CIS 721 - Real-Time Systems
Lecture 27: Embedded System Design

Mitch Neilsen
neilsen@ksu.edu

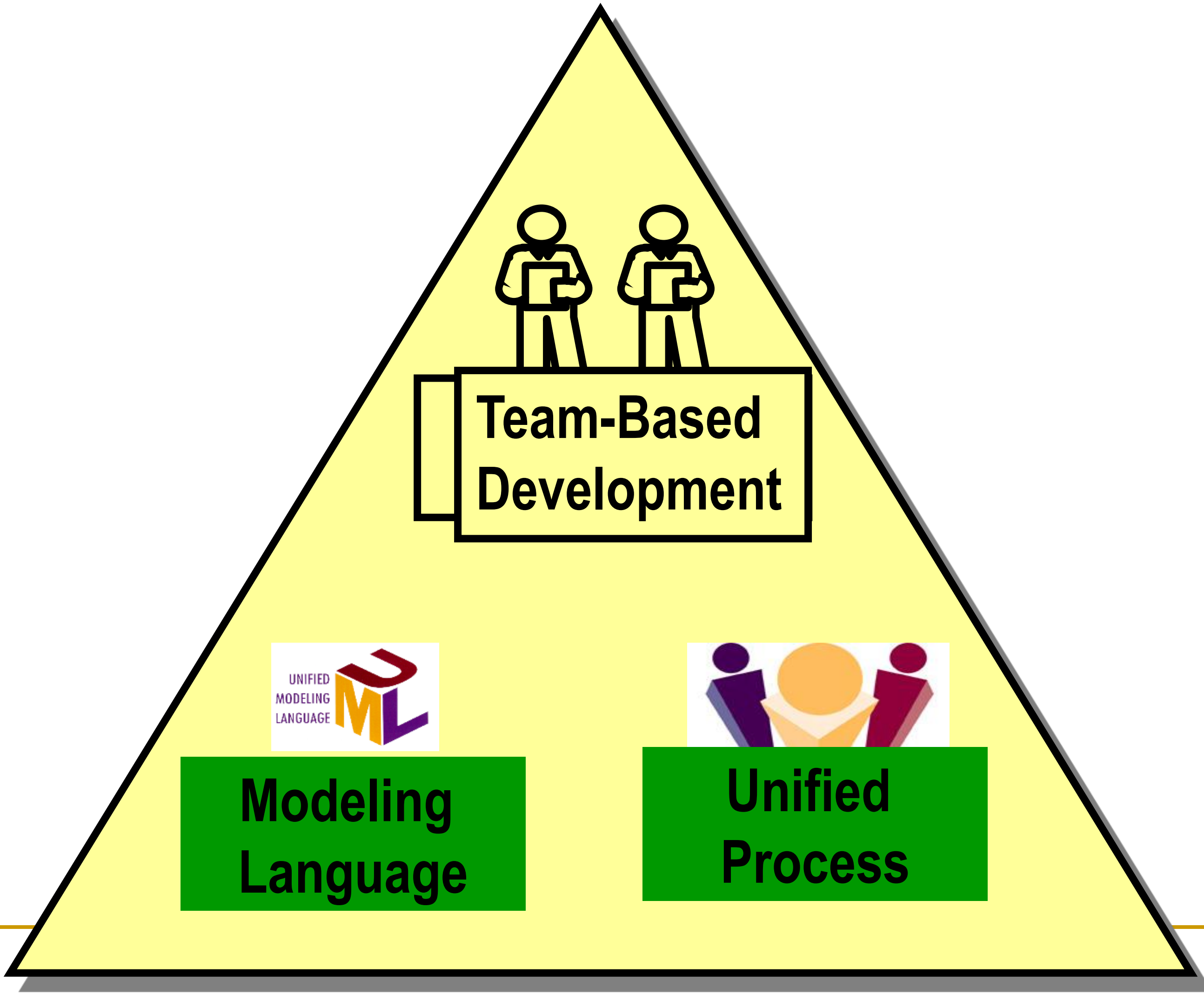
Outline

- Embedded System Design
  - Review – Unified Modeling Language (UML) and Rational's Unified Process (RUP)
    - Requirements Analysis
      - Use Cases
      - Relationships Between Use Cases
  - MARTE: UML Profile for Modeling and Analysis of Real-Time Embedded systems
    - Rational Rose RT -> Rhapsody
  - AADL: Architecture Analysis and Design Language

Design Methodology

- A **design methodology** consists of:
  - a **modeling language** consisting of a semantic framework and notational schema (UML, etc.), and
  - a **development process** governing the use of the language and the set of artifacts that result (RUP, etc.)

Software Development Triangle



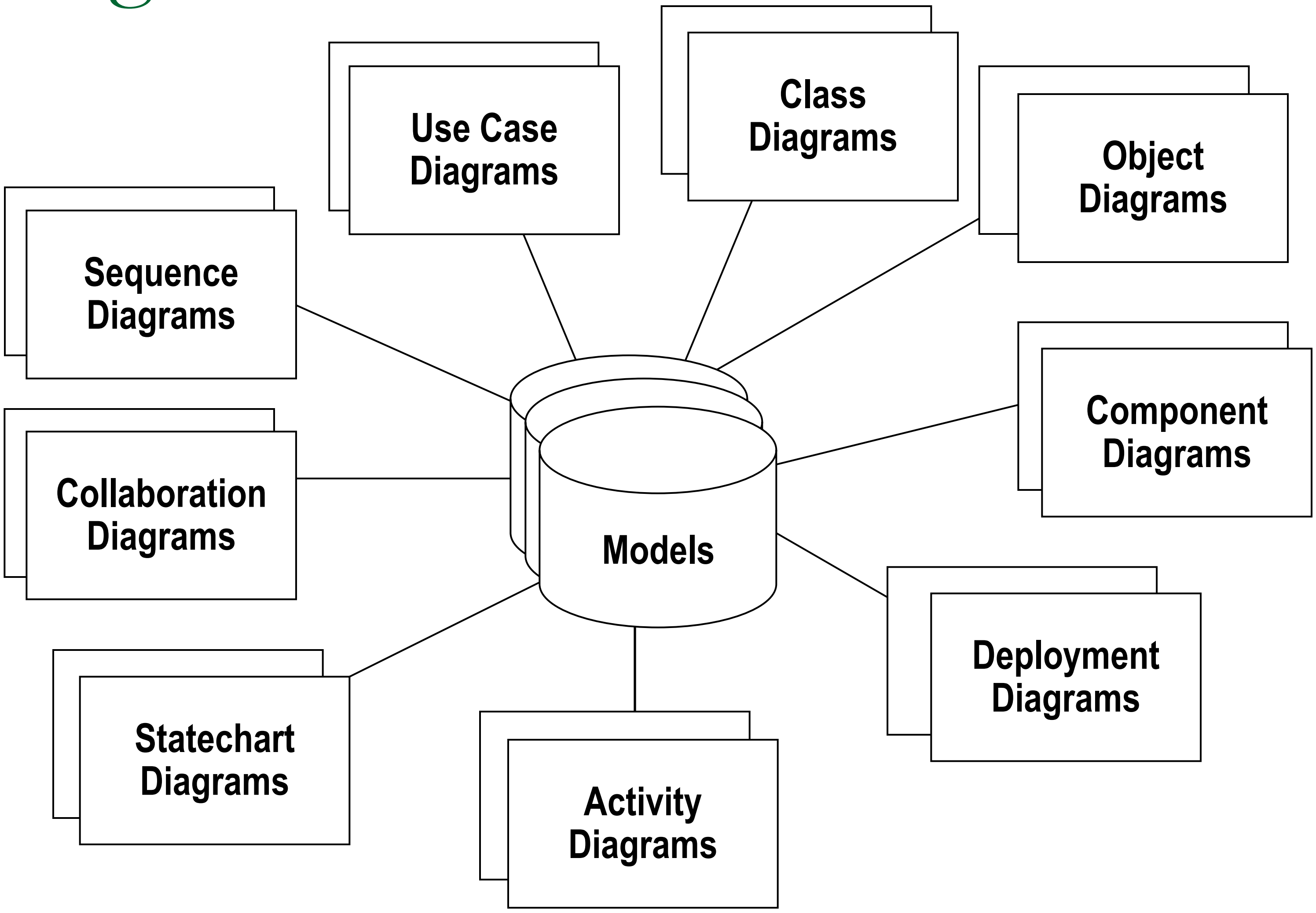
[Rational]

Development Process

- A **development process** defines:
  - who is doing what,
  - when it should be done, and
  - how to reach a specific goal.
- It describes **activities** that govern use of the language and the **design artifacts** (models, etc.) that are produced.

Design Artifacts

A **model** is a description of the system from a particular perspective.



Why Bother With A Process?

- To produce systems with consistent quality.
- To manage the development of complex systems.
- To predict completion time and development cost.
- To identify measurable milestones and generate iterative prototypes.
- To enable team-collaboration on large-scale systems.

Development Phases

- **Analysis** - identify essential characteristics of a correct solution.
- **Design** - define a particular solution based on the optimization of some criterion.
- **Implementation** - create an executable, deployable realization of the design.
- **Testing** - verify the translation and validate correctness of the implementation.

Sequencing Development Phases

- **Waterfall Lifecycle**
  - sequential ordering of Analysis, Design, Implementation, and Testing phases.
- **Iterative Lifecycle**
  - spiral cycles consisting of Analysis, Design, Implementation, and Testing phases to produce **Iterative Prototypes**.
  - enabling technology - automatic translation of description models into executable models.

Example: Network Architecture

Think Horizontally

Application Layer

Transport Layer

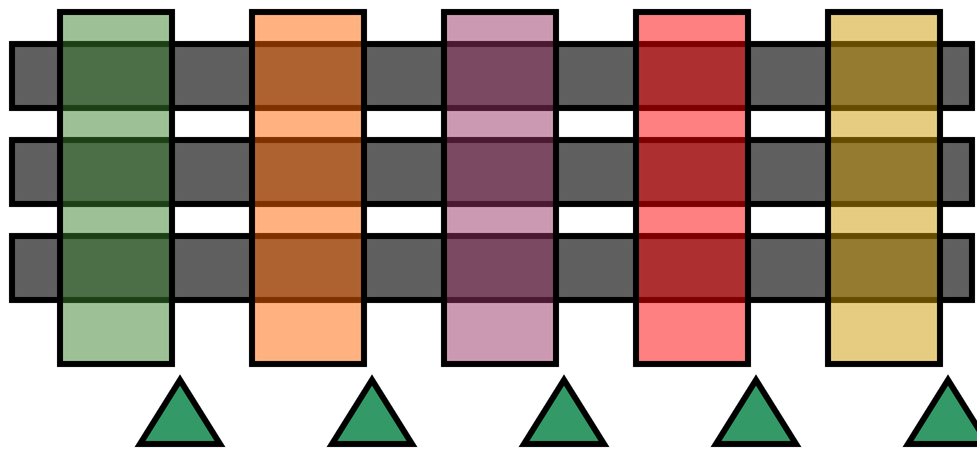
Data Link Layer

Construct Vertically

Application Layer

Transport Layer

Data Link Layer

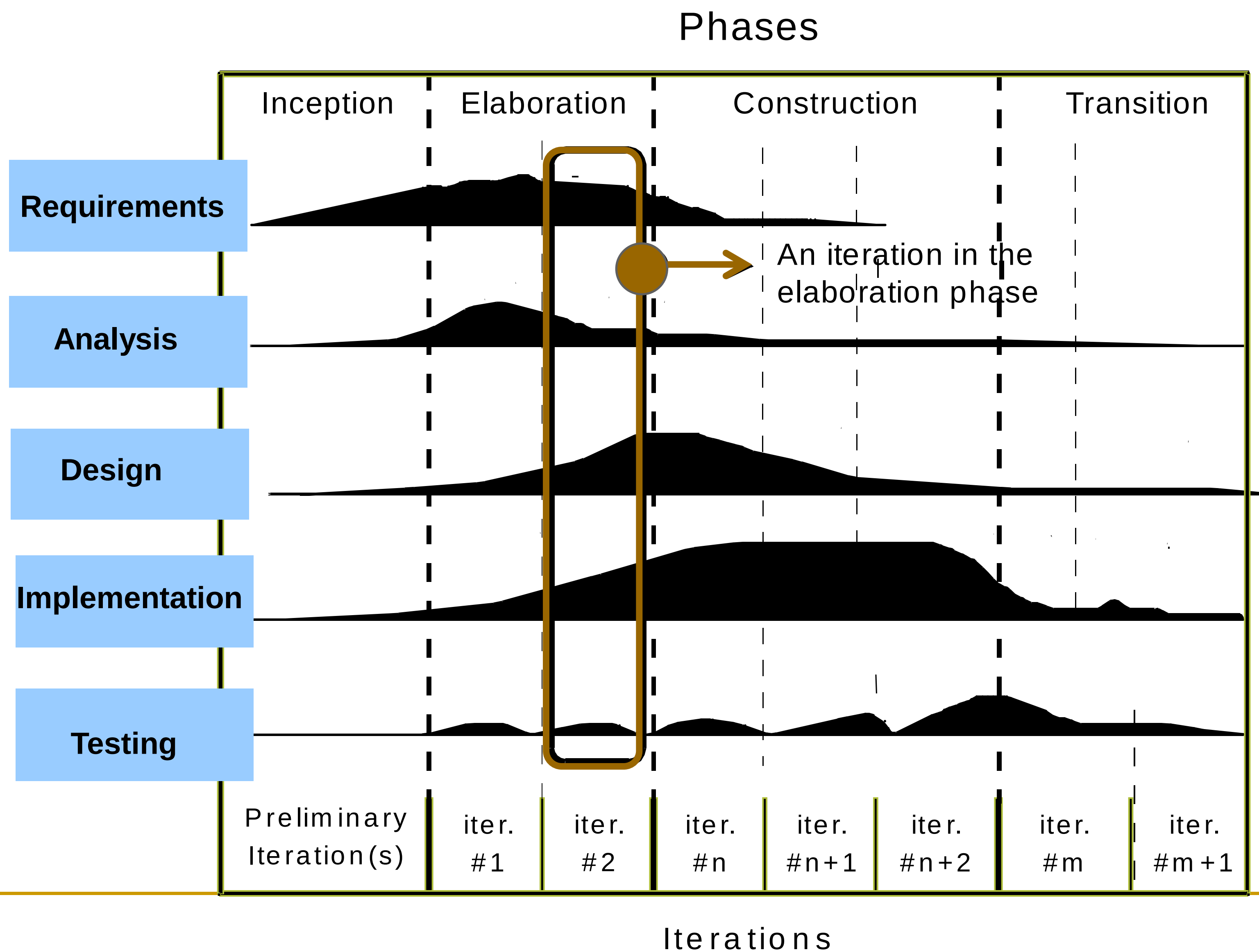


Vertical Prototypes

Iterations and Milestones

- An **iteration** is a sequence of activities with an established plan and criteria for evaluation, resulting in a release.
- Each **milestone** is completed after one or more iterations through each of the phases (Analysis, Design, Implementation, and Testing).

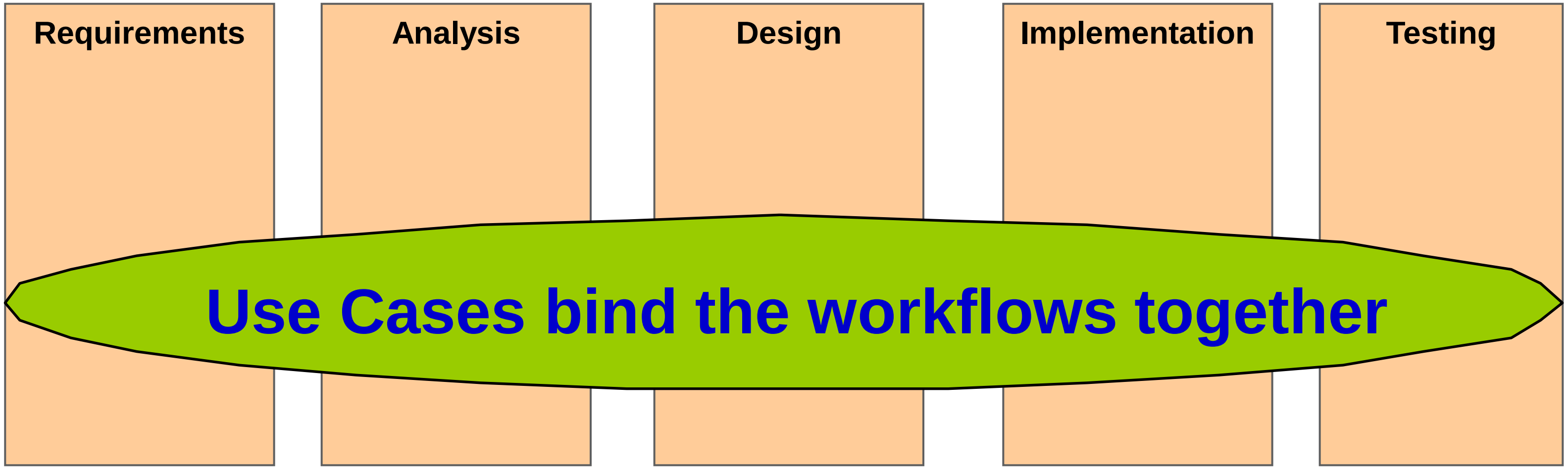
Iterations and Workflow



Unified Software Development Rational Unified Process (RUP)

- Iterative and Incremental
- Use Case Driven
- Architecture-Centric

Use Case Driven



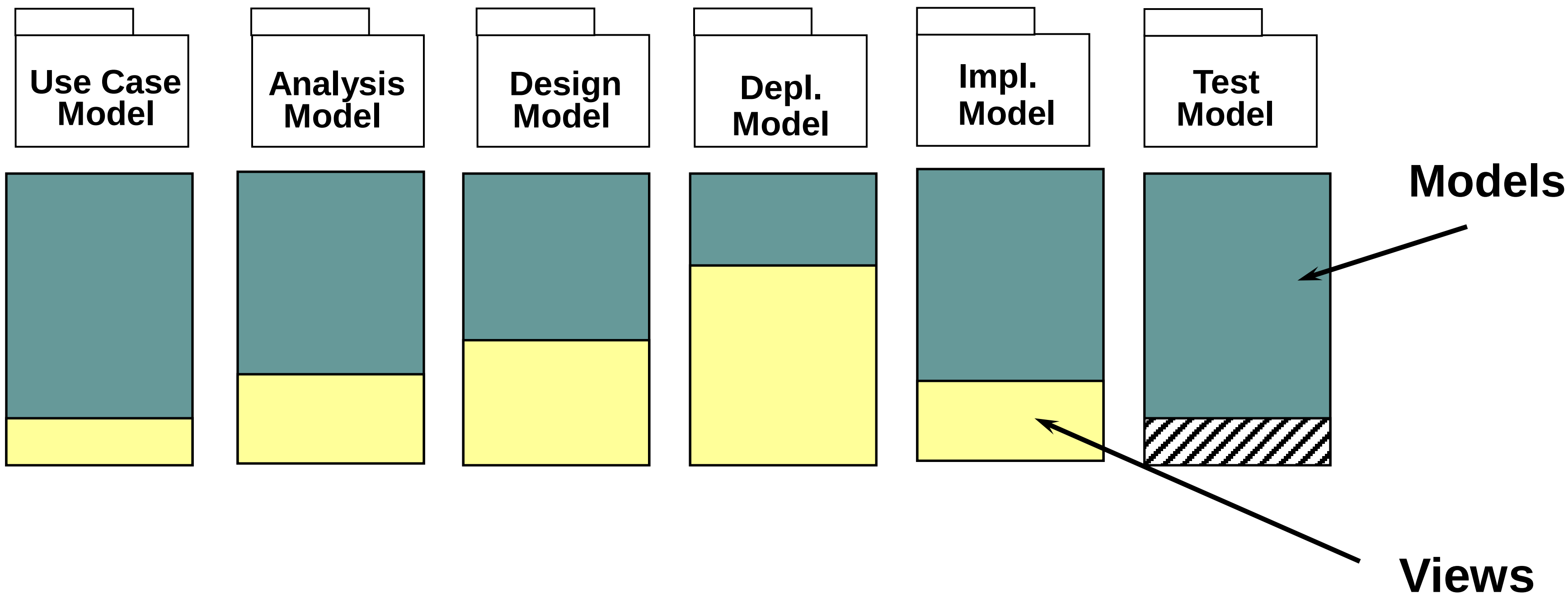
Use Case Driven Iterations

- Use Cases drive development activities:
  - Creation/validation of the system's architecture
  - Definition of test cases and procedures
  - Planning of iterations
  - Creation of user documentation
  - Deployment of system
- They also help to **synchronize** the content of different models.

Architecture-Centric

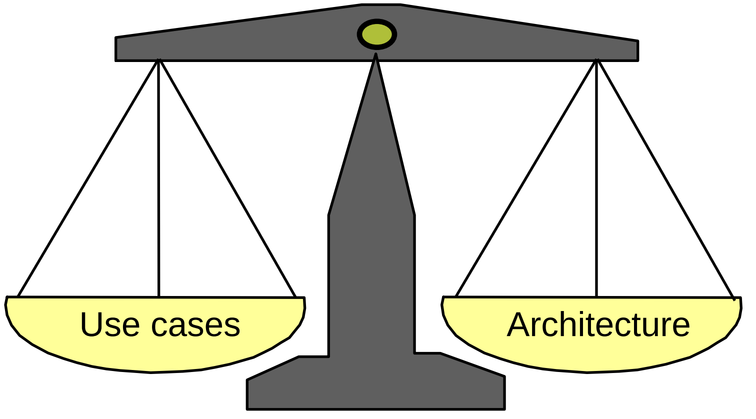
- Models are vehicles for visualizing, specifying, constructing, and documenting the architecture.
- The Unified Process prescribes the successive refinement of an executable architecture.

Architecture and Models



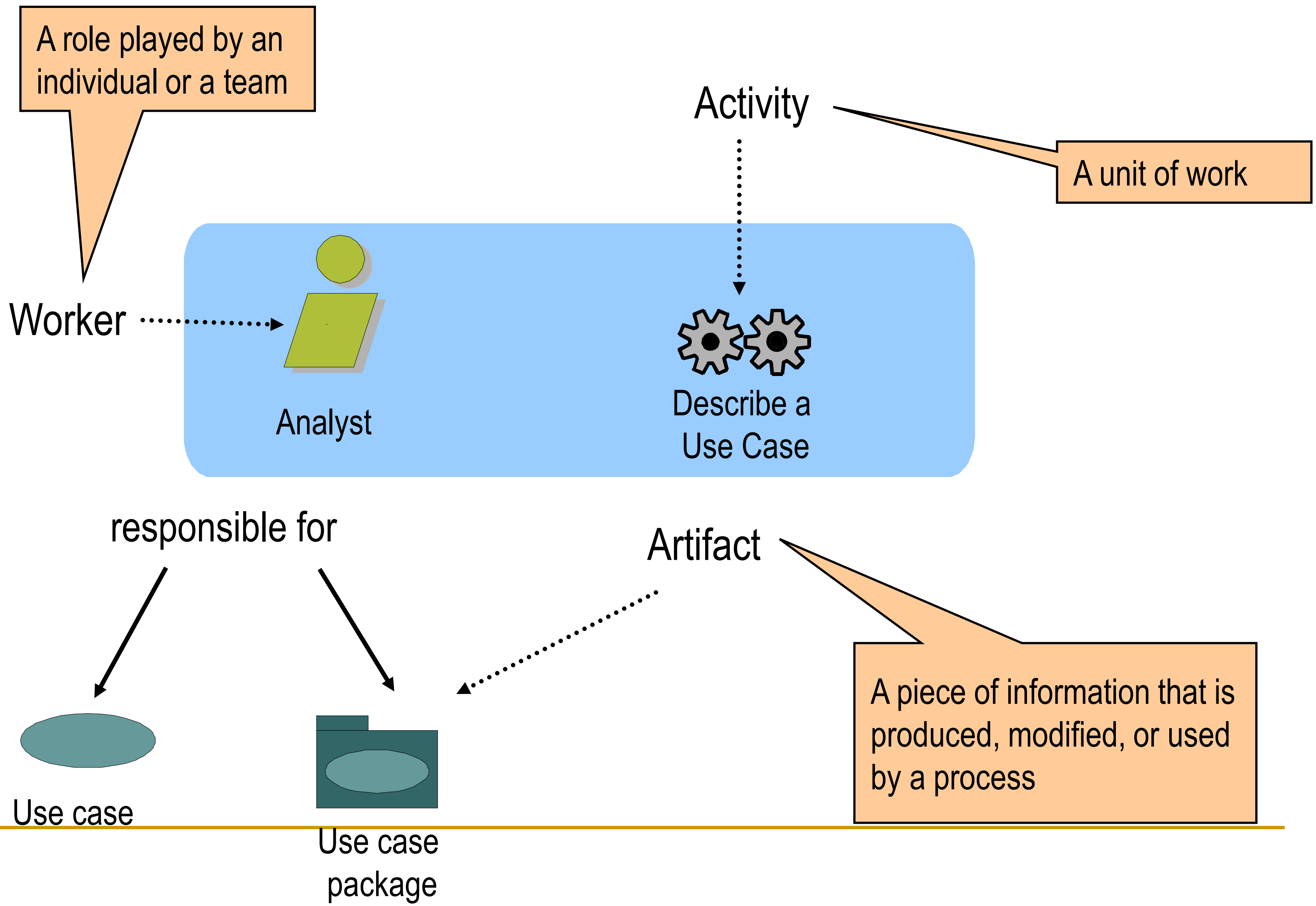
The **architecture** includes a set of views of the models.

Function versus Form

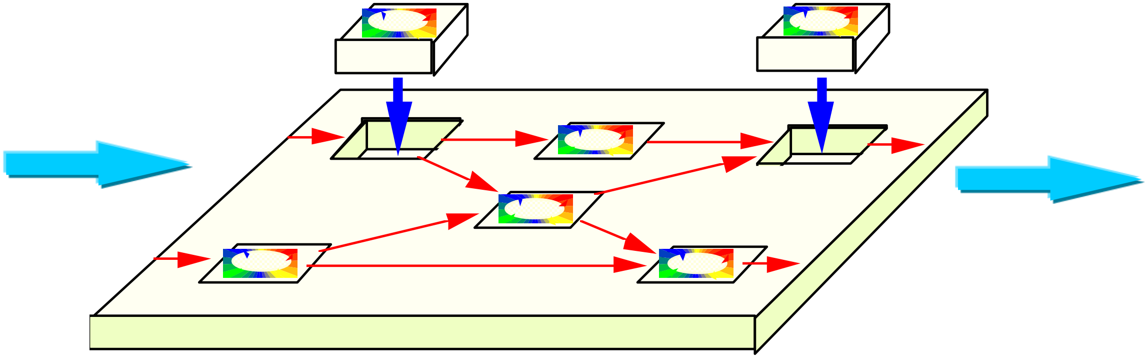


- Use cases specify **function** and the architecture specifies **form**.
- Use cases and architecture must be balanced.

The Unified Process is Engineered



Process Frameworks



There is no single Universal Process Framework.

Process frameworks (RUP, etc.):

- allow a variety of lifecycle strategies
- specify what artifacts to produce
- define activities and workers
- used to model concepts

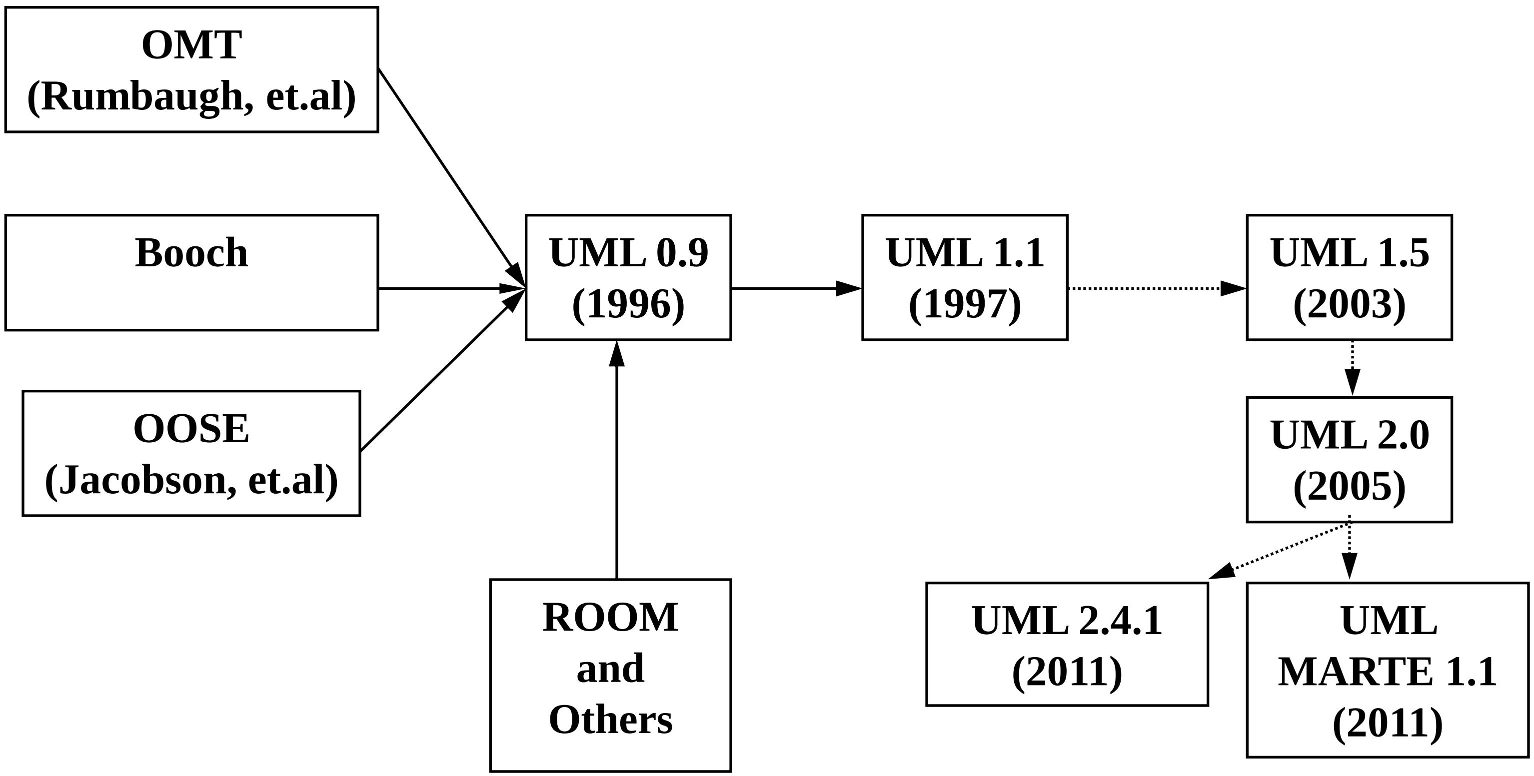
Two Parts of a Unified Whole

The Unified Modeling Language	+	Unified Design Process
OMG Standard 2.5		Convergence through process frameworks

Unified Modeling Language

- The **Unified Modeling Language (UML)** is a language for specifying, constructing, visualizing, and documenting artifacts of a software-intensive system.
- It focuses on a standard modeling language which represents the convergence of several popular object-oriented methodologies [*UML*, vers. 1.5, 2.5, www.omg.org/spec/UML/2.5].

Convergence of OO Methodologies



UML Reference: OMG: http://www.omg.org

UML Profile for MARTE

Date: June 2011

UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems

Version 1.1

Only 754 pages

OMG Document Number: formal/2011-06-02
Standard document URL: http://www.omg.org/spec/MARTE/1.1
Associated Files*: http://www.omg.org/spec/MARTE/20100801
http://www.omg.org/spec/MARTE/20100802

Original files: ptc/2010-08-33 (XMI), ptc/2010-08-34 (model library XMI)

Brief History

- 1967: Simula programming language
- 1970's: Smalltalk programming language
- 1980's: Theoretical foundations, C++, etc.
- 1990's: Object-oriented analysis and design methods (Booch, OMT, ROOM, etc.)
- 1997-2005: UML standardized by the Object Management Group (OMG)

Advantages

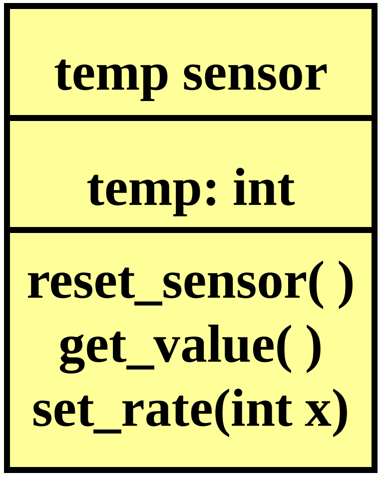
- Consistency of model views
- Improved problem-domain abstraction
- Improved reuse and scalability
- Improved reliability and safety
- Inherent support for concurrency

Disadvantages

- Perceived as immature technology for embedded systems
- Lack of compilers and other tools
- Perceived inefficiency of objects
- Lack of trained developers

Terms and Concepts

- An **object** is used to model a unique real-world or conceptual entity and includes:
  - Identity
  - Attributes (values)
  - Behaviours



- **Objects** are instances of classes.
- A **class** is an abstraction of the elements commonly shared by a set of objects.

Terms and Concepts

- Classes relate to other classes by **relations**:
  - **Associations** bind classes together to enable communication via messages.
  - **Links** are instances of associations between objects at a specific point in time.
  - **Aggregations** apply when one object contains another object.
  - **Composition** is a strong form of aggregation.
  - **Generalization** is when one class is a specialization of another class.

Terms and Concepts

- **Messages** are an abstraction of object communication.
- **Use cases** describe the primary and secondary functions of a system.
- A **scenario** is a specific path (sequence of operations on objects) through a use case.
- **Actors** are interacting objects outside the scope of a system.

Objects

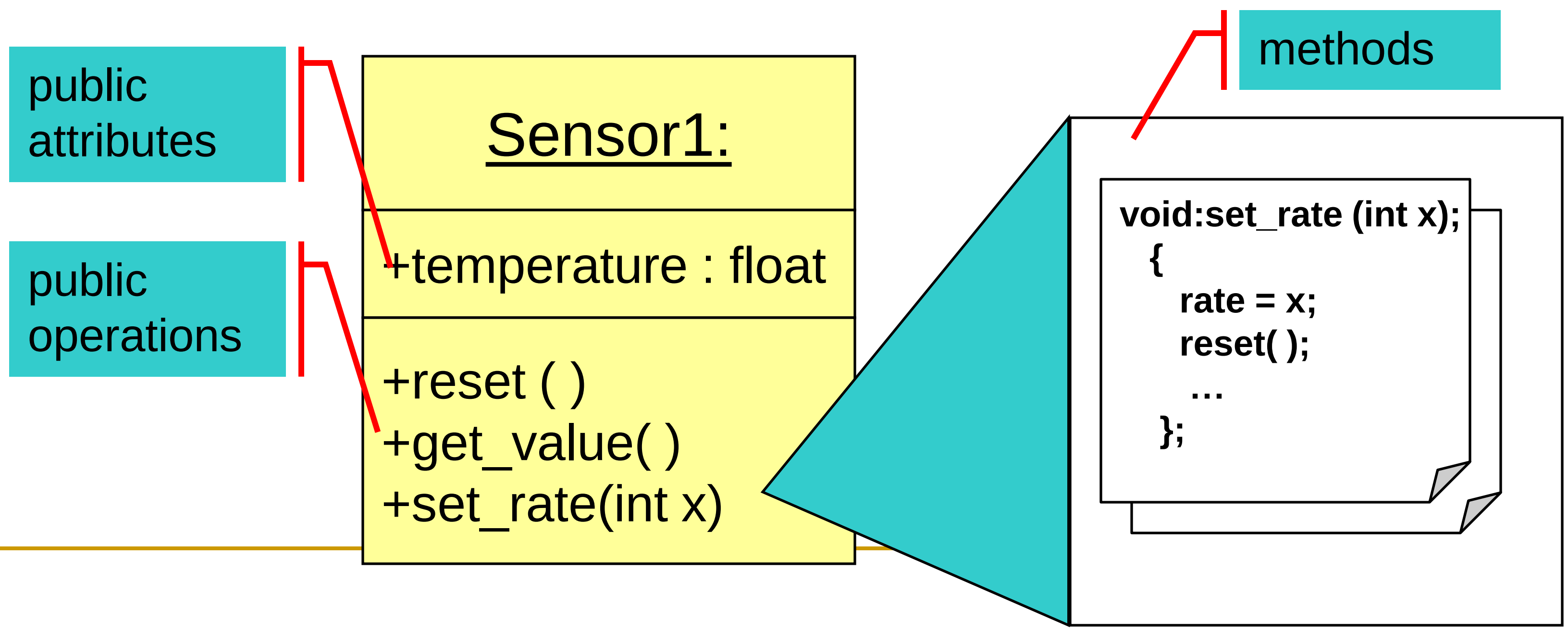
- All objects are entities that model some **physical or conceptual** entity. They have several aspects at run-time:
  - Identity
  - Attributes (data or named property)
  - Behavior (operation or method)
  - State (memory)
  - Responsibilities

Example: Sensor Object

- **Attributes:** Sensor Value, Rate of Change (RoC)
- **Behavior:** Acquire, Report, Enable, ...
- **State:** Last Sensor Value, Last RoC
- **Identity:** Instance for robot arm joint
- **Responsibility:** Provide information about the location of the robot arm in absolute space coordinates.

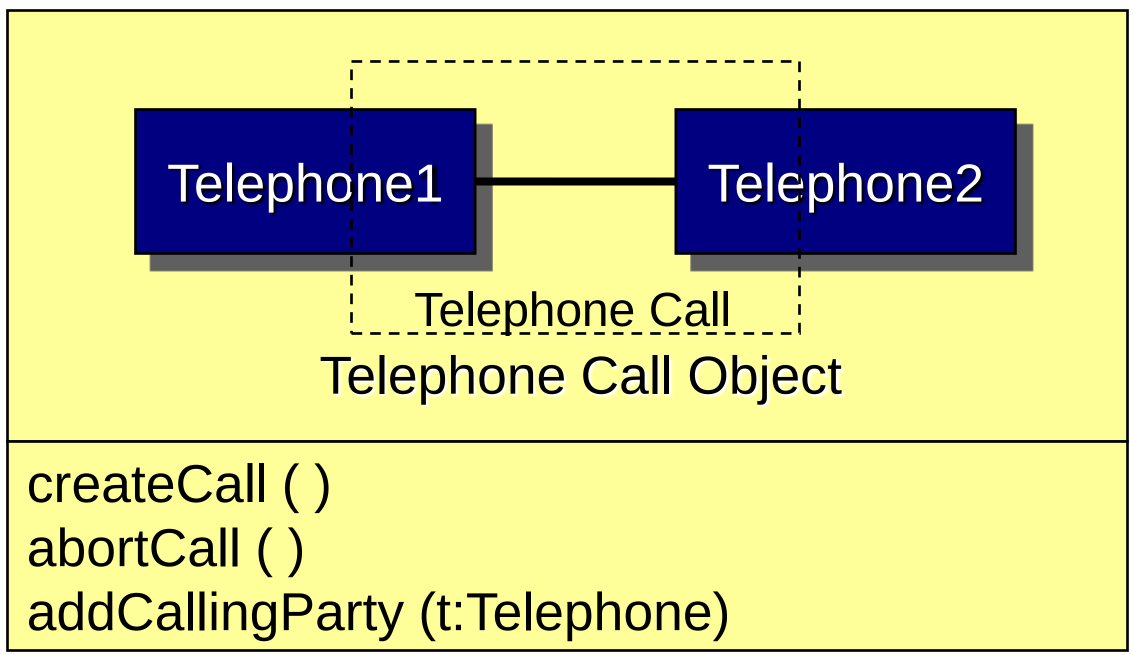
Object Components

- Public interface
- Hidden (encapsulated) implementation



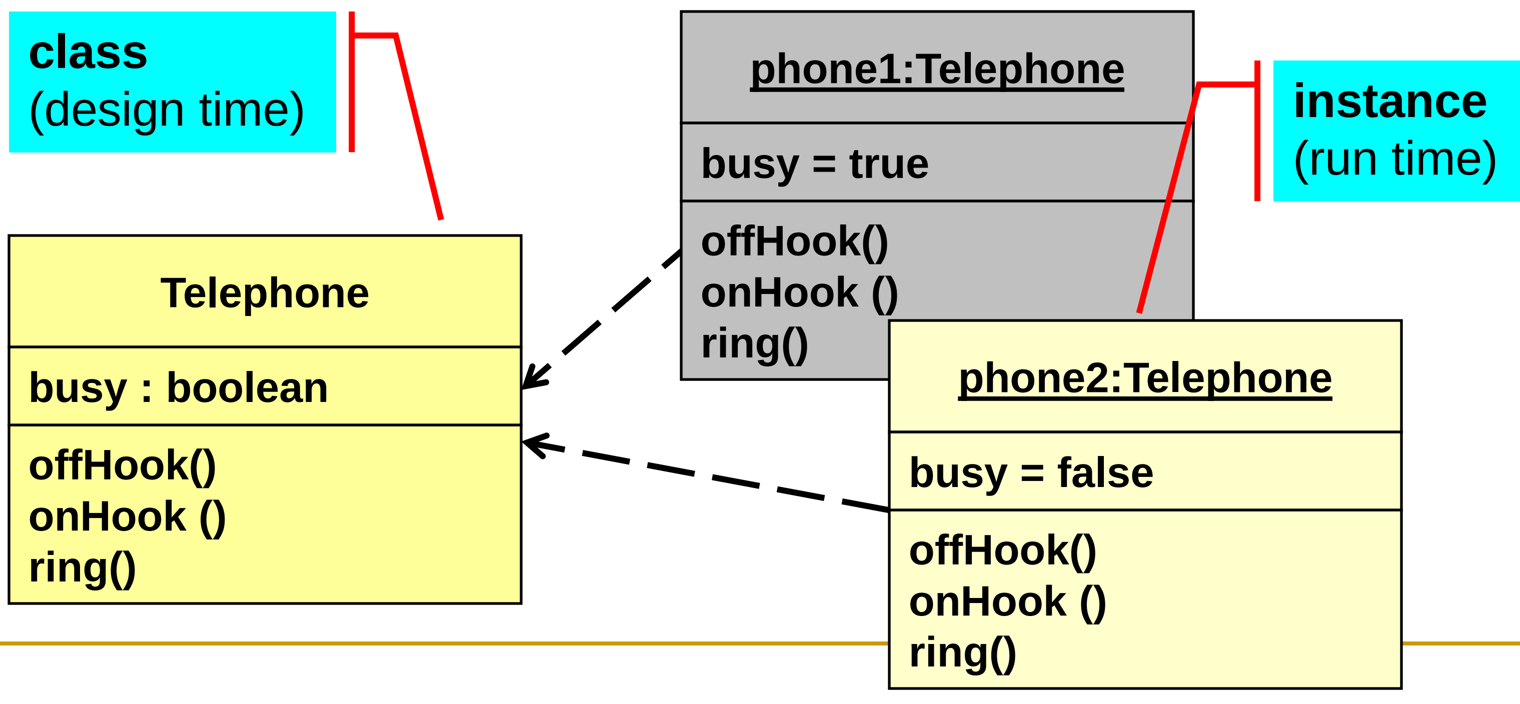
Conceptual Objects

- Not all objects represent physical entities.
- For example, the “telephone call” object:



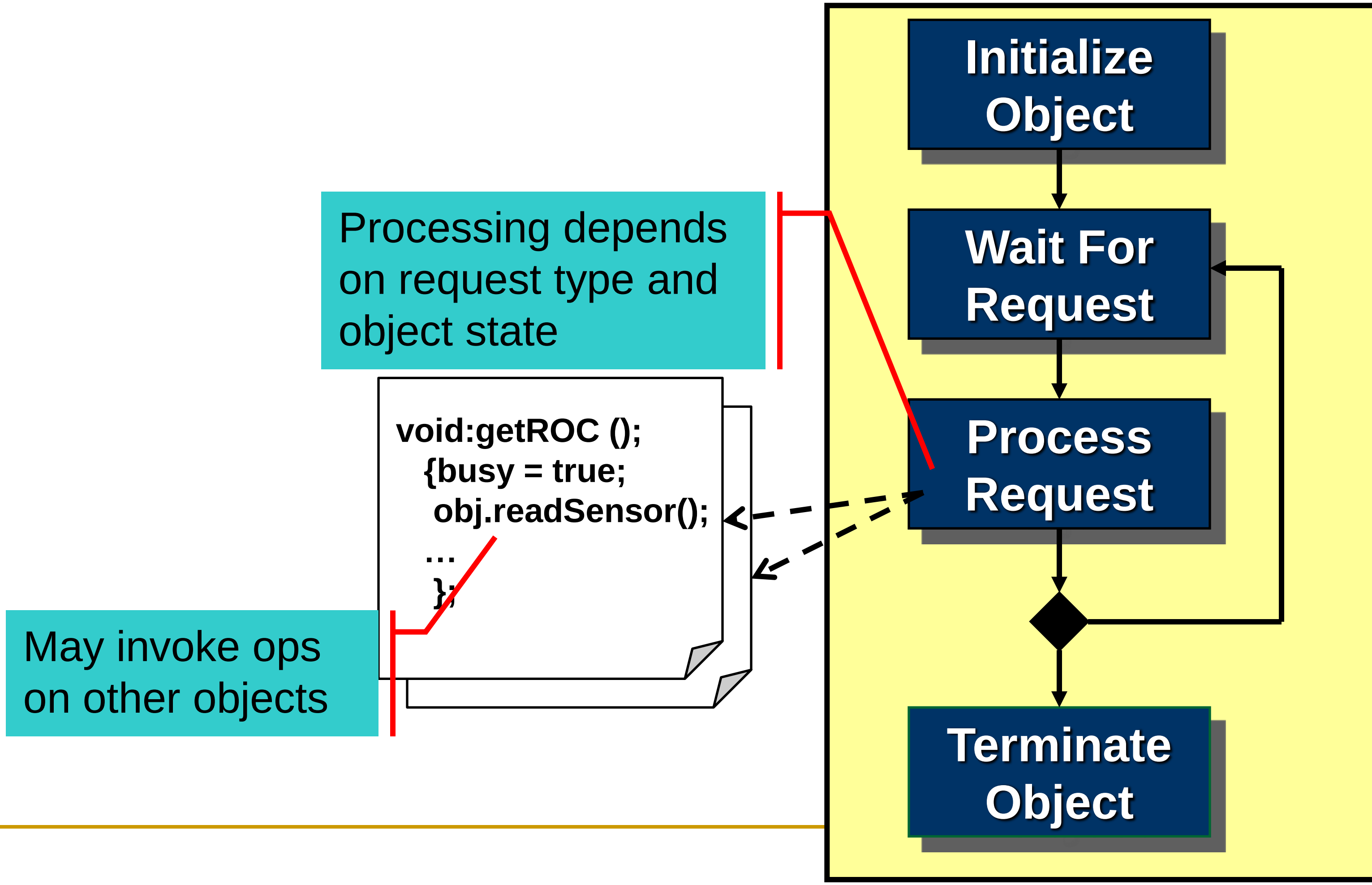
Classes and Instances

- Design-time specifications can be used to instantiate one or more distinct objects at run-time with a common form (structure and behavior)



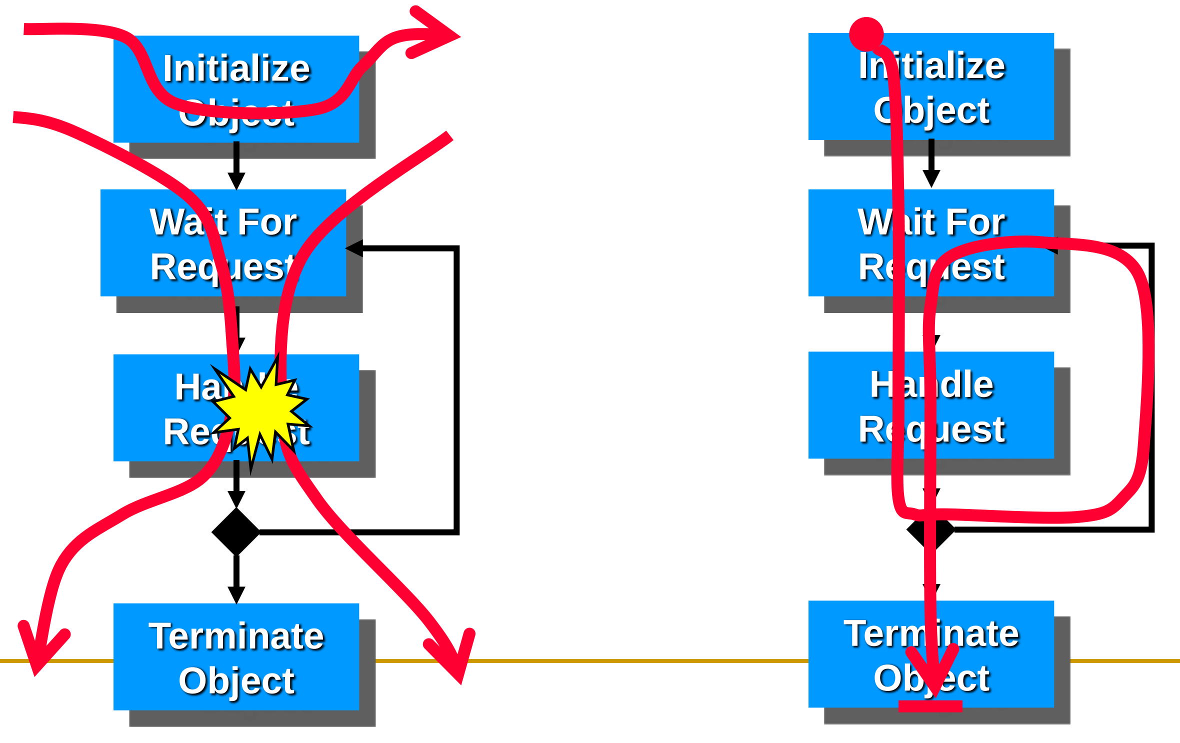
Object Behavior

- Example: Simple **reactive server model**:

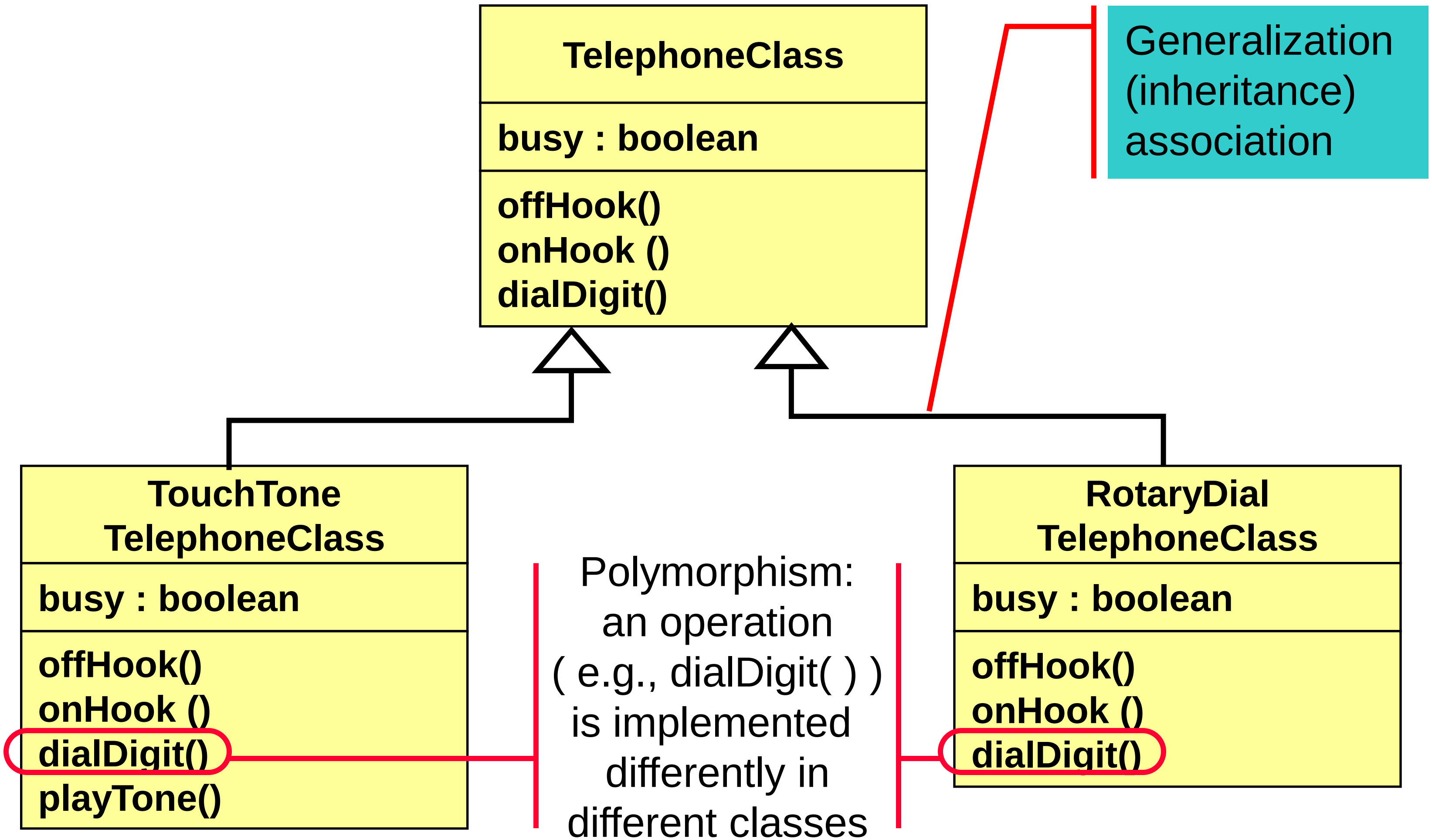


Types of Objects

- Passive Objects - invoked by external threads
- Active Objects - include own single thread



Inheritance and Polymorphism



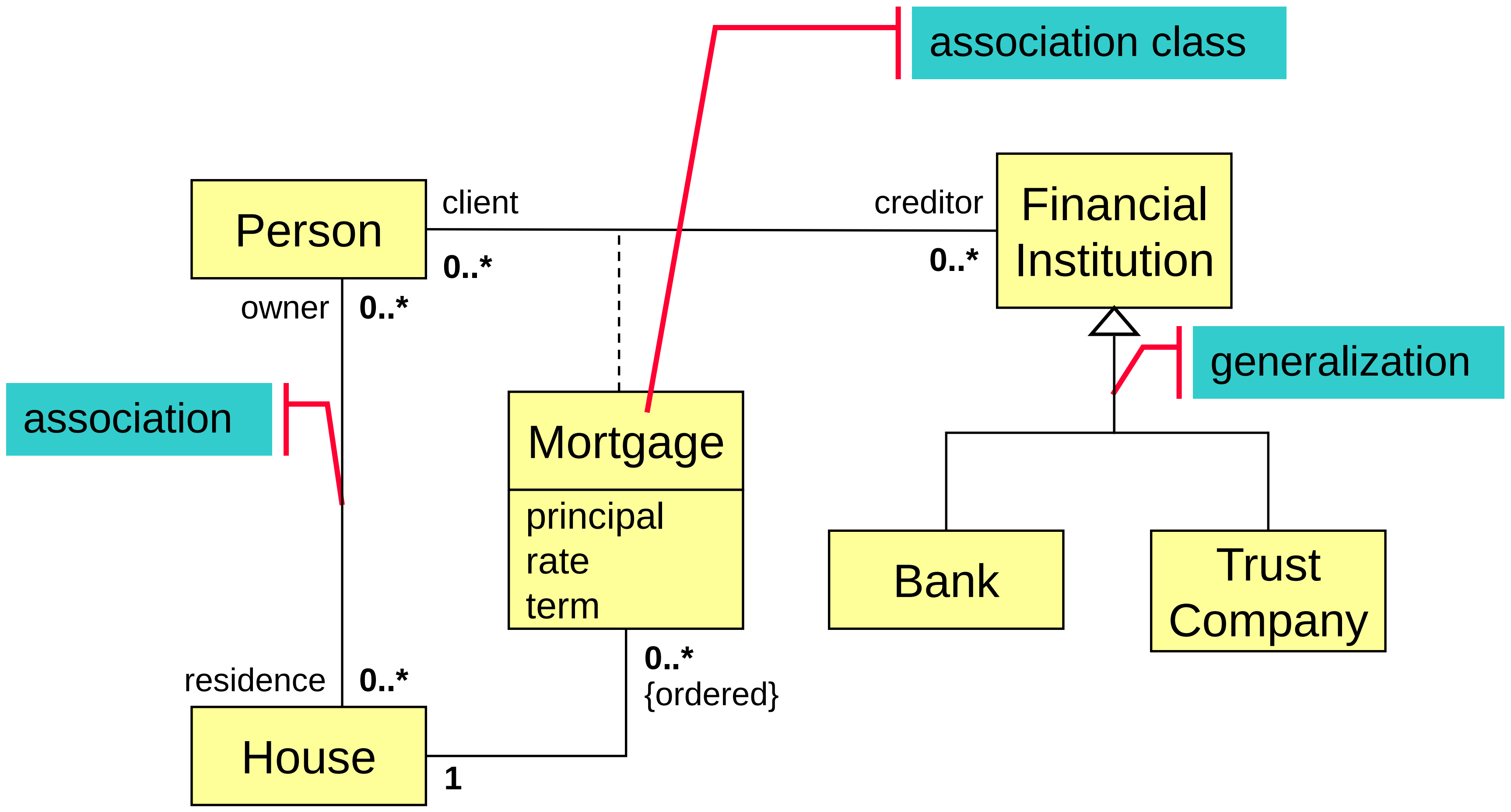
Why Build Models?

- Understand the problems better
- Facilitate communication between customers and developers
- Find errors or omissions in the design
- Plan out the design and analysis
- Automate code generation

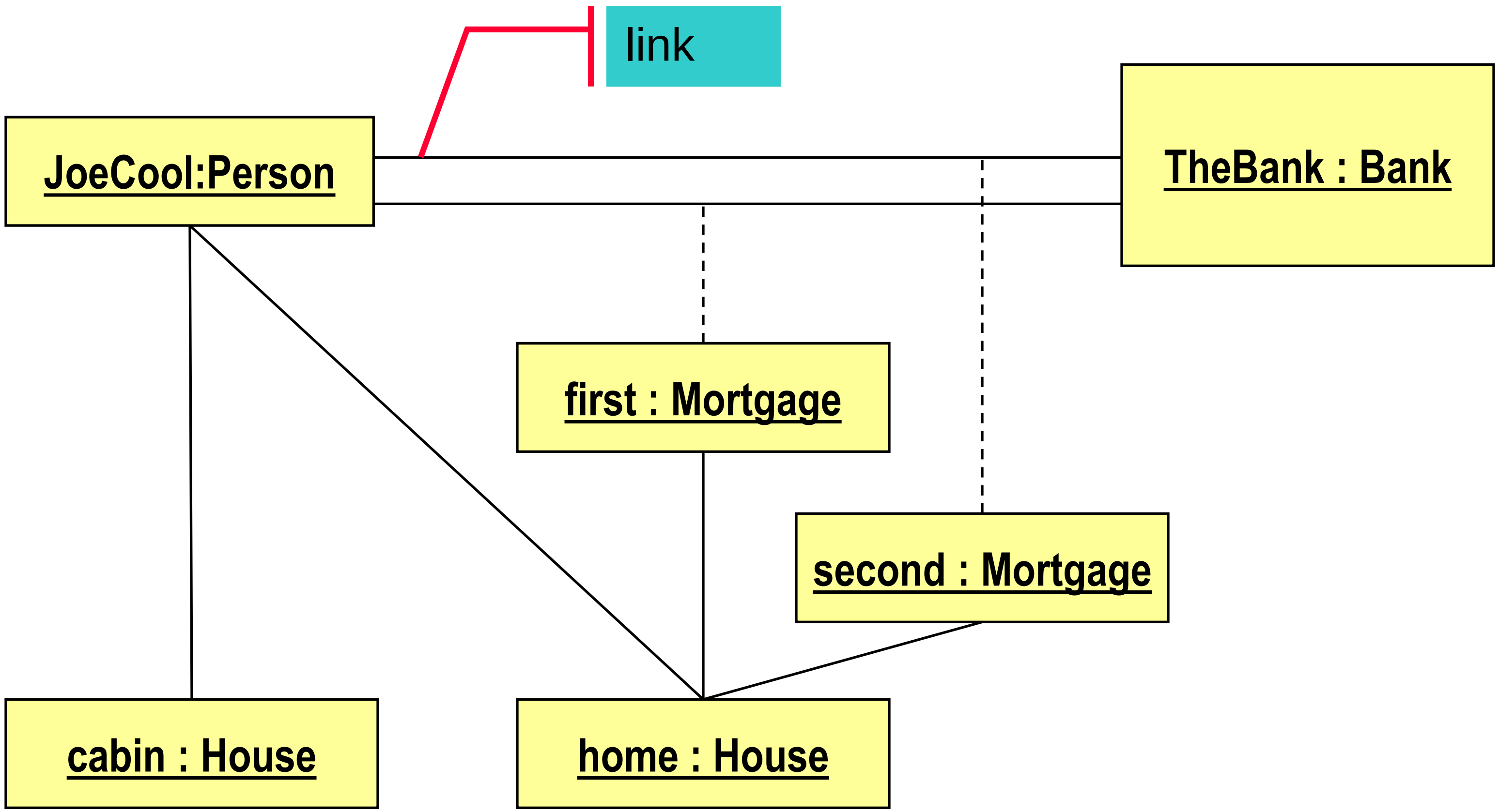
UML Models

- Requirements (use case diagrams)
- Static structure (class diagrams)
- Dynamic behavior (state machines)
- Interactive behavior (activity, sequence, and collaboration diagrams)
- Physical implementation structures (component and deployment diagrams)

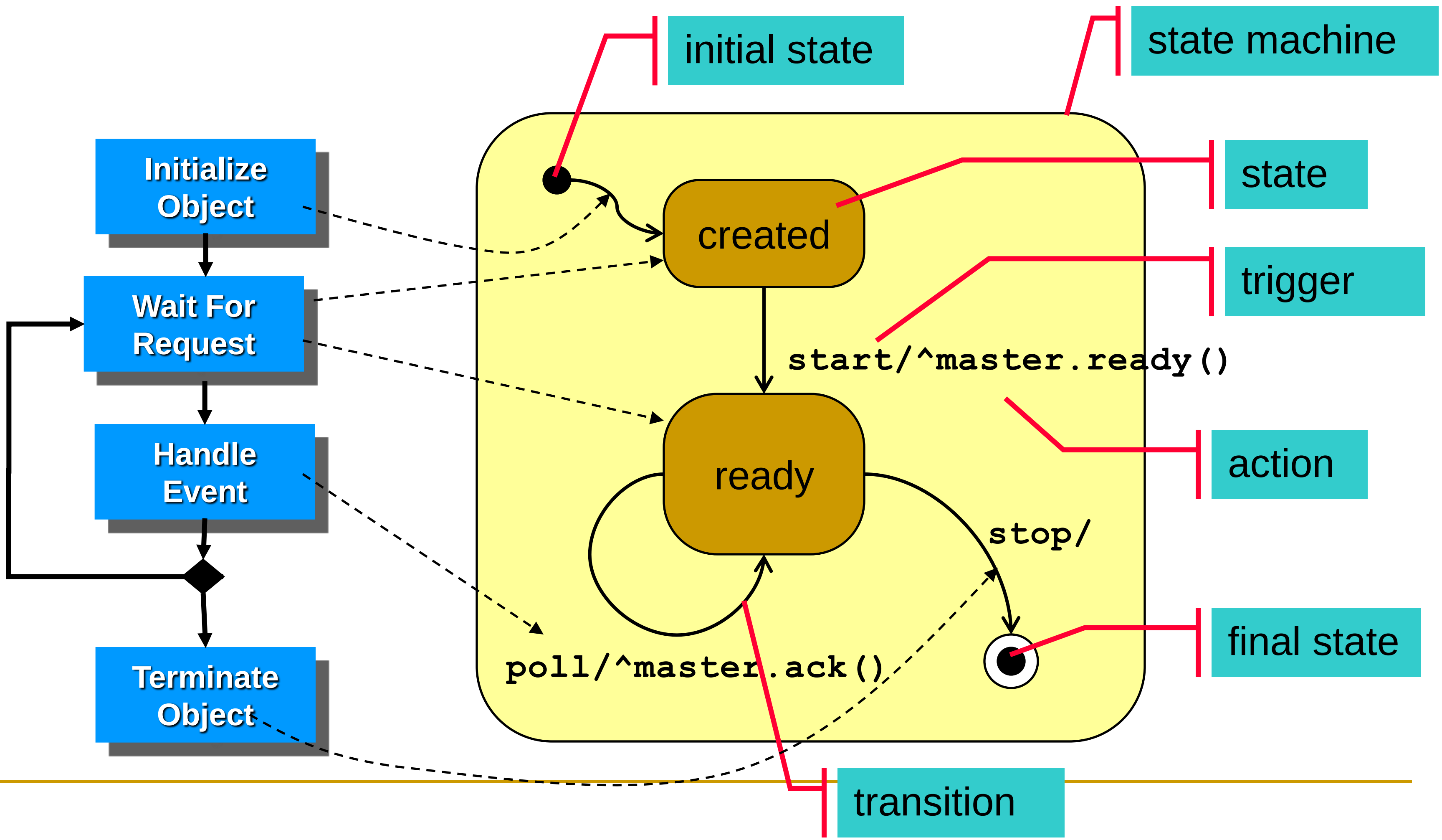
Class Diagram - Static Structure



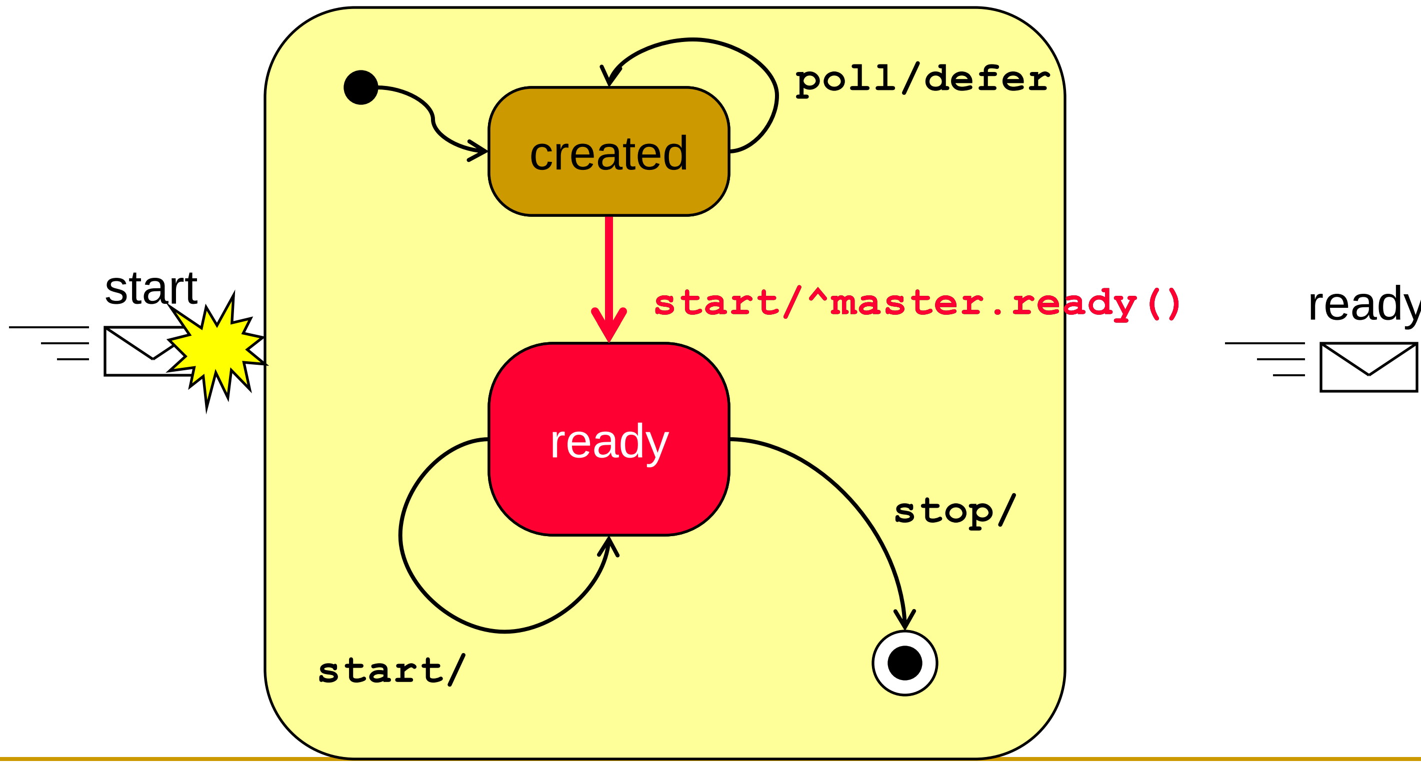
Object Instance Diagram



State Machine Diagram

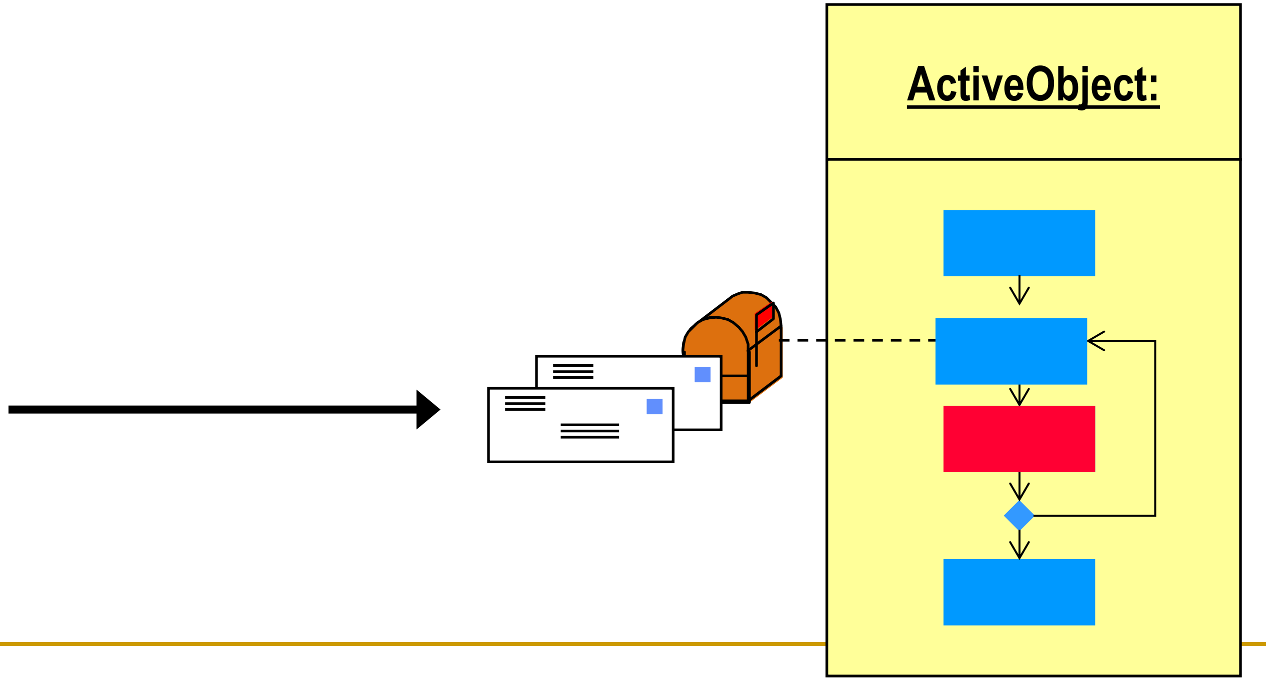


State Machine Behavior



Active Objects in the UML

- Concurrent incoming events are queued and handled one-at-a-time regardless of priority; e.g., run-to-completion (RTC) execution model

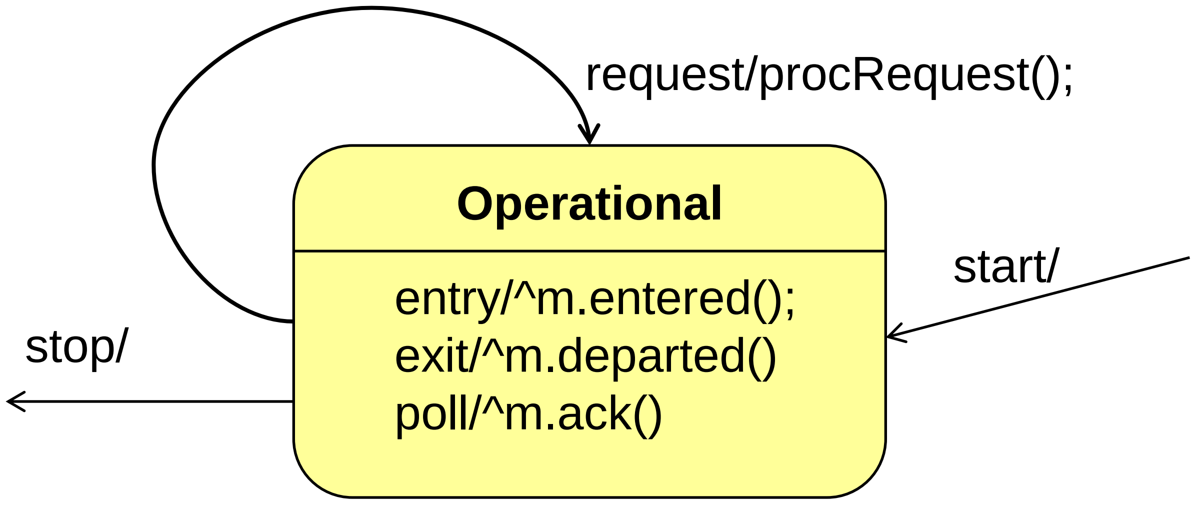


RTC and Concurrency

- Eliminates need to write synchronization code:
  - if all passive objects are encapsulated by an active object, only a single thread can pass through each passive object
  - an active object acts as an implicit critical section

Types of Actions

- **Entry action:** executed on state entry
- **Exit action:** executed on state departure
- **Internal transition:** a self transition



Requirements Analysis

- **Requirements** are used to specify the functionality that must be provided by the system. They are typically understood and specified by **domain experts**.
- A **use case model** documents the system’s intended functions (use cases), surroundings (actors), and relationships between actors and use cases.

System Requirement Categories

- **Functional Requirements** - define system behavior as viewed from the outside (system as a black box).
  - Example: When a sensor detects a weed, the corresponding sprayer should be activated.
- **Quality of Service (QoS) Requirements** - specify performance, reliability, and safety properties of functional requirements.
  - Ex: Actuate the sensor within 15 msec.

Use Case

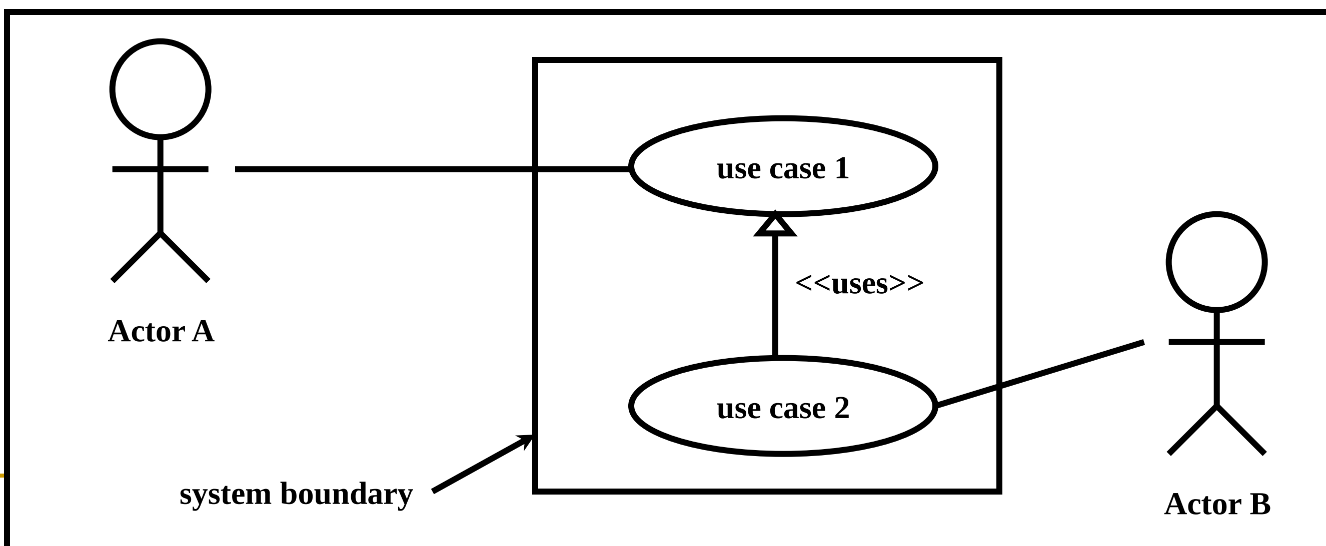
- The main tool used to capture **functional requirements**.
- A coherent piece of functionality visible (in black box form) from outside the system.
- Strictly behavioral, does **not** define or imply a specific internal structure (objects or classes).

Use Case Motivation

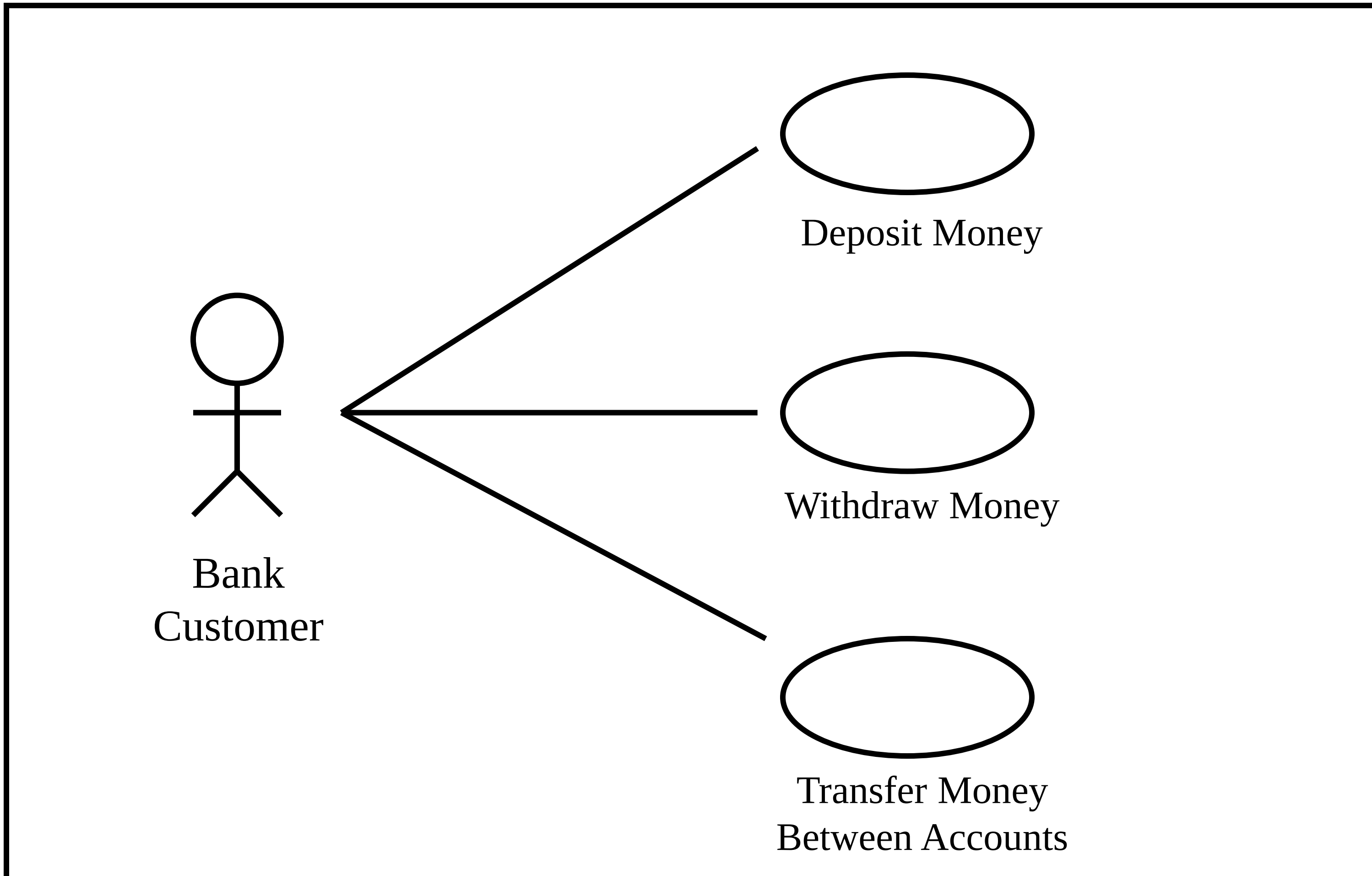
- Use cases help with the three of the most difficult aspects of development:
  - capturing requirements,
  - planning iterations of development, and
  - system testing
- They were first introduced by Ivar Jacobson (in the early 1990’s).

Actors

- Objects in the system’s external universe that interact with the system.
- Human users, external subsystems, or devices.



A Simple Use Case Diagram



Jacobson, "Software Reuse", Fig. 3.16

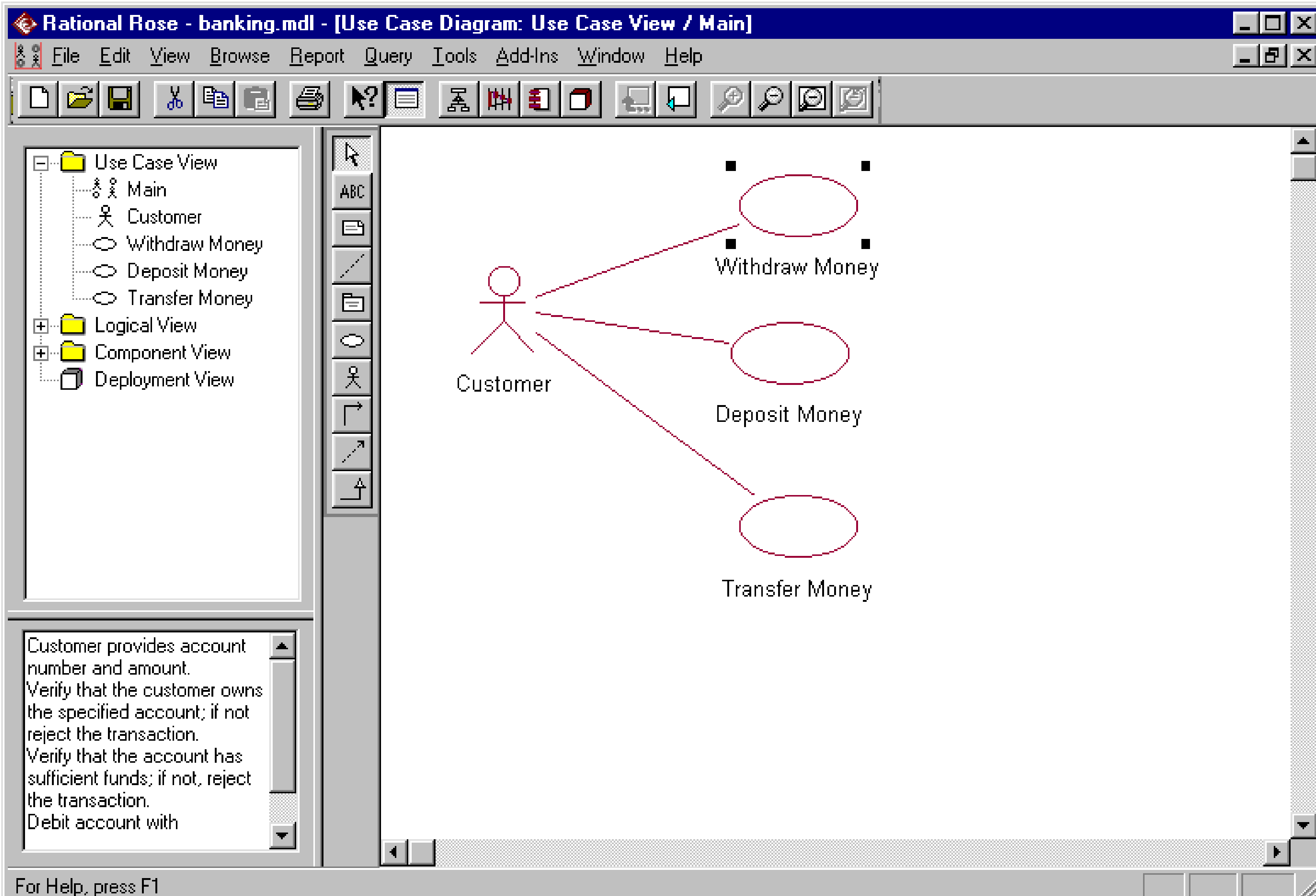
Actors

- An **actor** is an object in the system's external universe that interacts with the system; e.g., human users, external subsystems, or devices.
- More specifically, an actor **represents** a class of users; e.g., a bank may have many customers represented by one actor.
- An actor is represented as a stick figure in a use case diagram.

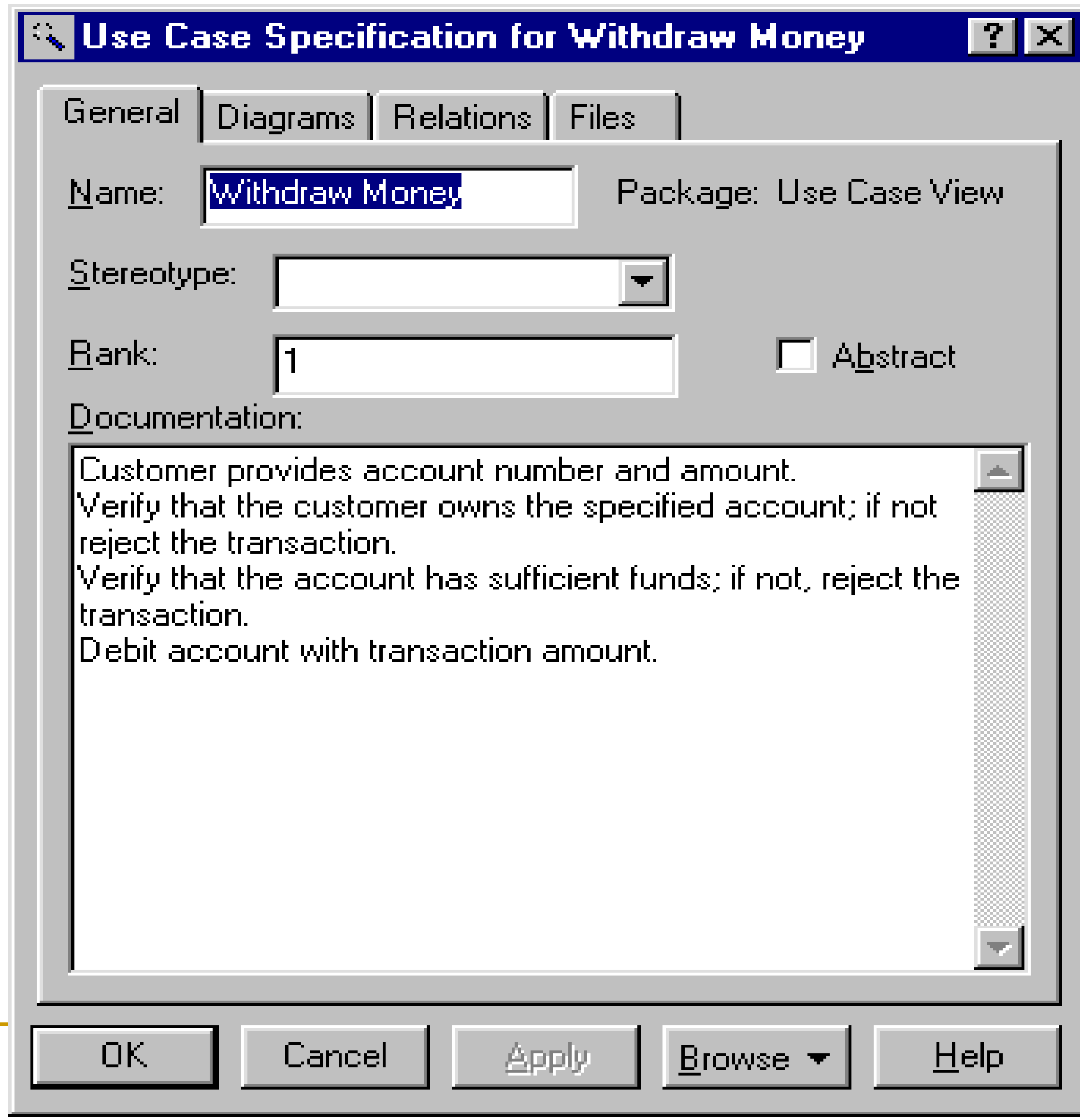
Use Cases

- A **use case** is shown on a use case diagram as a named oval. The name describes the coherent unit of work; e.g. Withdraw Money.
- A **use case** includes a description of the sequence of messages exchanged between the system and any actors, and actions performed by the system in response.

Use Case Diagram



Textual Part of Use Case



Requirements Capture

- Use cases help in requirements capture by providing a structured way to:
  - identify the actors
  - for each actor, identify
    - **what they need from the system**
    - **interactions they expect to have with the system**
    - **use cases in which they participate**

Notes

- Some aspects of system behavior may not show up as use cases for actors.
- Some use cases may not interact with any actor; these use cases are called **abstract**.
- In Rational Rose, relationships between use cases are called “generalizations”; note that use case generalizations are different than generalizations between classes.

Development Planning

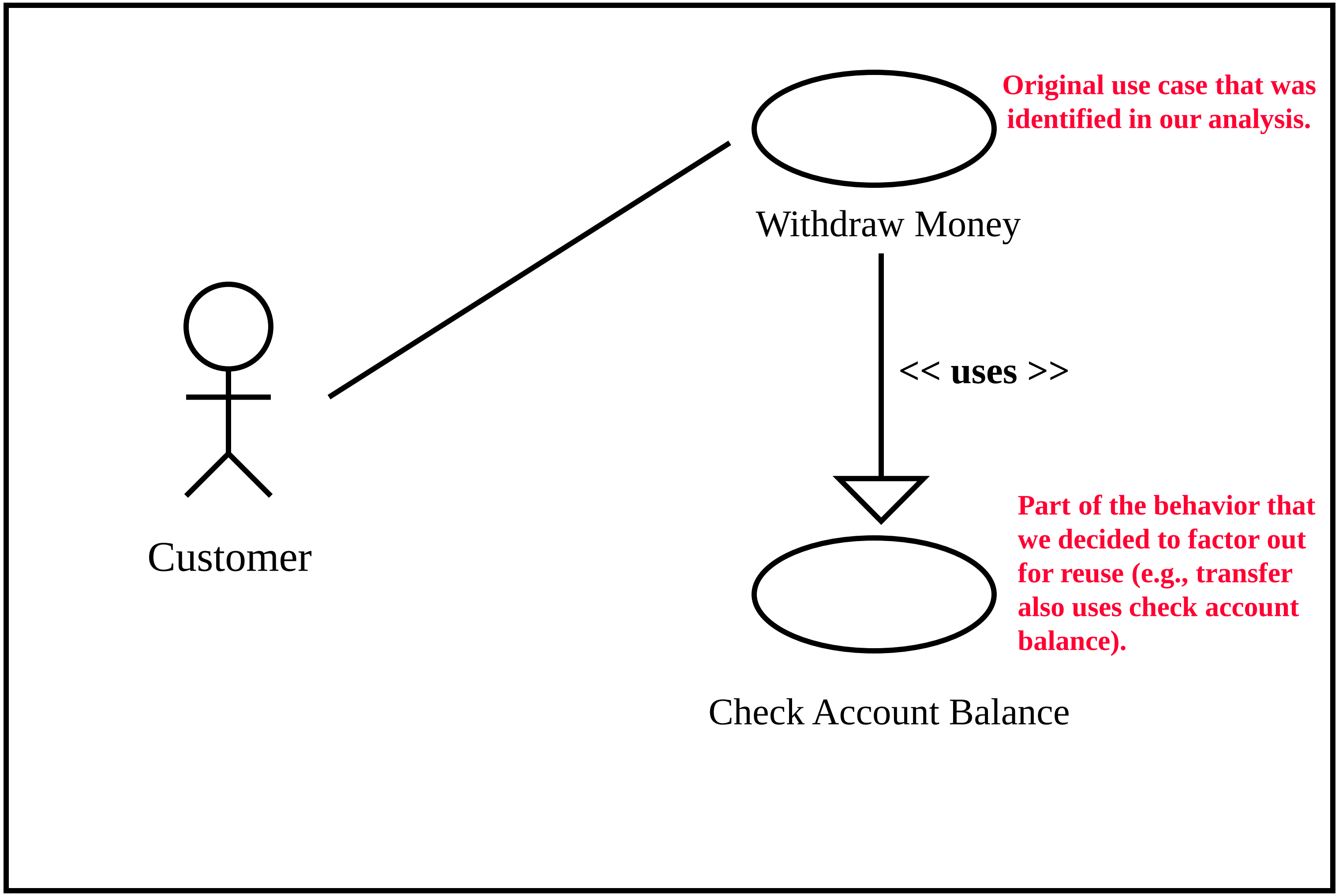
At the end of the Elaboration Phase (or by the end of analysis), we should have generated a complete list of use cases with:

- an understanding of what is important to whom,
- which use cases carry the most risk including requirements risk, technological risk, safety risk, performance risk, and skills risk, and
- a plan for how long it should take to implement each use case.

Relationships Between Use Cases

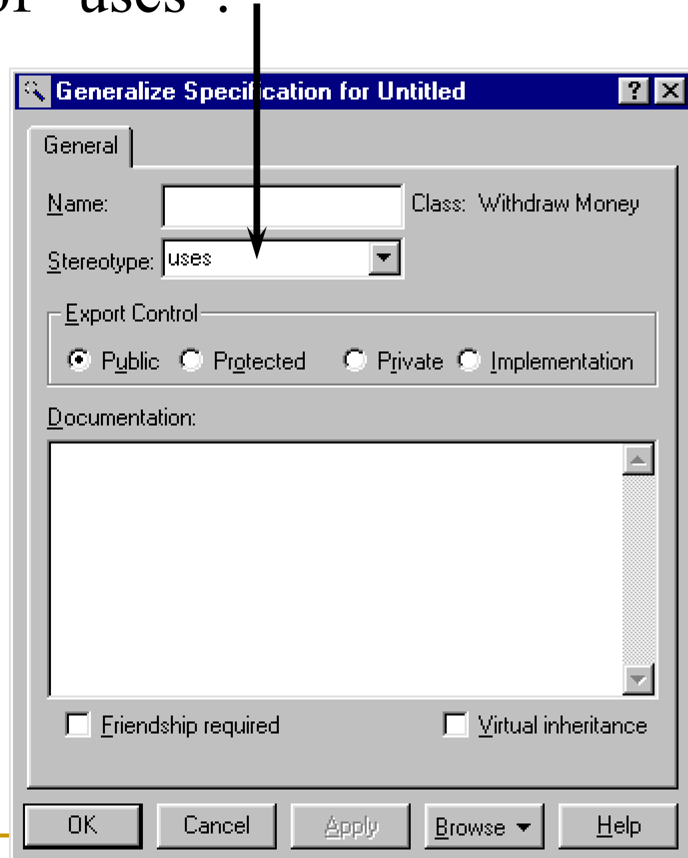
- Represented as an open-headed arrow on the use case diagram.
- Two types of relationships are distinguished by giving different stereotypes:
  - << uses >> - to reuse a use case; the source use case makes use of the target use case
  - << extends >> - to separate variant behavior; the source use case specializes or extends the behavior of the target use case

<< uses >>

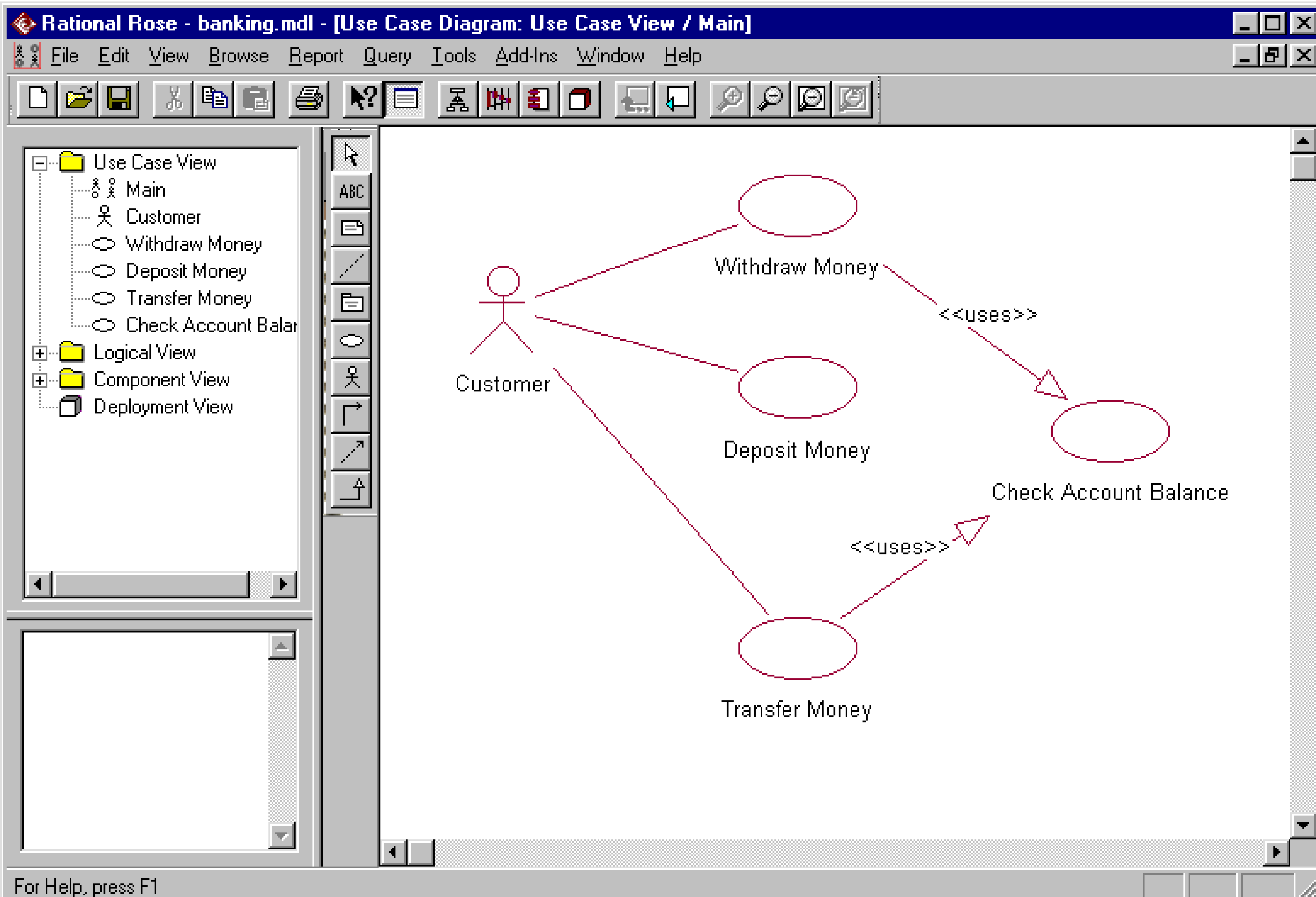

Original use case that was identified in our analysis.
Withdraw Money
<< **uses** >>
Customer
Part of the behavior that we decided to factor out for reuse (e.g., transfer also uses check account balance).
Check Account Balance

Rational Rose Example

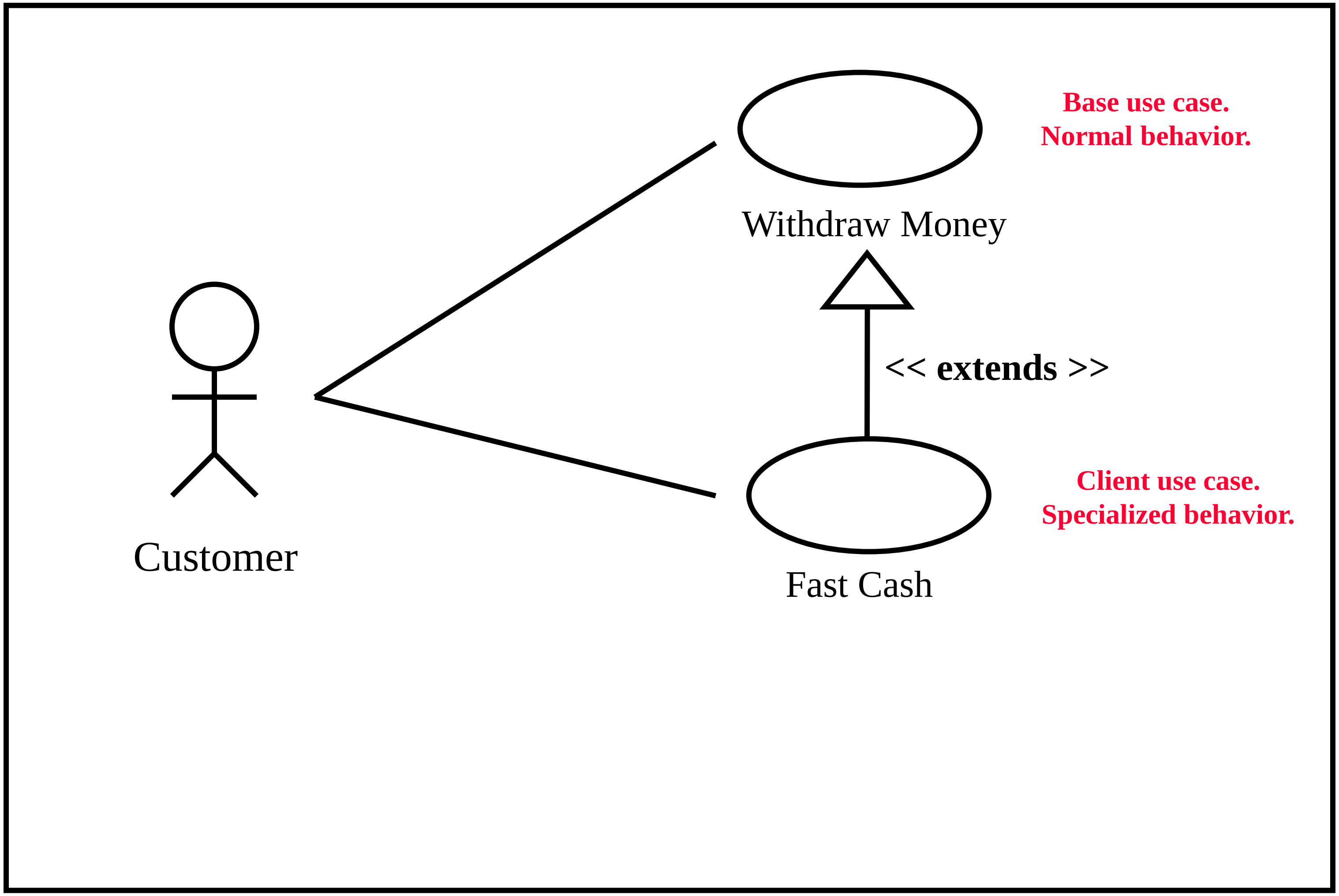
Double-click on the generalization arrow and specify a stereotype of “uses”.



Example (continued)



<< extends >>


Base use case.
Normal behavior.
Withdraw Money
<< **extends** >>
Customer
Client use case.
Specialized behavior.
Fast Cash

Problems With Use Cases

- Focusing on use cases may cause developers to lose sight of the system architecture; e.g., designing a top-down, function-oriented, inflexible system.
- Developers may mistake requirements for design; e.g., focus only on operational requirements.
- By focusing on actors, developers may miss some use cases; e.g., internal ones.

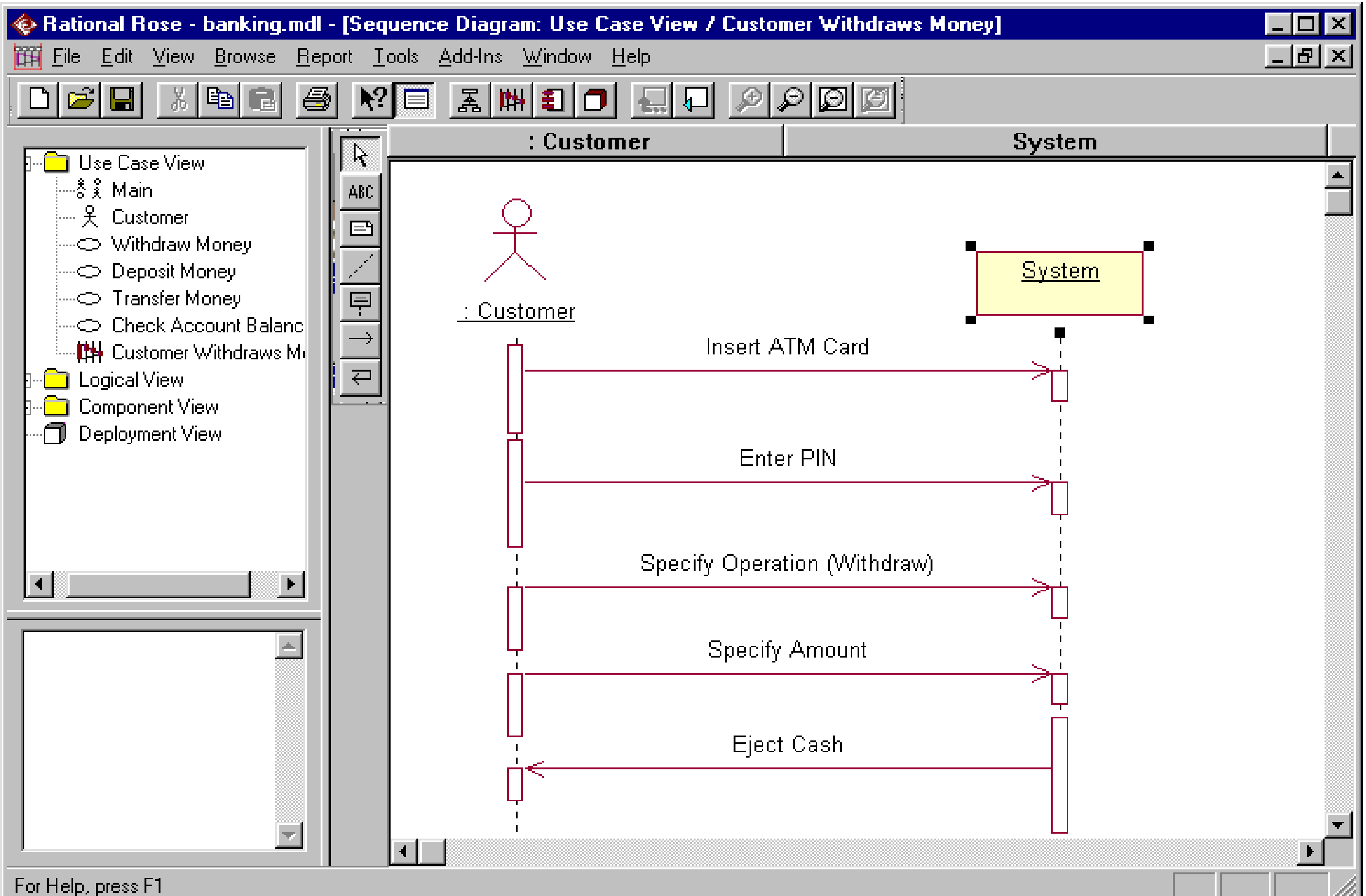
Detailing Use Case Behavior

- Textual Description - textual part of use case (previous slide)
- Scenarios
  - Sequence Diagrams - show the sequence of messages between objects
  - Collaboration Diagrams - show collaboration between objects
- Statecharts

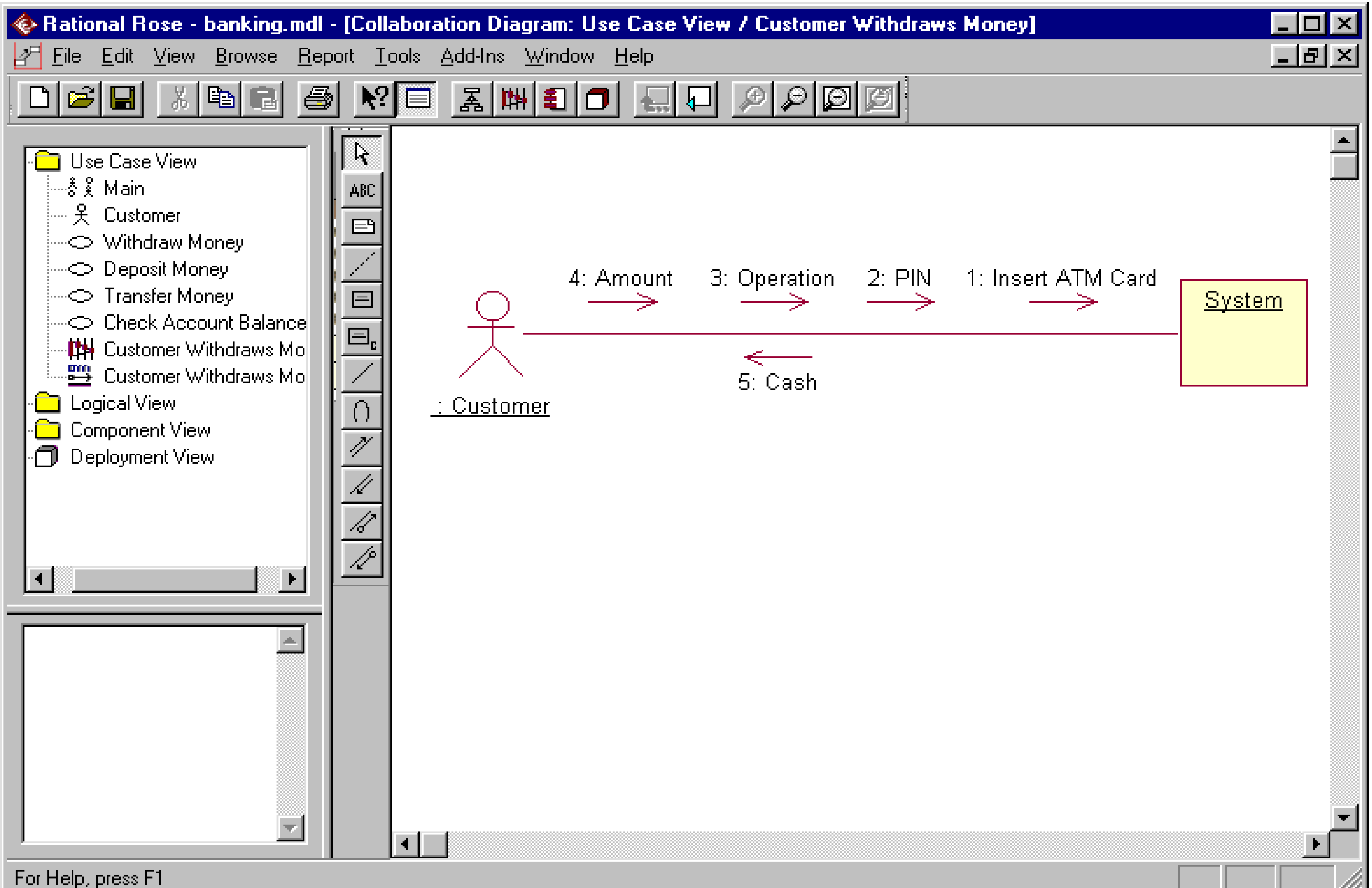
Sequence Diagrams

- Initial sequence diagrams at the use case level specify messages exchanged between actors and the system; e.g., there is only one object -- the system.
- Later, sequence diagrams are refined to represent more details; e.g., timing constraints, etc.

Example Sequence Diagram



Example Collaboration Diagram



External Event List

- Detailed list of environmental messages and events of interest to the system, including:
  - Event
  - Description
  - Direction (to system, or to specific actor)
  - Arrival pattern (periodic, sporadic, jitter, etc.)
  - Response performance (deadline)

Summary

- The UML is an industry standard for analysis and design of object-oriented systems, and provides users with an expressive visual modeling language.
- The UML can be used in many different domains to capture domain-specific concepts. It also provides extensibility and specialization mechanisms.
- Latest draft version UML 2.5 is a work in progress: http://www.omg.org

Summary

- Modelling and Analysis of Real-Time Embedded Systems (MARTE)
  - http://www.omg.org/spec/MARTE/
  - Current version: 2011-06-02
- Real-time Development Environments
  - Rational Rose Real-Time
  - Rhapsody
- References
  - Selic & Gerard, “Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE”, 1st Ed., 2013.