

E-Government Modeling and Design Patterns

Li Xiaoshan

Department of Computer and Information Science

Faculty of Science and Technology

University of Macau

Email: xsl@umac.mo

Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

1. Introduction

- E-government is a software system which can provide external services to citizens and business, and also provide services to internal government agencies as well.
- Motivation is to make government work effectively and efficiently.
- E-government is a new application area of computing system.

eMacao: Electronic Government System of Macao

- eMacao is a big and complex software system with some changes.
 - G2C (different public service systems)
 - G2B (business license application system,)
 - G2G (financial system, human resource management system, administration system)
- Complexity & Change

Benefit Application System (BAS)

Informal Description:

BAS provides online benefit application services to Macao local people through the Internet, such as view information of different benefits, download application forms, submit applications as well as check the status of their applications. And the system can also provide functions for the government officers to deal with those submitted applications, review applications and approve application, as well as other necessary relevant functions. Of course, the system should guarantee the security of personal information, and other non-functional requirements.

2: Overview of UP

Contents

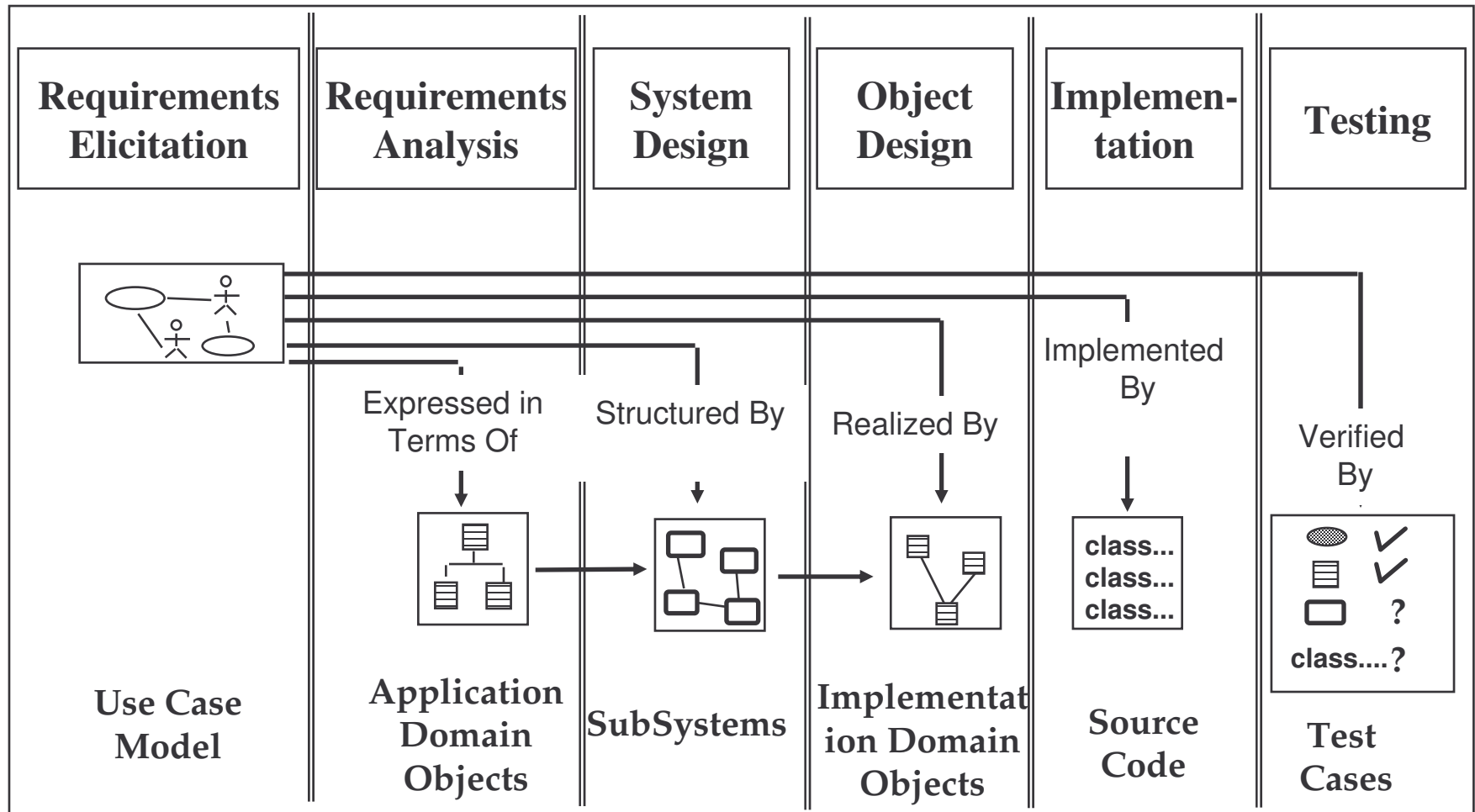
1. Introduction
2. *Unified Process (UP)*
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

Software Development Process

A Software Development Process comprises the activities needed to transform user requirements into a software system.



Software Development Activities



Unified Process (UP): *Use-Case Driven, Architecture-Centric, Iterative and Incremental*

- Software evolves toward bigger, more complex system.
- A software development process defines *who* is doing *what*, *when* and *how* to reach a certain goal. (technologies, tools, people, organization patterns)
- Unified Software Development Process: the outcome of more than 30 years of experience.

Four Ps: *People, Project, Product* and *Process*.

UP is Use-Case Driven

1. *User (Actors & Stakeholders)* represents someone or something that interacts with the system being developed.
2. An interaction is a *use case* (action), a piece of functionality in the system. (use-case view)
3. Use cases are not just a tool for specifying the requirements of system. They also drive its design, implement, and test: that is, they drive the development process – use cases are specified, use cases are designed, testing cases.

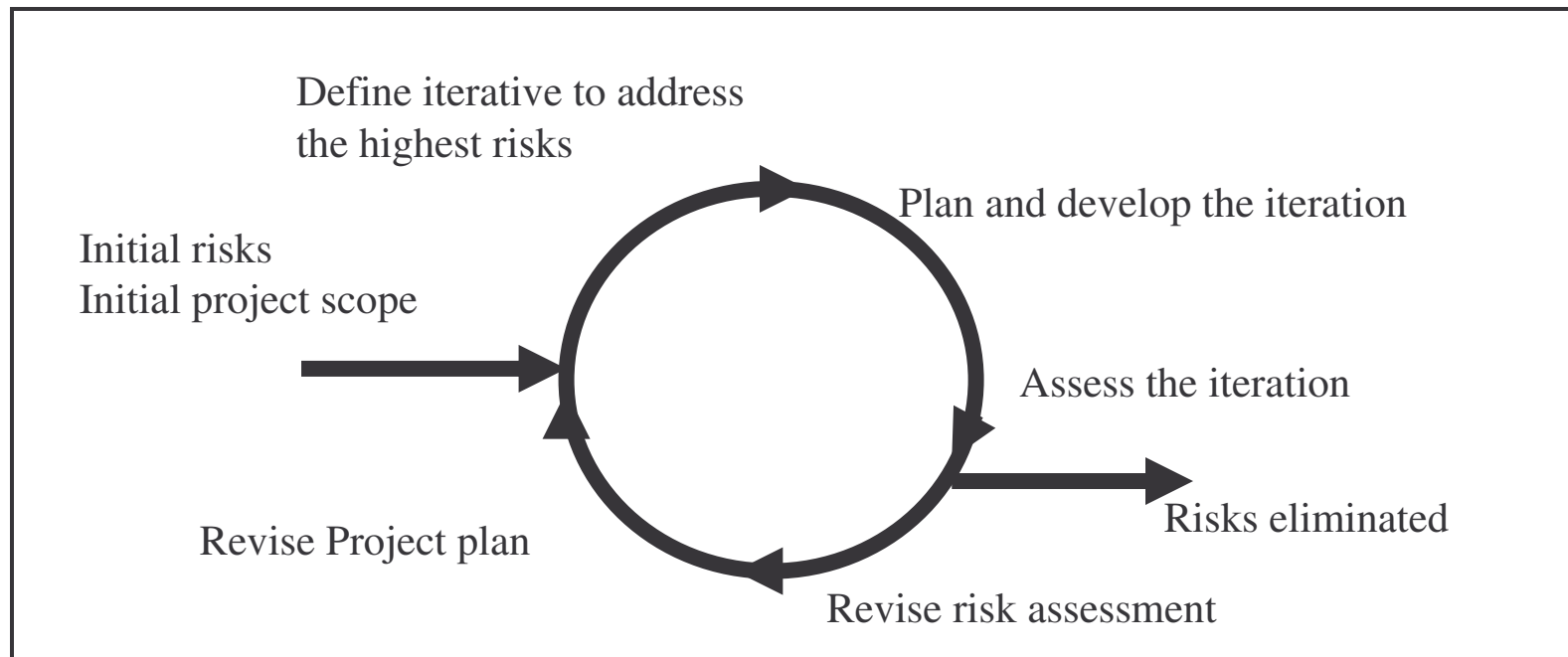
UP is Architecture-Centric

Software architecture captures the most significant static and dynamic aspects of the system.

Architecture Patterns

UP is Iterative and Incremental

- Each mini-project is an iteration that results in an increment (rolling snow ball). Iterative and incremental development proceeds as a series of iterations that evolve into the final system.



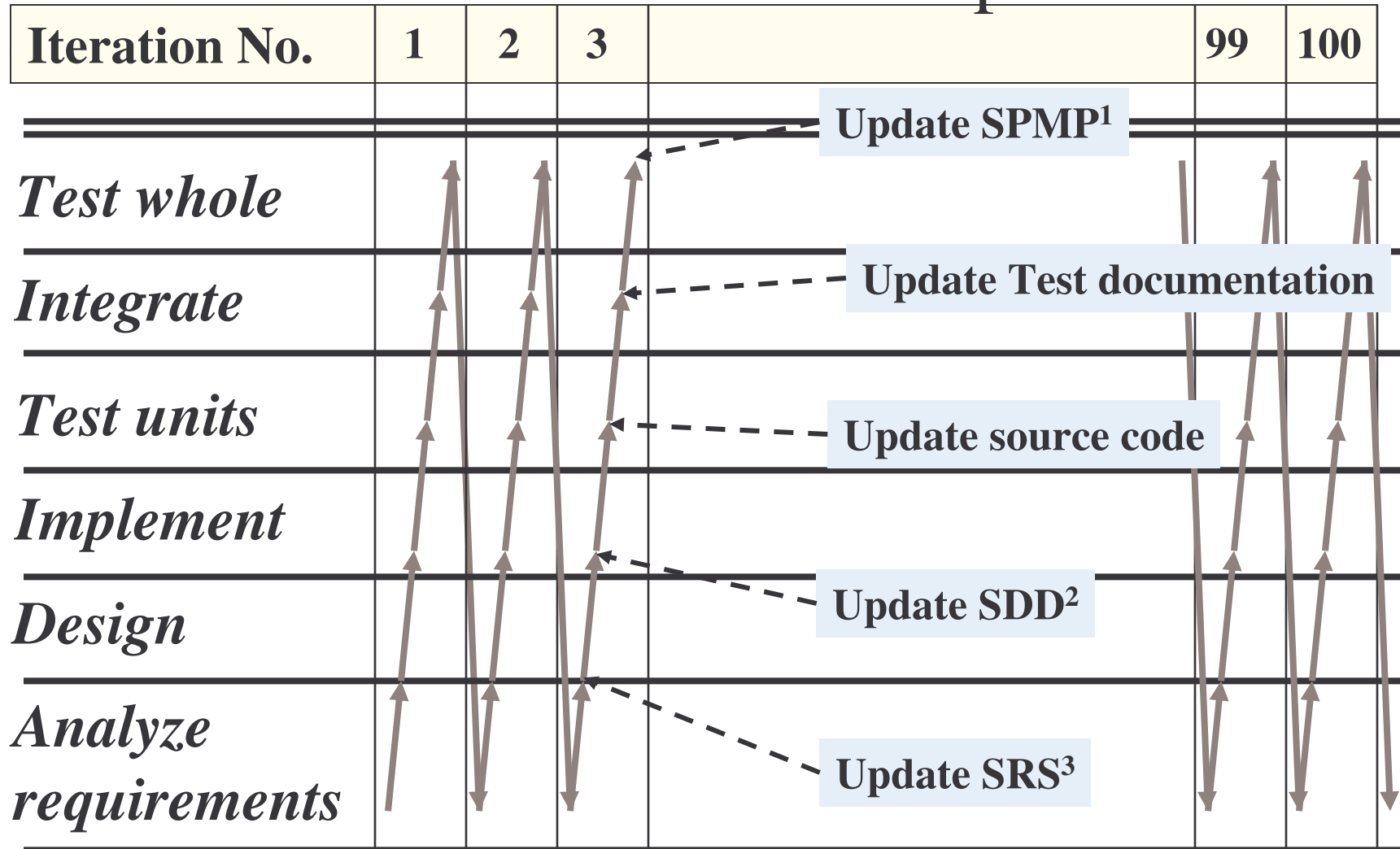
UP is Iterative and Incremental

- Each iteration consists of one or more of the following process components: requirements capture, analysis and design, implementation, and test.
- Developers do not assume that all requirements are known at the beginning of the life cycle; indeed change is anticipated throughout all phases.

4 Phases in Development Lifecycle

- **Inception**—specifying the project vision
- **Elaboration**—planning the necessary activities and required resources; specifying the features and designing the architecture.
- **Construction**—building the product as a series of incremental iterations.
- **Transition**—supplying the product to the user community(manufacturing, delivering, and training).

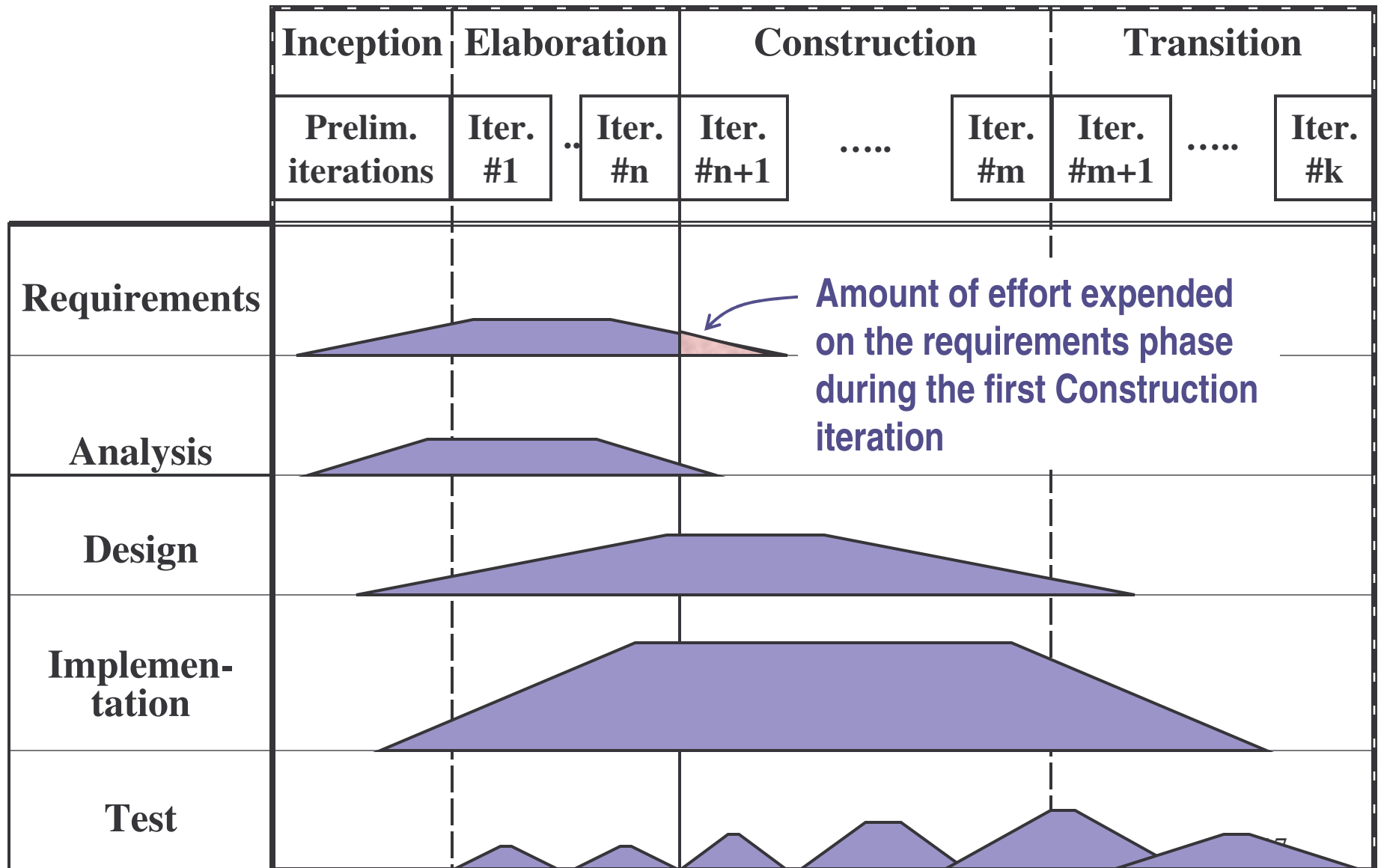
Incremental Development



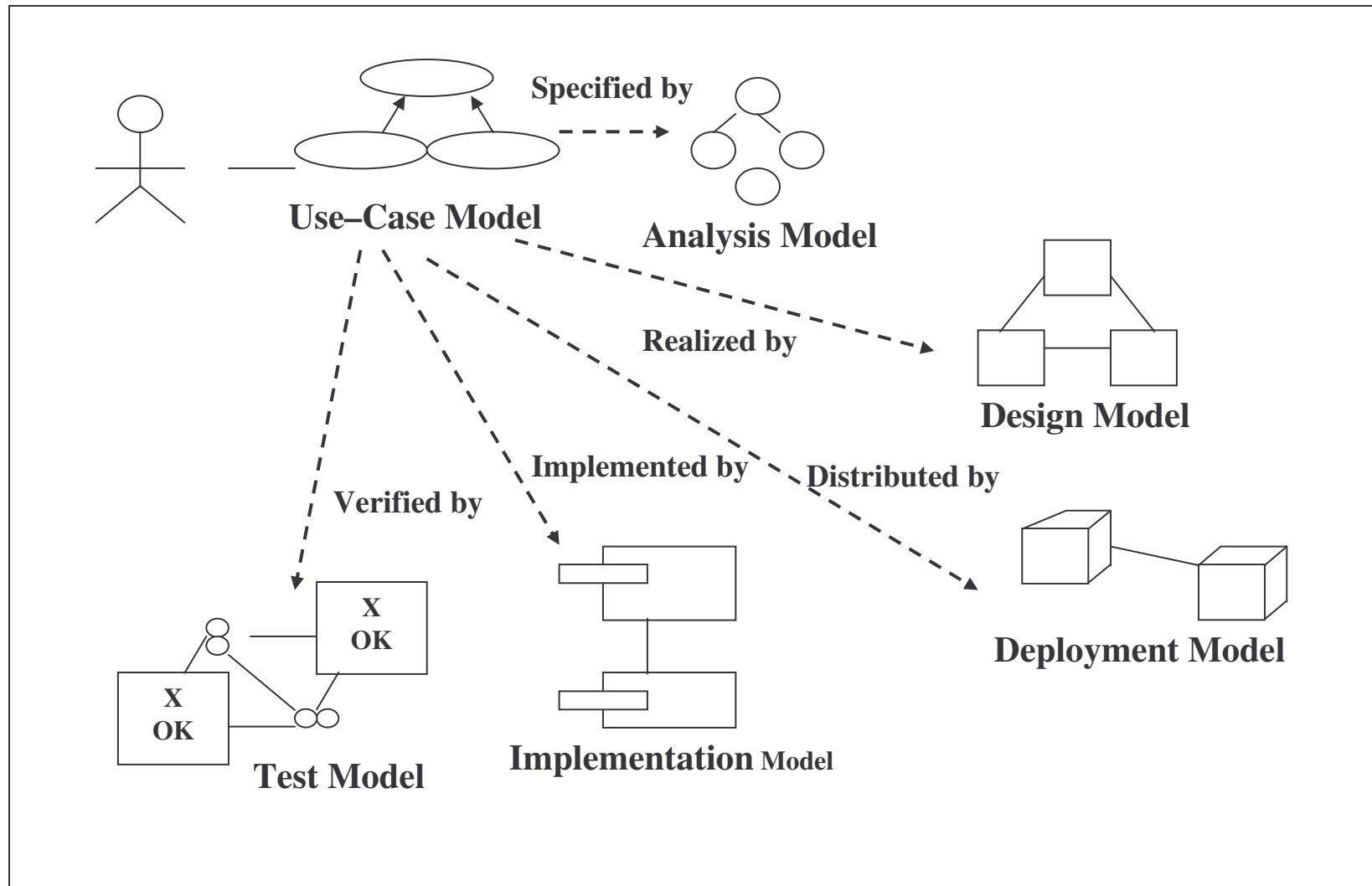
¹ Software Project Management Plan; ² Software Design Document; ³ Software Requirements Specification

Unified Process Matrix

Jacobson *et al*: USDP



Models of the Unified Process

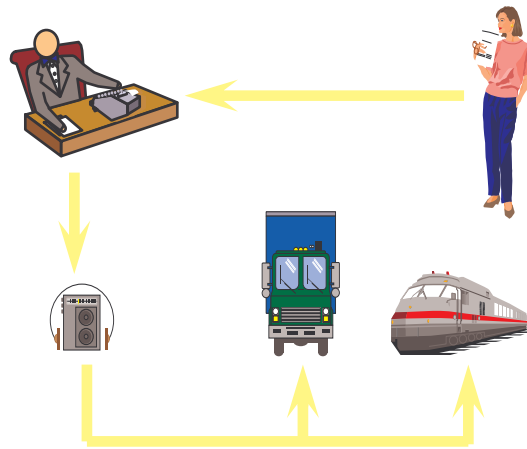


3: Visual Modeling and UML

Contents

1. Introduction
2. Unified Process (UP)
3. *Visual Modeling and UML*
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

What is Visual Modeling?

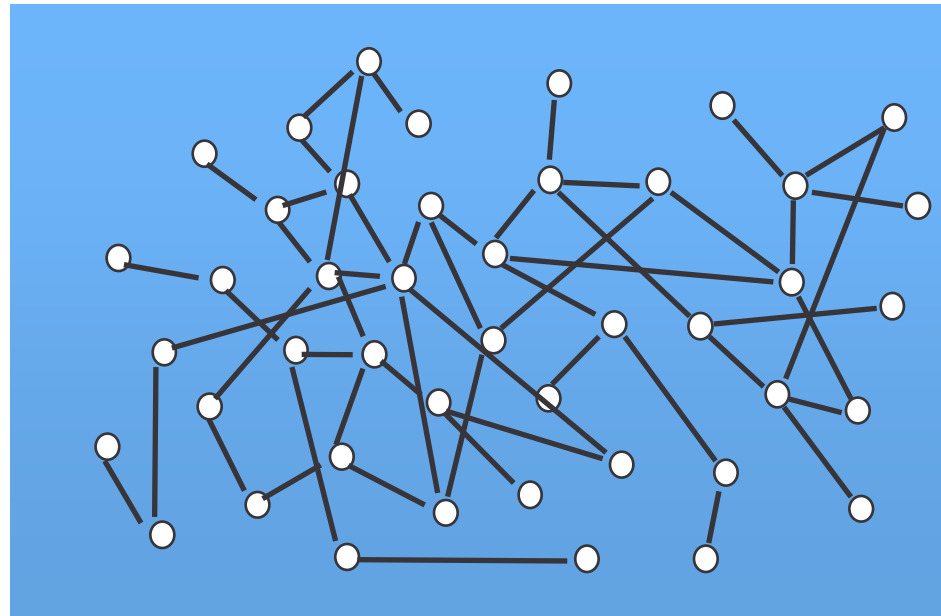
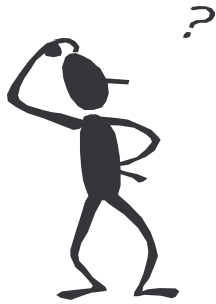


“Modeling captures essential parts of the system.”



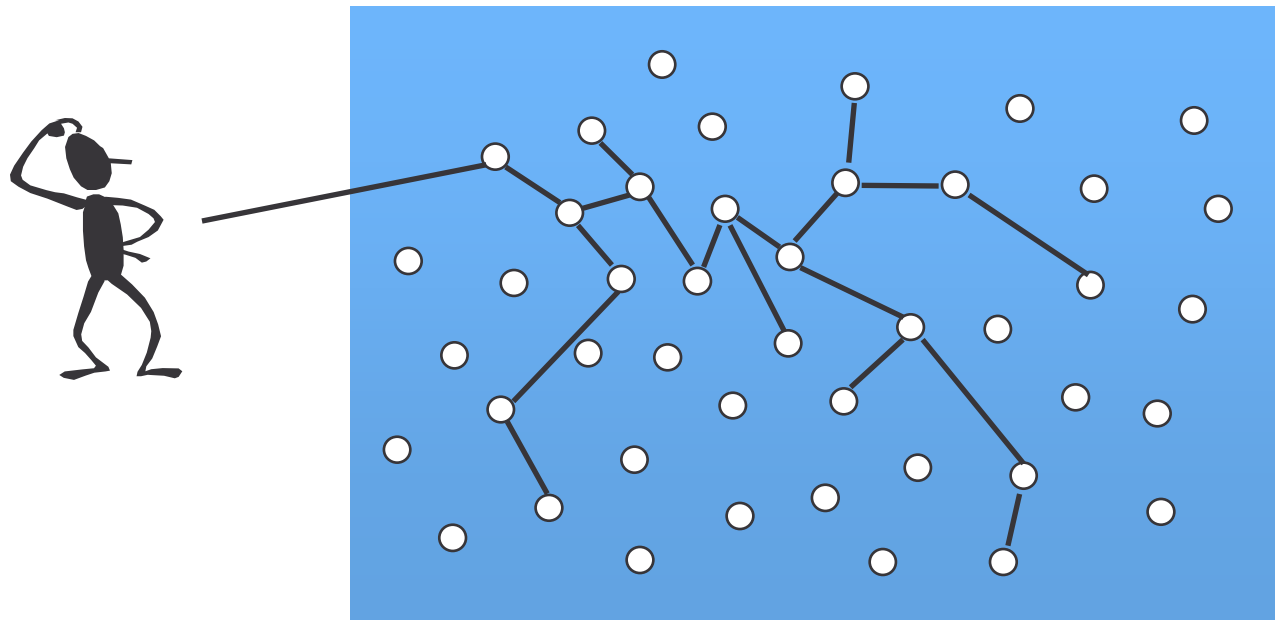
Visual Modeling is modeling using standard graphical notations

Modeling Captures Business Process



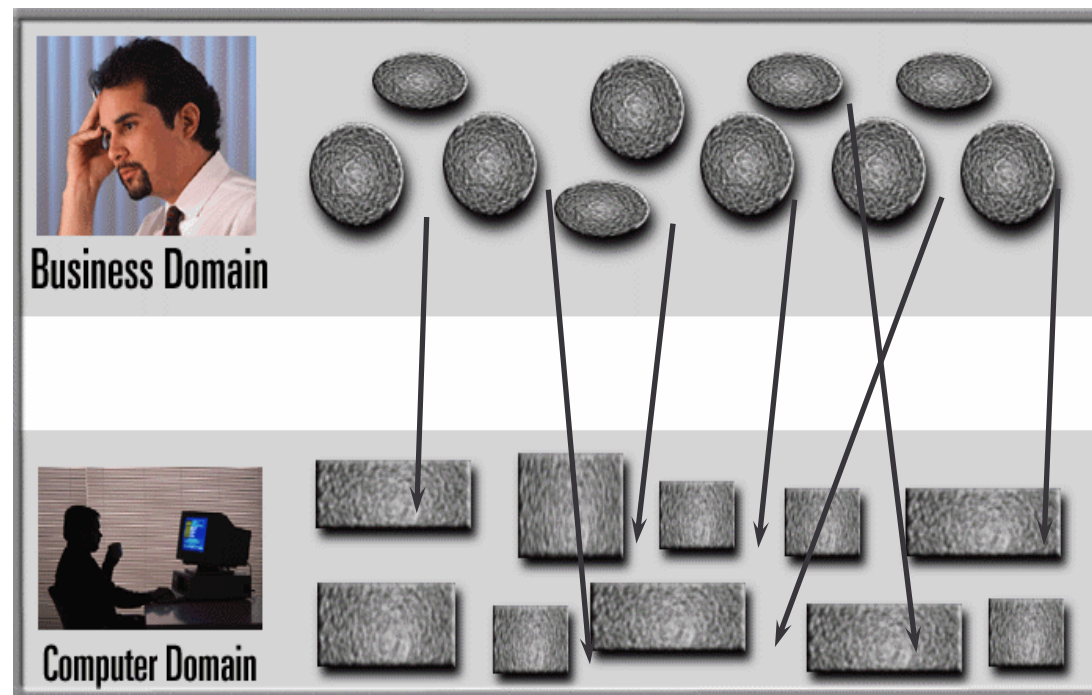
Modeling Captures Business Process

Use Case is a technique to capture business process from user's perspective



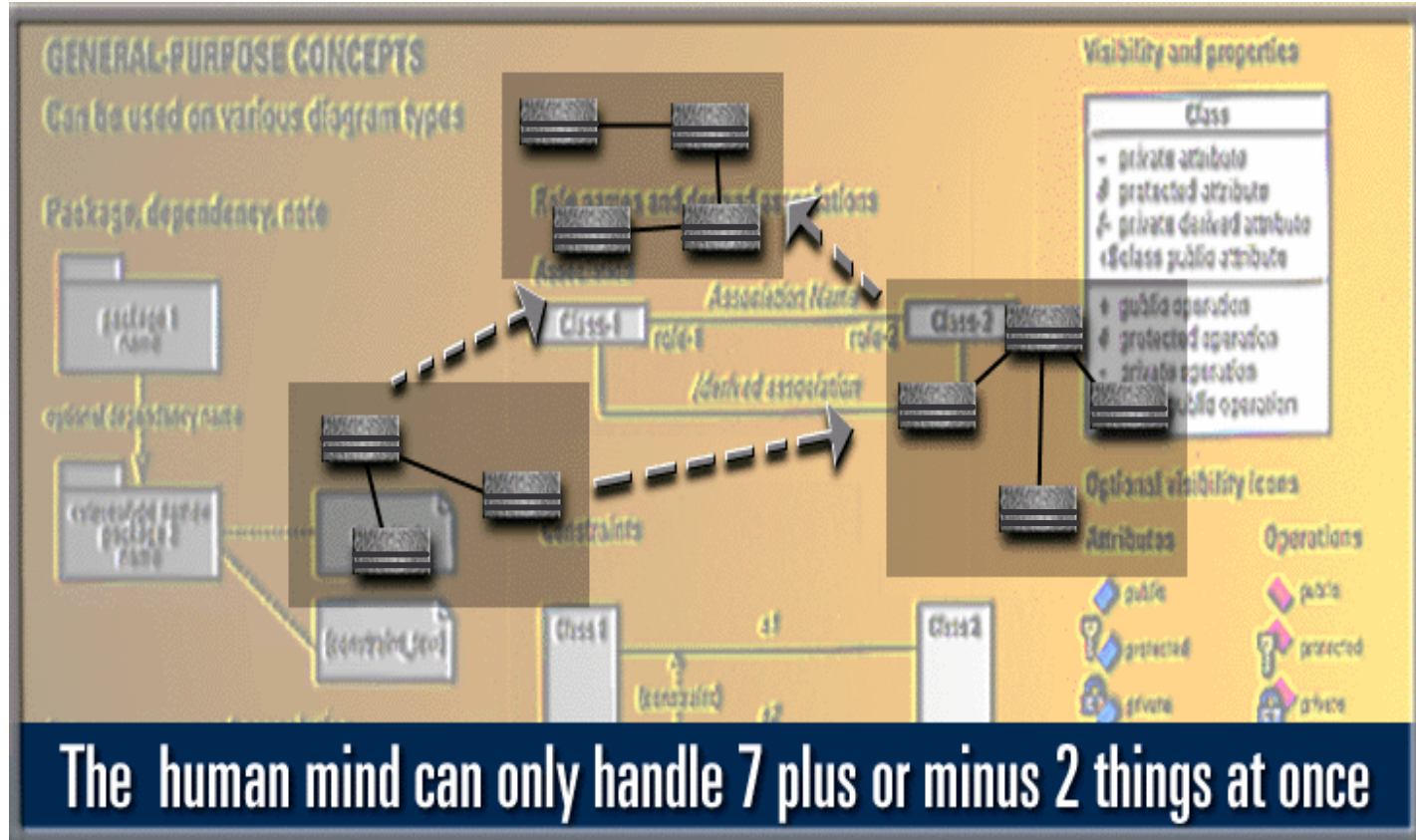
Visual Modeling is a Communication Tool

Use visual modeling to capture business objects and logic

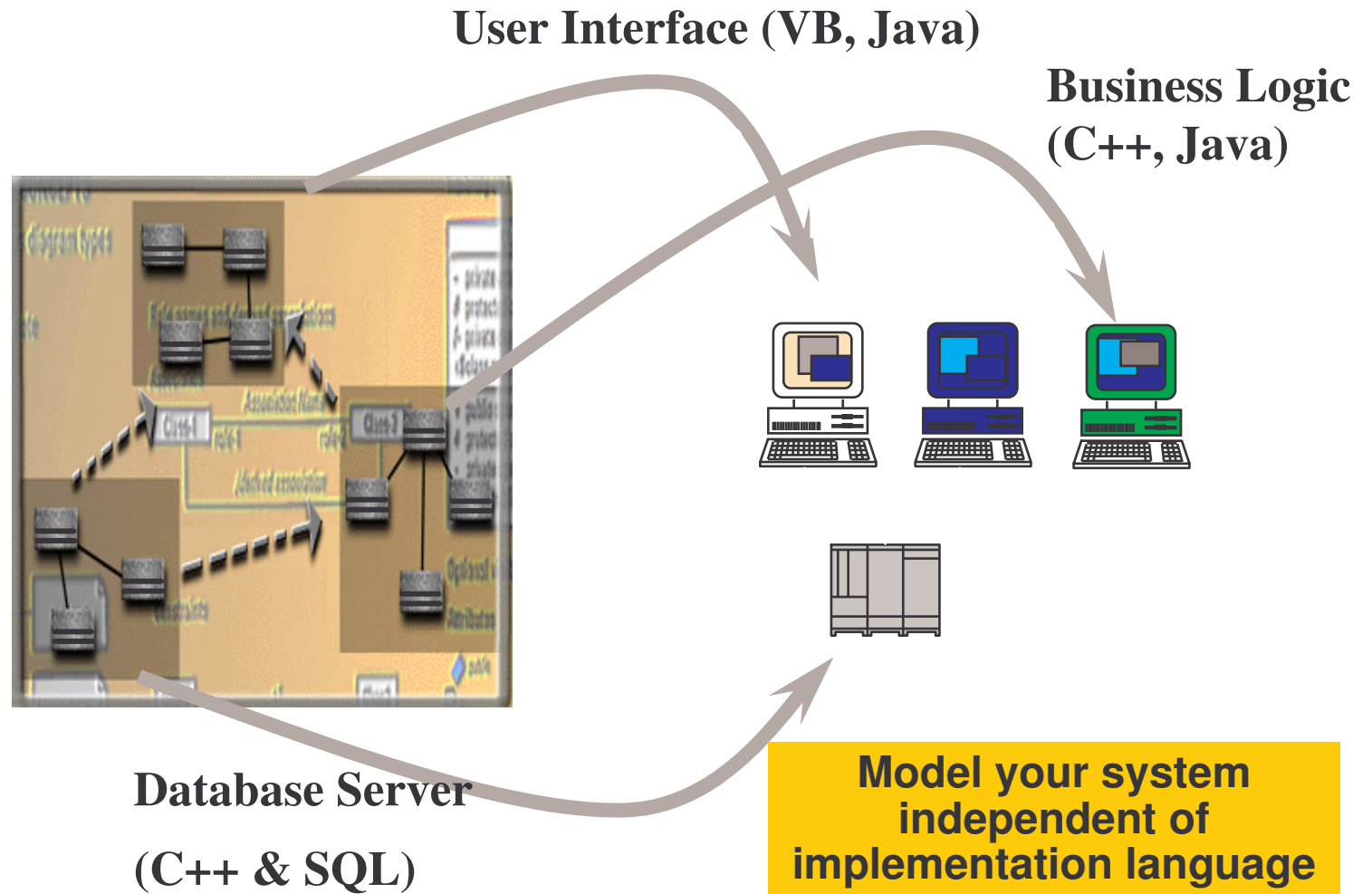


Use visual modeling to analyze and design your application

Visual Modeling Manages Complexity

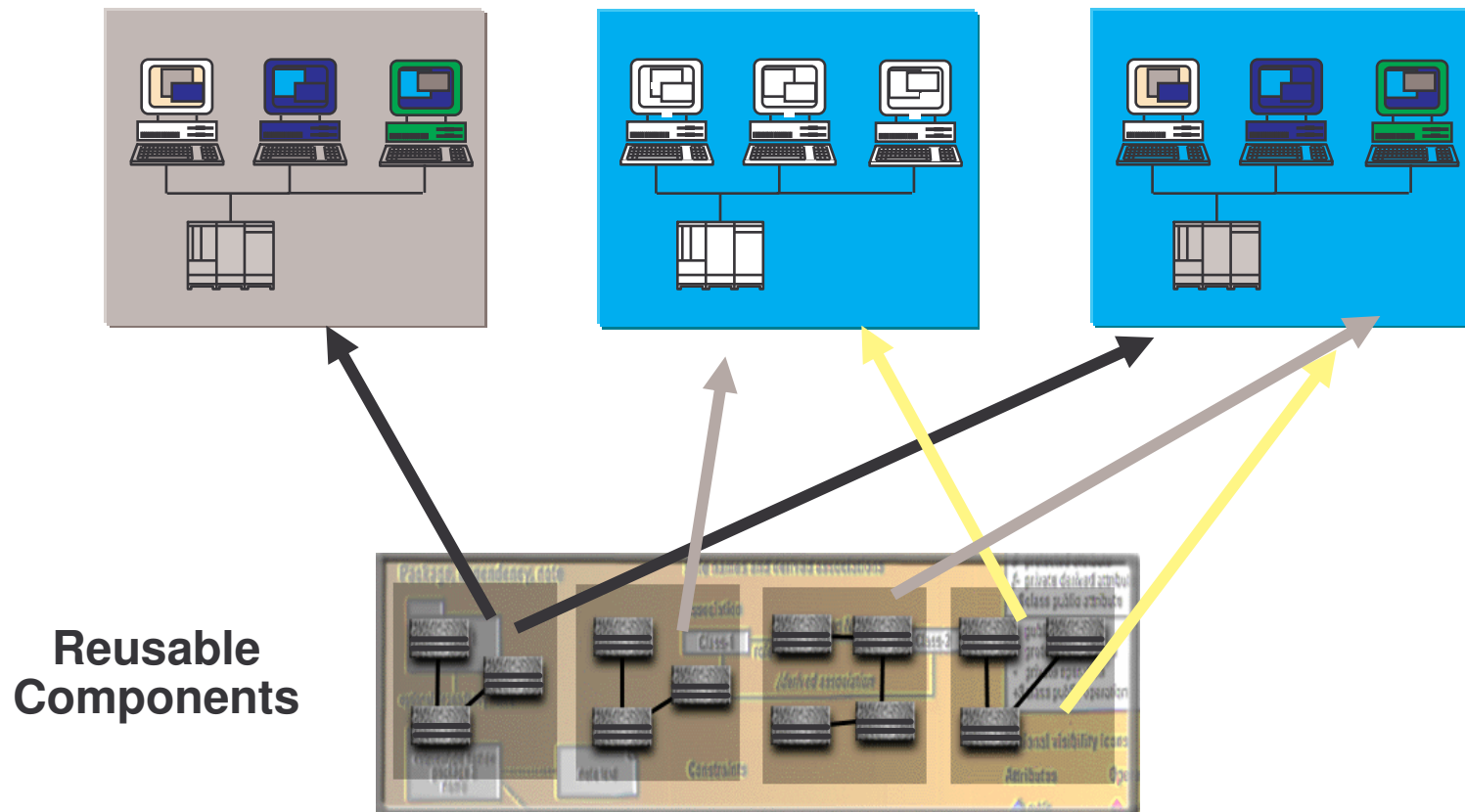


Modeling Defines Software Architecture



Modeling Promotes Reuse

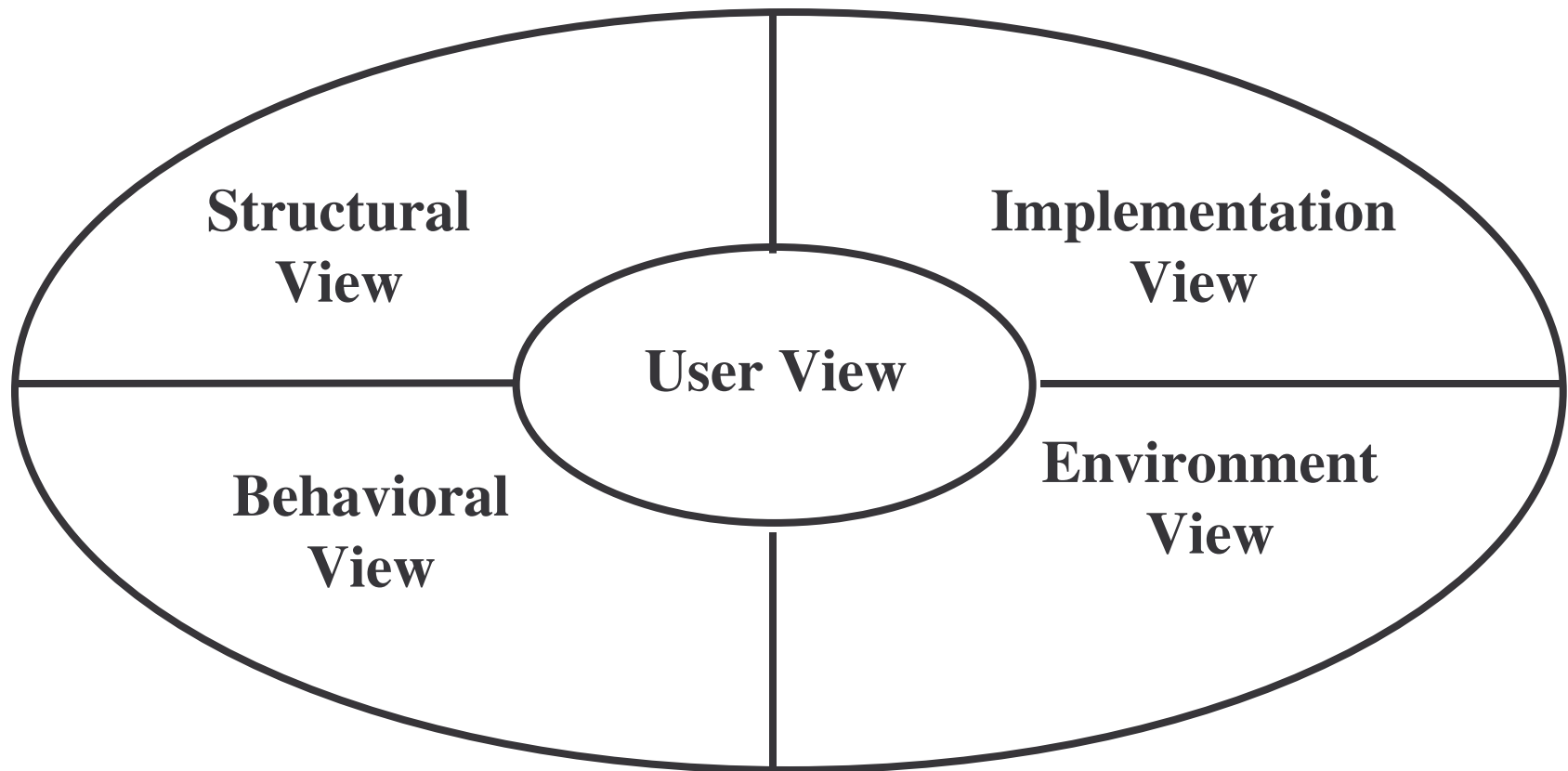
Multiple Systems



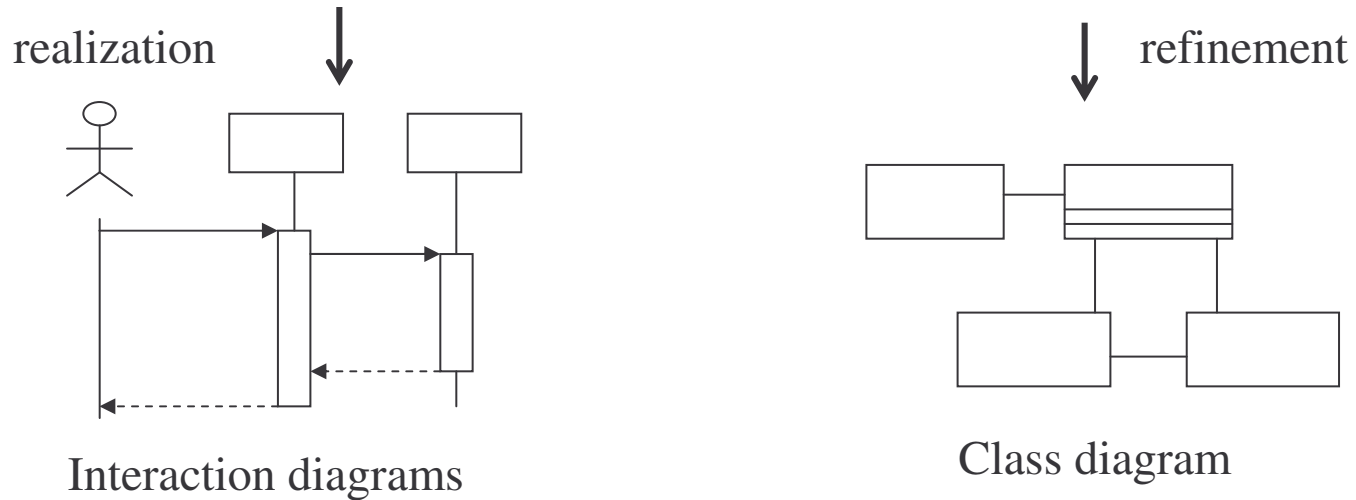
What is the UML?

- UML stands for Unified Modeling Language
- The UML is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system
- It can be used with all processes, throughout the development life cycle, and across different implementation technologies.
- There are over 12 kinds of diagrams in UML2.0 now.

UML Multi-view Modeling: 4+1 Views

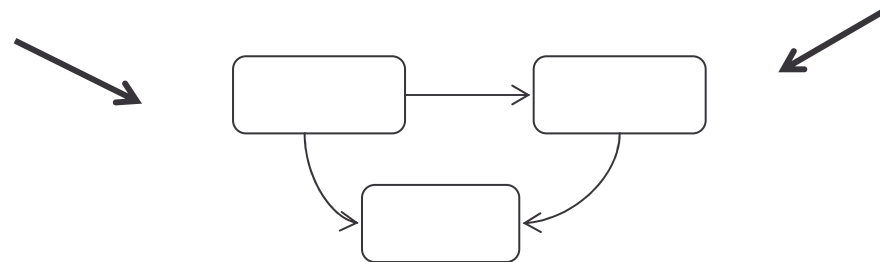


Realization and Refinement with Use Cases



Interaction diagrams

Class diagram



Statecharts

UML Concepts

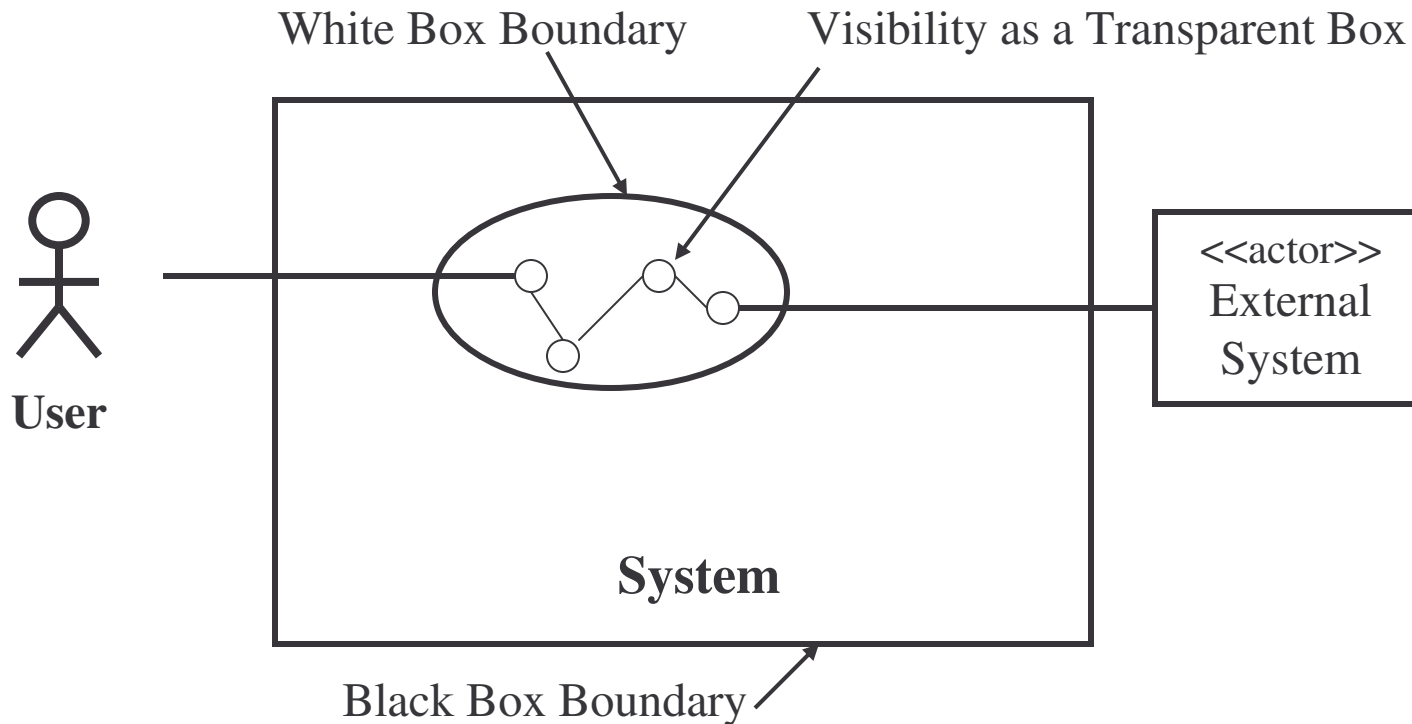
- The UML may be used to:
 - Display the boundary of a system & its major functions using use cases and actors
 - Illustrate use case realizations with interaction diagrams
 - Represent a static structure of a system using class diagrams
 - Model the behavior of objects with state transition diagrams
 - Reveal the physical implementation architecture with component & deployment diagrams
 - Extend your functionality with stereotypes

System

- Three kinds of systems: natural systems, social systems and artificial systems
- Systems are organized collections of interacting and connected components cooperating to accomplish a purpose. These components are called system elements.
- The UML is system independent. It may be used to describe different types of systems (hardware, software systems as well as human interaction systems).

System

A system can be understood by three basic models: black box, white box, and transparent box.



Systems, Models, and Views

- A *model* is an abstraction describing system or a subset of a system
- A *view* depicts selected aspects of a model
- A *notation* is a set of graphical or textual rules for representing views
- Views and models of a single system may overlap each other

Model

- *Models* are complete abstractions of systems. Or Models are simplifications of reality.

A dog house—a house for a family— a high-rise building

- Curiously, a lot of software development organizations start out wanting to build high buildings but approach the problem as if they were knocking out a dog house.
- We build models so that we can better understand the system we are developing. We build models of complex systems because we cannot comprehend such a system in its entirety.

Why model software?

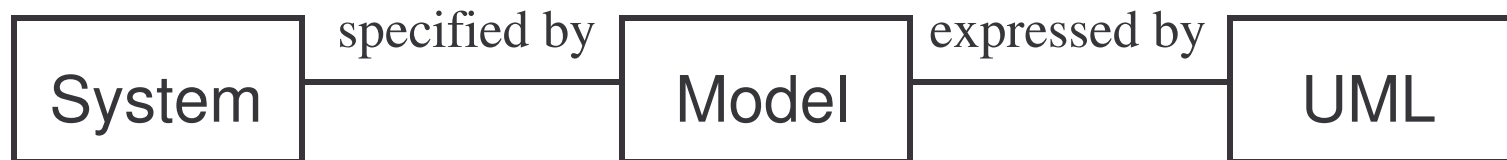
Software is already an abstraction: why model software?

- Software is getting larger, not smaller
 - NT 5.0 ~ 40 million lines of code
 - A single programmer cannot manage this amount of code in its entirety.
- Code is often not directly understandable by developers who did not participate in the development
- We need simpler representations for complex systems
 - Modeling is a mean for dealing with complexity

Diagram and Language

- Diagrams are graphical projections of sets of model elements.
- Languages are means of expressing and communicating content or information.
 1. *Notations and syntax*
 2. *Concepts and semantics*

Relationship among System, Model and UML

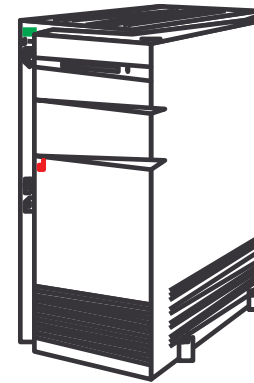
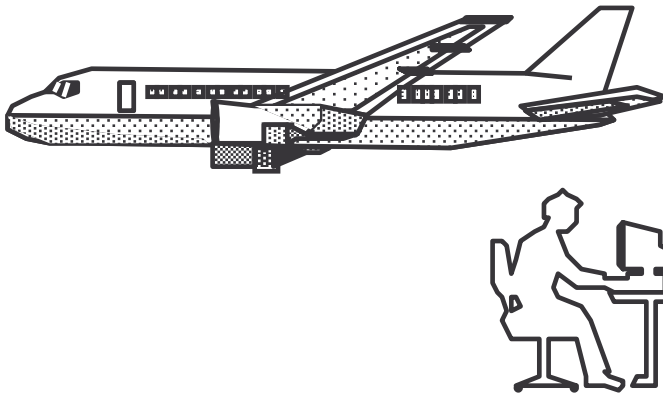


4: Use Case Model

Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
- 4. *Use Case Model***
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

Object-Oriented Modeling

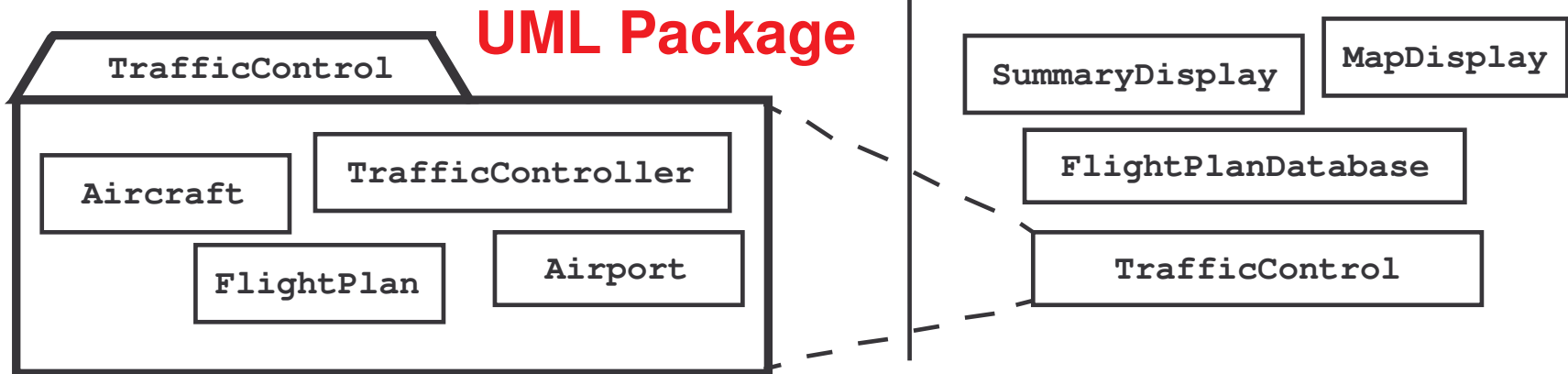


Application Domain

Solution Domain

Application Domain Model

System Model

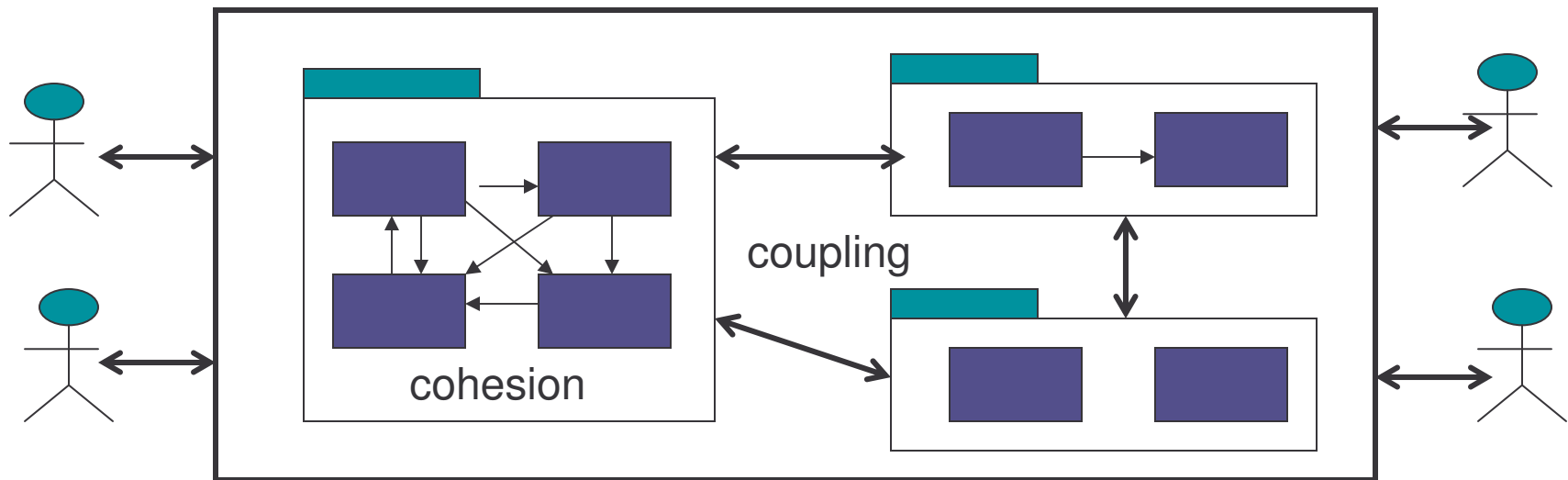


Application and Solution Domain

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

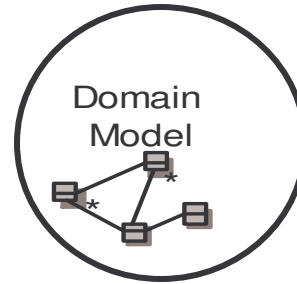
Divide and Conquer Principle

- Vertical decomposition (layer architecture)
- Horizontal decomposition (subsystem)
- Dynamic and Static views
- Low coupling and high cohesion



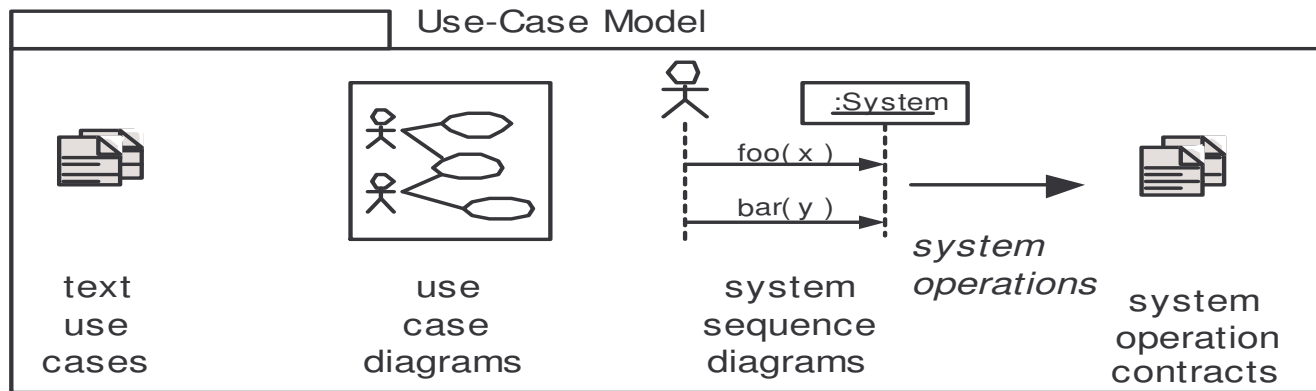
Artifacts in the Use-Case Model

Business Modeling

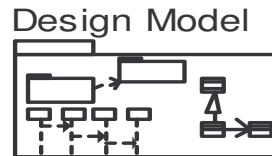


Partial artifacts, refined in each iteration.

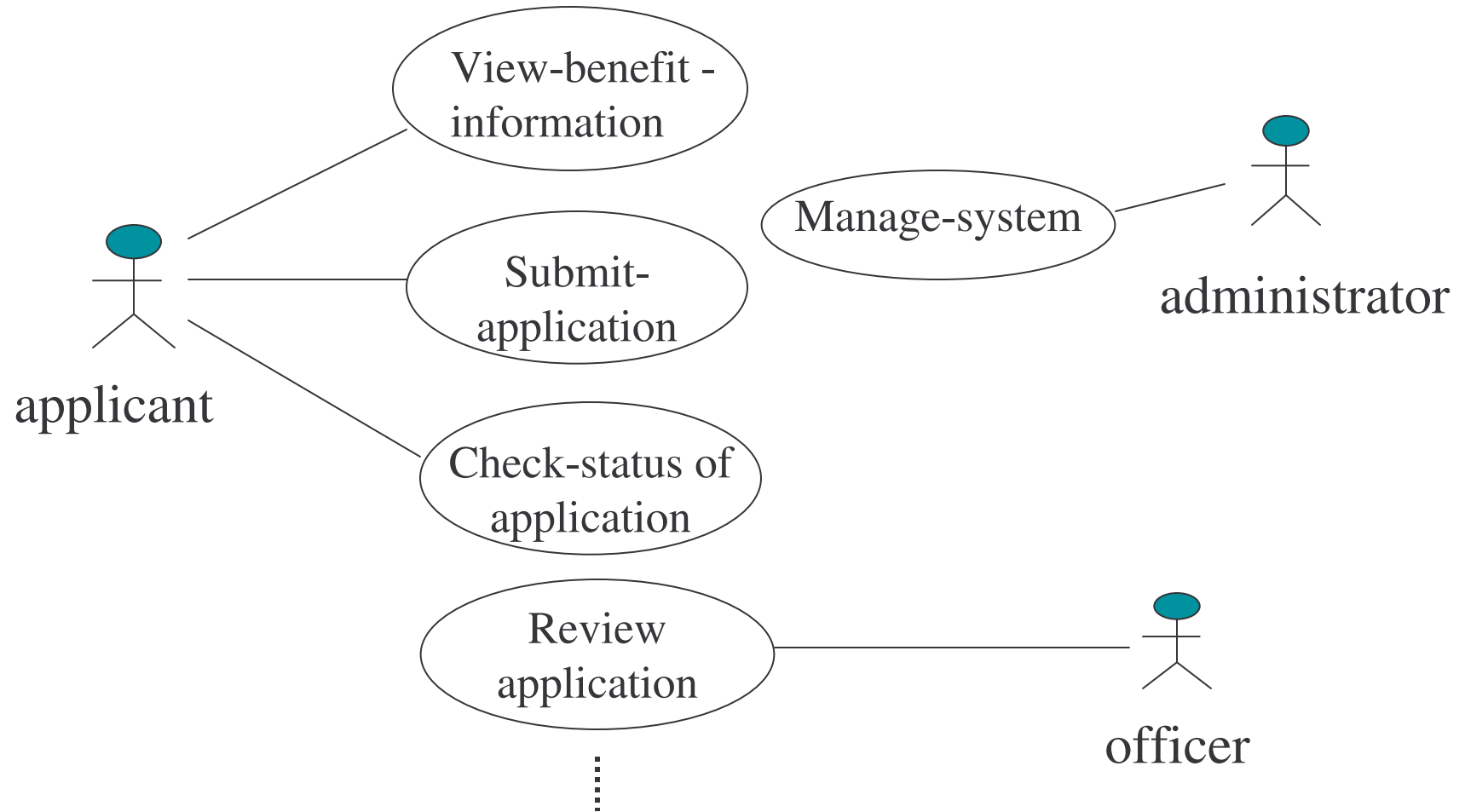
Requirements



Design



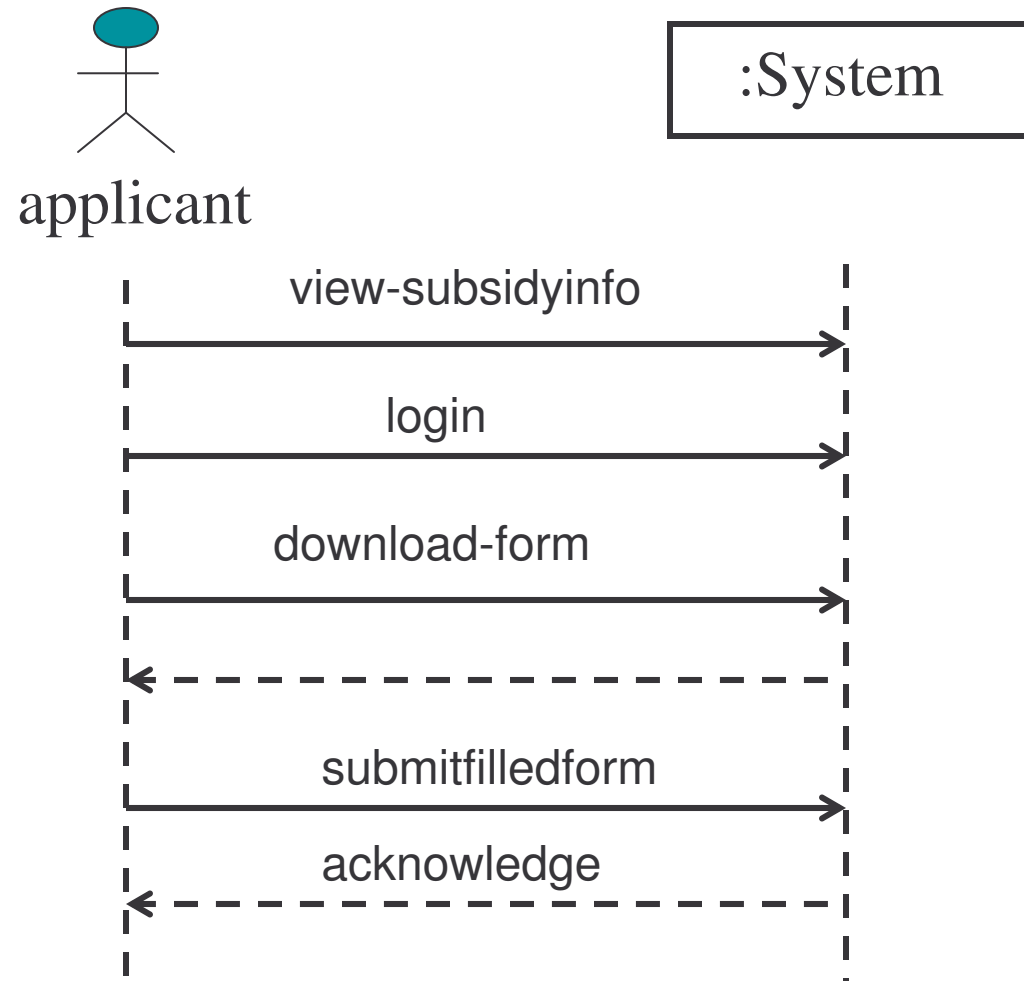
Partial Use Case Diagram of Benefit Application System



Description of Use Case

- A flow of events document is created for each use cases
 - Written from an actor point of view
- Details what the system must provide to the actor when the use cases is executed
- Typical contents
 - How the use case starts and ends
 - Normal flow of events
 - Alternate flow of events
 - Exceptional flow of events

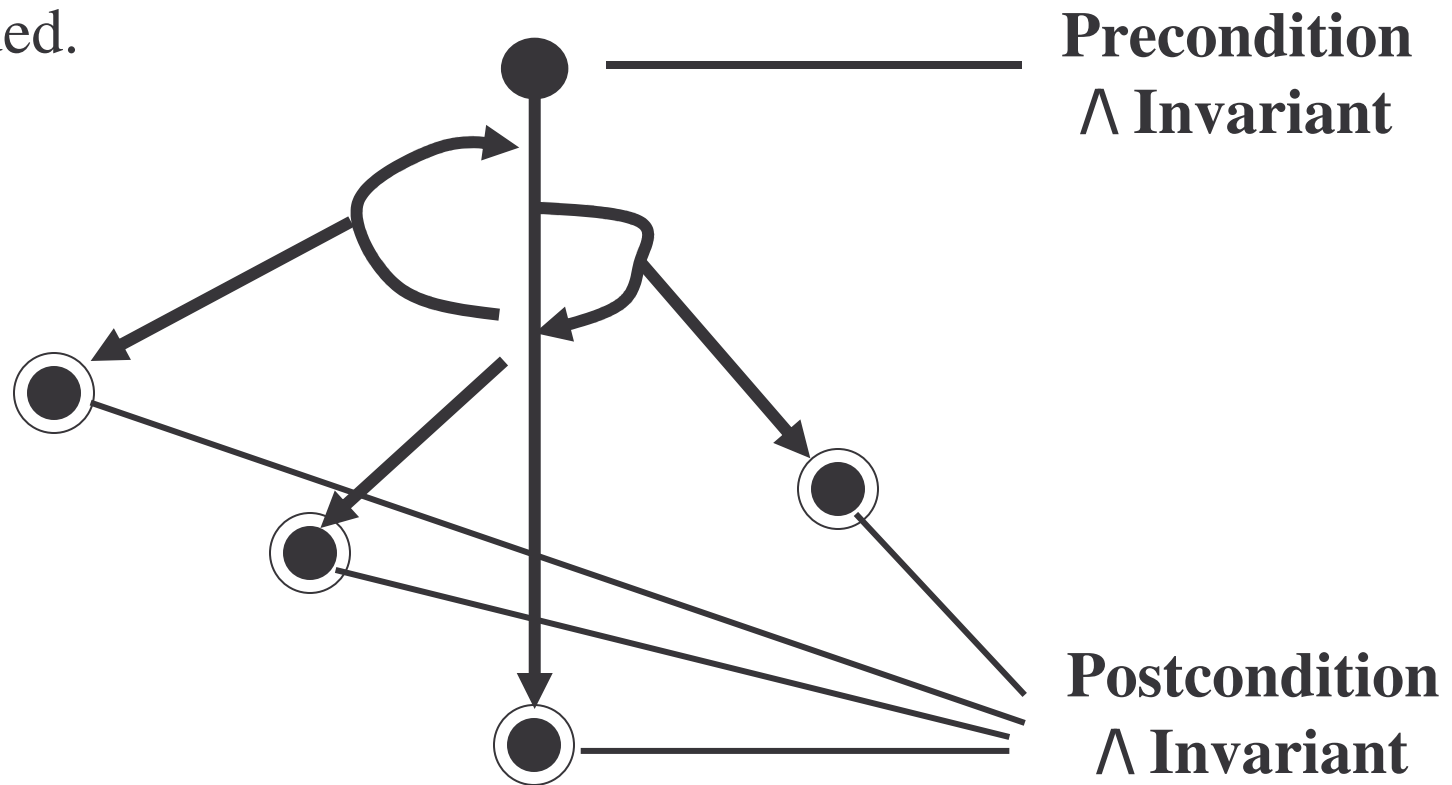
System Sequence Diagram of Use Case *Submitapplication*



Precondition and Postcondition

A **precondition** is the state of the system and its surroundings that is required before the use case (operation) can be started.

A **postcondition** is the state the system can be after the use case has ended.



5. Domain Model

Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
- 5. Domain Model***
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

Definition & Motivation: Domain Model

- A *Domain Model* visualizes, using UML class diagram notation (conceptual class diagram), noteworthy concepts or objects.
 - It is a kind of “visual dictionary.”
 - *Not* a picture of software classes.
- It helps us identify, relate and visualize important information.
- It provides inspiration for later creation of software design classes, to reduce “representational gap.”

Concepts and Phenomena

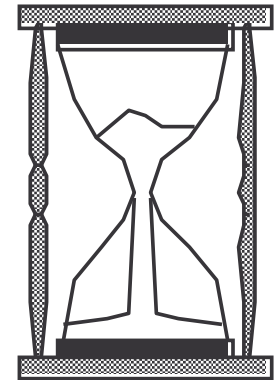
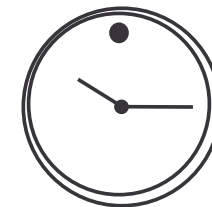
Name

Purpose

Members

Clock

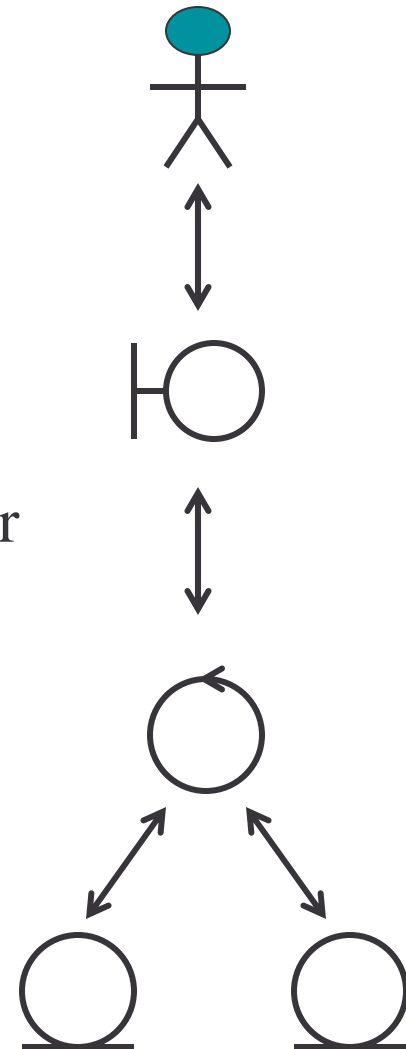
A device that
measures time.



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

Different Object Types

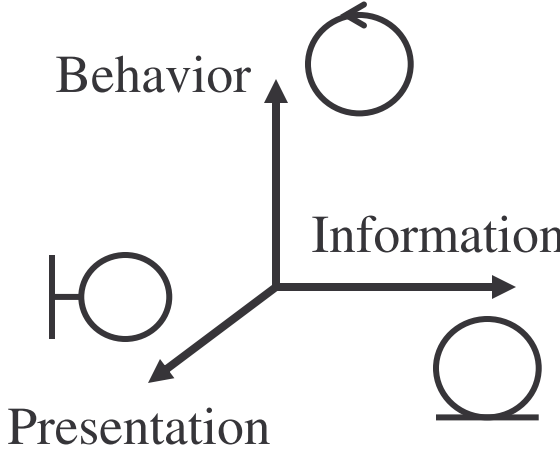
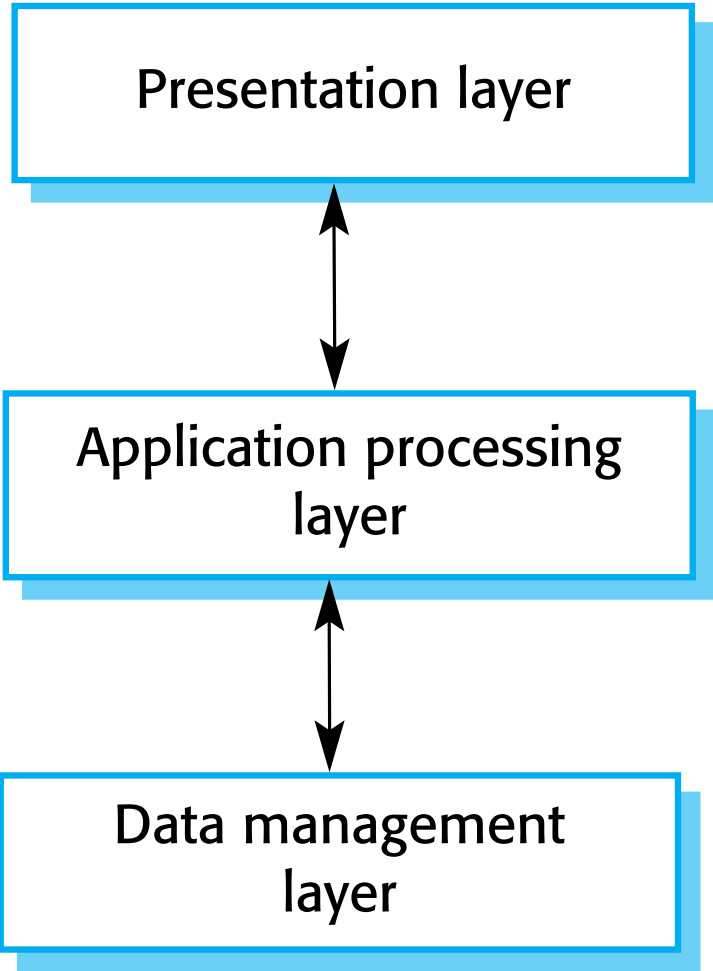
- Boundary objects
 - Implement the interaction with the user
 - Constructed from UI components
 - Subject to most modification
- Control objects
 - Implement the transactions with the user
 - Constructed with Command objects
 - Modified frequently but less often than boundary objects
- Entity objects
 - Represent the domain model
 - Often represent persistent data
 - Least often modified



Layered application architecture

- Presentation layer
 - Concerned with presenting the results of a computation to system users and with collecting user inputs.
- Application processing layer
 - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management layer
 - Concerned with managing the system databases.

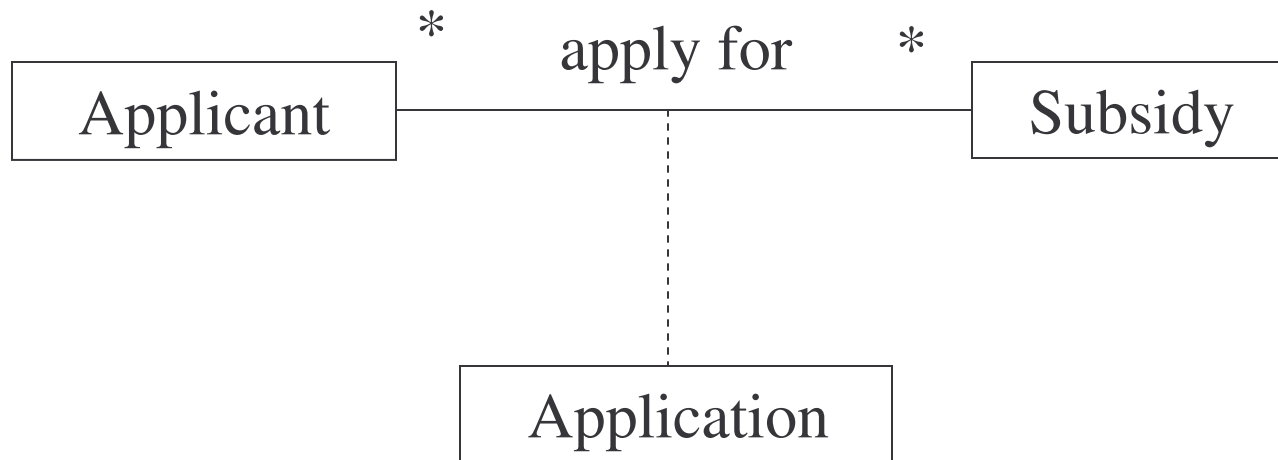
Application layers



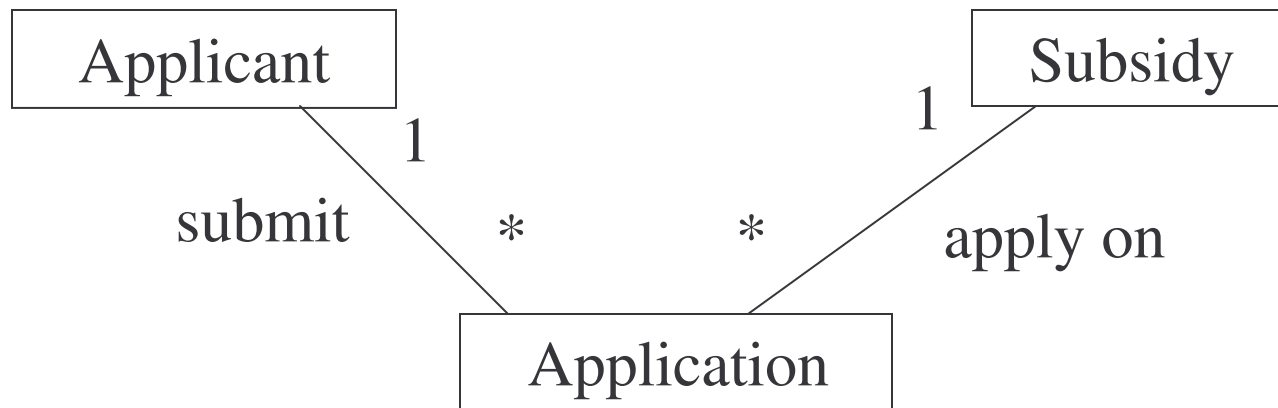
Domain Model of Benefit Application System



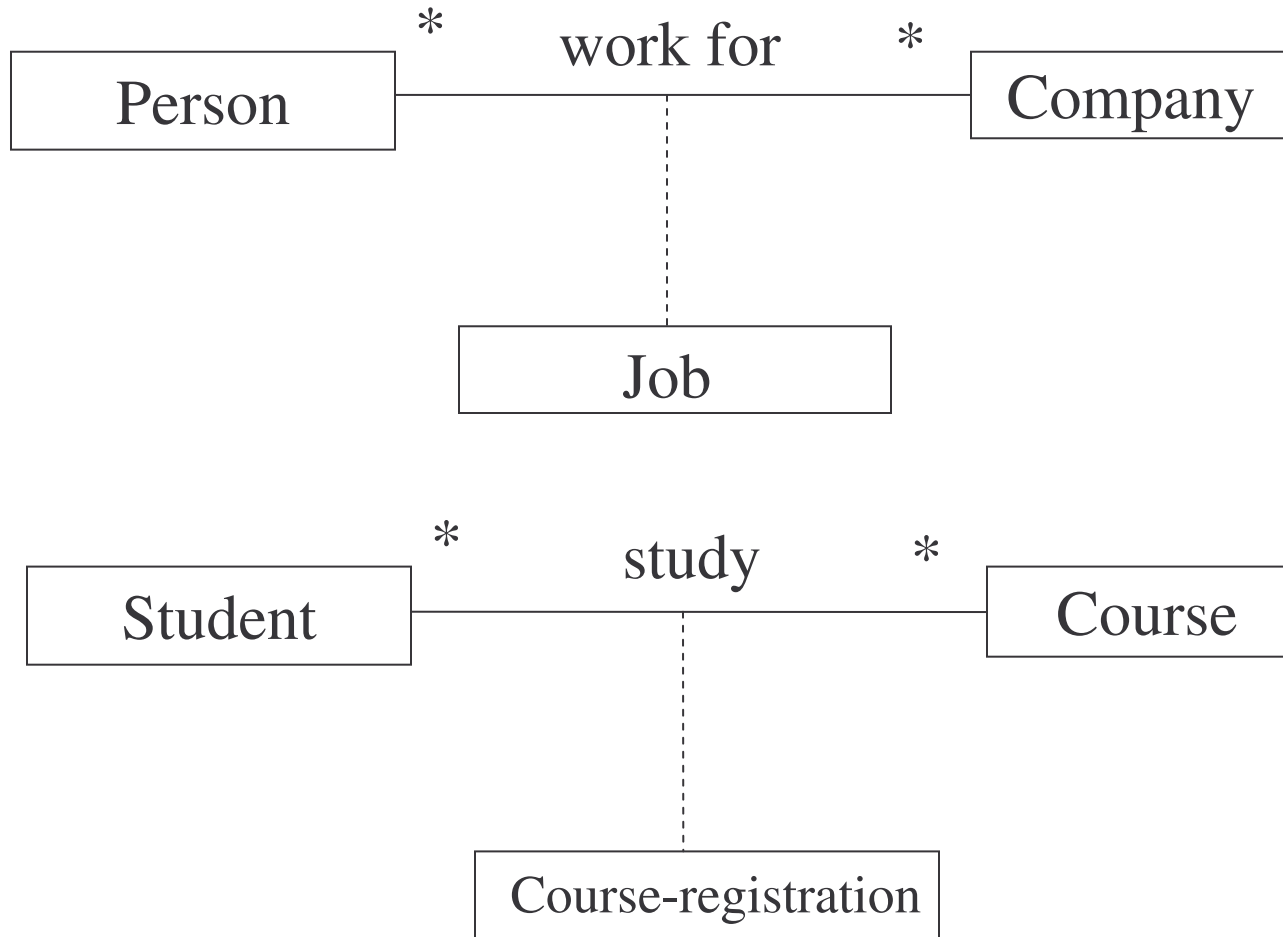
Domain Model of Benefit Application System



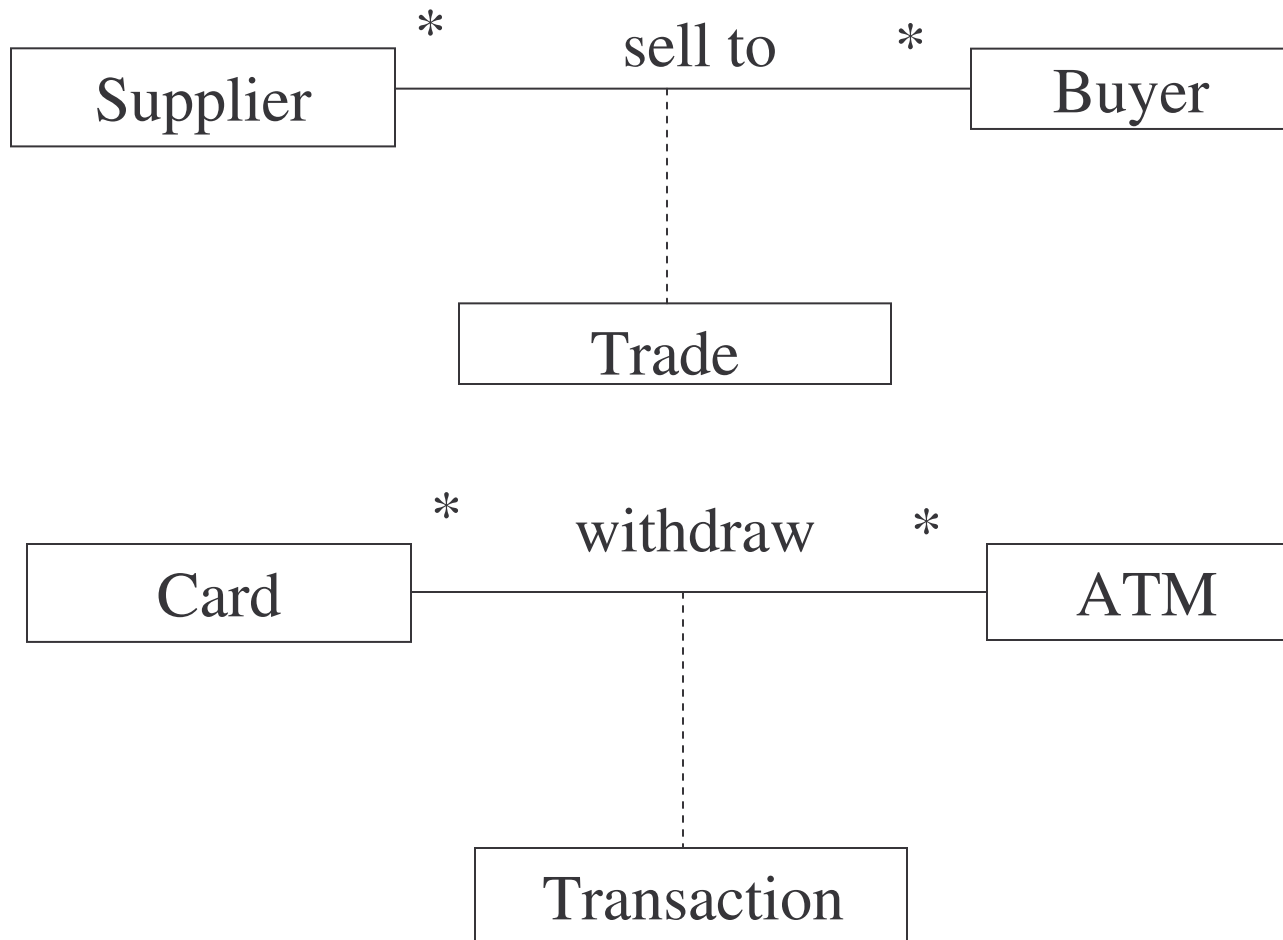
Domain Model of Benefit Application System



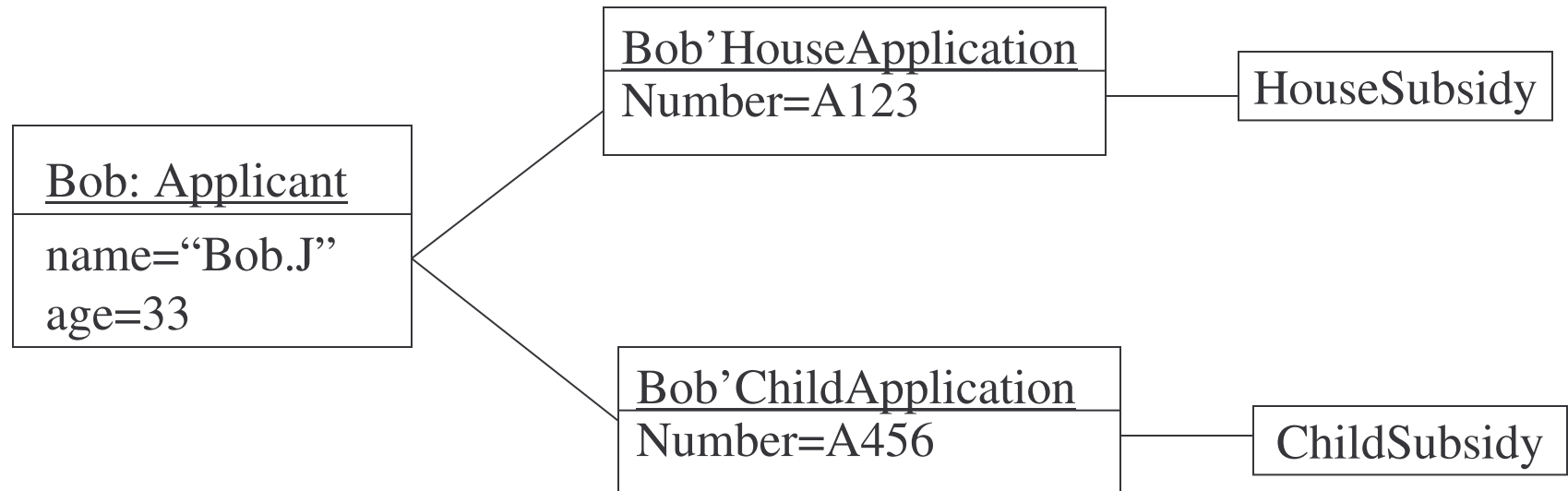
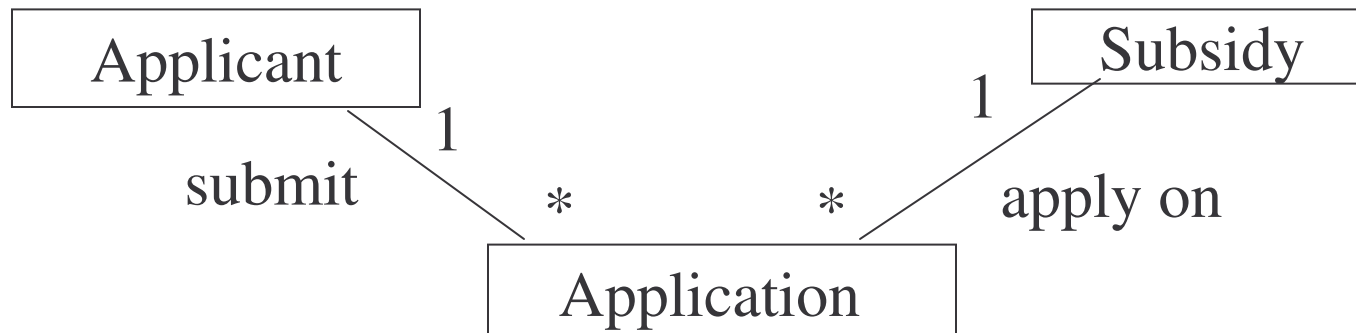
Domain Model with Association Class



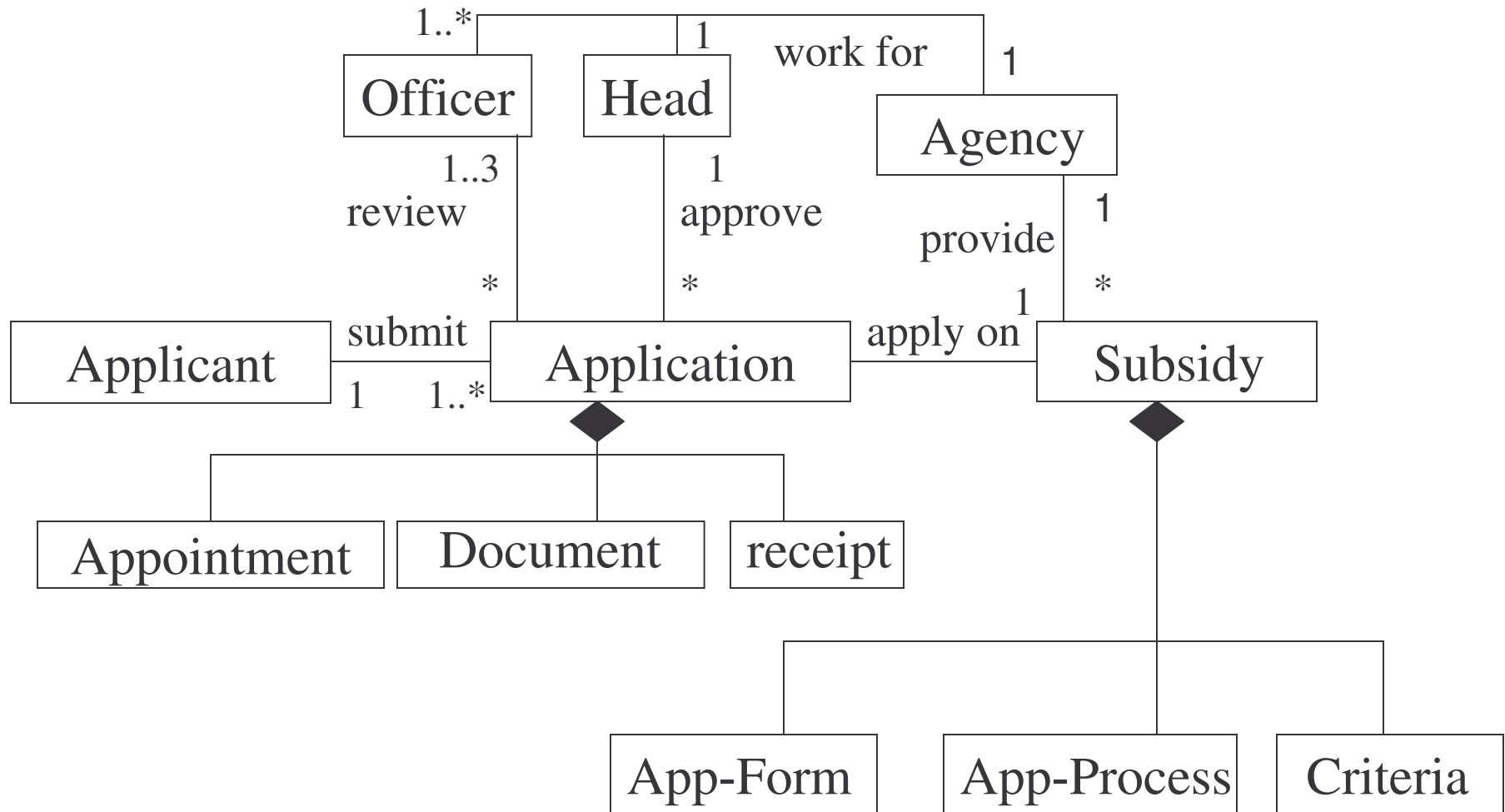
Domain Model with Association Class



Object and Class Diagrams



Conceptual Class Diagram of BAS



6: Object Design with Patterns

Contents

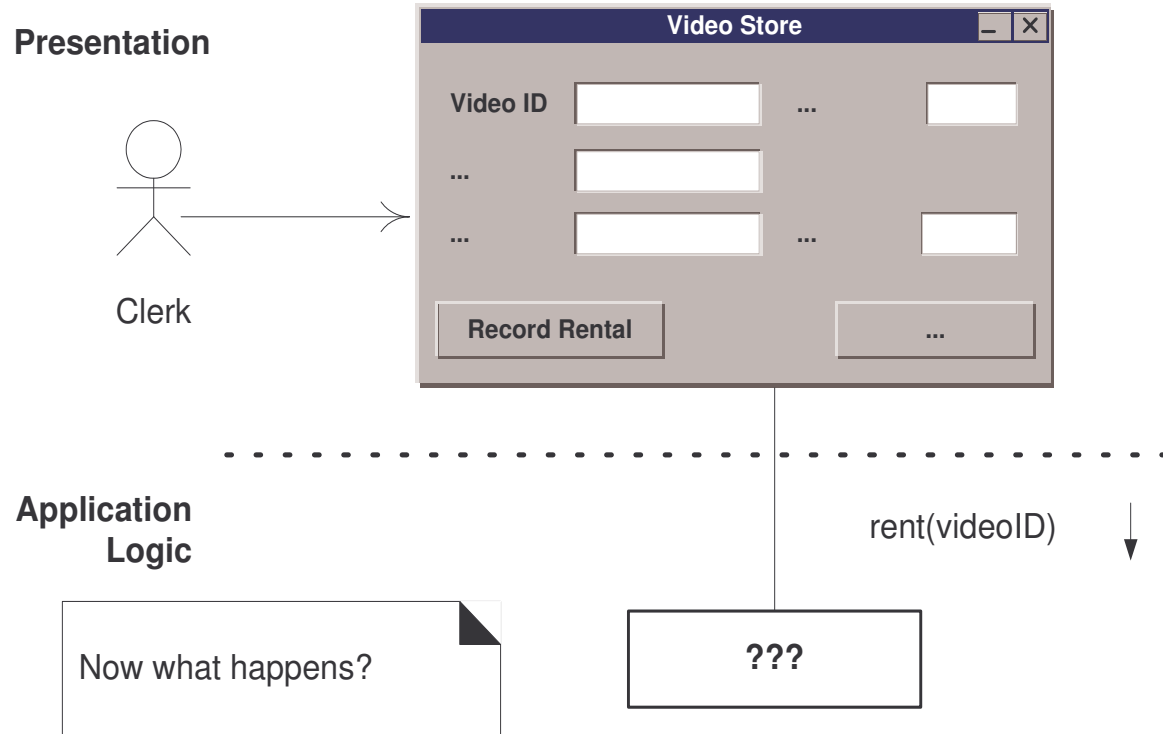
1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. *Object Design with Patterns*
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

Overview

- A critical skill is *designing or thinking in objects*.
- You will be able to:
 - Apply Expert, Creator, Controller, Low Coupling, High Cohesion
 - Design for low representational gap
 - Define use case realizations
 - Relate the UP artifacts

Motivation: Introduction

- Now what happens?
- What object should receive this message?
- What objects should interact to fulfill the request, and how?



Responsibility-Driven Design (RDD)

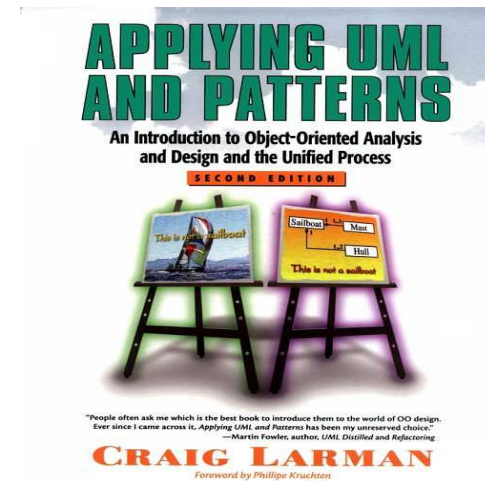
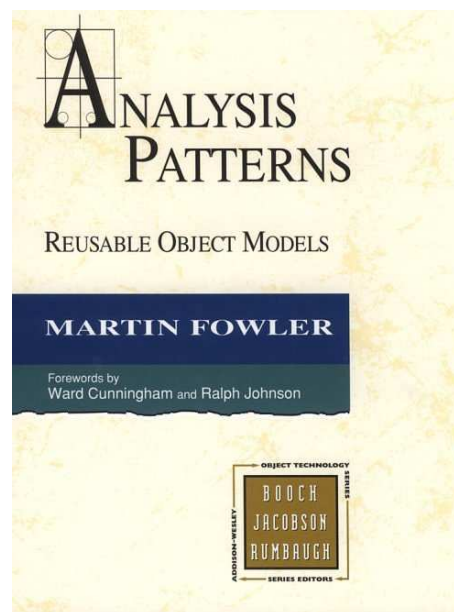
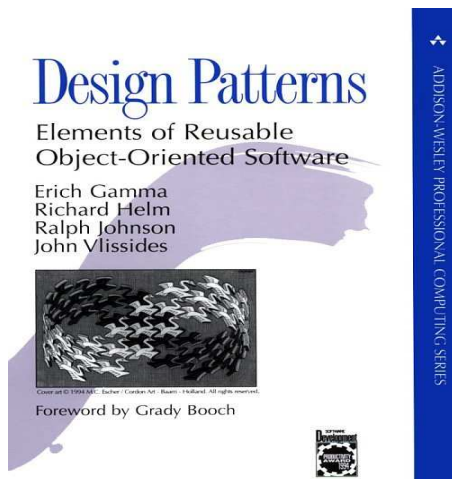
- Detailed object design is usually done from the point of view of the *metaphor* of:
 - Objects have responsibilities
 - Objects collaborate
- In RDD we do object design such that we will ask questions such as:
 - What are the responsibilities of this object?
 - Who does it collaborate with?

Definition: Responsibilities

- Responsibilities are an abstraction.
 - The responsibility for persistence.
 - Large-grained responsibility.
 - The responsibility for the sales tax calculation.
 - More fine-grained responsibility.
- They are implemented with methods in objects.
 - 1 method in 1 object
 - 5 methods in 1 object
 - 50 methods across 10 objects

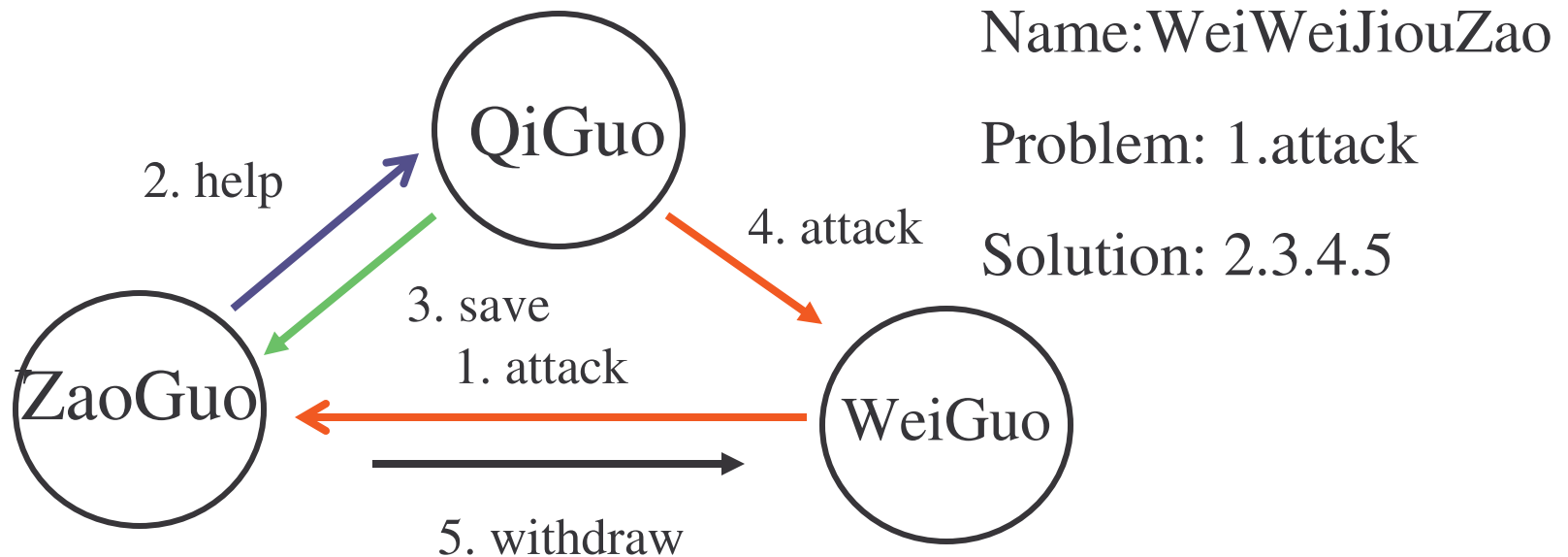
Patterns

- Patterns are *named* problem-solution pairs to common problems, typically showing a popular, robust solution.
 - “Façade” “Information Expert” ...
- They provide a *vocabulary* of design.



Patterns

All patterns have *names*. The **problem** description states when the pattern is used and which problem it tries to solve. The **solution** is described as a number of classes and objects, their structure, and dynamic collaboration.



Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - **GRASP patterns**
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - Summary

GRASP Patterns

- What guiding principles to help us assign responsibilities?
- These principles are captured in the **GRASP** patterns.
 - **General Responsibility Assignment Software Patterns.**
 - Very fundamental, basic principles of object design.

The 9 GRASP Patterns

1. Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

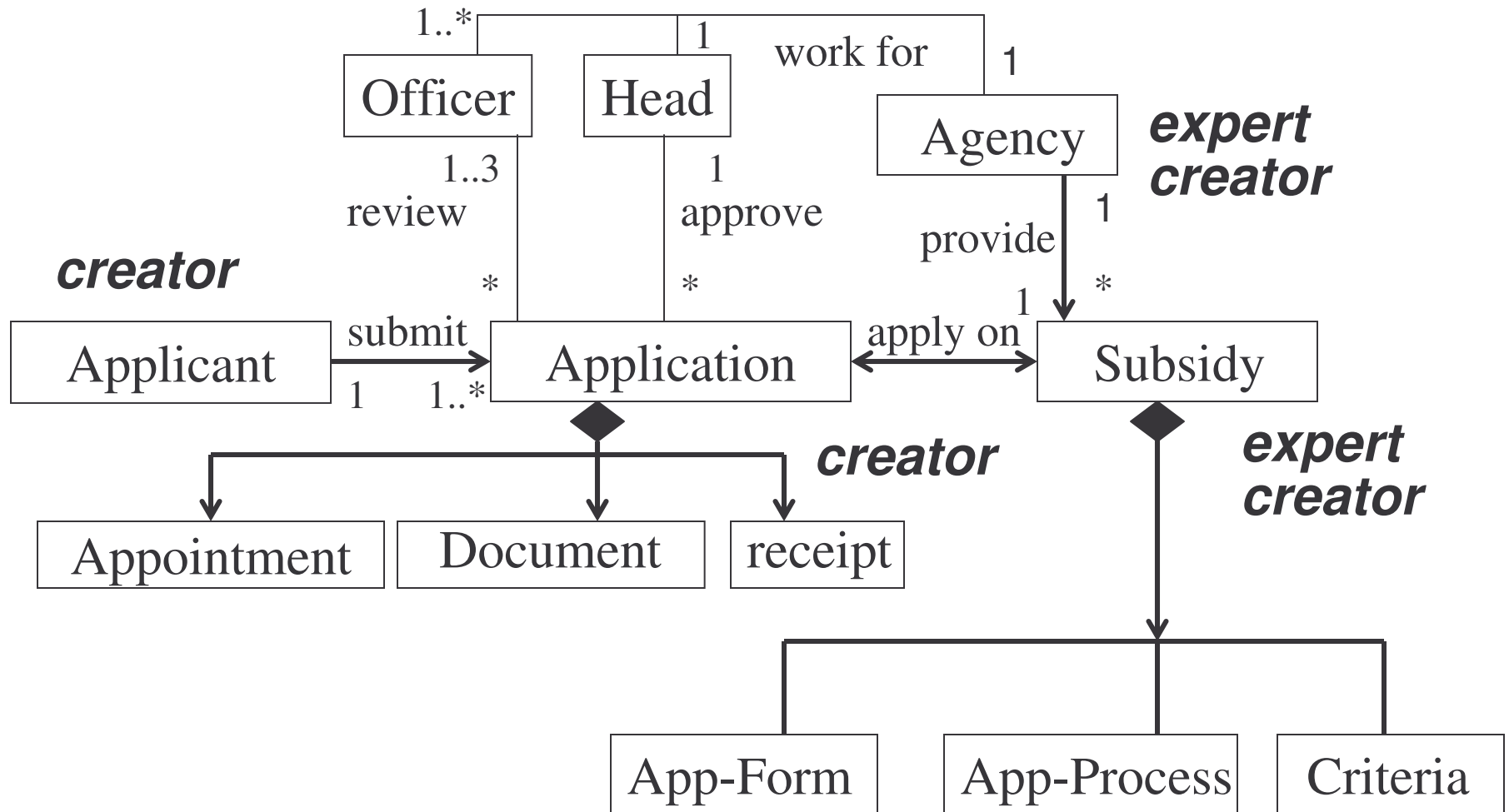
Information Expert

- What is most basic, general principle of responsibility assignment?
- Assign a responsibility to the object that has the information necessary to fulfill it.
 - “That which has the information, does the work.”
 - E.g., What software object calculates total benefits for applicant?
 1. What information is needed to do this?
 2. What object or objects has the majority of this information.

Creator Pattern

- What object creates an *X*?
 - Ignores special-case patterns such as *Factory*.
- Choose an object *C*, such that:
 - *C* contains or aggregates *X*
 - *C* closely uses *X*
 - *C* has the initializing data for *X*

Expert and Creator Patterns to BAS

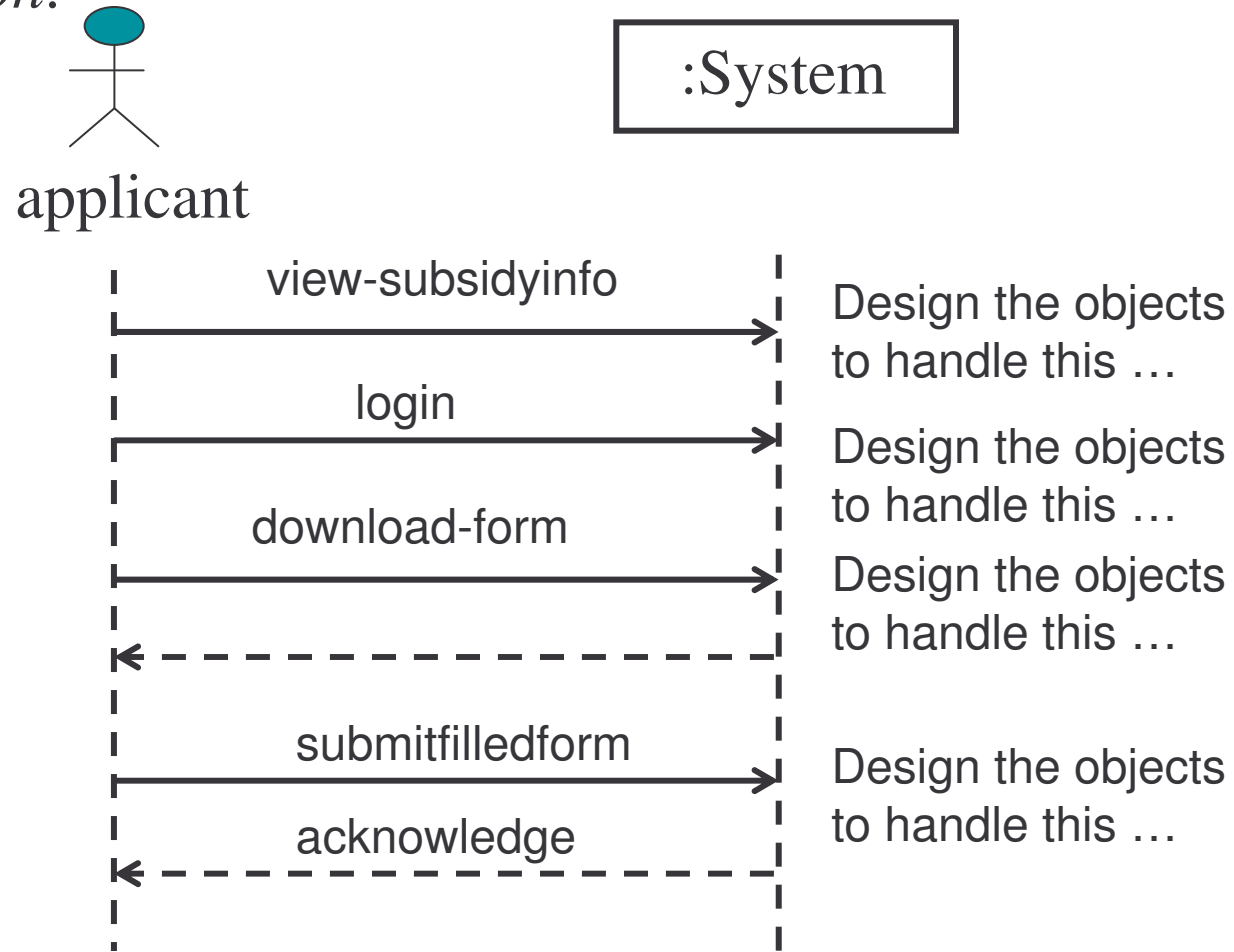


Controller Pattern

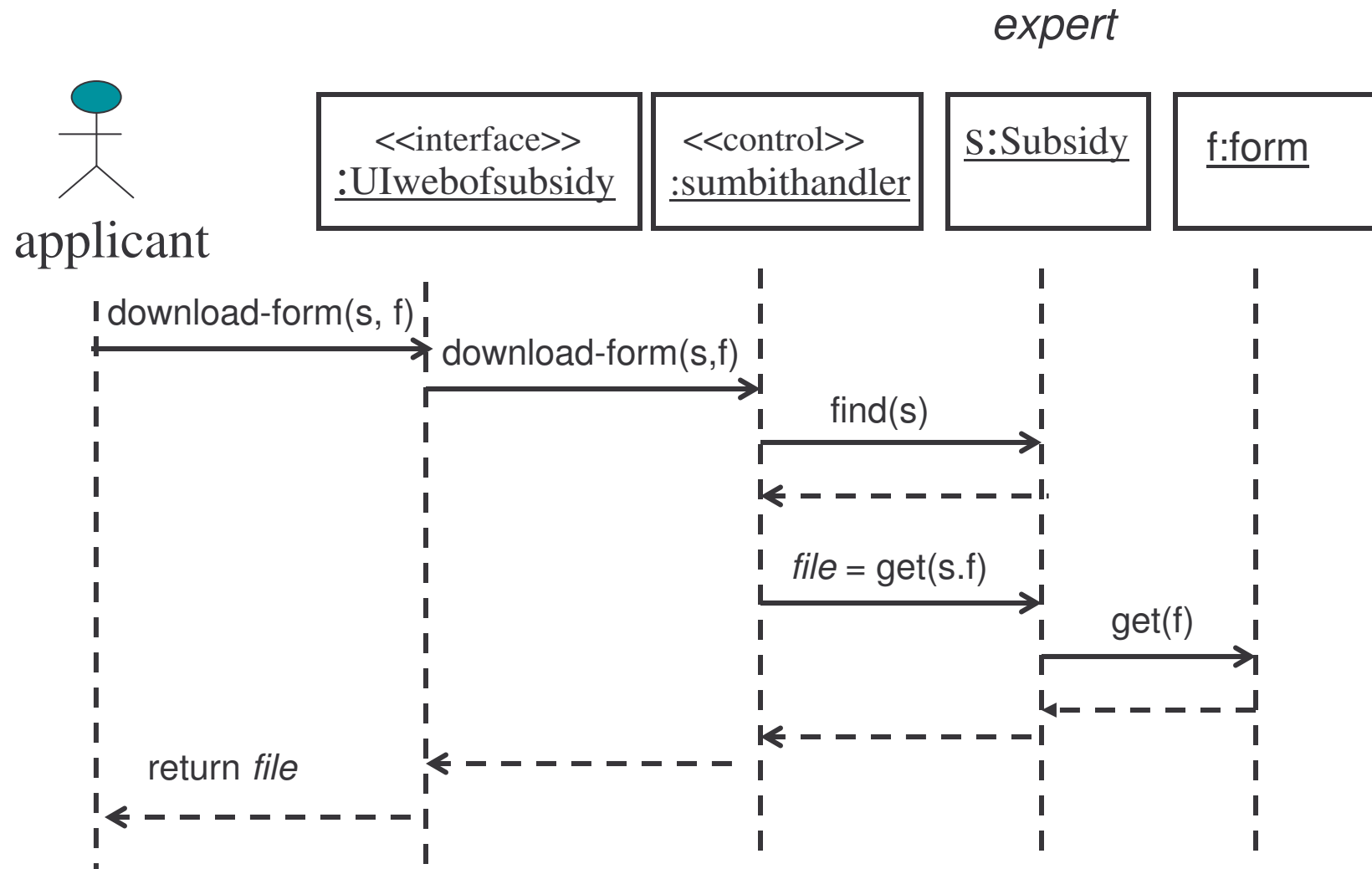
- What object in the domain (or application coordination layer) receives requests for work from the UI layer?
- Solution: Choose a class whose name suggests:
 - The overall “system,” device, or subsystem
 - A kind of Façade class
 - Or, represents the use case scenario or session

Use Case Realizations

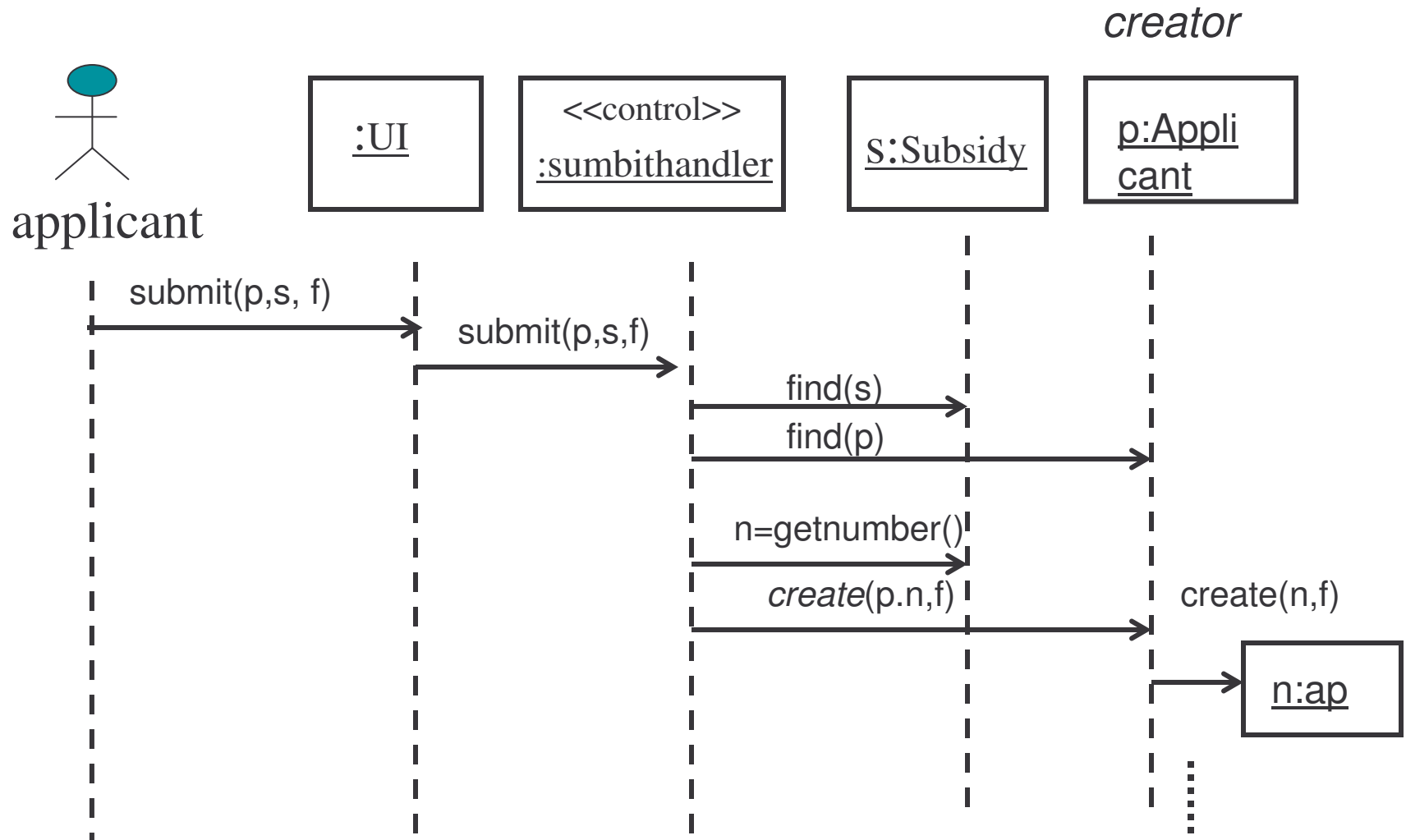
- When we design the objects to handle the systems operation for a scenario, we are designing a *use case realization*.



SubmitHandler: Download Operation



SubmitHandler: Summit Operation



PATTERN: Low Coupling

- Assign responsibilities so that coupling remains low.
- What does low coupling mean?

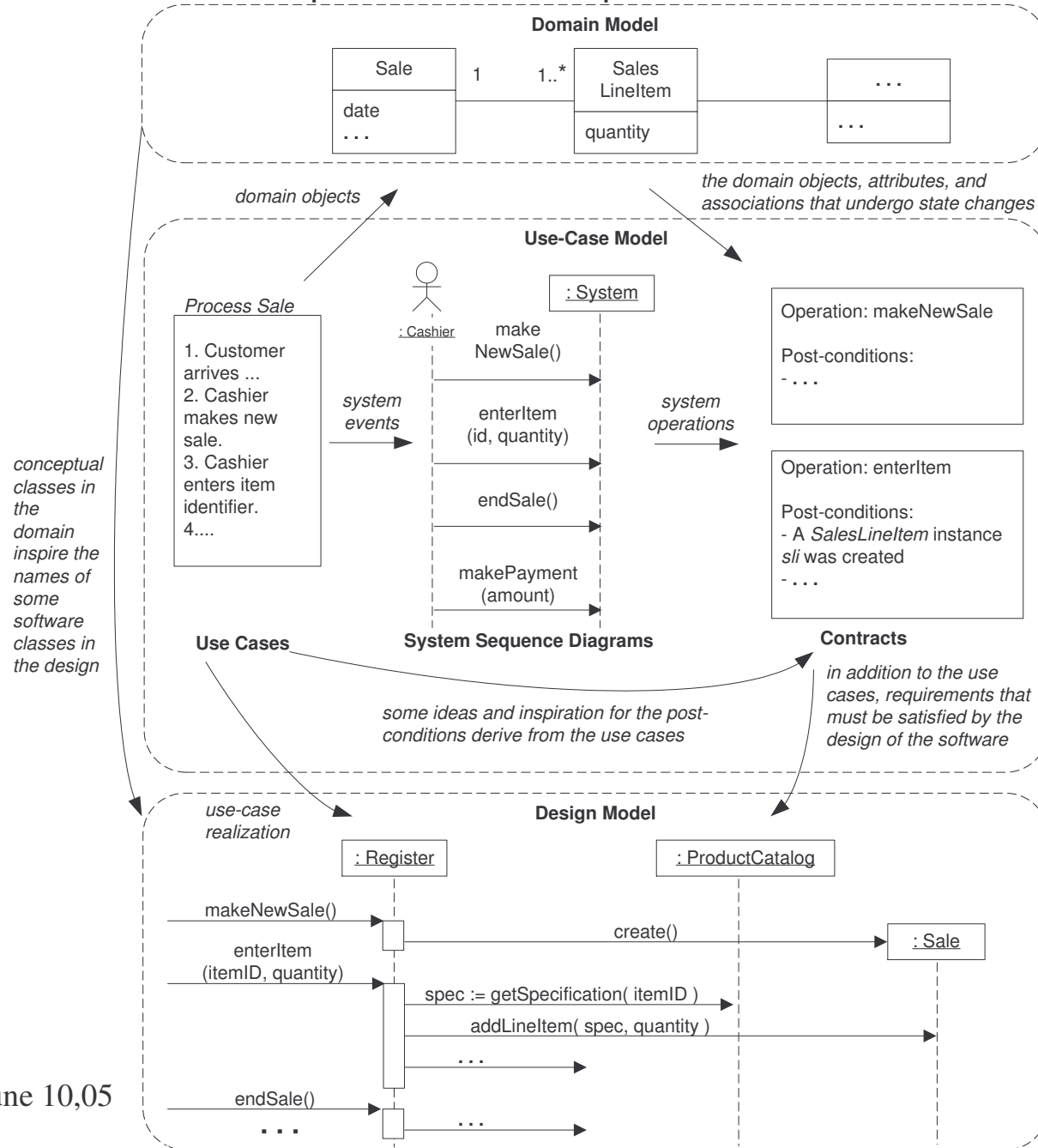


PATTERN: High Cohesion

- Assign responsibilities so that cohesion remains high?
- What does high cohesion mean?



Sample UP Artifact Relationships for Use-Case Realization



Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - **Business rules**
 - State and activity diagram
 - GoF's design patterns
 - Summary

Business Rule

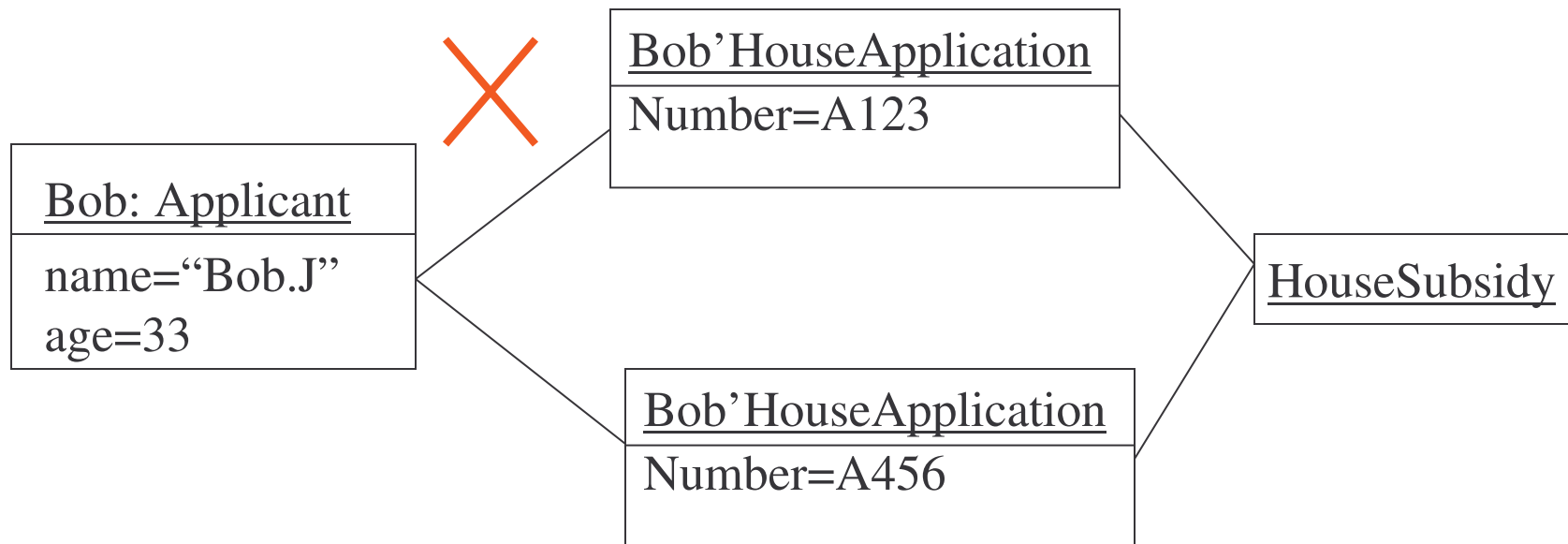
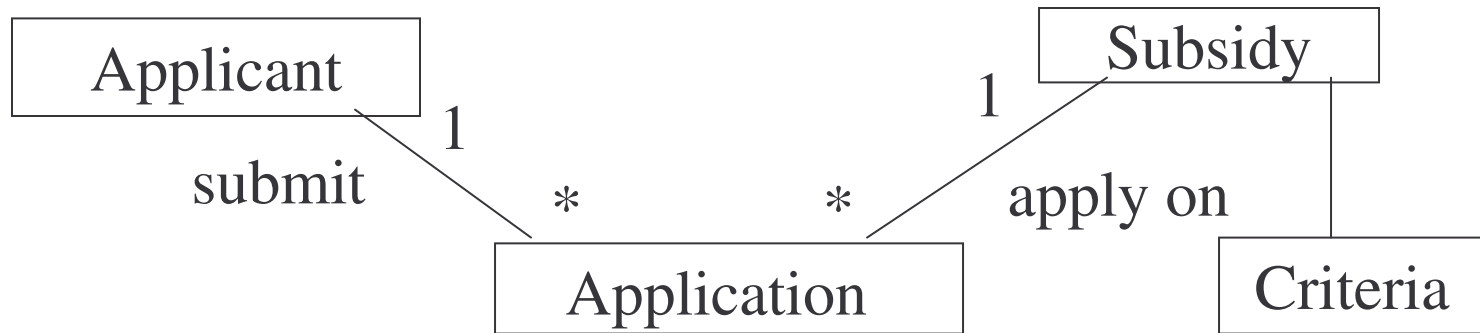
- Business Rules constraint either the possible structure or the behavior of objects or processes.
- Three kinds of constraints: structural, operation/behavior, and stimulus/response.
- Structural Rules – types, associations, invariant conditions on static aspects, like multiplicity.
- Operational/behavioral Constraints – guard condition, pre/post conditions.
- Stimulus/Response Rules – certain actions should be performed when certain events are generated in the business.

Object Constraint Language (OCL)

- OCL is a *specification* language designed to formally specify constraints in software modules
 - An OCL expression simply specifies a logical fact (a constraint) about the system that must remain **true**
 - A constraint cannot have any side-effects. It cannot compute a non-Boolean result nor modify any data.
 - OCL statements in class diagrams can specify what the values of attributes and associations must be

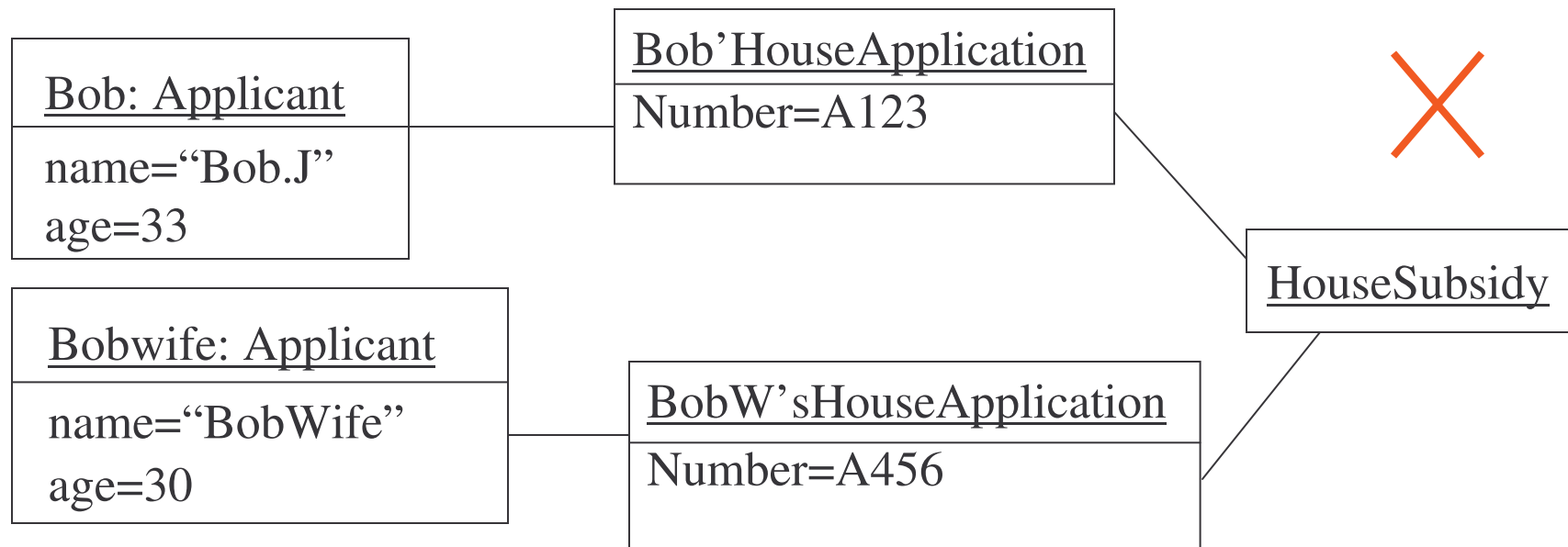
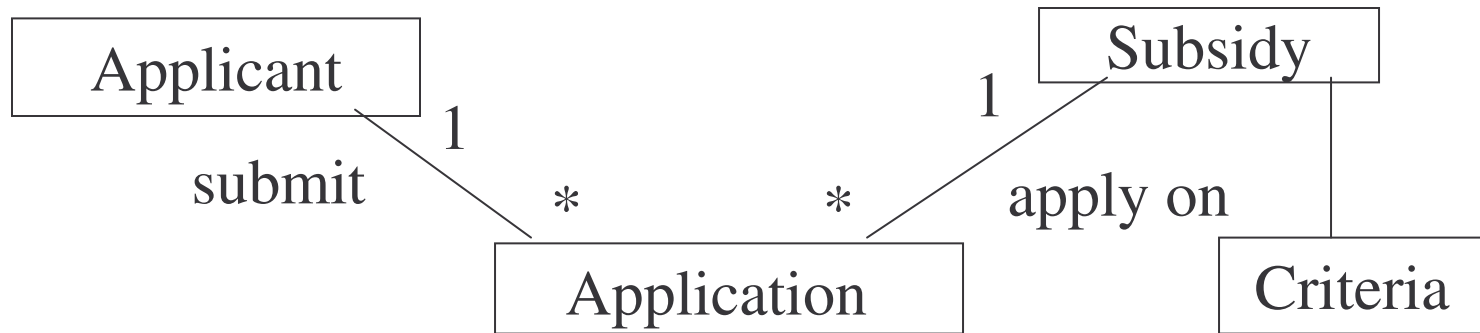
Constraints on BAS

One applicant cannot submit two applications on the house-subsidy.



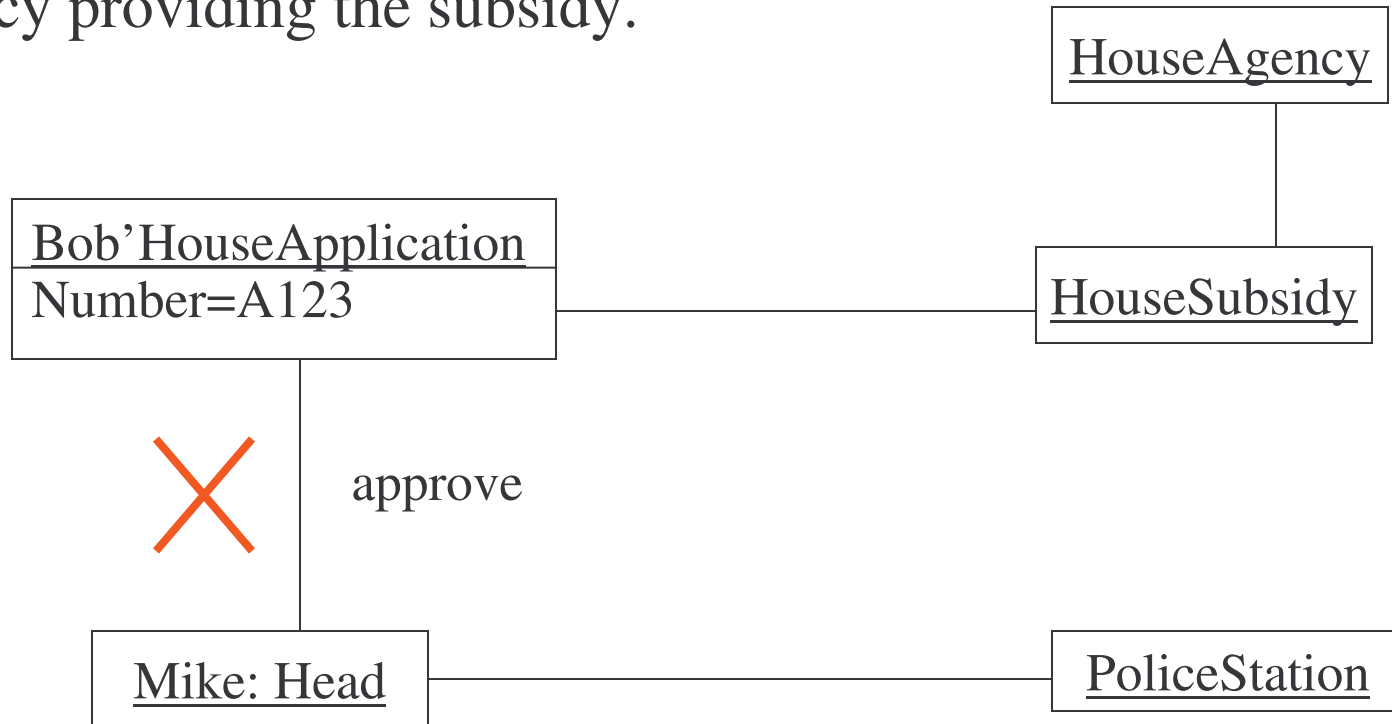
Constraints on BAS

One family can only submit one application on the house-subsidy.



Constraints on BAS

An application must be approved by the corresponding head of agency providing the subsidy.



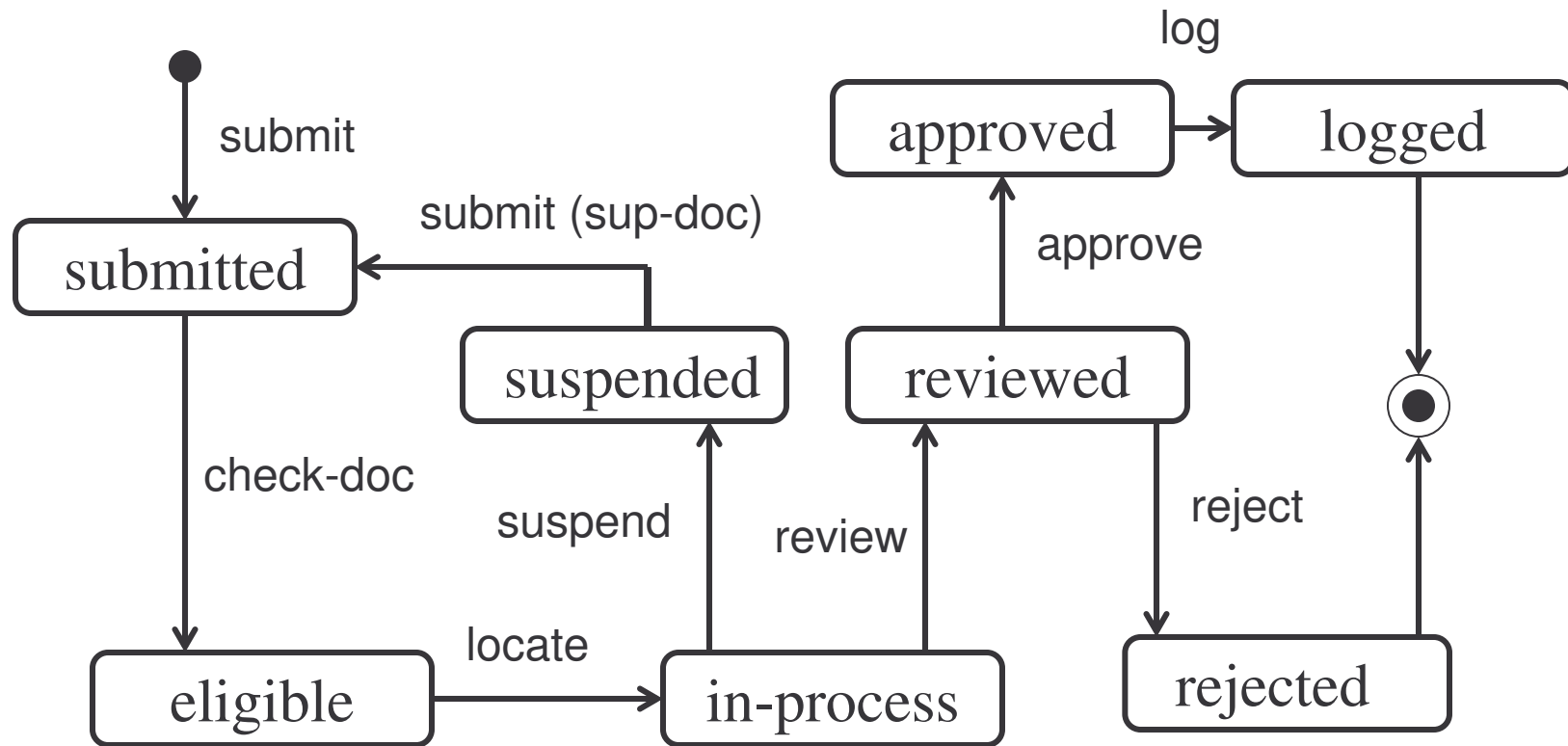
Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - *State and activity diagram*
 - GoF's design patterns
 - Summary

State Machine Diagram

- UML state machine diagrams show a dynamic view.
- A state machine diagram shows the lifecycle of an object: what events it experiences, its transitions, and the states it is in between these events.
- Consider state machines for state-dependent objects with complex behavior, not for state-independent objects.
- In general, business information systems have few complex state-dependent classes. It is seldom helpful to apply state machine modeling.
- By contrast, process control, device control, protocol handlers, and telecommunication domains often have many state-dependent objects.

State Machine Diagram for an Application



Activity Diagram

- A **UML activity diagram** shows sequential and parallel activities in a process. They are useful for modeling business processes, workflows, data flows, and complex algorithms.
- An activity definition contains activity nodes. An activity node represent the execution of a statement in a procedure or the performance of a step in a workflow. Nodes are connected by control flows and data flows.

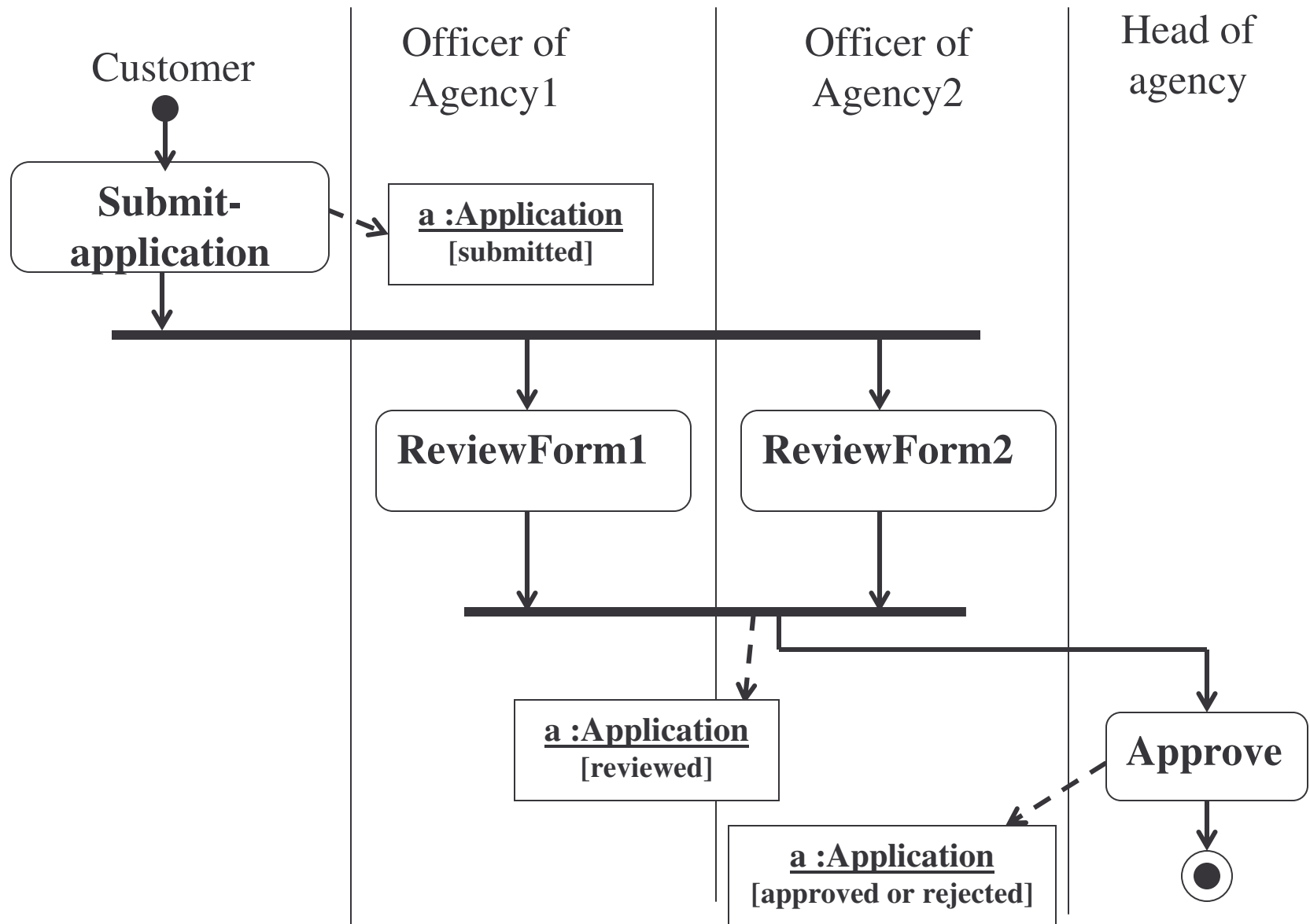
Concurrent Activities

- Concurrent threads represent activities that can be performed concurrently by different objects or persons in an organization.
- Frequently concurrency arises from aggregation, in which each object has its own concurrent thread.
- Concurrent activities can be performed simultaneously or in any order.

Action

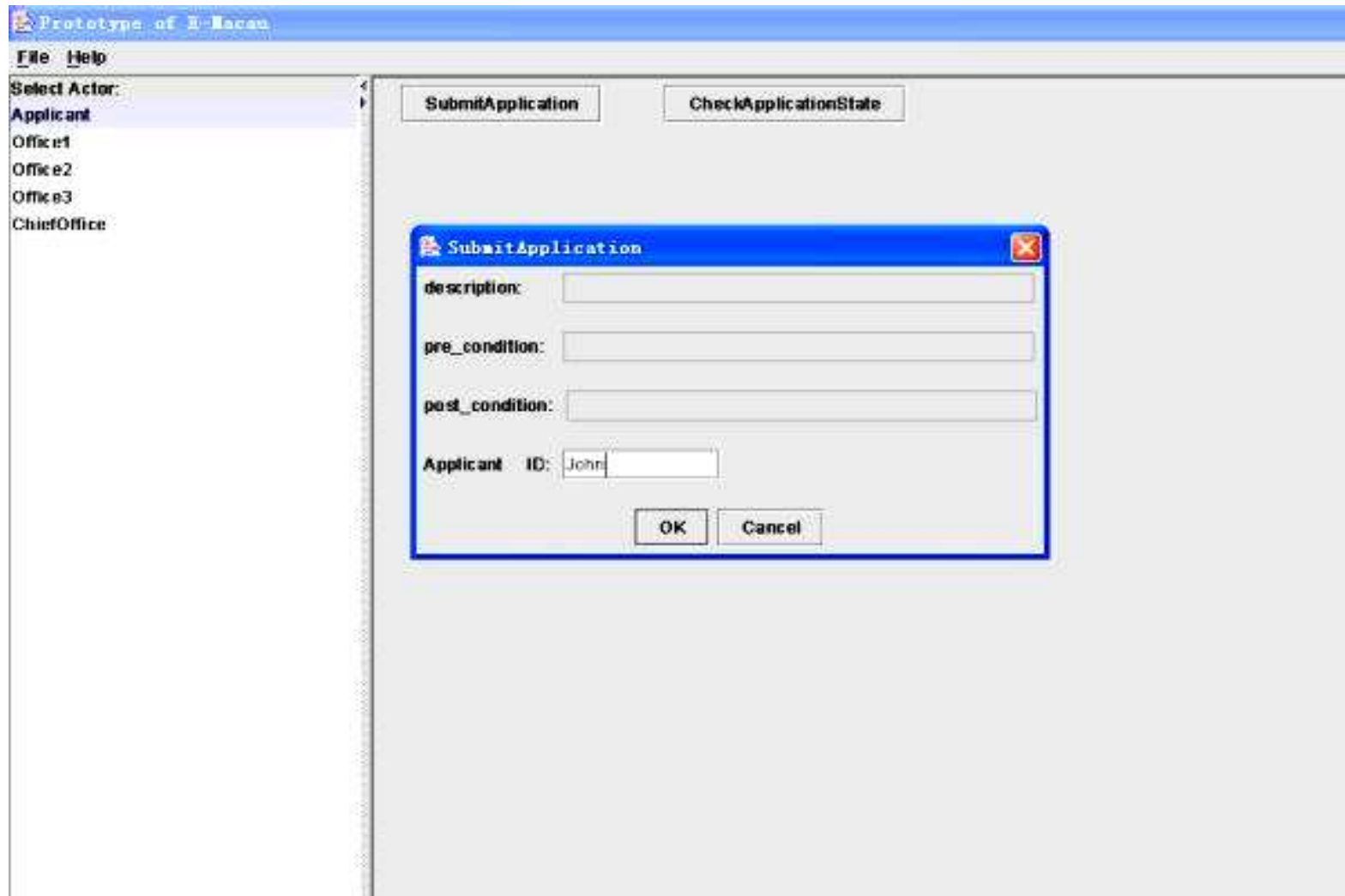
- An action is a basic, atomic, predefined activity.
- Atomic actions:
 1. Find an object
 2. Get an attribute
 3. Set an attribute
 4. Create an object
 5. Delete an object
 6. Build a link
 7. Delete a link

Activity Diagram with Swimlane

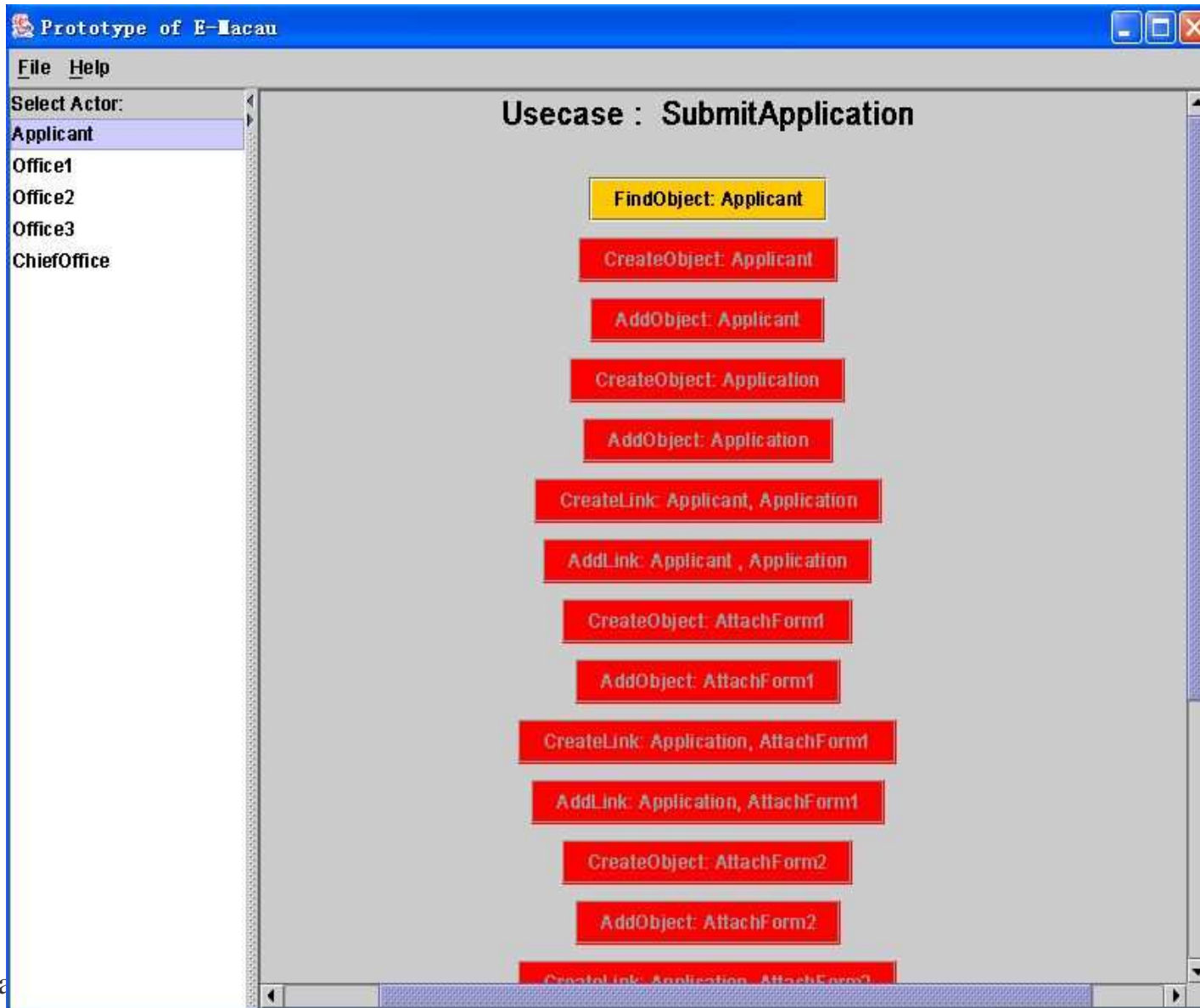


Prototype of A Simple Application System

Usecase:SubmitApplication (initiate)



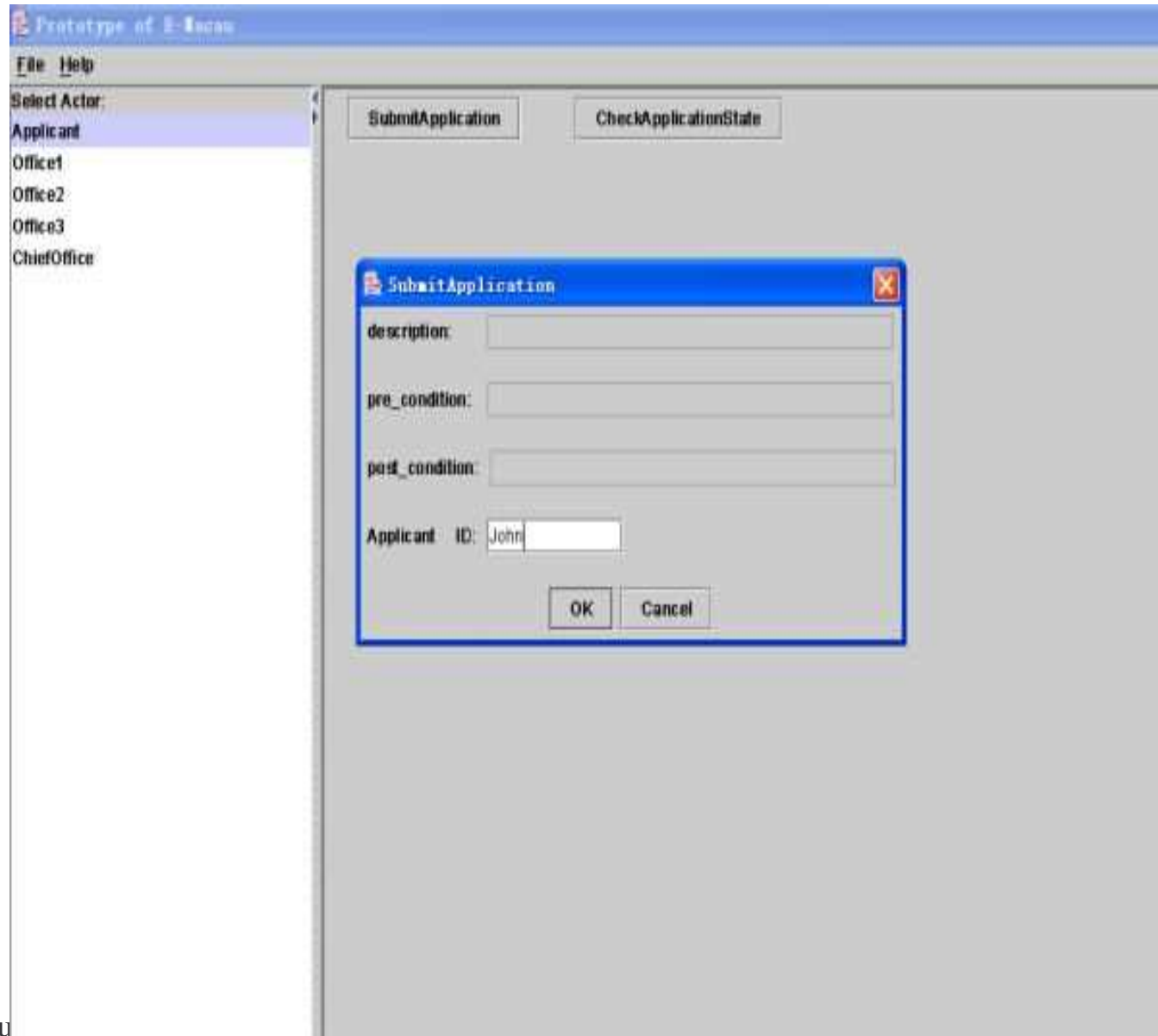
Usecase:SubmitApplication (initial)



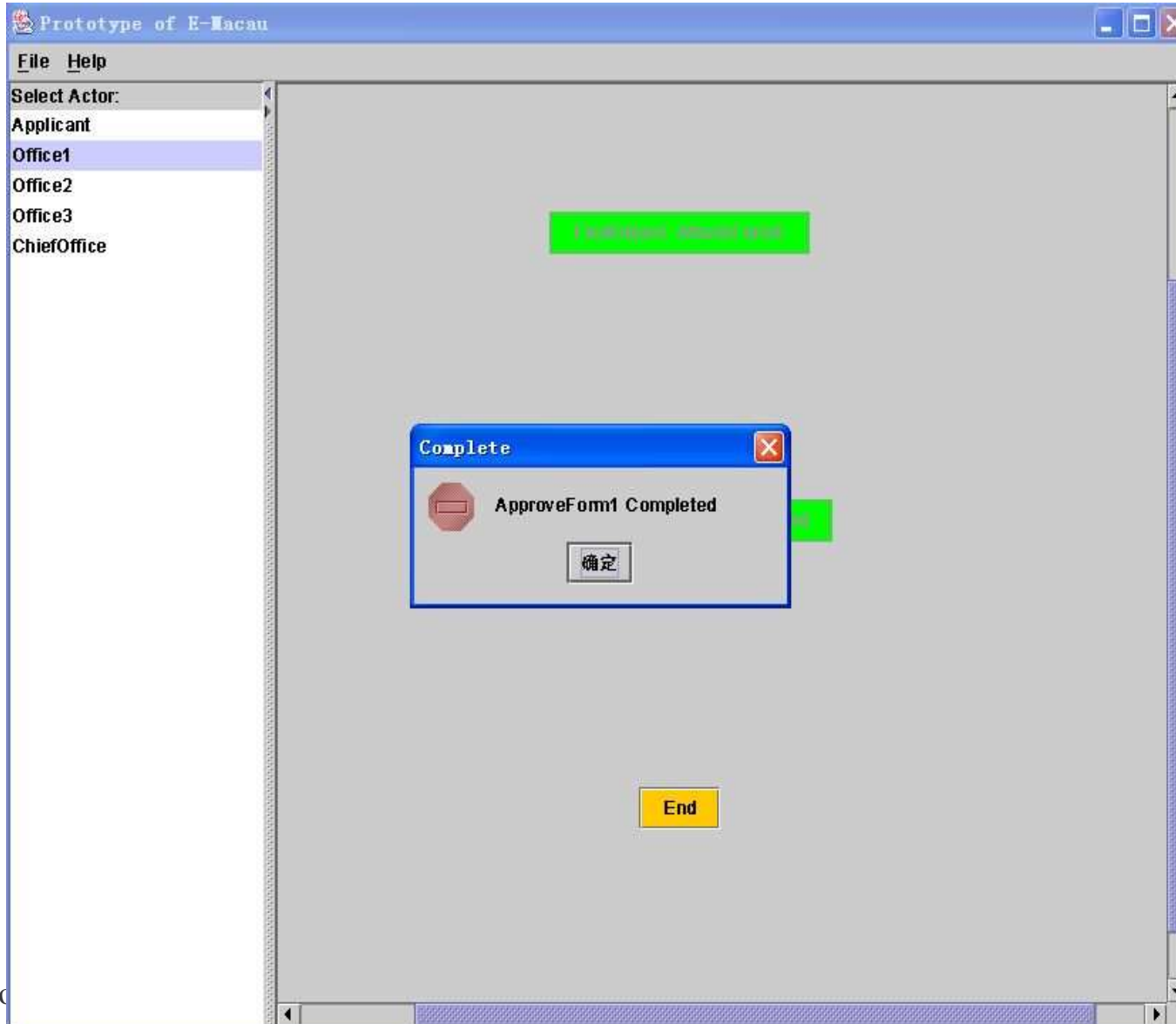
Usecase:SubmitApplication (finish)



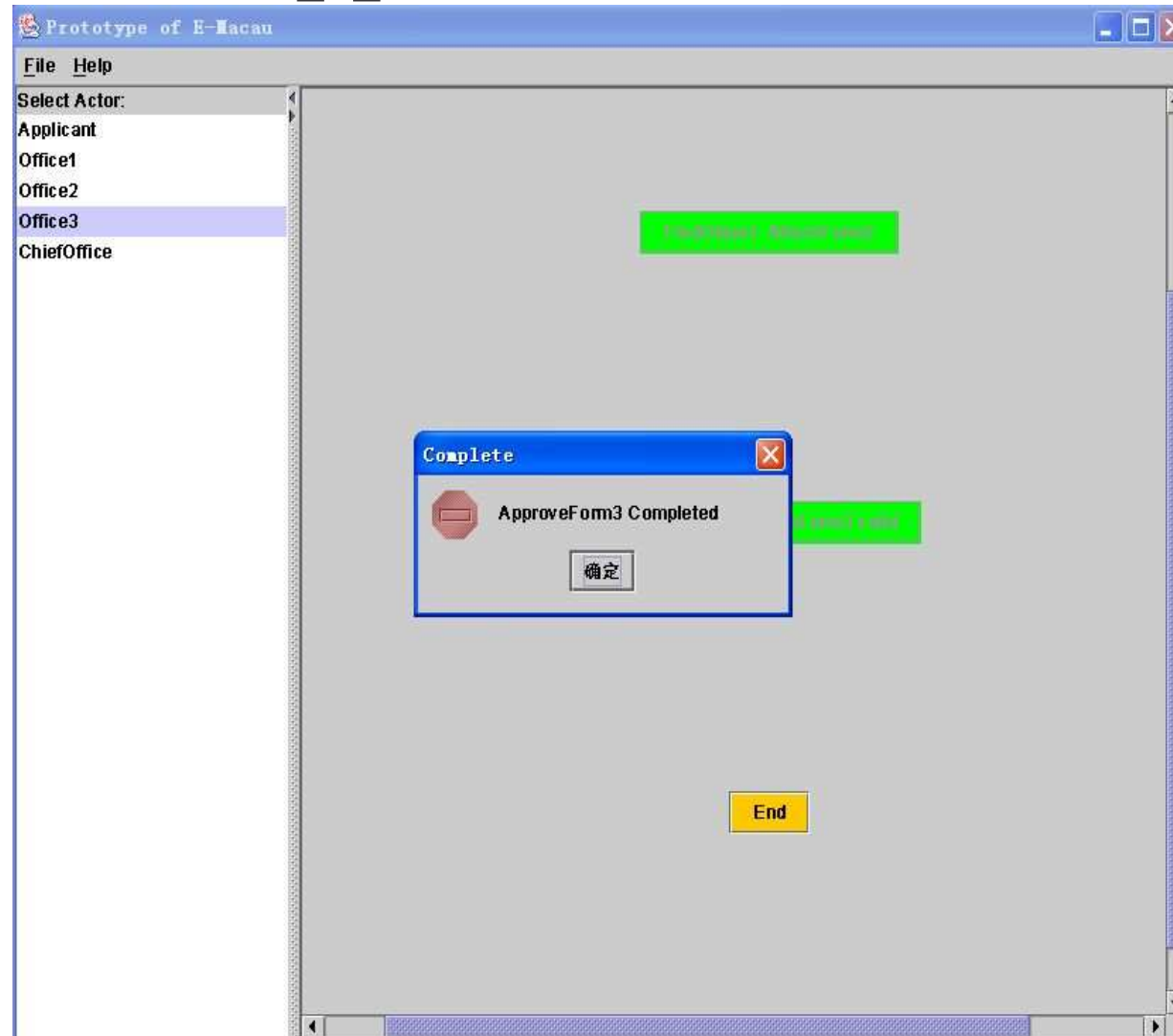
Usecase: ApproveForm1 (initial)



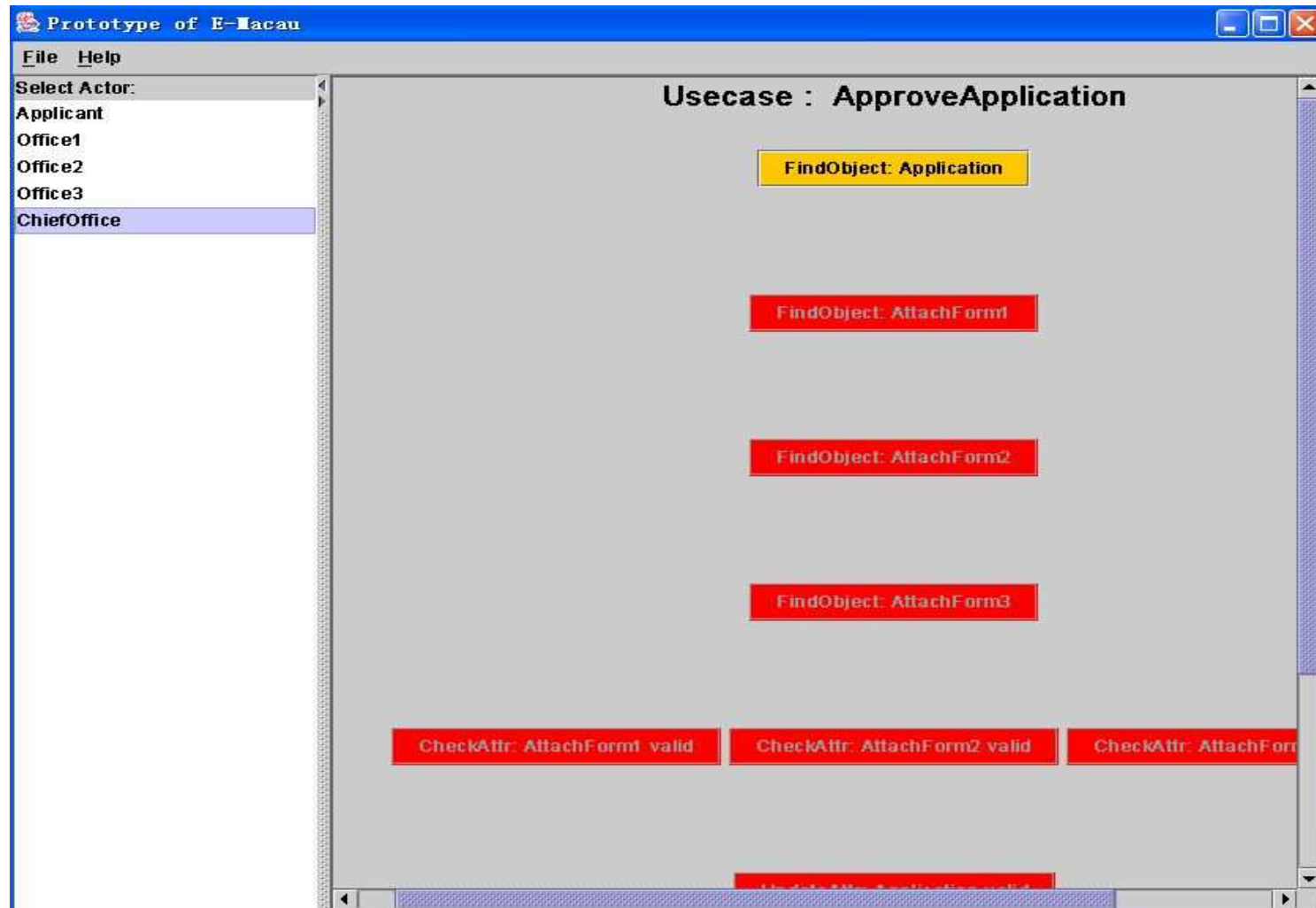
Usecase: ApproveForm1 (finish)



Usecase: ApproveForm3 (finished)

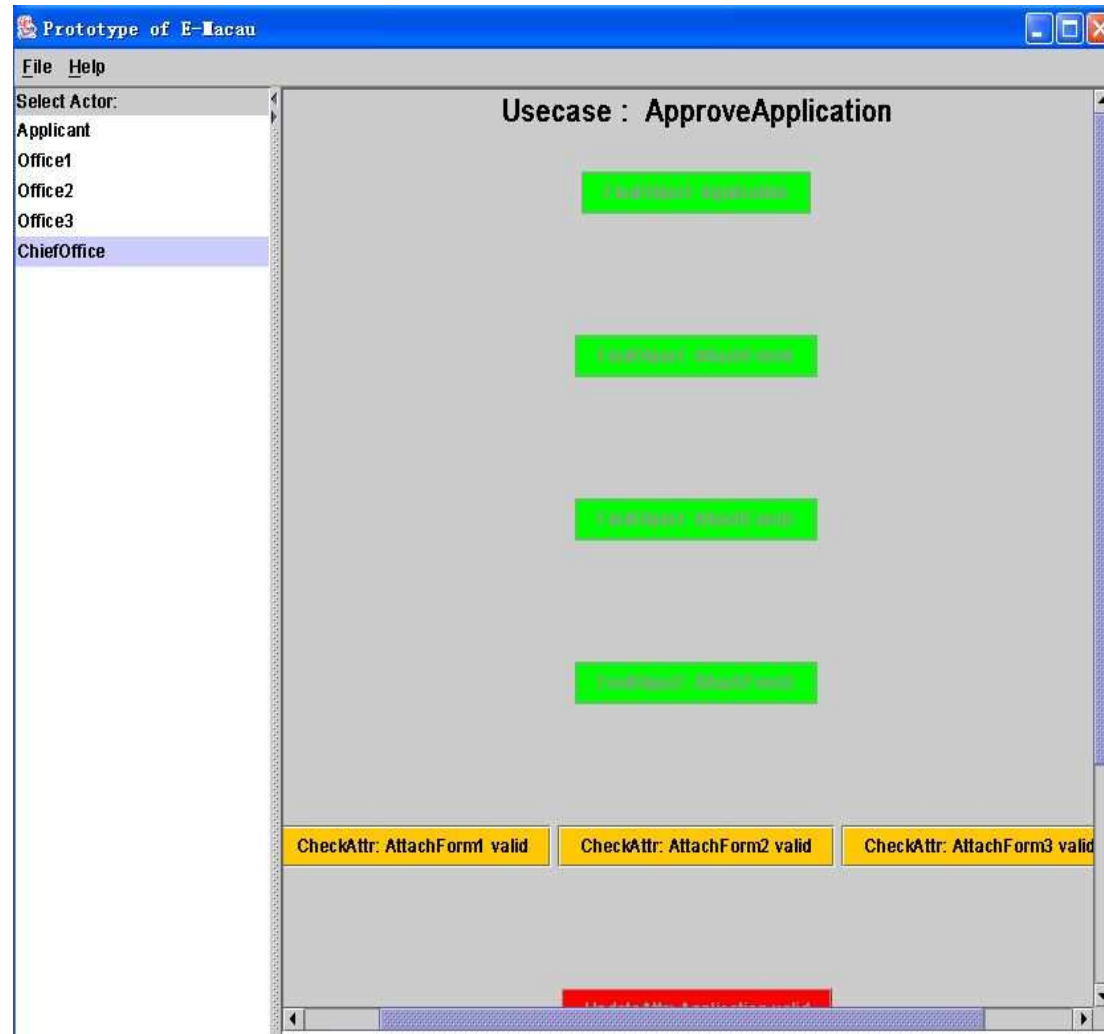


Usecase: ApproveApplication (initial)



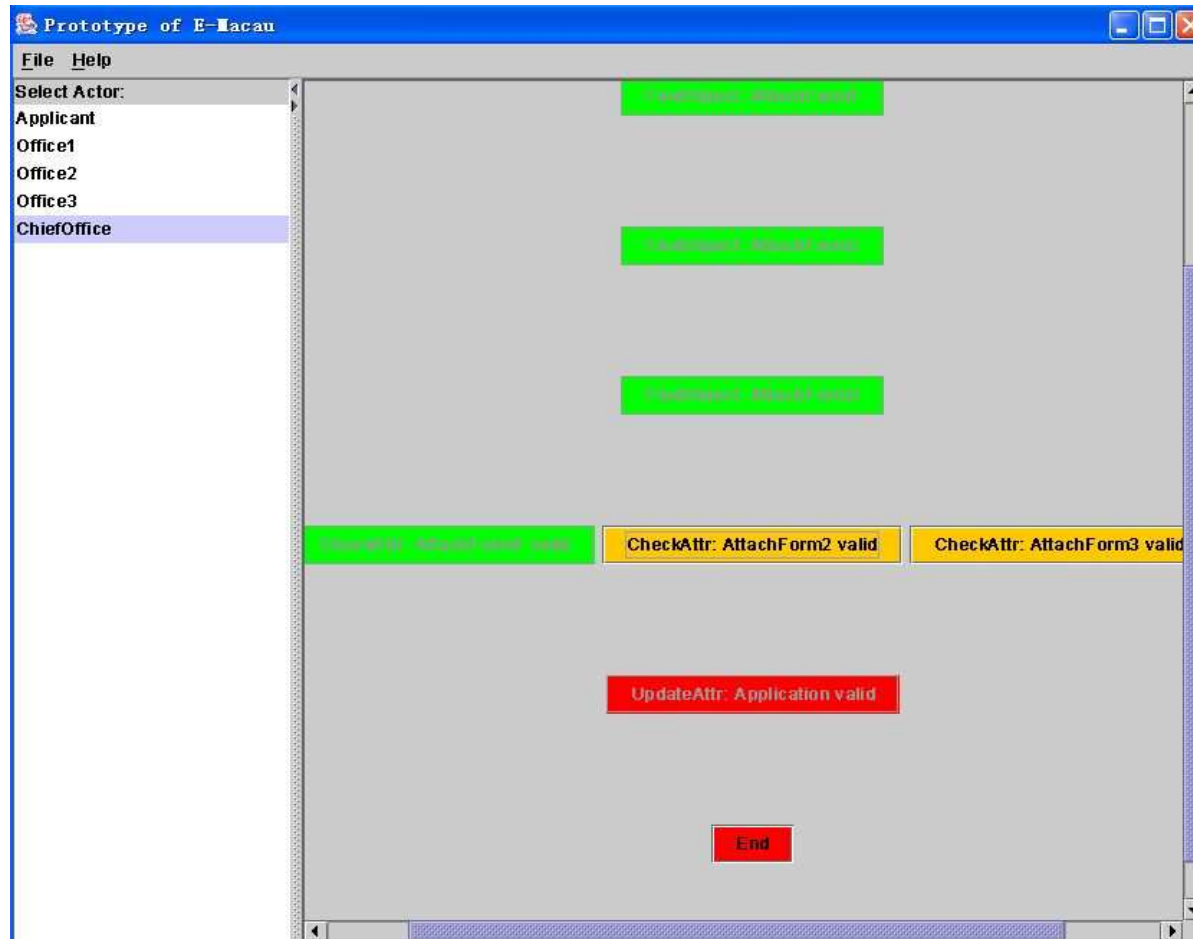
Usecase: ApproveApplication

- The 3 conditions should all be hold.



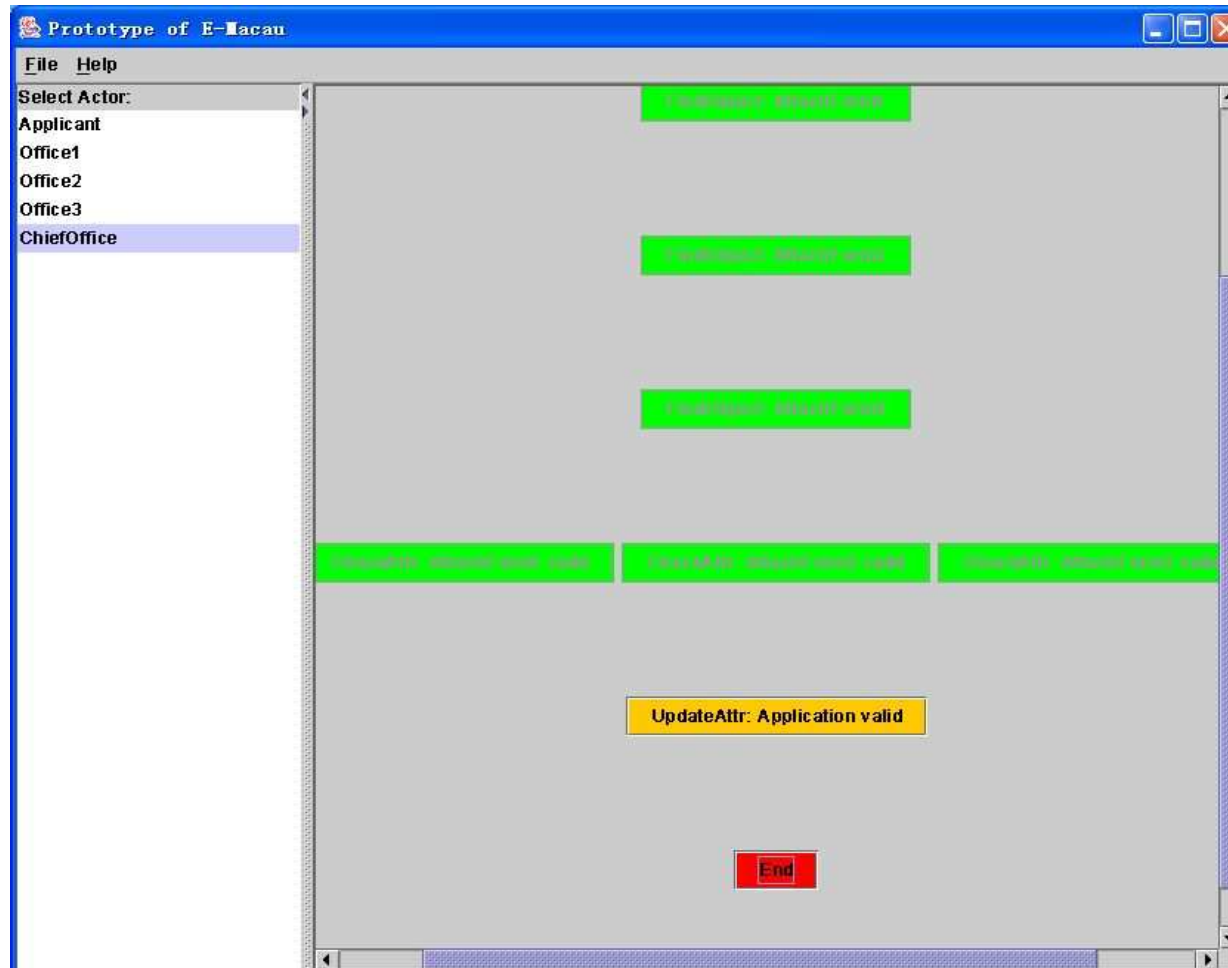
Usecase: Approve Application

- First condition is satisfied.



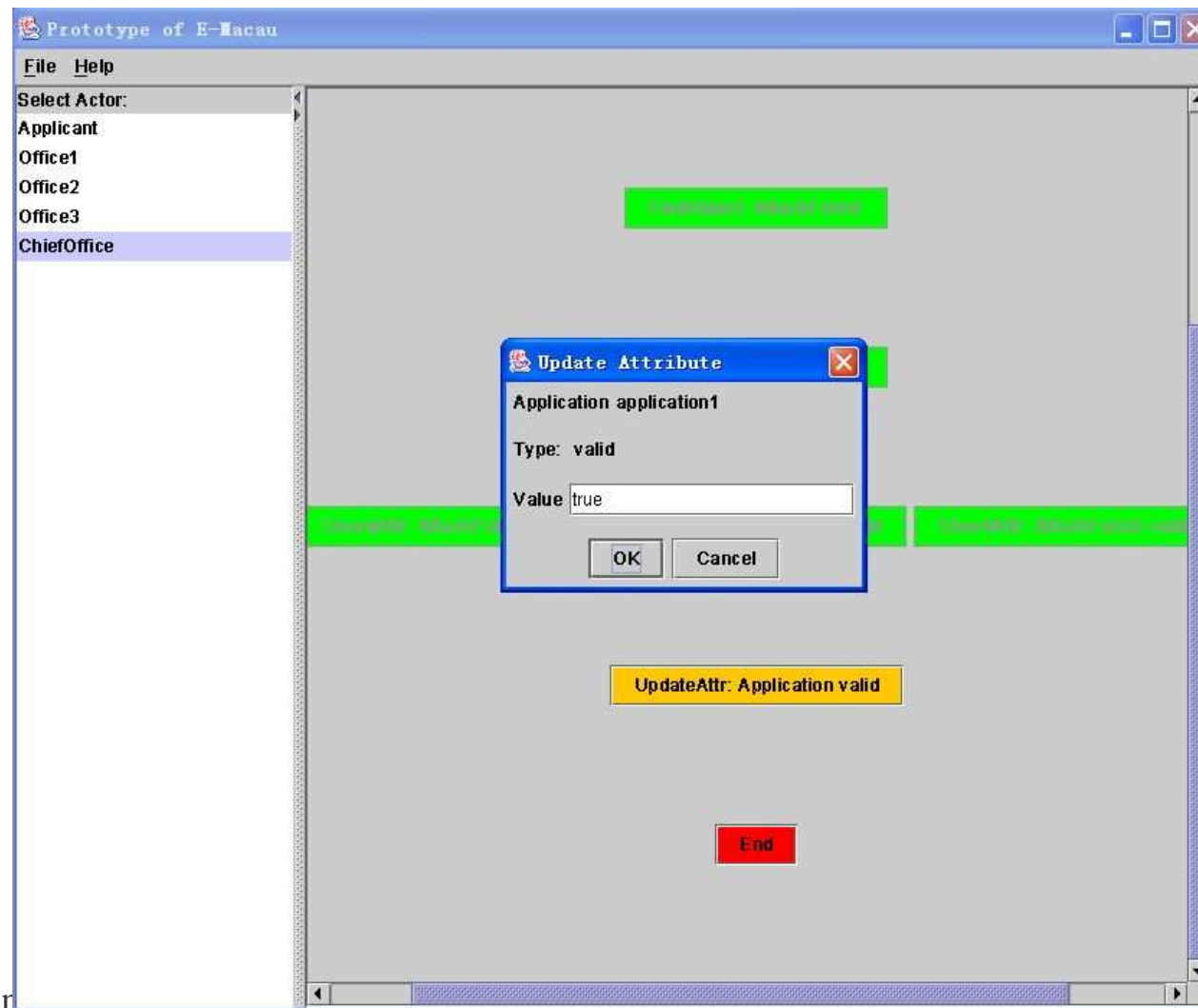
Usecase: Approve Application

- All satisfy, then the next step is activated.

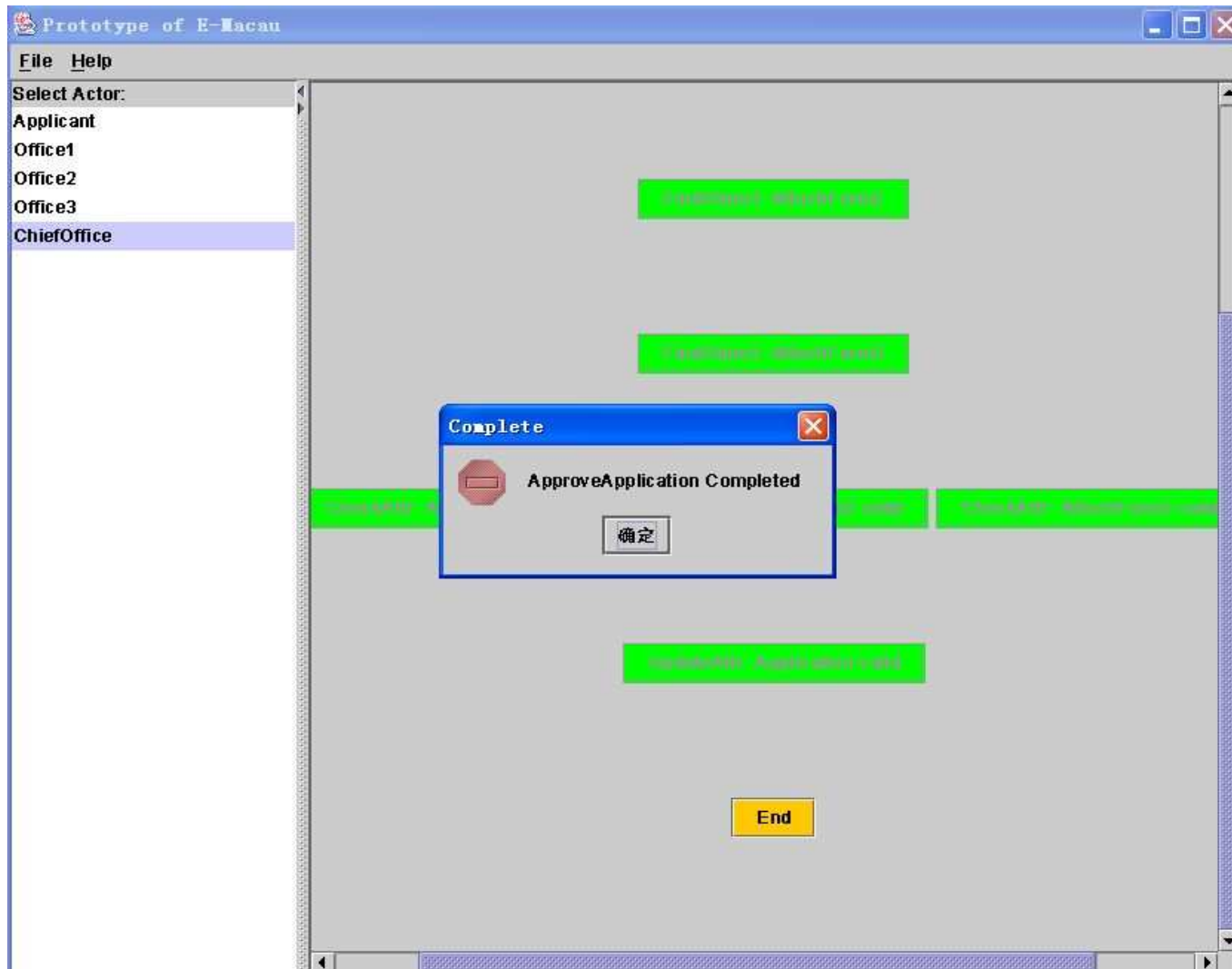


Usecase: Approve Application

- Update the valid attribute of application.



Usecase: ApproveApplication



Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - *GoF's design patterns*
 - Summary

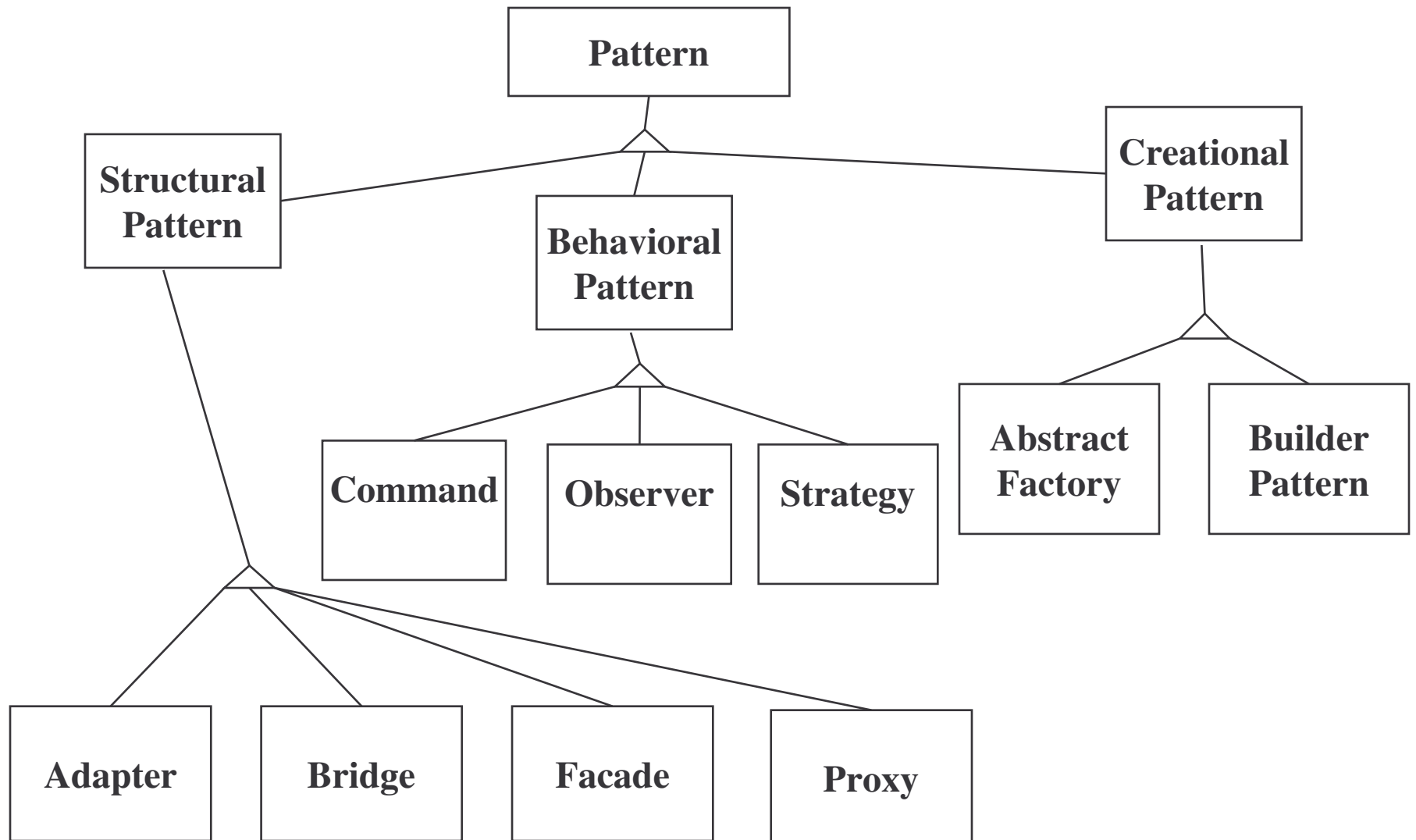
3 kinds of Design Patterns

GoF's patterns book defined a base catalog of 23 patterns, which defined in three categories of patterns:

1. *Creational patterns*: Handle the instantiation process(how, when, and what object are created) and the configuration of classes and objects. Allow a system to work with “product” objects that vary in structure and functionality.
2. *Structural patterns*: Handle the way classes and objects are used in larger structures, and separate interfaces from implementation.
3. *Behavioral patterns*: Handles algorithms and the division of responsibility between objects, and dynamic interaction between classes and objects.

Many design patterns use a combination of inheritance and delegation.

A Pattern Taxonomy

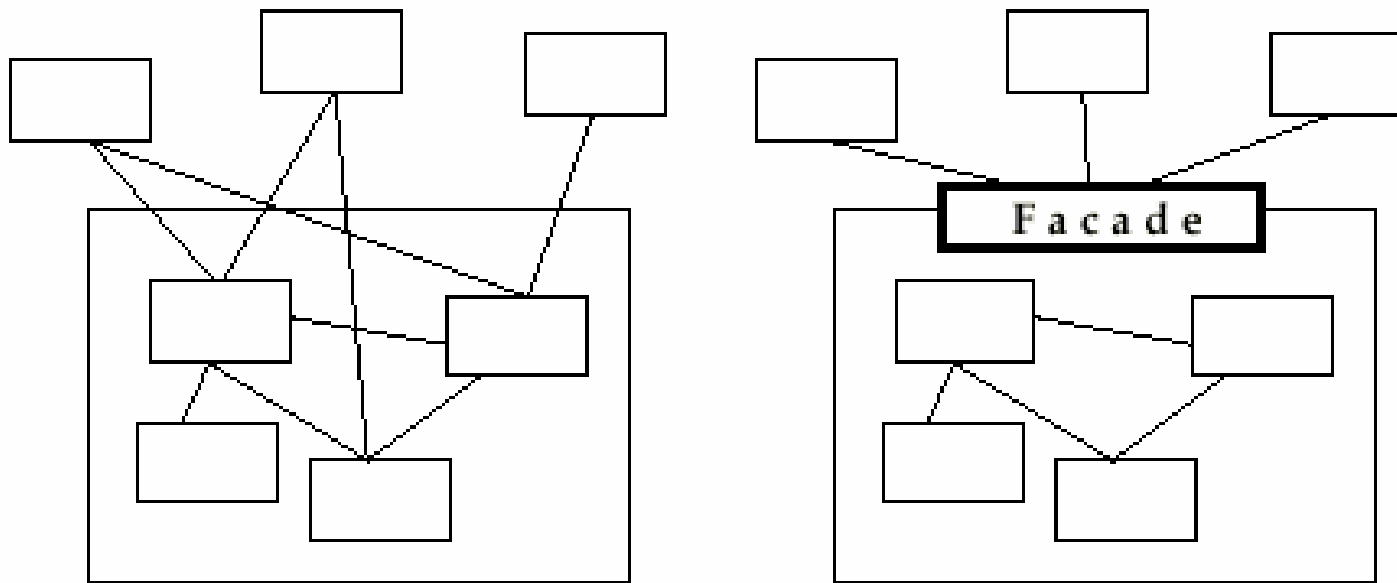


Ideal Structure of a Subsystem: Façade, Adapter, Bridge

- A subsystem consists of
 - an interface object
 - a set of application domain objects (entity objects) modeling real entities or existing systems
 - some of the application domain objects are interfaces to existing systems
 - one or more control objects
- Realization of Interface Object: Facade
 - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
 - Provides the interface to existing system (legacy system)
 - The existing system is not necessarily object-oriented!

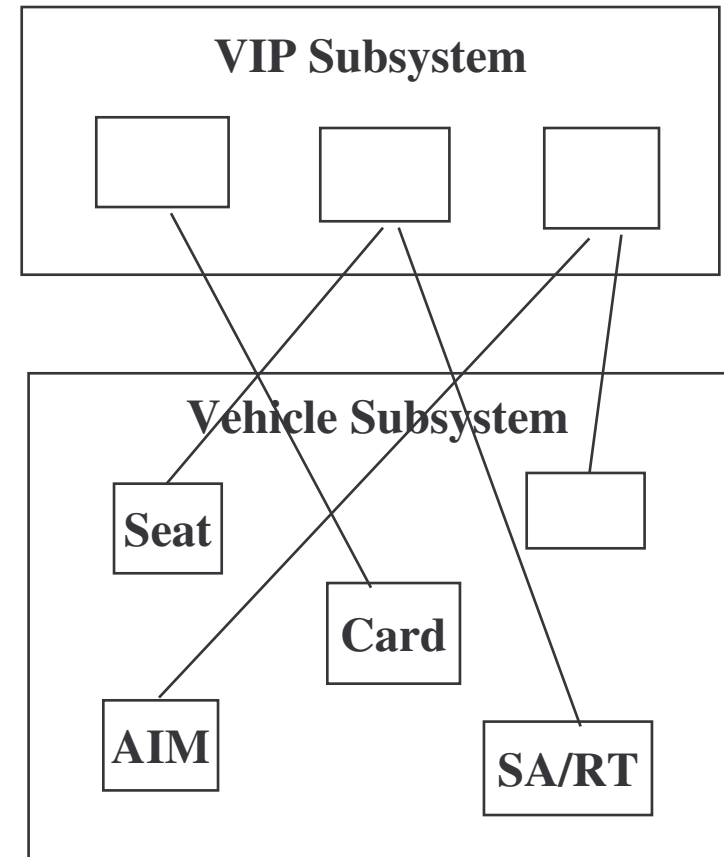
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



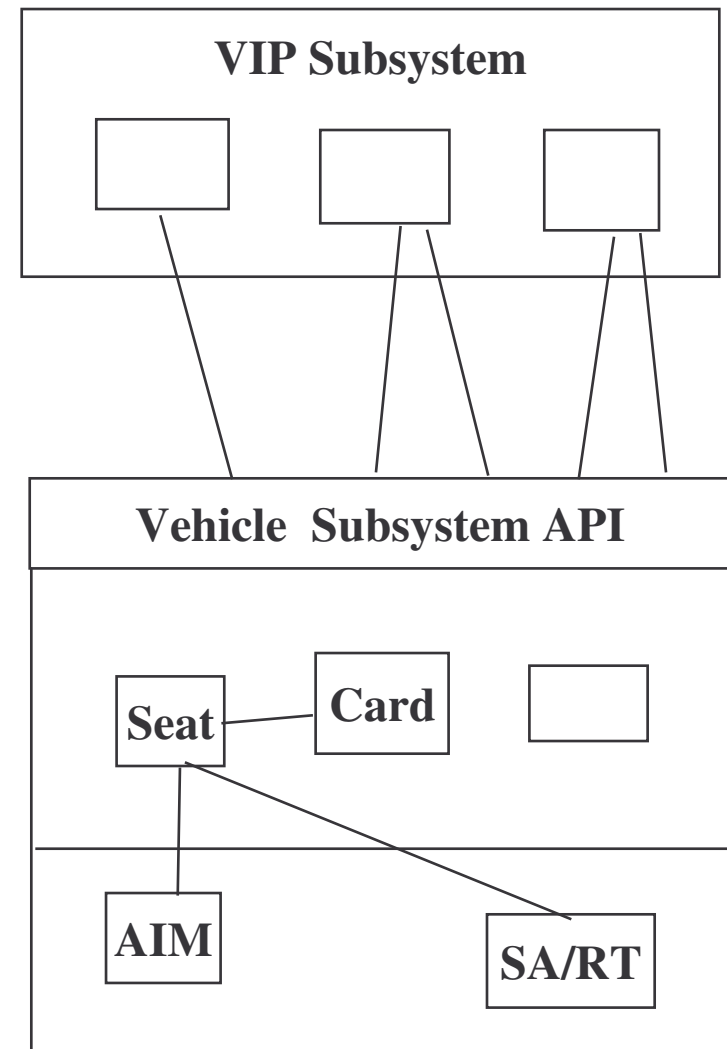
Open vs Closed Architecture

- Open architecture:
 - Any client can see into the vehicle subsystem and call on any component or class operation at will.
- Why is this good?
 - Efficiency
- Why is this bad?
 - Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
 - We can be assured that the subsystem will be misused, leading to non-portable code

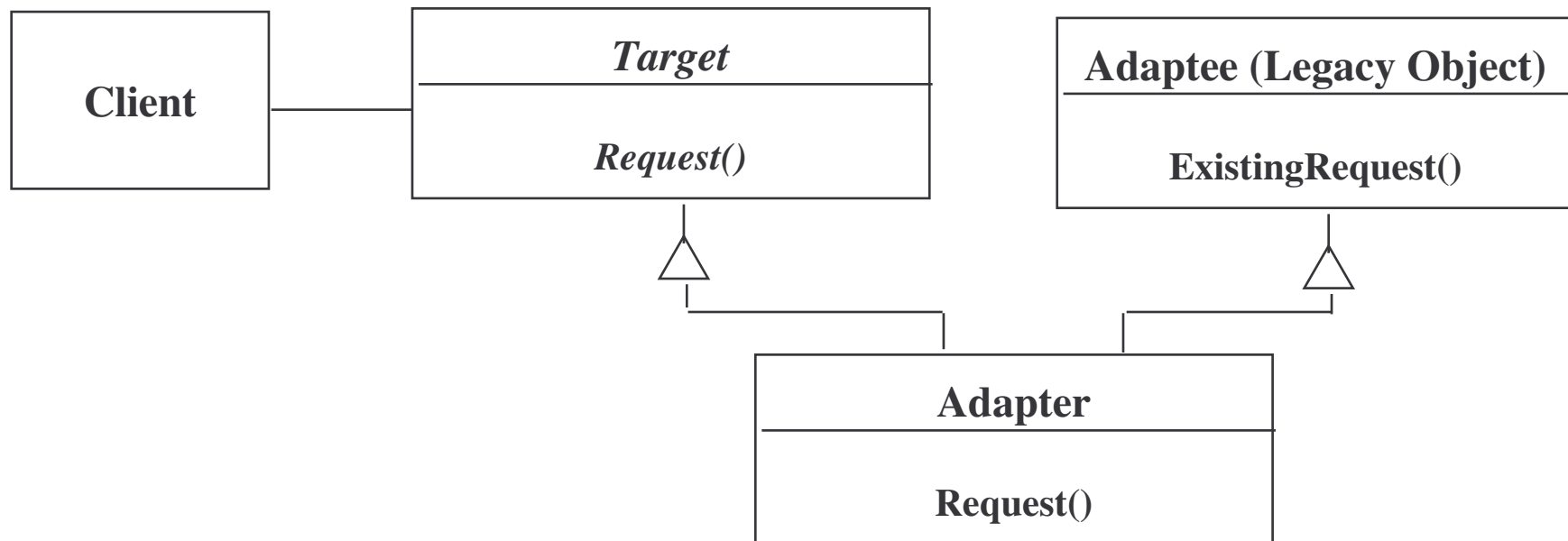


Realizing a Closed Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used the subsystem can be used in an early integration test
 - We need to write only a driver

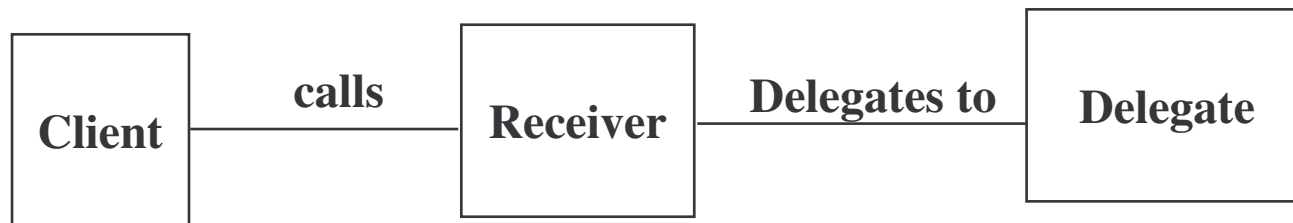


Class Adapter Pattern (based on Multiple Inheritance)



Delegation

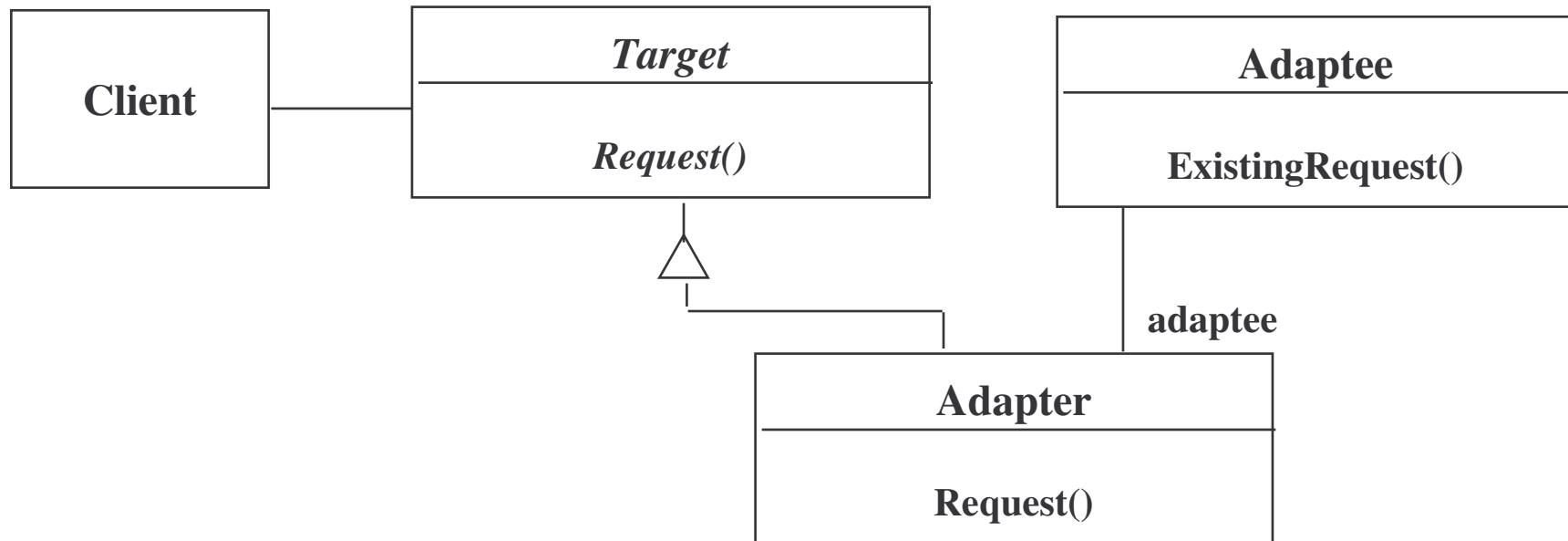
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
 - A receiving object delegates operations to its delegate.
 - The developer can make sure that the receiving object does not allow the client to misuse the delegate object



Adapter Pattern

- “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
 - Class adapter:
 - Uses multiple inheritance to adapt one interface to another
 - Object adapter:
 - Uses single inheritance and delegation
- We will mostly use object adapters and call them simply adapters

Adapter pattern



- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is used to specify the interface of the **Adapter** class.
- **Target** and **Adaptee** (usually called legacy system) pre-exist the **Adapter**.
- **Target** may be realized as an interface in Java.

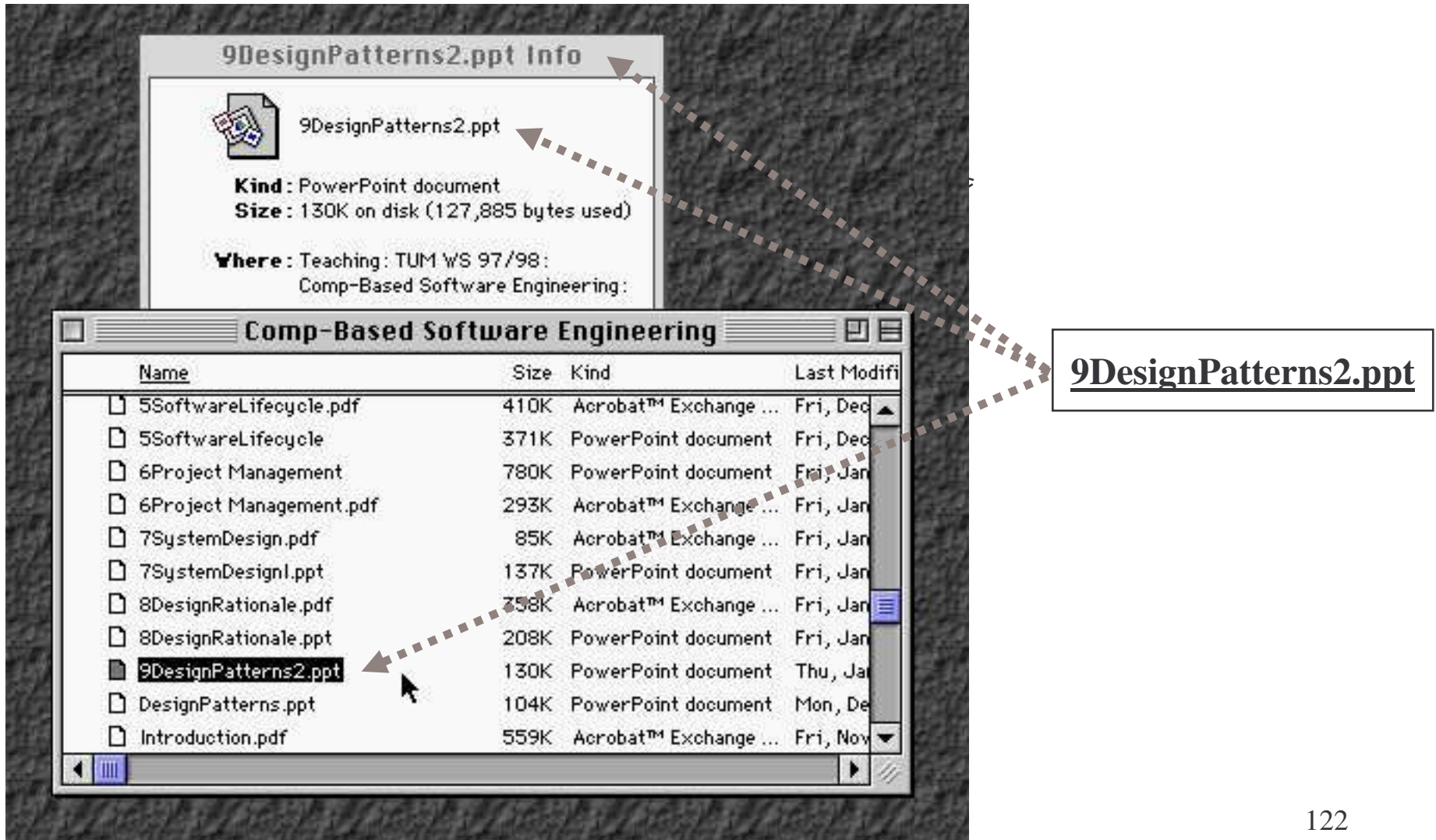
Observer pattern

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”
- Also called “Publish and Subscribe”
- Uses:
 - Maintaining consistency across redundant state
 - Optimizing batch changes to maintain consistency

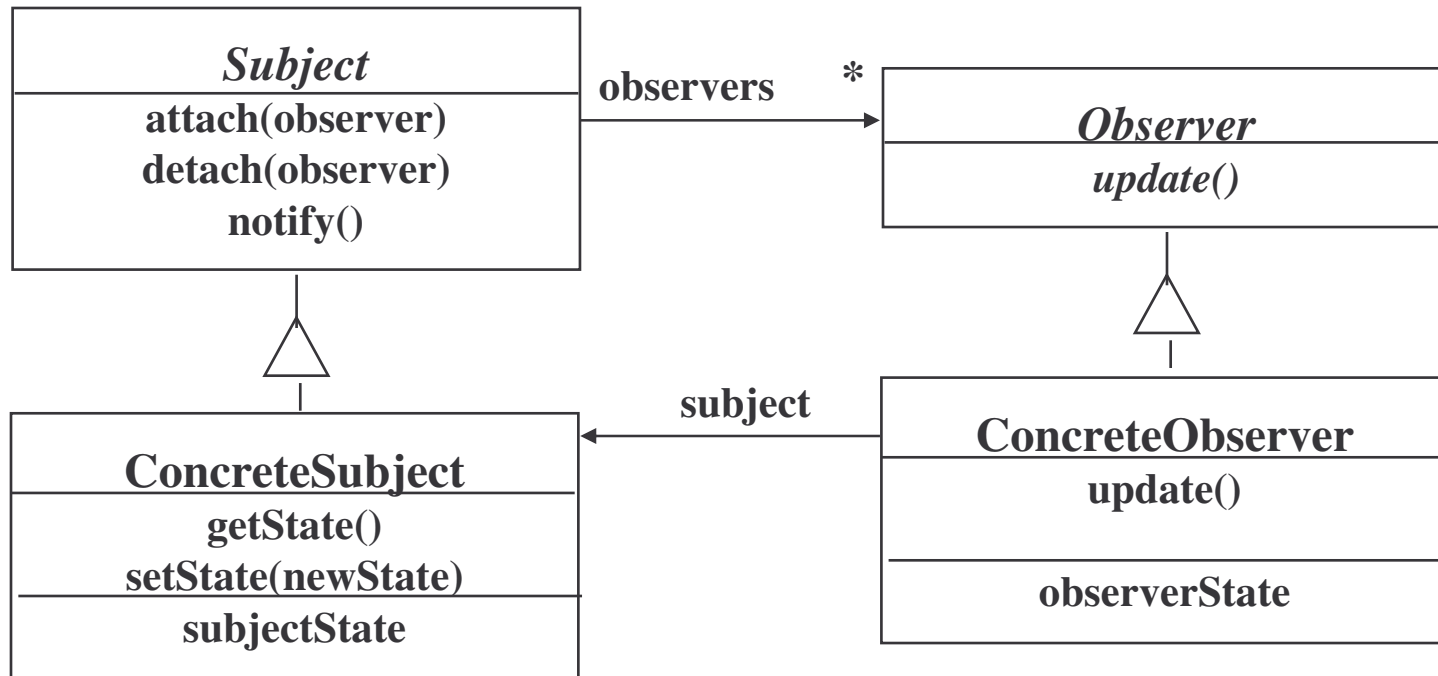
Observer pattern

Observers

Subject

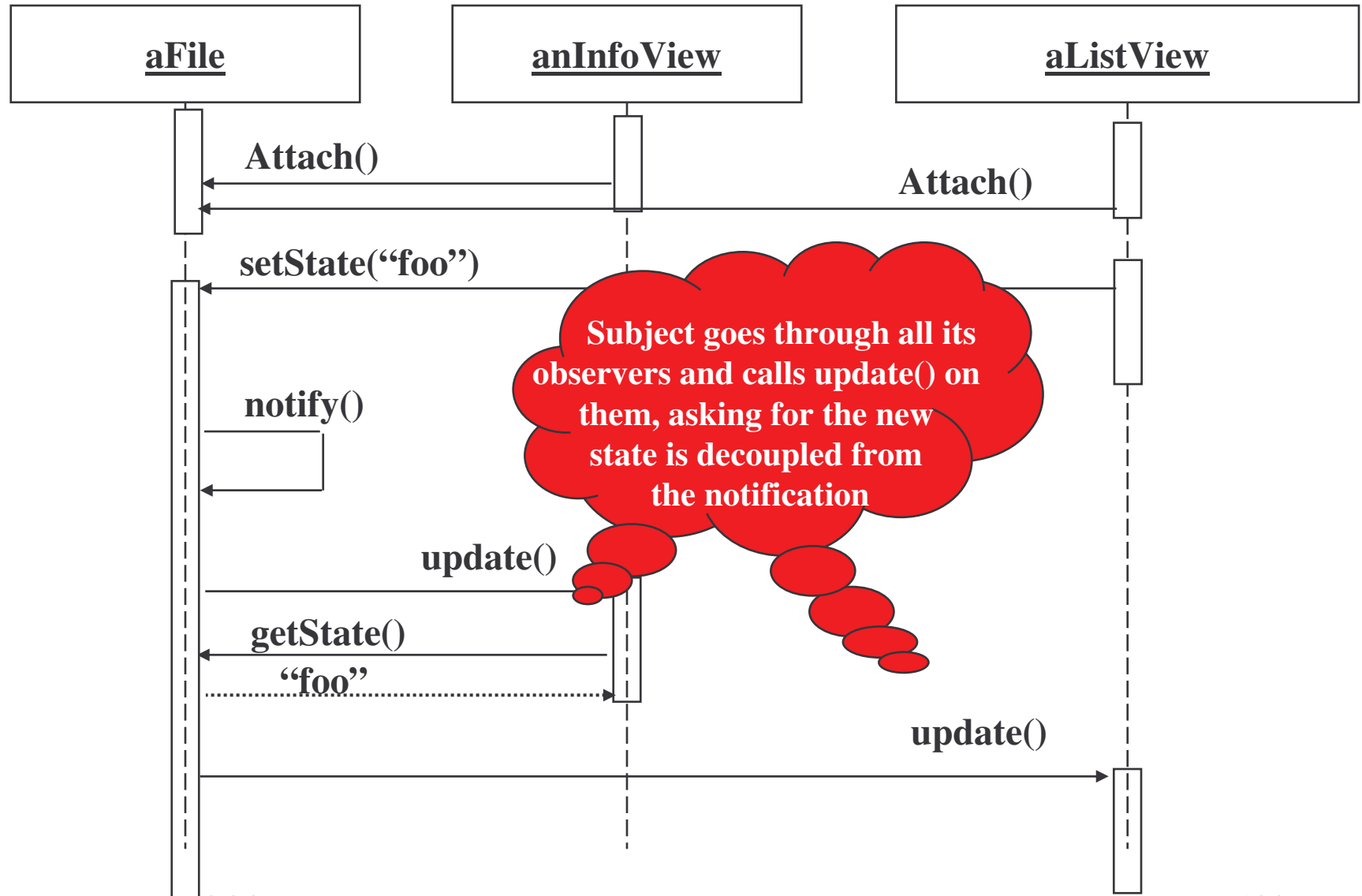


Observer pattern

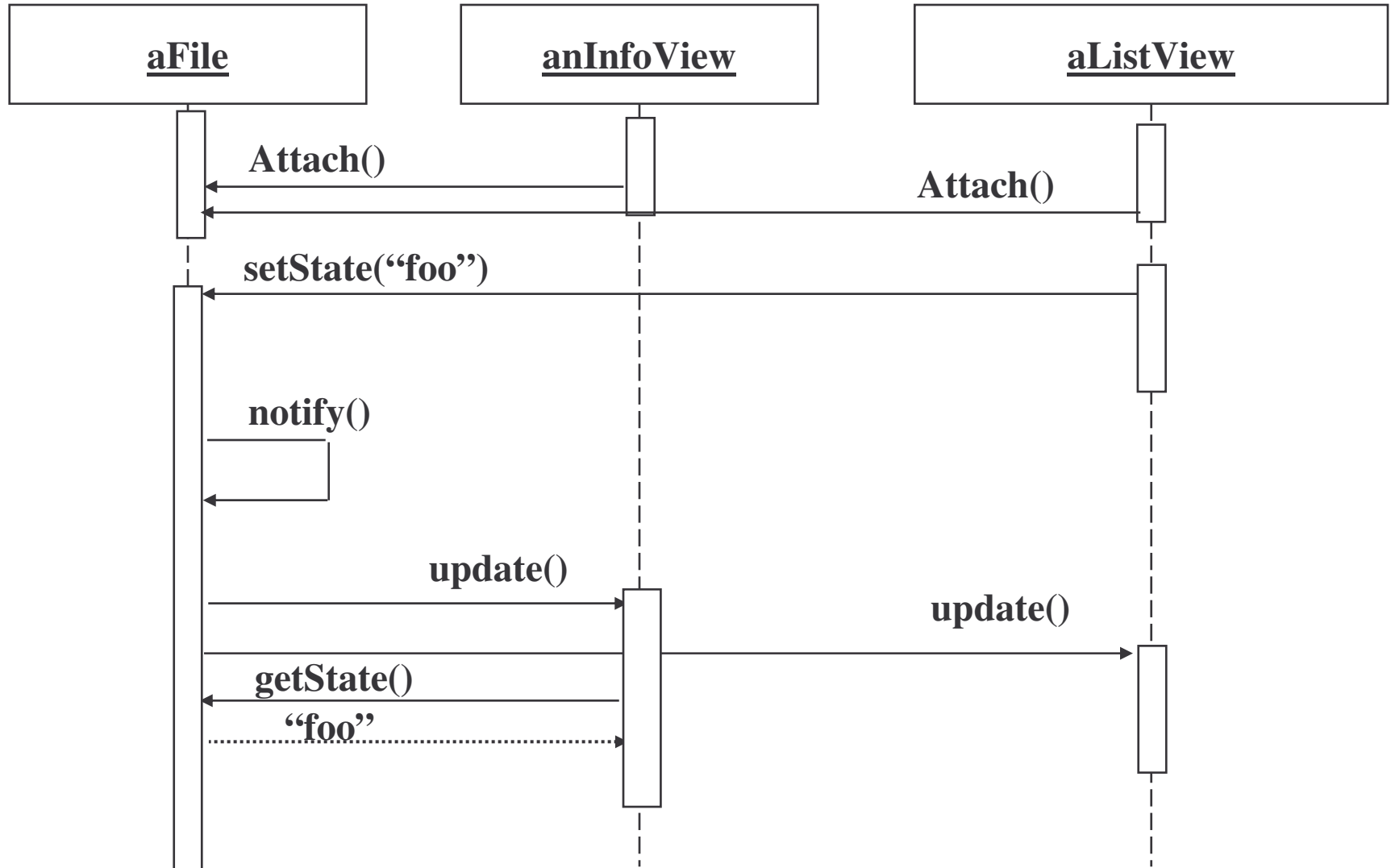


- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) *not* an interface.

Sequence diagram for scenario



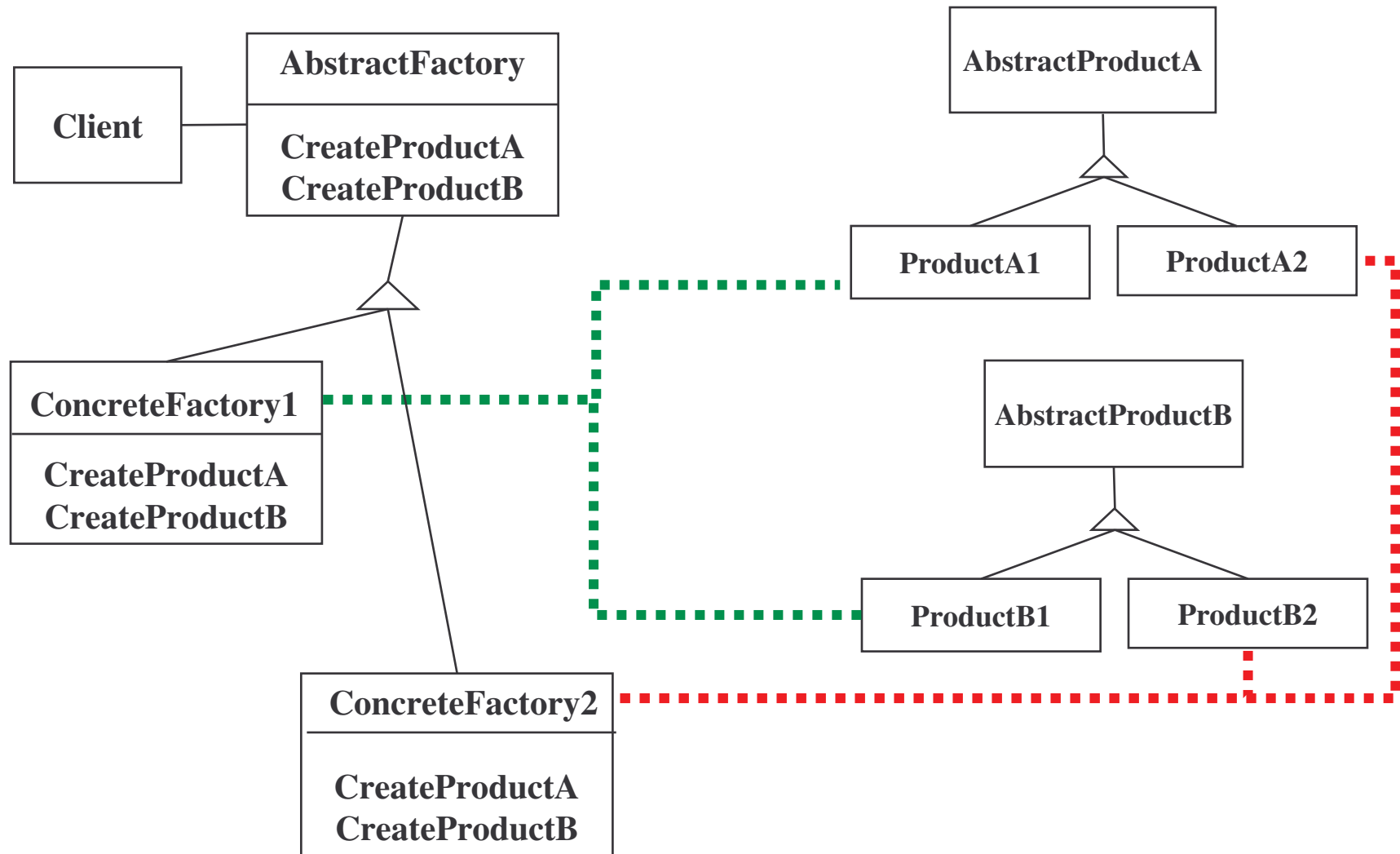
Animated Sequence diagram



Abstract Factory Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in Mac OS.
 - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens' Instabus, Johnson & Control Metasys or Zumtobe's proprietary standard.
 - How can you write a single control system that is independent from the manufacturer?

Abstract Factory



Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
 - The system should be independent of how its products are created, composed or represented
- Manufacturer Independence:
 - A system should be configured with one of multiple family of products
 - You want to provide a class library for a customer (“facility management library”), but you don’t want to reveal what particular product you are using.
- Constraints on related products
 - A family of related products is designed to be used together and you need to enforce this constraint
- Cope with upcoming change:
 - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

Contents

1. Introduction
2. Unified Process (UP)
3. Visual Modeling and UML
 - Visual modeling
 - Overview of UML
 - System, model and view
4. Use Case Model
5. Domain Model
6. Object Design with Patterns
 - Overview
 - GRASP patterns
 - Business rules
 - State and activity diagram
 - GoF's design patterns
 - *Summary*

Design Patterns encourage good Design Practice

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
 - The facade will delegate requests to the appropriate components within the subsystem.
- Adapters should be used to interface to any existing proprietary components.
 - For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.

What makes a design modifiable?

- Low coupling and high coherence
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
 - Abstract classes
 - Delegation

Summary

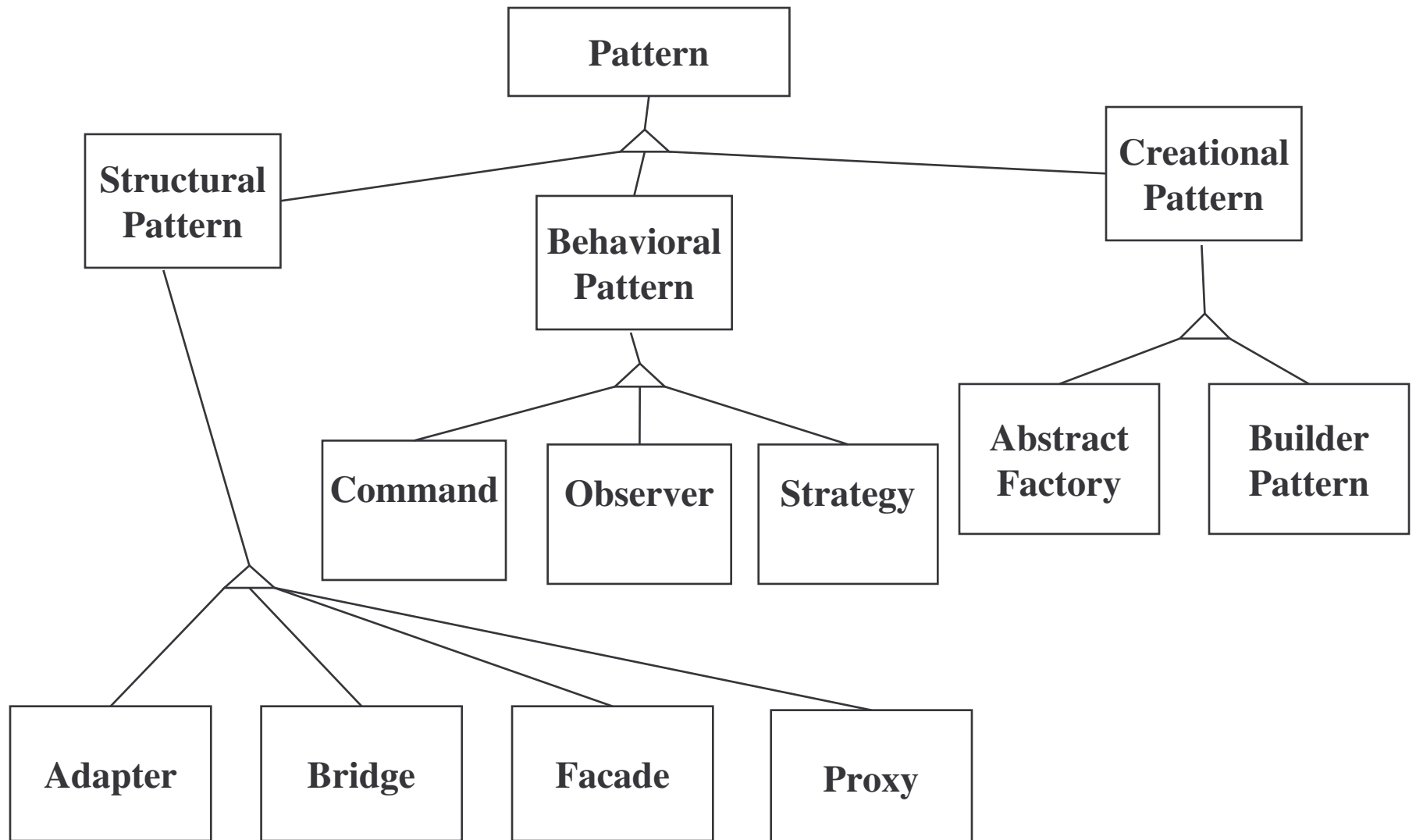
- Structural Patterns
 - Focus: How objects are composed to form larger structures
 - Problems solved:
 - Realize new functionality from old functionality,
 - Provide flexibility and extensibility
- Behavioral Patterns
 - Focus: Algorithms and the assignment of responsibilities to objects
 - Problem solved:
 - Too tight coupling to a particular algorithm
- Creational Patterns
 - Focus: Creation of complex objects
 - Problems solved:
 - Hide how complex objects are created and put together

Potential Applications to BAS

- Observer Pattern can be used to workflow engine for acknowledging applicants when approving events and events of supplementary documents needed occur, as well as in scheduling application tasks to officers.
- Façade Pattern in unified interfaces of system components.
- Business Process Patterns can also be used in dealing with system workflow coordination.

....

A Pattern Taxonomy



References

- [1] Larman,G: *Applying UML and Patterns: an introduction to object-oriented analysis and design and iterative development*, 3rd edition, Prentice Hall, 2005.
- [2] Gamma, E. Helm,R, Johnson, R. and Vlissides, J: *Design Patterns*, Addison-wesley, 1995.
- [3] Fowler, M: *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1999.