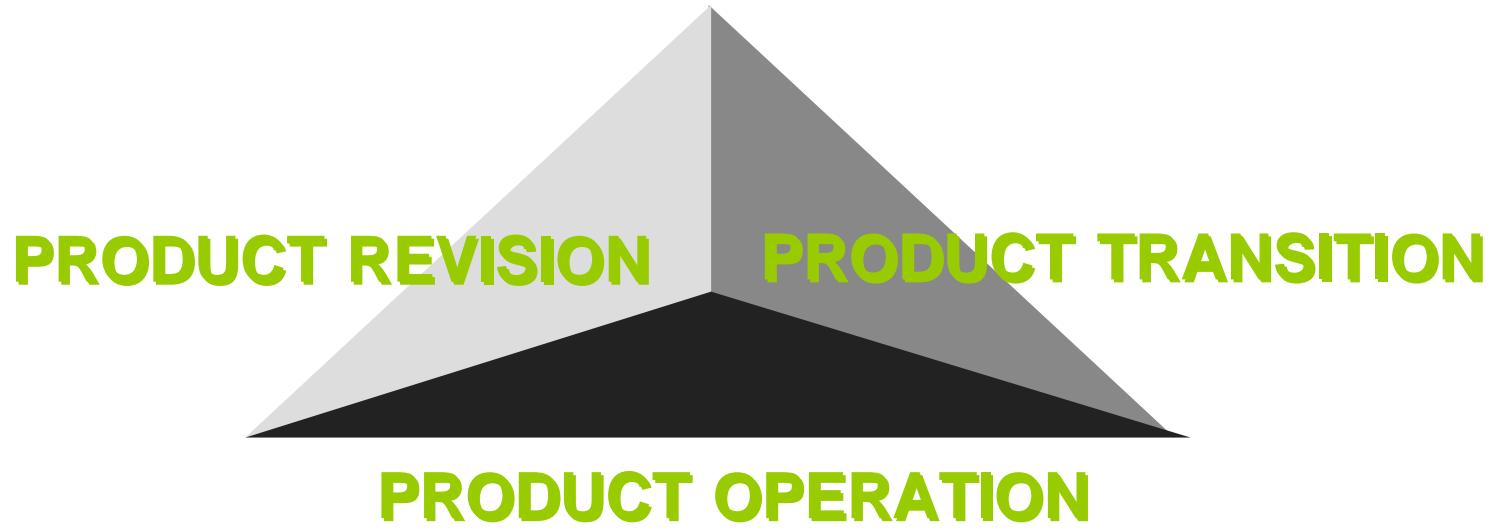
FURPS

- Functionality - features of system
- Usability – aesthetics, documentation
- Reliability – frequency of failure, security
- Performance – speed, throughput
- Supportability – maintainability

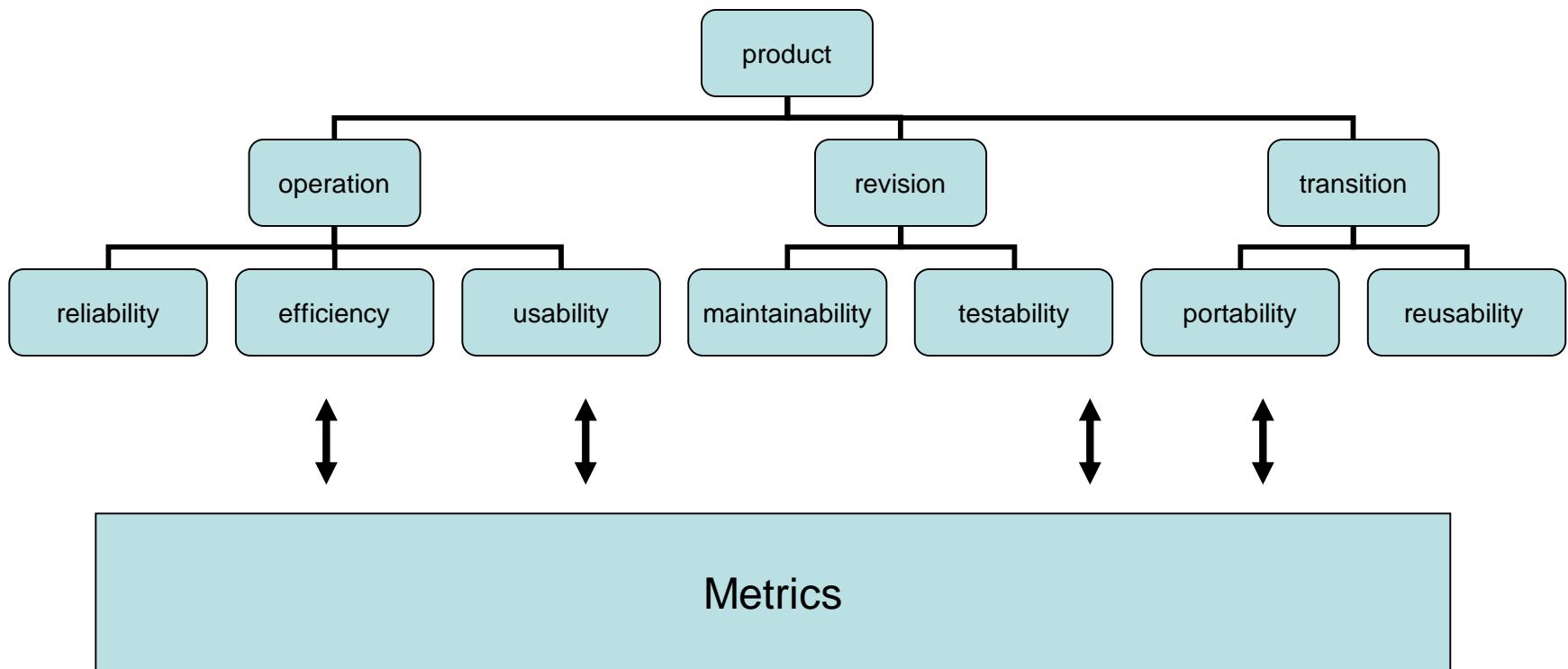
Measures of Software Quality

- Correctness – degree to which a program operates according to specification
 - Defects/KLOC
 - Defect is a verified lack of conformance to requirements
 - Failures/hours of operation
- Maintainability – degree to which a program is open to change
 - Mean time to change
 - Change request to new version (Analyze, design etc)
 - Cost to correct
- Integrity - degree to which a program is resistant to outside attack
 - Fault tolerance, security & threats
- Usability – easiness to use
 - Training time, skill level necessary to use, Increase in productivity, subjective questionnaire or controlled experiment

McCall's Triangle of Quality



Quality Model



Using Metrics

- The Process
 - Select appropriate metrics for problem
 - Utilized metrics on problem
 - Assessment and feedback
- Formulate
- Collect
- Analysis
- Interpretation
- Feedback

OO Metrics

- Measuring on class level
 - Coupling
 - Inheritance
 - Cohesion
 - Size
 - Structural Complexity
- Measuring on package / higher level

Metrics for the Object Oriented

- Chidamber & Kemerer '94
- Metrics specifically designed to address object oriented software
- Class oriented metrics
- Direct measures

Weighted Methods per Class

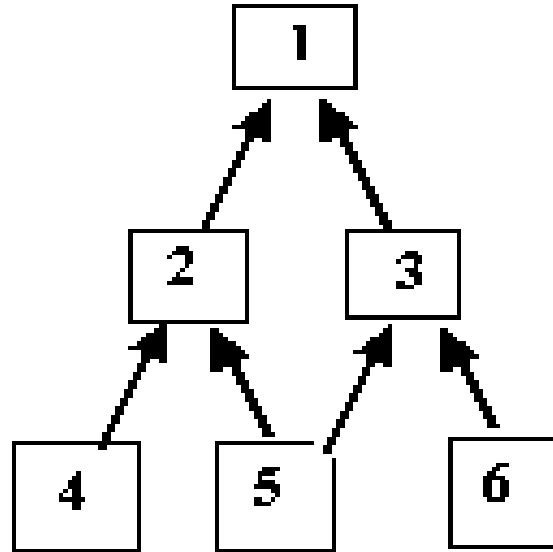
$$\text{WMC} = \sum_{i=1}^n c_i$$

- c_i is the complexity (e.g., volume, cyclomatic complexity, etc.) of each method
- Viewpoints: (of Chidamber and Kemerer)
 - The number of methods and complexity of methods is an indicator of ***how much time and effort is required to develop and maintain*** the object
 - The ***larger the number of methods in an object, the greater the potential impact on the children***
 - Objects with ***large number of methods*** are likely to be more application specific, ***limiting the possible reuse***

Depth of Inheritance Tree

- DIT is the maximum length from a node to the root (base class)
- Viewpoints:
- Lower level subclasses inherit a number of methods making behavior harder to predict
- Deeper trees indicate greater design complexity

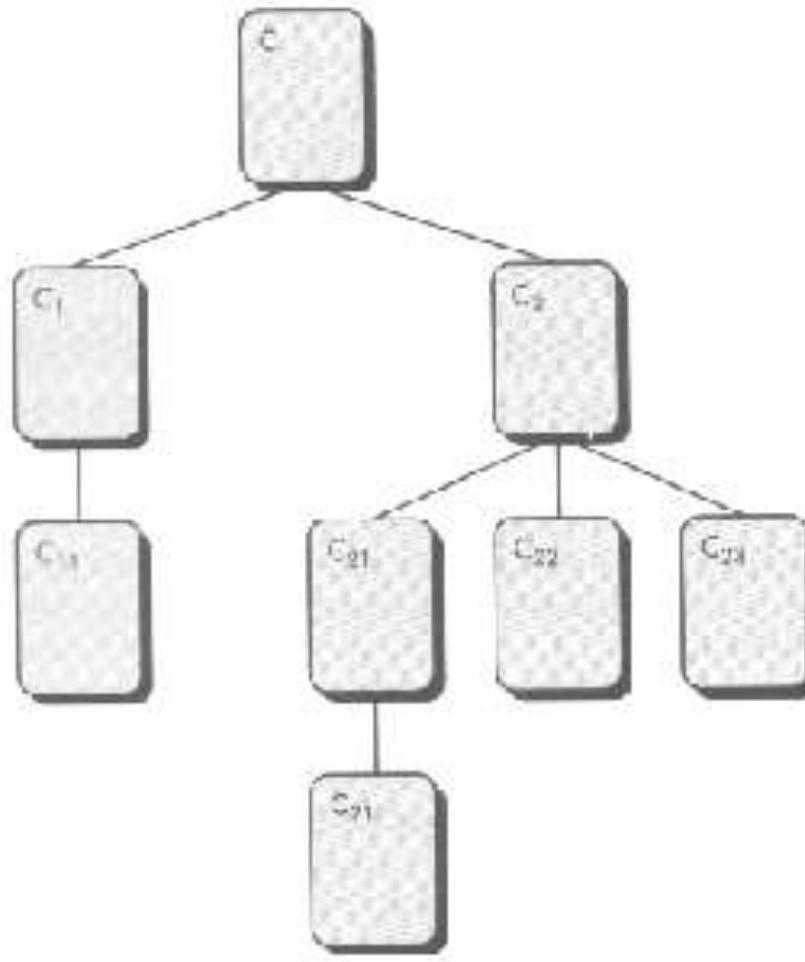
DIT Example



DIT level of subclass 4 and 5 is = 2

Number of Children

- NOC is the number of subclasses immediately subordinate to a class
- **Viewpoints:**
- As *NOC* grows, *reuse increases* - but the abstraction may be diluted
- *Depth is generally better* than breadth in class hierarchy, since it *promotes reuse* of methods through inheritance
- Classes *higher up in the hierarchy* should have *more sub-classes* than those lower down
- NOC gives an idea of the potential *influence a class has on the design*: classes with large number of children may require more testing



In the preceding example, the number of children for C2 is 3 (C21, C22, C23).

Coupling between Classes

- CBO is the number of collaborations between two classes (fan-out of a class C)
 - the number of other classes that are referenced in the class C (a reference to another class, A, is an reference to a method or a data member of class A)
- **Viewpoints:**
 - As collaboration increases reuse decreases
 - High fan-outs represent class coupling to other classes/objects and thus are undesirable
 - High fan-ins represent good object designs and high level of reuse
 - Not possible to maintain high fan-in and low fan outs across the entire system

Response for a Class

- RFC is the number of methods that could be called in response to a message to a class (local + remote)
- Viewpoints:
 - As RFC increases
 - testing effort increases
 - greater the complexity of the object
 - harder it is to understand

Lack of Cohesion in Methods

- LCOM
- Class C_k with n methods M_1, \dots, M_n
- I_j is the set of instance variables used by M_j

LCOM

- There are n such sets I_1, \dots, I_n
 - $P = \{(I_i, I_j) \mid (I_i \cap I_j) = \emptyset\}$
 - $Q = \{(I_i, I_j) \mid (I_i \cap I_j) \neq \emptyset\}$
- If all n sets I_i are \emptyset then $P = \emptyset$
- $\text{LCOM} = |P| - |Q|$, if $|P| > |Q|$
- $\text{LCOM} = 0$ otherwise

Example LCOM

- Take class C with M_1, M_2, M_3
- $I_1 = \{a, b, c, d, e\}$
- $I_2 = \{a, b, e\}$
- $I_3 = \{x, y, z\}$
- $P = \{(I_1, I_3), (I_2, I_3)\}$
- $Q = \{(I_1, I_2)\}$

- Thus $\text{LCOM} = 1$

Explanation

- LCOM is the number of empty intersections minus the number of non-empty intersections
- This is a notion of degree of similarity of methods
- If two methods use common instance variables then they are similar
- LCOM of zero is not maximally cohesive
- $|P| = |Q|$ or $|P| < |Q|$

Some other cohesion metrics

LCOM3	<p>Consider an undirected graph G, where the vertices are the methods of a class, and there is an edge between two vertices if the corresponding methods use at least an attribute in common.</p> <p>LCOM3 is then defined as the number of connected components of G.</p>
LCOM4	<p>Like LCOM3, where graph G additionally has an edge between vertices representing methods m and n, if m invokes n or vice versa.</p>
Co (connectivity)	<p>Let V be the number of vertices of graph G from measure LCOM4, and E the number of its edges. Then</p> $Co = 2 \cdot \frac{ E - (V - 1)}{(V - 1) \cdot (V - 2)}.$
LCOM5	<p>Consider a set of methods $\{M_i\}$ ($i=1, \dots, m$) accessing a set of attributes $\{A_j\}$ ($j=1, \dots, a$). Let $\mu(A_j)$ be the number of methods which reference attribute A_j.</p> <p>Then</p> $LCOM5 = \frac{\frac{1}{a} \left(\sum_{j=1}^a \mu(A_j) \right) - m}{1 - m}$

Metrics Suites – C & K

- Proposed by Chidamber & Kemerer, Sloan School of Mgmt., in 1991,
 - Based on measurement theory
 - Language independent
 - Theoretical validation using Weyuker's Axioms
 - Provided Empirical validation in 1994
 - Applied for management decision making in 1998
 - Quantifies Complexity using Cohesion, coupling, Inheritance
 - Applies to class
- Weighted Methods per Class (WMC)
 - Sum of complexities of methods
- Depth of Inheritance Tree (DIT)
 - Depth of inheritance of a class Starting from root
- Number of Children of a Class (NOC)
 - Number of immediate subclasses
- Coupling between Object Classes (CBO)
 - Count of number of other classes coupled
- Response set for a Class (RFC)
 - Cardinality of set of methods that can be executed
- Lack of Cohesion in Methods (LCOM)
 - Pairs of methods sharing instance variables

Class Size

- CS
 - Total number of operations (inherited, private, public)
 - Number of attributes (inherited, private, public)
- May be an indication of too much responsibility for a class

Number of Operations Overridden

- NOO
- A large number for NOO indicates possible problems with the design
- Poor abstraction in inheritance hierarchy

Number of Operations Added

- NOA
- The number of operations added by a subclass
- As operations are added it is farther away from super class
- As depth increases NOA should decrease

Method Inheritance Factor

$$\text{MIF} = \frac{\sum_{i=1}^n M_i(C_i)}{\sum_{i=1}^n M_a(C_i)} .$$

- $M_i(C_i)$ is the number of methods inherited and not overridden in C_i
- $M_a(C_i)$ is the number of methods that can be invoked with C_i
- $M_d(C_i)$ is the number of methods declared in C_i

MIF

- $M_a(C_i) = M_d(C_i) + M_i(C_i)$
- All that can be invoked = new or overloaded + things inherited
- MIF is $[0,1]$
- MIF near 1 means little specialization
- MIF near 0 means large change

Coupling Factor

$$CF = \frac{\sum_i \sum_j is_client(C_i, C_j)}{(TC^2 - TC)} .$$

- $is_client(x,y) = 1$ iff a relationship exists between the client class and the server class. 0 otherwise
- $(TC^2 - TC)$ is the total number of relationships possible
- CF is $[0,1]$ with 1 meaning high coupling

Polymorphism Factor

$$PF = \frac{\sum_i M_o(C_i)}{\sum_i [M_n(C_i) * DC(C_i)]}$$

- $M_n()$ is the number of new methods
- $M_o()$ is the number of overriding methods
- $DC()$ number of descendent classes of a base class
- The number of methods that redefines inherited methods, divided by maximum number of possible distinct polymorphic situations

Encapsulation

- MHF - Method Hiding Factor
- AHF - Attribute Hiding Factor
- Proposed as “measures of encapsulation”
- Earlier as measures of “the use of information hiding concept”

Visibility

- $\text{Is_visible}(M, C)$
 - 1 iff class C may call method M and M is in another class
 - 0 otherwise
- $V(M) = \text{sum of } \text{Is_visible} \text{ for method } M \text{ over all classes divided by number of other classes}$
 - percentage of other classes that can call this method

MHF definition

- Summation over all methods in all classes of 1 minus the $V(M)$ divided by the total number of methods

Encapsulation – MHF and AHF

- The *method hiding factor* (MHF) and the *attribute hiding factor* (AHF) attempt to measure the encapsulation.

$$\text{MHF} = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{m,i}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

$$\text{AHF} = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{m,i}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

- Where $A_d(C_i)$ is the number of attributes declared in a class and
- $V(A_{m,i})$ is the visibility of an attribute, $A_{m,i}$

MHF

- ▶ **Method Hiding Factor**
- ▶ **It is a fraction in which the denominator is the number of total methods whereas the numerator is the total of encapsulated methods defined in all the classes.**
- ▶ **If all methods are **private/protected**, $MHF = 100\%$, High encapsulation decreases the complexity since encapsulated methods dictate the scope from which they may be accessed therefore limiting the number of locations which makes the debugging process easier.**
- ▶ **If all methods are **public**, $MHF = 0\%$ shows methods are unprotected and chances of errors are high.**

MHF Example

```
class Polygon {  
    protected: int width, height;  
    public: void setParameters (int w, int  
h);  
    {  
        width = w;  
        height = h;  
    }  };
```

```
class Rectangle : public Polygon {  
    public: int calculateArea ()  
    { return (width*height); }  };
```

```
class Triangle : public Polygon {  
    public int calculateArea ()  
    { return( (width*height)/2); }  };
```

MHF = the total of encapsulated methods/ total methods
= 0

AHF

- **Attribute Hiding Factor**
- It measures the total number of attributes encapsulated in the class.
- The AHF may be expressed as a fraction in which the denominator is the number of total attributes whereas the numerator is the total of encapsulated attributes defined in all the classes.
- If all attributes are **private/protected** – MHF = 100%.
- If all attributes are **public** – MHF = 0% shows methods are unprotected and chances of errors are high.
- Same as MHF

AHF Example

```
#include <iostream>
using namespace std;
class A {
public:
    A() { a_member = 0; }
protected:
    int a_member;

class B{
public:
    B() { b_member = 0; }
protected:
    int b_member; };

class C : public A, public B {
public:
    C() : A(), B() { c_member = 0; }
public:
    int c_member; };
```

AHF = the total of encapsulated attributes/ total attributes
= 2/3 = .6667

MOOD

- Attribute Hiding Factor(AHF)
 - ideally 100 %
- Method Hiding Factor (MHF)
 - low: insufficient abstraction
 - high: little functionality
- Attribute Inheritance Factor (AIF)
- Method Inheritance Factor (MIF)
 - AIF and MIF should be within an acceptable range
- Polymorphism Factor (PF)
 - should be within acceptable range
- Coupling Factor (CF)
 - high CF values should be avoided
- MOOD conclusion
 - designed to measure overall quality of an OO project
 - not appropriate for projects relying mostly on forms and standard modules

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_v(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_v(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_s(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_s(C_i)}$$

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_d(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_u(C_i) \times DC(C_i)]}$$

$$COF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_client(C_i, C_j) \right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}$$

Operational Oriented Metrics

- Average operation size (LOC, volume)
- Number of messages sent by an operator
- Operation complexity – cyclomatic
- Average number of parameters/operation
 - Larger the number the more complex the collaboration

Encapsulation

- Lack of cohesion
- Percent public and protected
- Public access to data members

Inheritance

- Number of root classes
- Fan in – multiple inheritance
- NOC, DIT, etc.

Metric tools

- McCabe & Associates (founded by Tom McCabe, Sr.)
 - The Visual Quality ToolSet
 - The Visual Testing ToolSet
 - The Visual Reengineering ToolSet
- Metrics calculated
 - McCabe Cyclomatic Complexity
 - McCabe Essential Complexity
 - Module Design Complexity
 - Integration Complexity
 - Lines of Code
 - Halstead

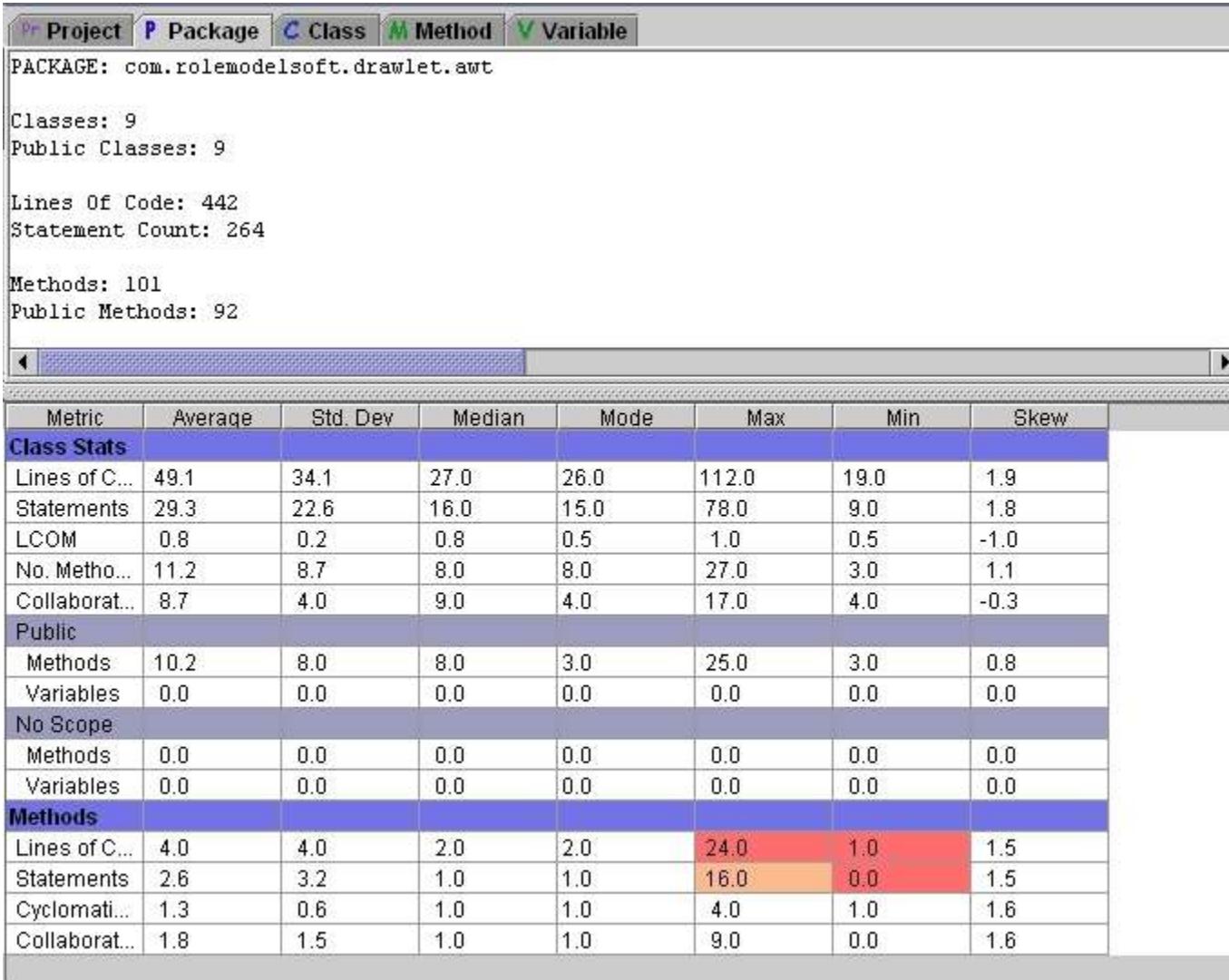
CCCC

- A metric analyser C, C++, Java, Ada-83, and Ada-95 (by Tim Littlefair of Edith Cowan University, Australia)
- Metrics calculated
 - Lines Of Code (LOC)
 - McCabe's cyclomatic complexity
 - C&K suite (WMC, NOC, DIT, CBO)
- Generates HTML and XML reports
- freely available
- <http://cccc.sourceforge.net/>

Jmetric

- OO metric calculation tool for Java code (by Cain and Vasa for a project at COTAR, Australia)
- Requires Java 1.2 (or JDK 1.1.6 with special extensions)
- Metrics
 - Lines Of Code per class (LOC)
 - Cyclomatic complexity
 - LCOM (by Henderson-Seller)
- Availability
 - is distributed under GPL
- <http://www.it.swin.edu.au/projects/jmetric/products/jmetric/>

JMetric tool result



The screenshot shows the JMetric tool interface with the following statistics for the package com.rolemodelsoft.drawlet.awt:

- Classes: 9
- Public Classes: 9
- Lines Of Code: 442
- Statement Count: 264
- Methods: 101
- Public Methods: 92

Below the statistics is a detailed metrics table:

Metric	Average	Std. Dev	Median	Mode	Max	Min	Skew
Class Stats							
Lines of C...	49.1	34.1	27.0	26.0	112.0	19.0	1.9
Statements	29.3	22.6	16.0	15.0	78.0	9.0	1.8
LCOM	0.8	0.2	0.8	0.5	1.0	0.5	-1.0
No. Metho...	11.2	8.7	8.0	8.0	27.0	3.0	1.1
Collaborat...	8.7	4.0	9.0	4.0	17.0	4.0	-0.3
Public							
Methods	10.2	8.0	8.0	3.0	25.0	3.0	0.8
Variables	0.0	0.0	0.0	0.0	0.0	0.0	0.0
No Scope							
Methods	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Variables	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Methods							
Lines of C...	4.0	4.0	2.0	2.0	24.0	1.0	1.5
Statements	2.6	3.2	1.0	1.0	16.0	0.0	1.5
Cyclomatic...	1.3	0.6	1.0	1.0	4.0	1.0	1.6
Collaborat...	1.8	1.5	1.0	1.0	9.0	0.0	1.6

GEN++

(*University of California, Davis and Bell Laboratories*)

- GEN++ is an application-generator for creating code analyzers for C++ programs
 - simplifies the task of creating analysis tools for the C++
 - several tools have been created with GEN++, and come with the package
 - these can both be used directly, and as a springboard for other applications
- Freely available