CIS 721 - Real-Time Systems
Lecture 28: Analysis Modeling

Mitch Neilsen
neilsen@ksu.edu

Outline

- Embedded System Design
  - Requirement Modeling – functional requirements (Use Cases)
  - **Analysis Modeling**
    - **Structural Object Analysis – static model defining the relationships between classes (Class Diagrams, etc.)**
    - **Behavioral Object Analysis – model describing dynamic (behavioral) aspects (statecharts, etc.)**
  - Design Modeling – design a software architecture
    - Architectural Design – system-wide
    - Mechanistic Design – inter-object
    - Detailed Design – intra-object
  - Rational Rose Real-Time (RoseRT) -> Rhapsody -> Rational Architect

Structural Object Analysis

- Identify key objects and classes, and how they are related, within the system.
- Define the behavior of objects.
- Validate the object model for consistency, completeness, and correctness.

Objects

- An **object** is a model of some entity, either concrete or conceptual.
- Each object has three characteristics: **state**, **behavior**, and **identity**. State can affect behavior, and behavior can affect state.
- Objects have well-defined boundaries.
- Objects communicate by sending messages.
- Objects have an **identity**, internal data called **attributes**, and **methods**.
- The **state** of an object is determined by the value of its attributes.

Messages To Objects

- Messages sent to objects include a keyword (**selector**) and possibly one or more arguments.
- Example: Object BankAccount which accepts a message of the form:
  - add_Deposit(newDeposit: currency);
  - then, it would understand the message: add_Deposit(12.49).

Object Visibility

- An object’s **public interface** defines which messages it will accept.
- It may send itself other messages (defined by its **private interface**).

Classes

- A **class** defines the structure and behavior of a group of similar objects; that is, objects with common properties (attributes), common behaviors (operations), common relationships to other objects, and common semantics.
- A good class captures one and only one abstraction; e.g., it should have one major theme.

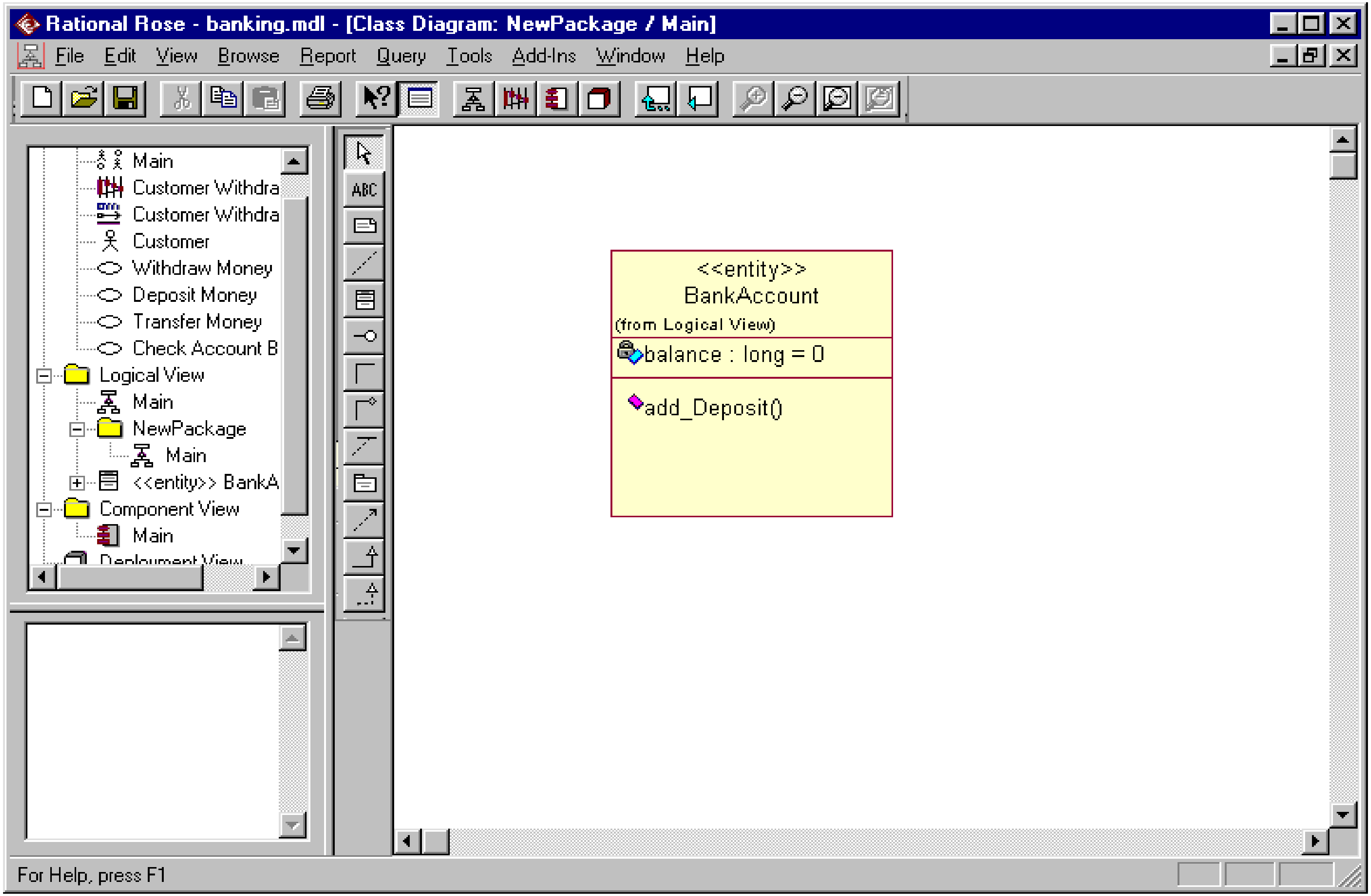
Objects and Classes

- **Type-instance dichotomy** – make a distinction between things (objects) and types (classes).
- **Classification:**
  - **Entity classes** - model information and behavior that is generally long lived.
  - **Boundary classes** - handle communication with the environment.
  - **Control classes** - model sequencing behavior.

Classes and Objects

- Every object belongs to a class, and the class of an object determines its interface.
- The process of creating a new object belonging to a particular class is called **instantiating** or creating an instance of the class.
- A **package** is a collection of related classes or packages, and is denoted by a folder.
- In UML, a class is denoted by a rectangle by giving its **name (identity)**, **attributes**, and **operations (methods)**.
- Visibility is shown by putting different symbols by each attribute or operation.

Class Diagram (Logical View)



Discovering Objects and Classes

- **Underline the nouns** - to obtain a first-cut, watch for redundant nouns.
- **Identify causal agents** - sources of actions, events and messages.
- **Identify coherent services** - operations that seem to be intrinsically bound.
- **Identify real-world items** - entities that exist in the real world; e.g., motor, forces, chemicals, etc.

Objects and Classes (cont.)

- **Identify physical devices** - sensors, actuators, etc.
- **Identify essential abstractions of domains** (bank accounts, CAN IDs, etc.)
- **Identify transactions** - finite instances of associations between objects; e.g., CAN message, etc.
- **Identify persistent information**

Objects and Classes (cont.)

- **Identify visual elements** - user interface elements.
- **Identify control elements** - provide interface for user to control system.
- **Execute scenarios on the object model.**

Object Oriented Modeling

- **Encapsulation**
  - Hide details of an object (e.g. structure and implementation) that do not contribute to its essential characteristics.
- **Abstraction**
  - Define essential characteristics of an object that distinguish it from all other kinds of objects; e.g., provide a well-defined conceptual boundary.

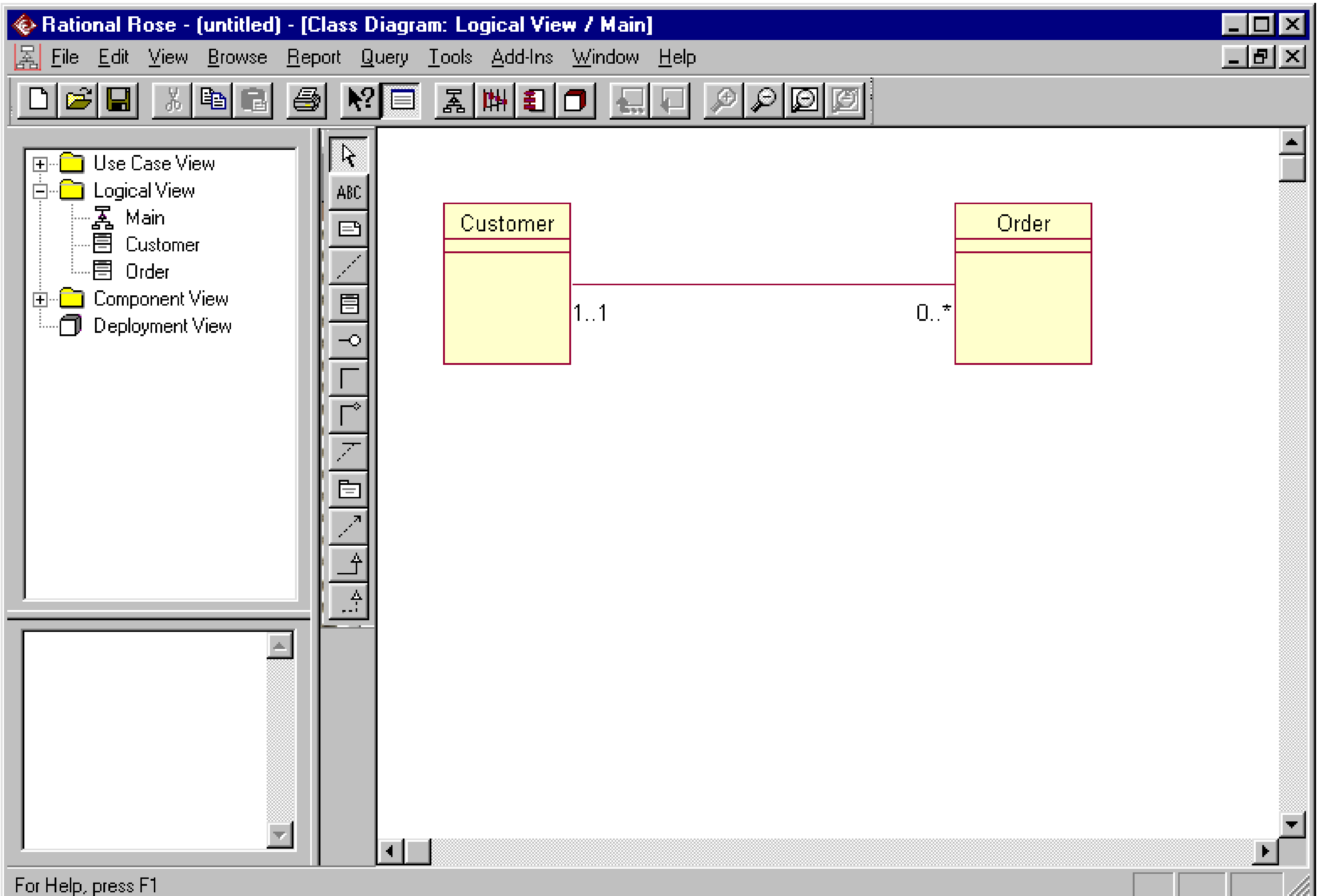
Relationships Between Classes

- Association “associates with”
- Generalization “is less specialized”
- Aggregation “is part of”

Association

- **Association** - some object of one class needs to know about the existence of another class; e.g., to send a message to an object of the other class or create an object of the other class.
- Example: Customer places an Order.
- Multiplicity - identifies how many objects of the other class are associated; e.g., 1, 0..* etc.

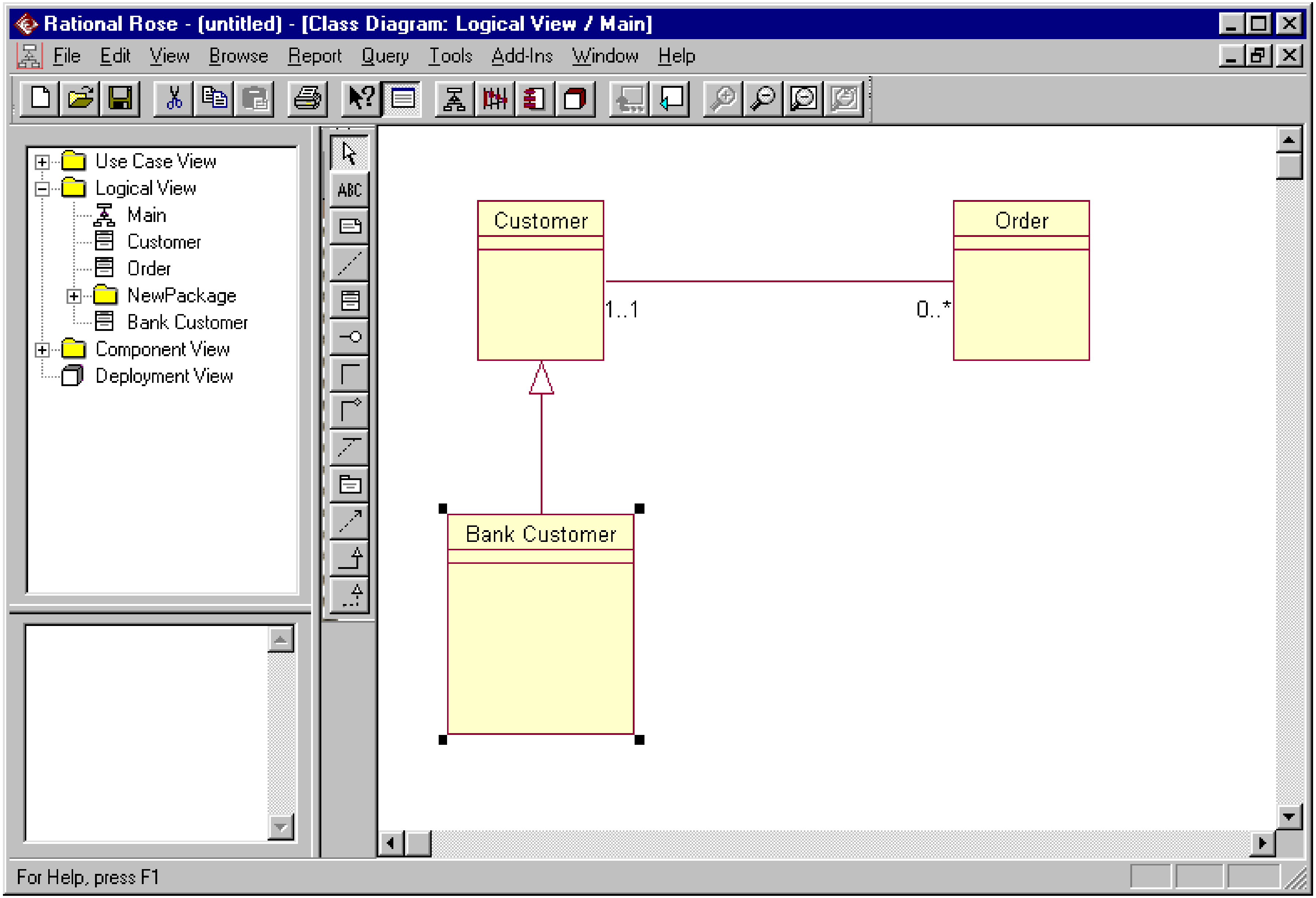
Association Example



Generalization

- An object in the **specialized class** should conform to the interface given for the **generalized class**.
- If some message is acceptable for an object in the generalized class, then it should be acceptable for an object in the specialized class as well.
- Example: A Customer is a generalization of a Bank Customer.

Generalization Example



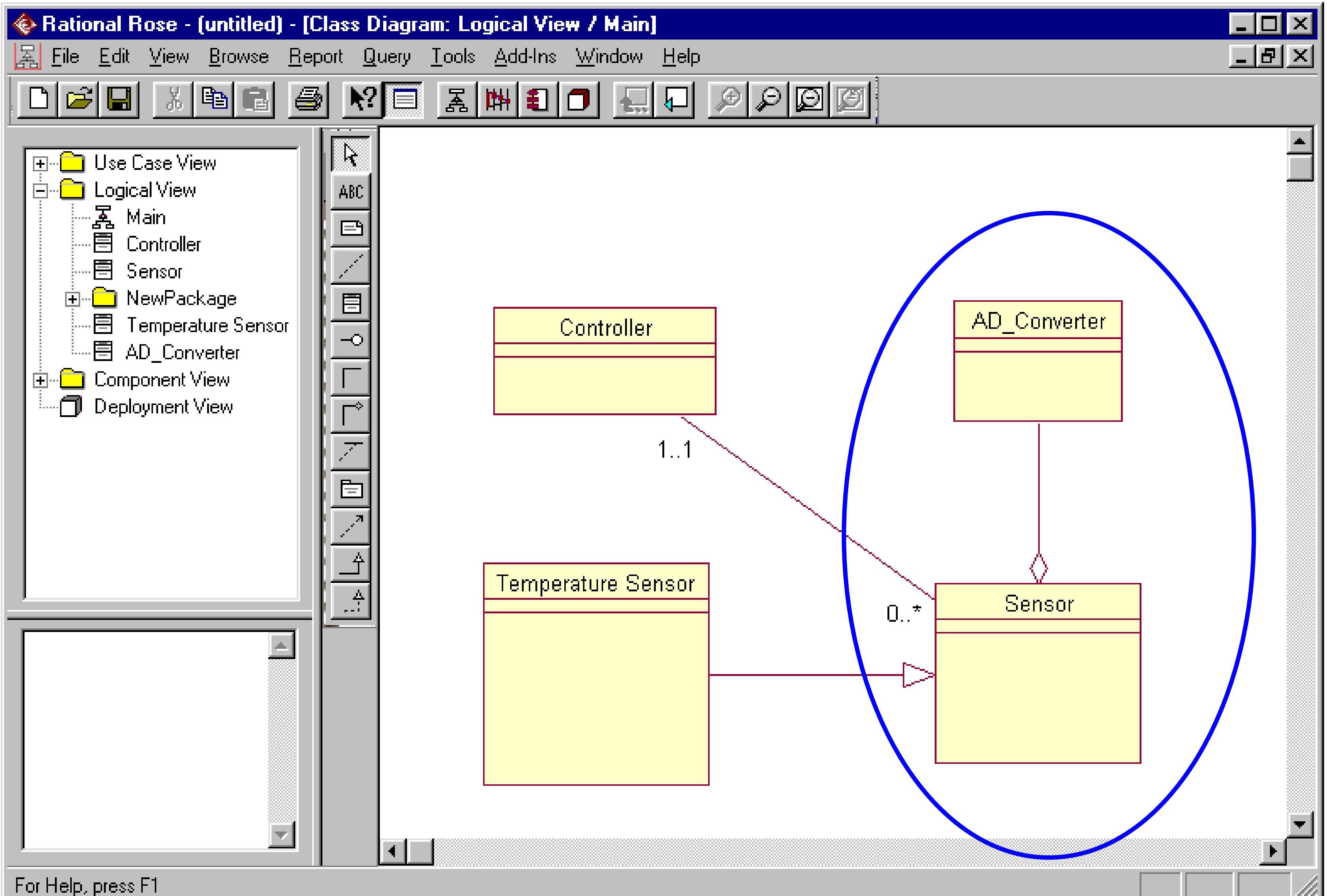
Inheritance

- **Inheritance** is one method that can be used to **implement** generalization.
- If we define a new class by inheritance, then we only need to write code for the new parts of the specialized class; e.g., operations to react to messages received by objects in the specialized class, but not by objects in the base class.

Aggregation

- An **aggregation** between two classes exists if an object B of one class **is part of** an object A of the other class.
- In most cases, object B lives and dies with object A.

Aggregation Example



Composition

- **Composition** is a strong form of aggregation.
- Part objects (called components) are solely the responsibility of the composite class.
- Composites must create and destroy their components.

Class Diagram

- The class diagram evolves as the development proceeds.
- The initial diagram should be conceptual:
  - It should not consider which direction(s) an association is navigable.
  - Attributes are recorded to indicate that objects have certain attributes, not that they are public or in which class they will eventually reside.
  - Aggregation and compositional relationships are at the conceptual level.

Rules for Good Class Diagrams

- Each diagram should have a “mission” in life.
- Build one diagram per collaboration (use case).
- Create one diagram per domain; large systems should be decomposed into **packages**.
- For rich hierarchies, create one diagram per generalization hierarchy.
- Don’t generally mix generalization and collaboration.

Rules (cont.)

- Create a master road map only for very small systems.
- Don’t show attributes and operations on class diagrams.
- Don’t cross lines.
- Show multiplicities for all associations.
- Show role names rather than association labels.

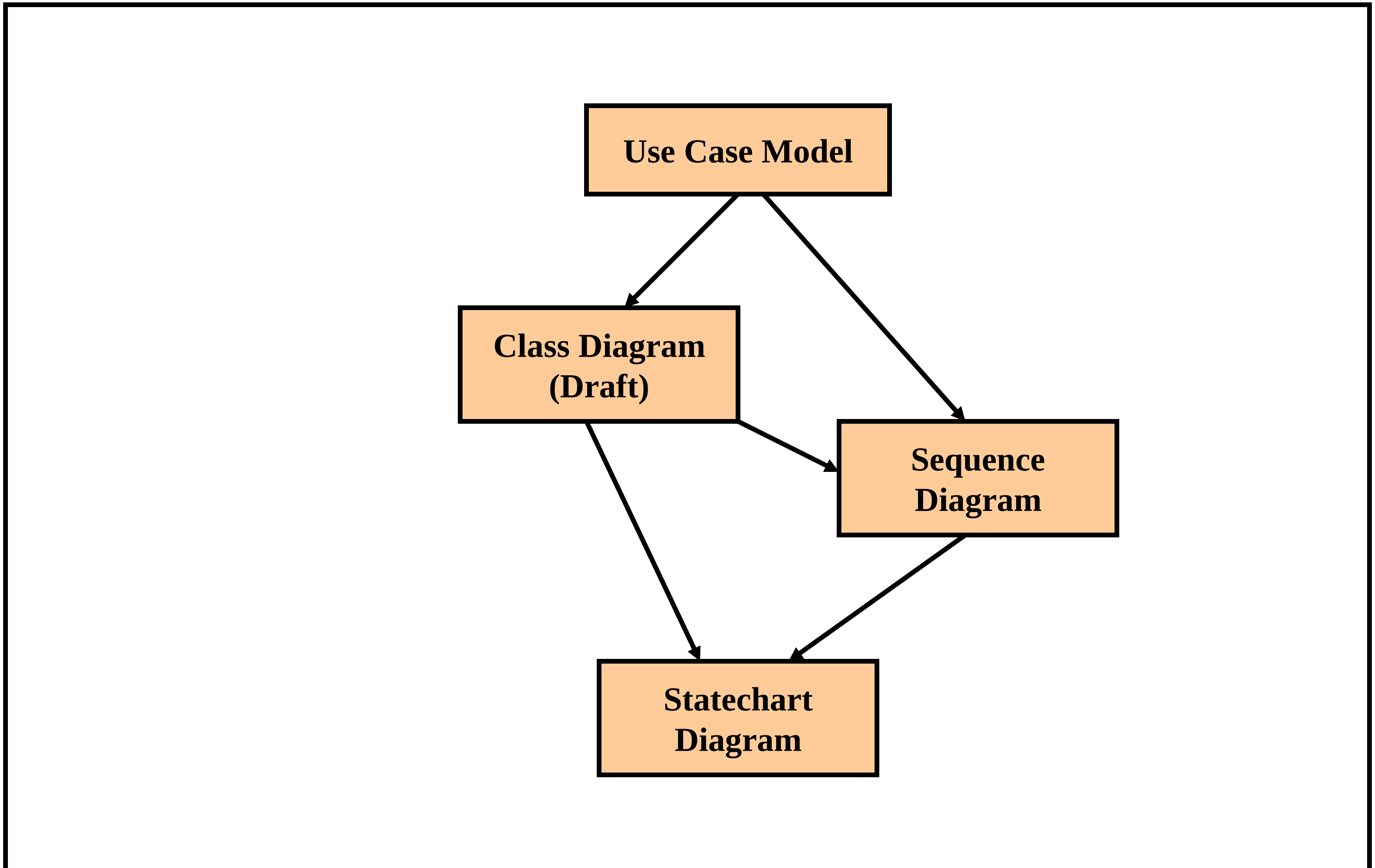
Rules (cont.)

- Show only one role name for unidirectional navigation association; show both role names for bidirectional associations.
- Use composition to represent layers of abstraction.
- Show constraints in note boxes, use { } to denote the constraint.
- Use a note box to label each diagram noting project, author, date of creation, and purpose.

Behavioral Object Analysis

- Define **“what”** an object does.
- Change perspective from **“outside”** to **“inside”** classes.
- Define dynamic behavior of each class formally using **state transition diagrams (statecharts).**

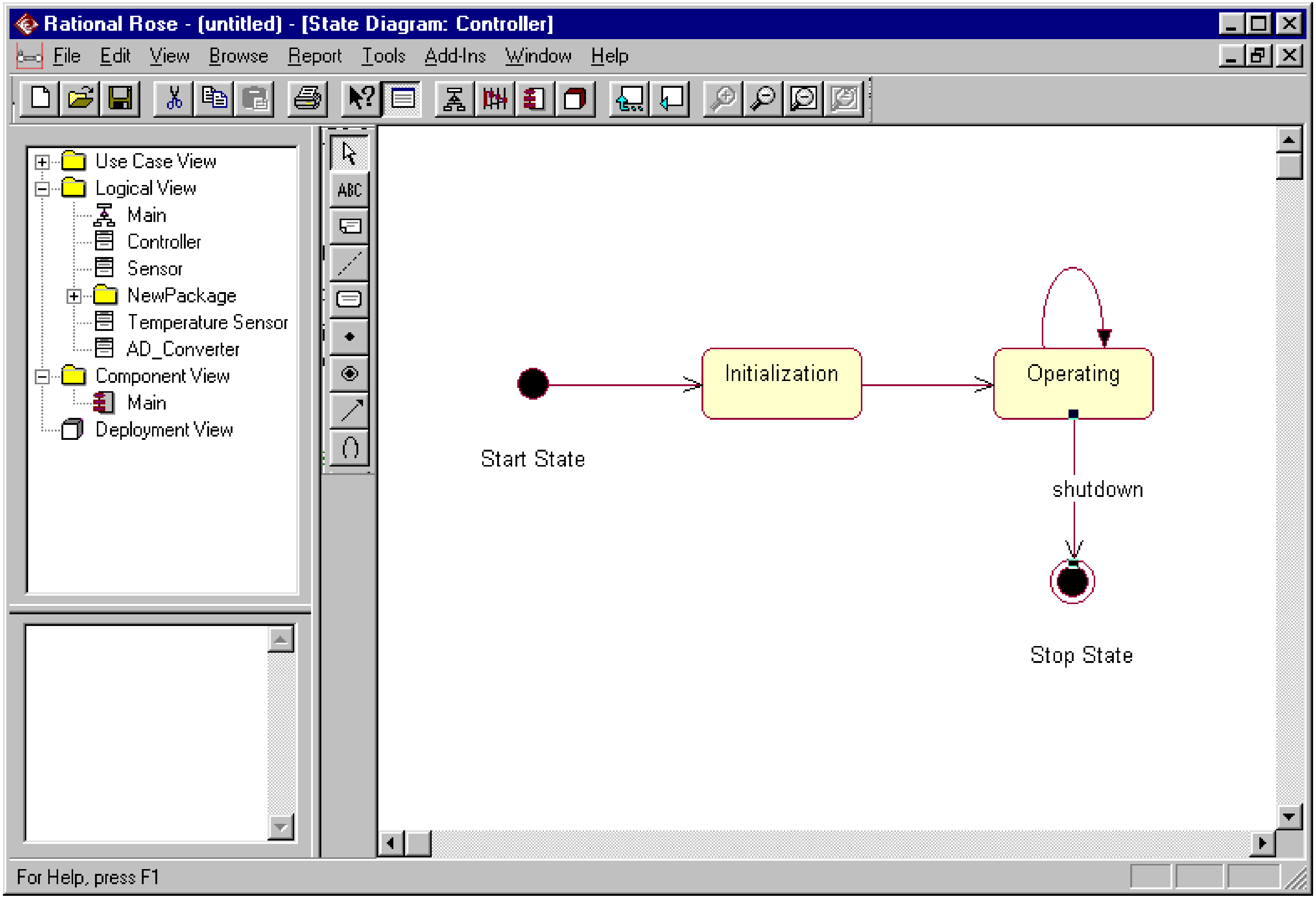
Diagrams To Be Developed



Statecharts

- **Purpose:** to **formally specify** the **behavior** of the instances of a given **class** in response to external stimuli.
- **UML Notation:** A directed **graph** of **states** connected by **transitions.**
- **Origins:**
  - finite state machines (FSM)
  - state transition diagrams
  - Harel’s statecharts

State Transition Diagram



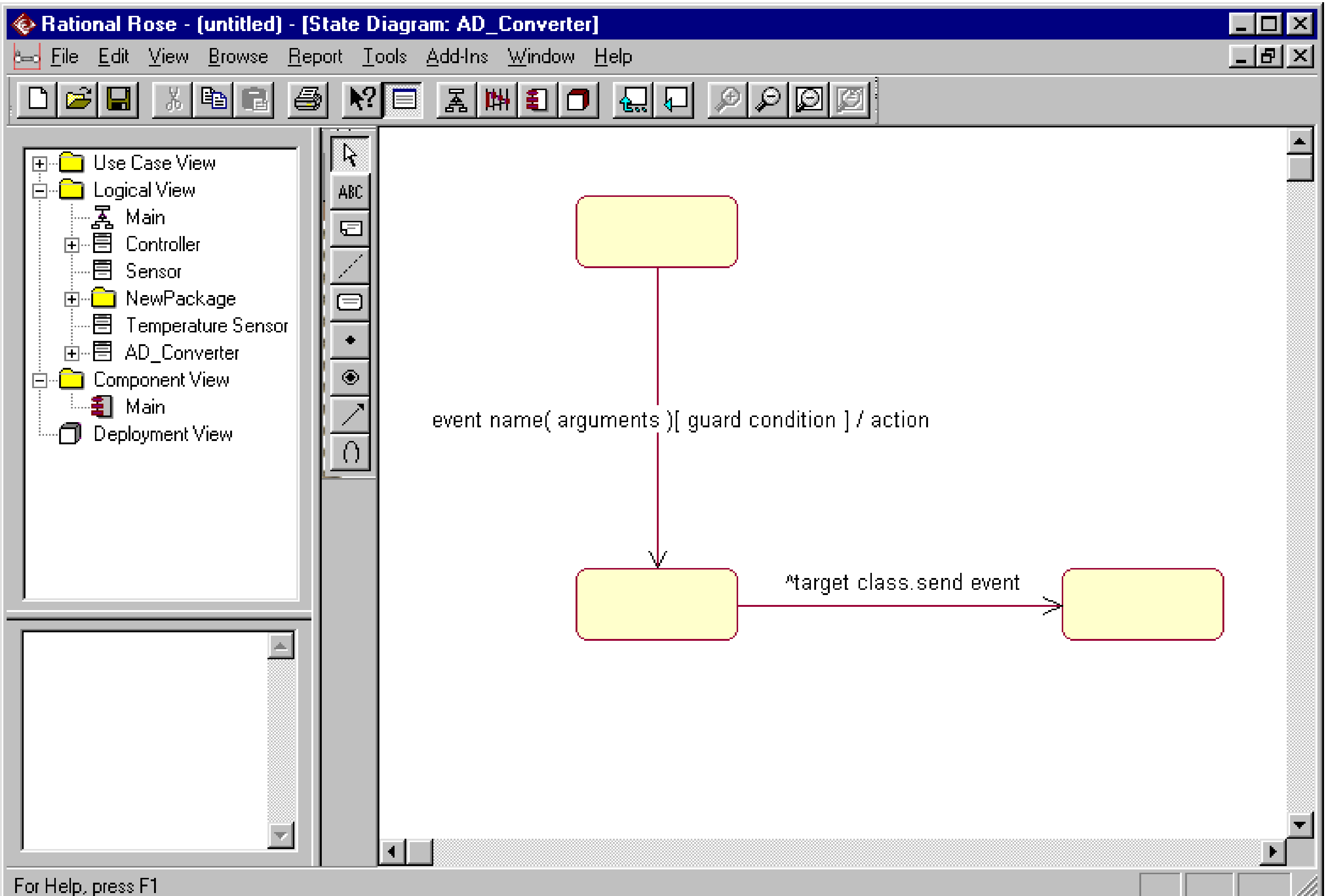
Statechart Semantics

- **State** - a finite period in the life of an object
  - when the object satisfies some condition, or
  - performs some action, or
  - waits for some event to occur.
- **Event** - an occurrence of
  - a change in truth value of a condition
  - a receipt of a message
  - the end of a designated period of time
- Events cause changes in state (**transitions**).

Transitions

- A state **transition** can have the following elements associated with it:
  - an **action** (behavior that occurs when the transition takes place), and/or
  - a **guard** (Boolean expression that must be true for the transition to be allowed).
- Actions and guards are behaviors, and typically become private operations.
- State transitions can also trigger events.

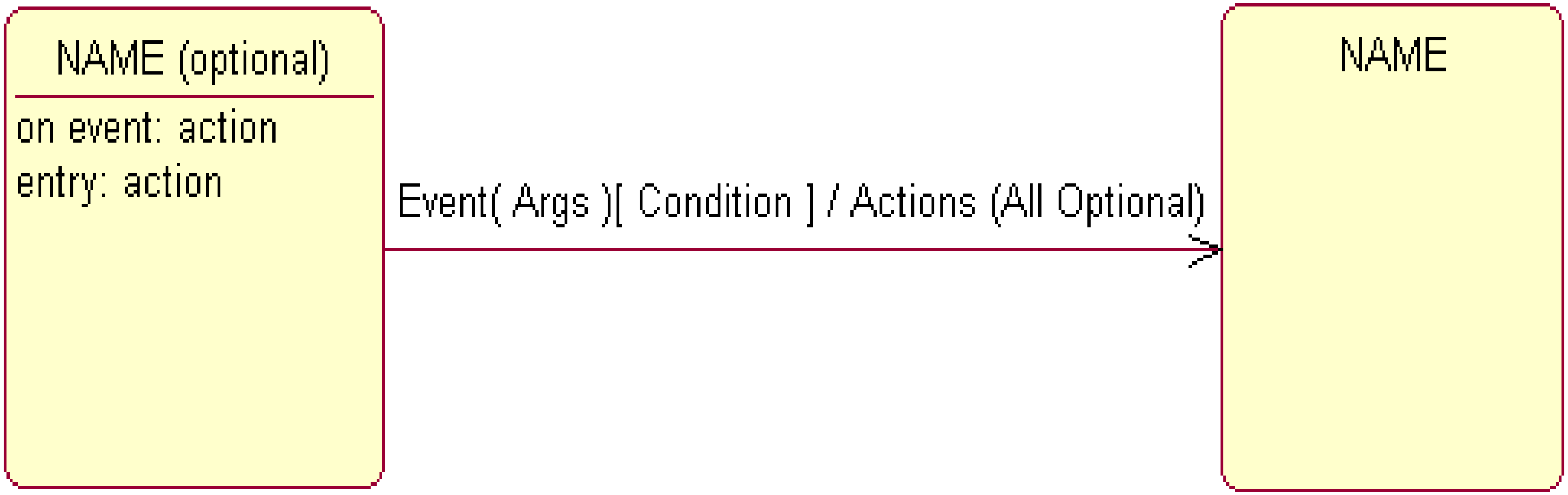
State Transitions



Event [Guard] / Action Transition

- An **event** prompts the transition between states.
- A **guard** is used to specify that this transition can occur only if the guard is true.
- An **action** is performed when the transition occurs.

Statechart Diagram: Basic Notation



Internal activities are performed in response to an events received (on entry, exit, or some other event -- usually internal activities that result in invocation of private operations).

State Details

- Actions that accompany state transitions into (out of) a state can be noted as **entry (exit) actions** within the state.
- Behavior within a state is called an **activity**.
- Activities can be a simple action or an event sent to another object.
- Activities are optional.

State Detail Notation

entry: simple action

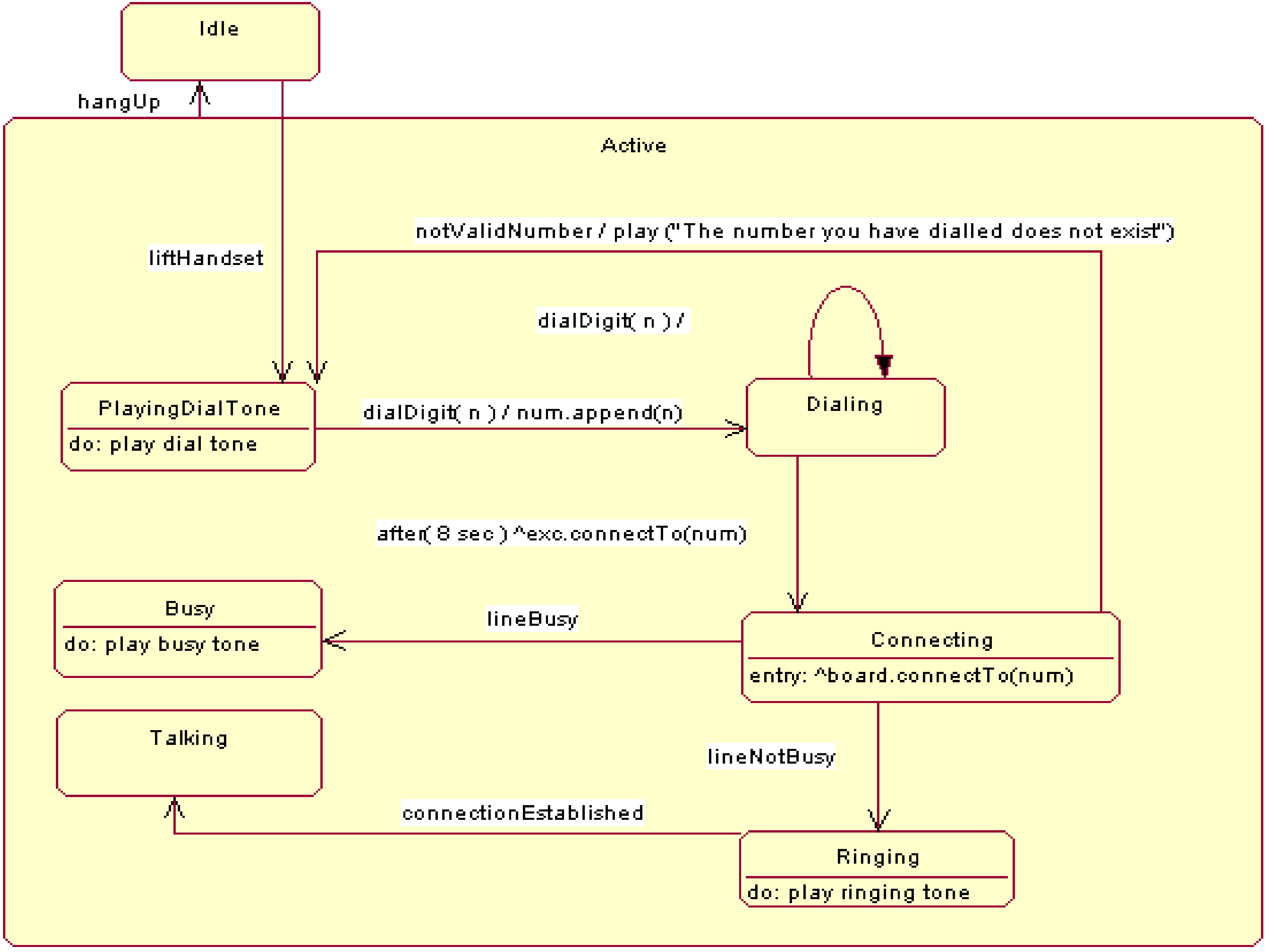
entry: ^destination class name.event name

do: simple action

do: ^destination class name.event name

exit: simple action

exit: ^destination class name.event name


Idle
hangUp
Active
notValidNumber / play ("The number you have dialled does not exist")
liftHandset
dialDigit(n) /
PlayingDialTone
do: play dial tone
dialDigit(n) / num.append(n)
Dialing
after(8 sec) ^exc.connectTo(num)
Busy
do: play busy tone
lineBusy
Connecting
entry: ^board.connectTo(num)
Talking
lineNotBusy
connectionEstablished
Ringing
do: play ringing tone

Composite States

- A **state** can be decomposed into:
  - a set of **mutually exclusive** substates **(or-decomposition) (UML + ROSE)**
  - a set of **concurrent** substates **(and-decomposition)**

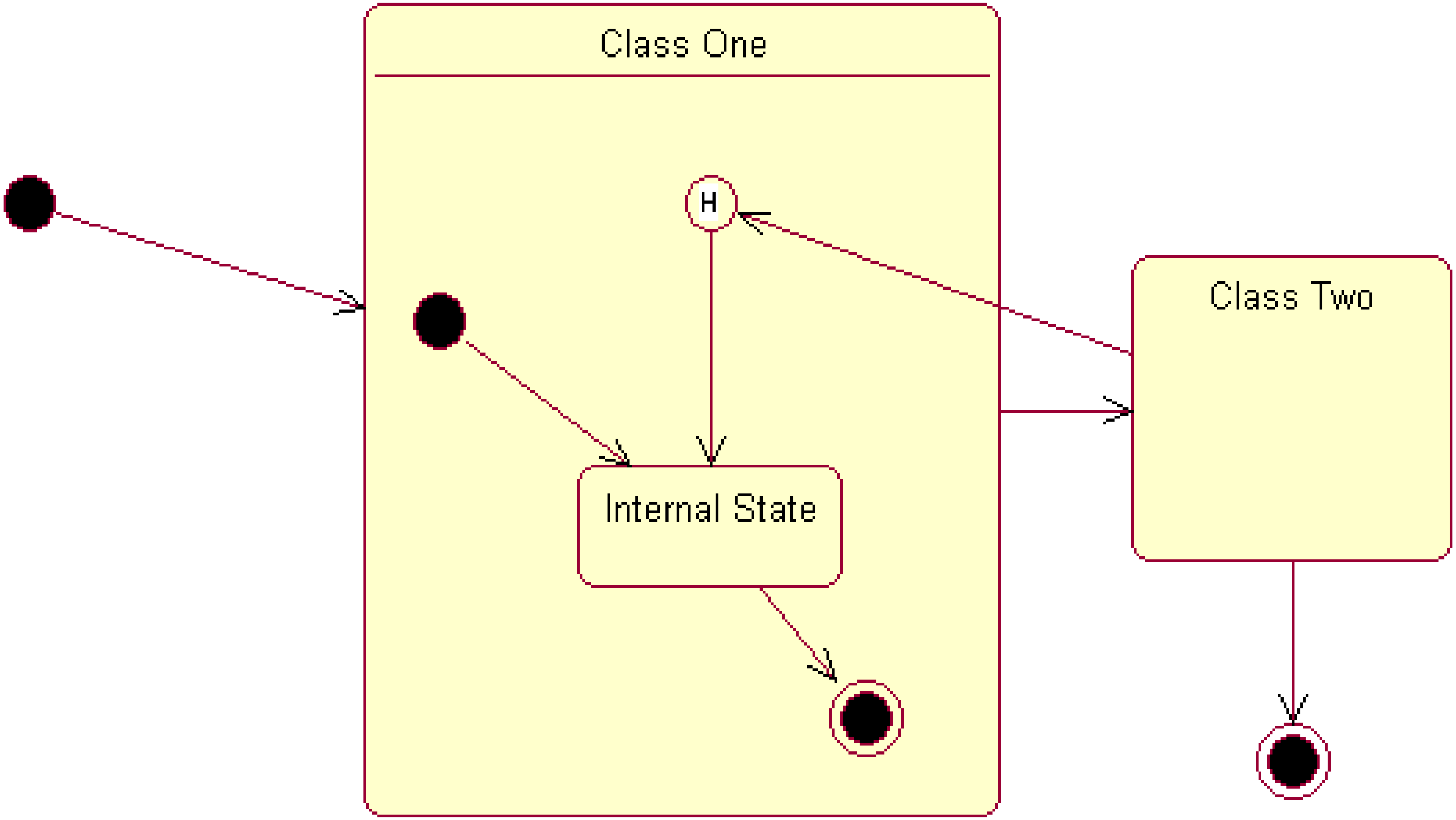
Special States and Activities

- **History State** - a state resumed upon re-entry.

H1

- **Activities -** operation within a state represented by a nested statechart.

Example



Role of Statecharts

- Formally specify behavior of objects.
- Increase understanding of classes.
- Describe what happens when events occur within the system and its environment.
- Provide abstract and partial descriptions of the actual code.

Statecharts and Class Diagrams

- Check consistency between Statecharts and Class Diagrams to ensure that:
  - messages that label transitions correspond to operations of the relevant class
  - attributes and associations referenced by transitions and state actions have been defined
  - messages sent to other objects correspond to operations in the classes of these objects

Operations

- An **operation** is the fundamental quantum of object behavior.
- The implementation of an operation within a class is called a **method**.

Protocol for Operations

- **Precondition invariants** - assumptions that must be true before an operation is invoked.
- **Signature** - ordered list of formal parameters and their types, and return type of the operation.
- **Post-conditional invariants**.
- **Rules** for thread-reliable interaction including synchronization.

Types of Operations

- **Constructor** - create an object
- **Destructor** - destroy an object
- **Modifier** - change values within objects
- **Selector** - read values or request services
- **Iterator** - provide orderly access to the components of an object; used with objects that maintain collections of objects called **collections** or **containers**.

Checking the Model

- **Homogenize** the model.
  - Combine classes
  - Split classes
  - Eliminate classes
  - Check consistency
  - Walk-through scenarios
  - Trace events in sequence diagrams
  - Review documentation

Combine Classes

- A class may be called by a different name in different parts of the model or by different teams of developers.
- Combine such classes.
- Choose the name which is closest to the language used by the customers.

Split Classes

- Classes should follow the golden rule of OO: "*A class should do one thing, and do it very well*"; e.g., classes should be **cohesive**.
- For example, sometimes what appears to be only an attribute ends up having structure and behavior unto itself, and should be split off to form its own class.

Eliminate Classes

- Classes may be eliminated from the model when:
  - they do not have any structure or behavior
  - they do not participate in any use cases
- For example, if a control class is only responsible for passing information through the class, it may be eliminated; e.g., get rid of the “middle man”.

Check Consistency

- The static view (class diagram) must be checked against dynamic views (use case diagram and interaction diagrams) in parallel to ensure consistency.
- Consistency checking should be integrated throughout the lifecycle.

Scenario Walk-Through

- Each message in a sequence diagram represents a behavior of the receiving class.
- Verify that each message is captured as an operation on the class diagram.
- Verify that interacting objects have a pathway to communicate via an association or aggregation.

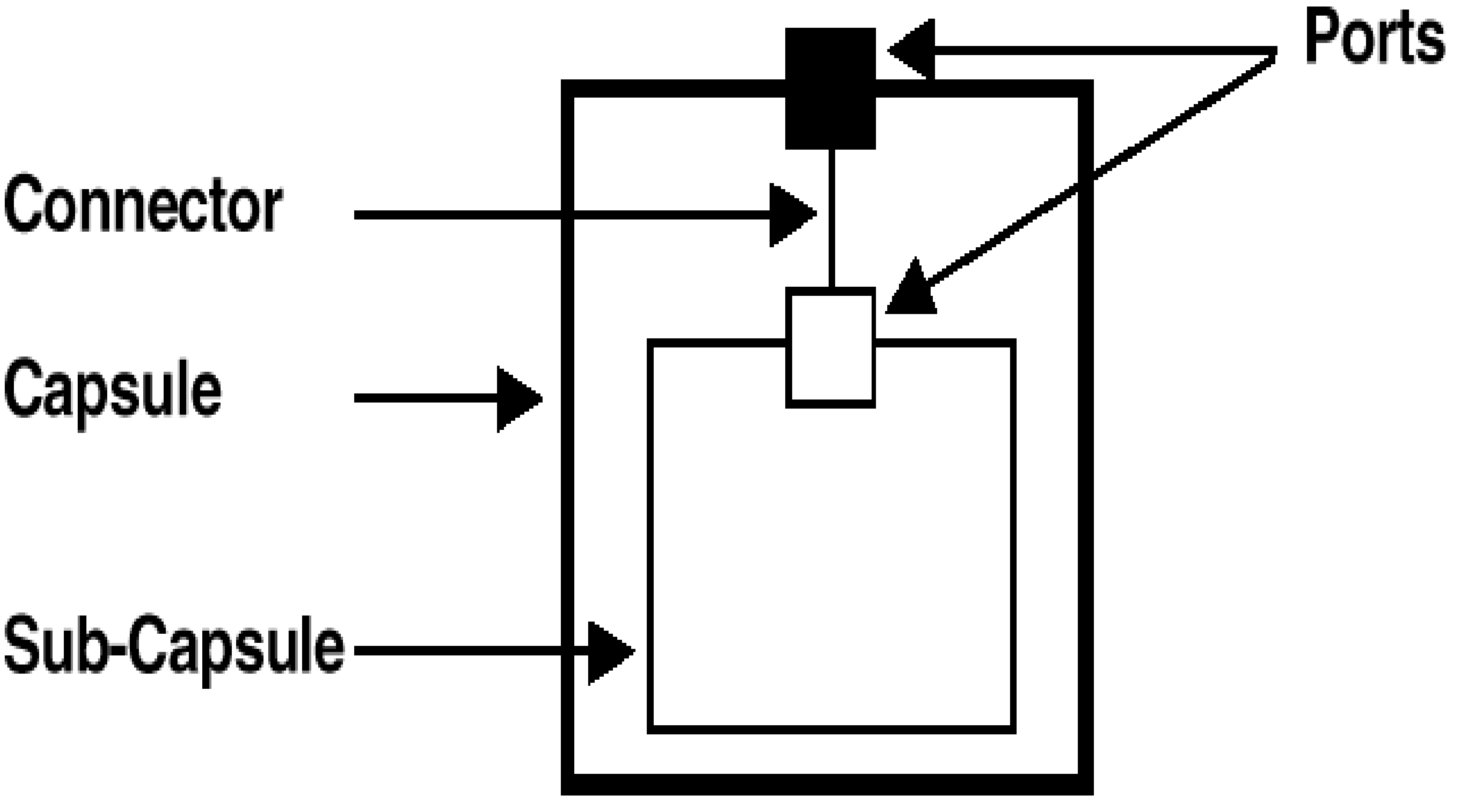
Event Tracing

- For every message in a sequence or collaboration diagram, verify that the sending class is responsible for sending the event, and the receiving class handles the message.
- If a statechart for the class exists, verify that the event is represented on the diagram.

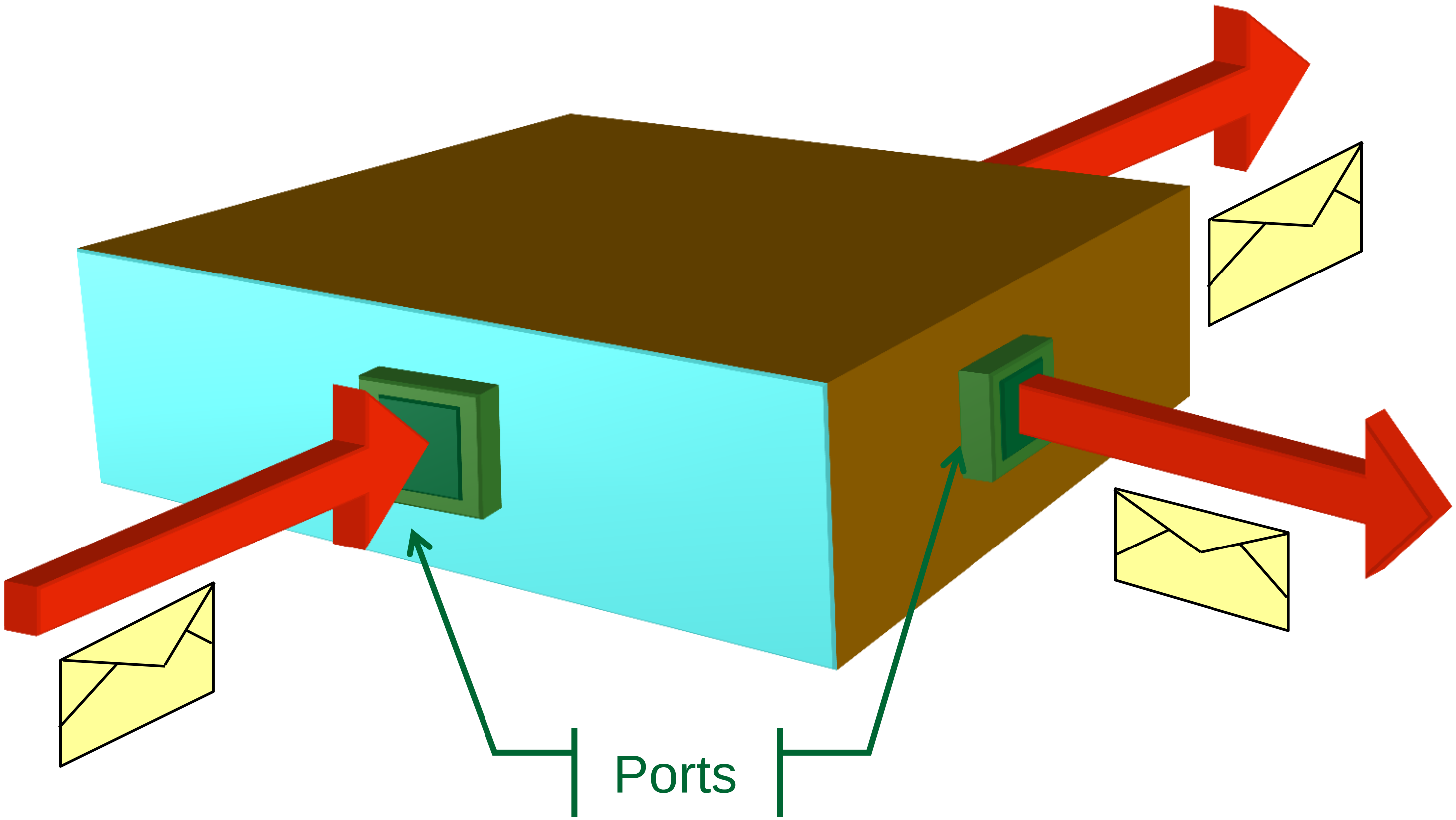
Real-Time UML Constructs

- For Modeling **Structure**
  - capsules (capsule classes)
  - ports
  - connectors
- For Modeling **Behavior**
  - protocols
  - state machines
  - time service

Capsule

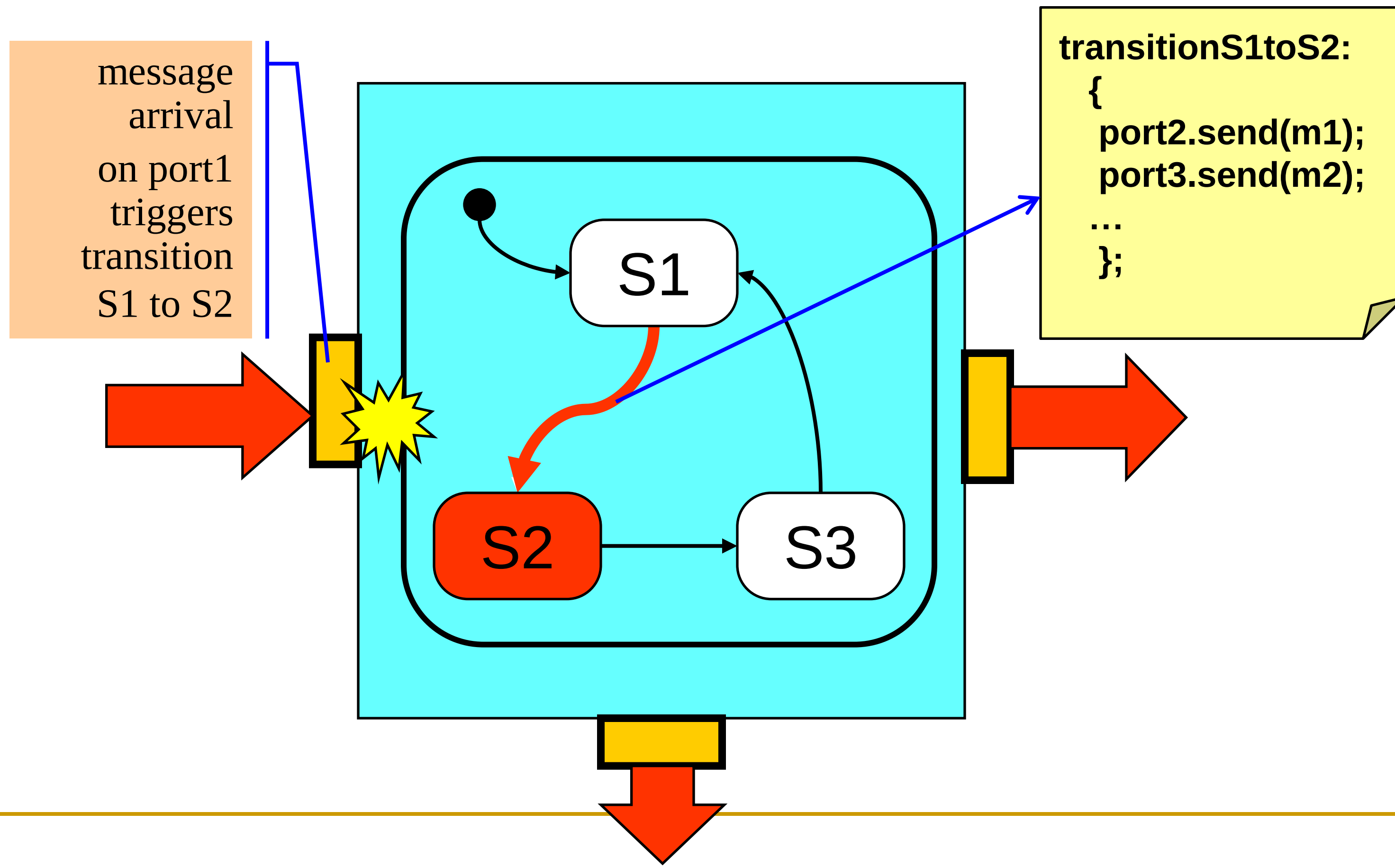


Capsules: Active Objects



Capsules: Behavior

- State machine



Capsules (Capsule Classes)

- Complex, physical, possibly distributed architectural objects that interact with surroundings through signal-based boundary objects called **ports**.
- A stereotype of a class (**<< capsule >>**).
- The fundamental modeling element of Rational Rose Real-Time (RoseRT).

Classes vs. Capsules

- Communication:
  - Classes: Public operations
  - Capsules receive messages through **public ports** which understand protocols.
- Attributes:
  - Classes: Public, private, and protected.
  - Capsules only have private attributes to enforce encapsulation.

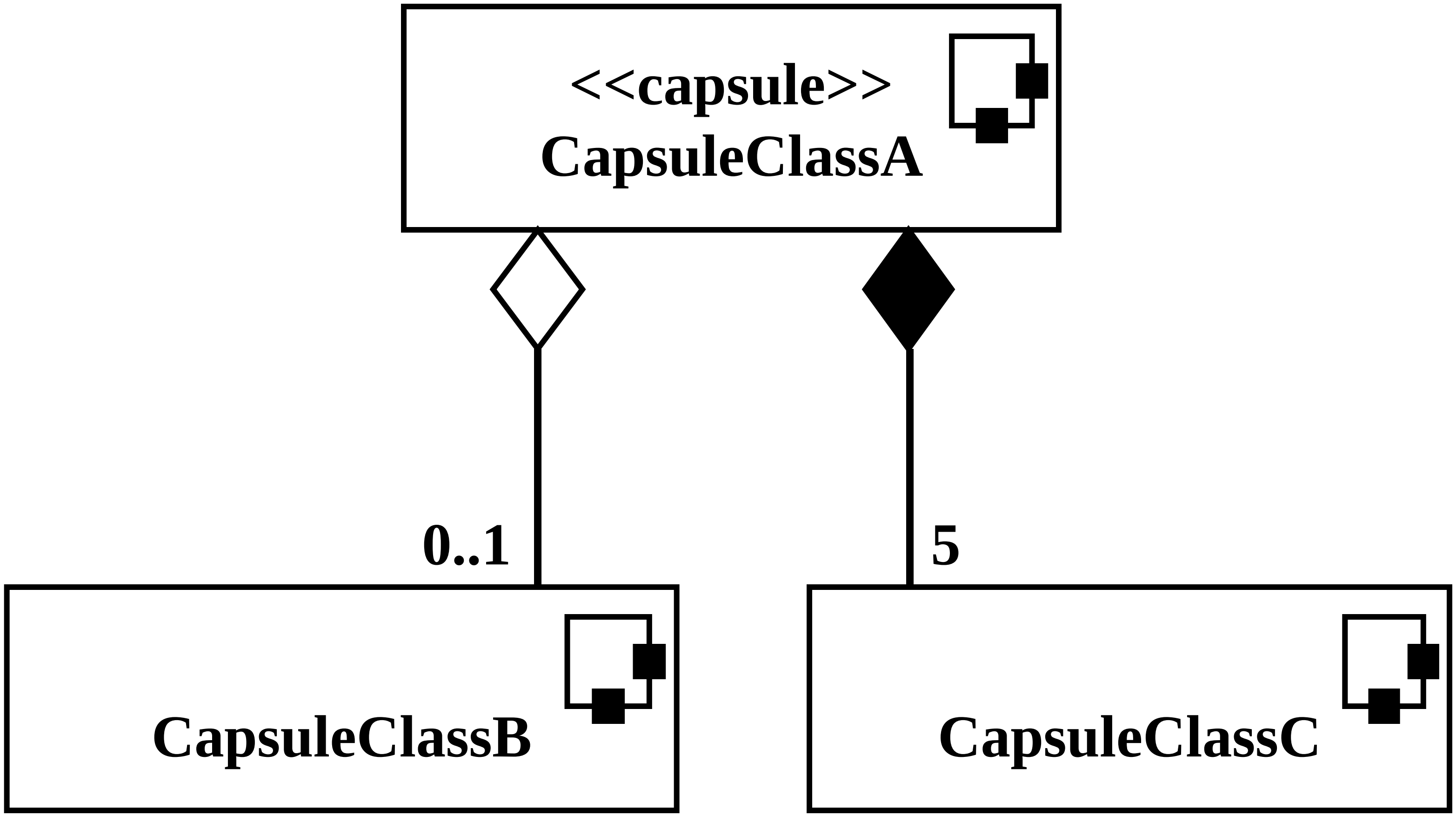
Classes vs. Capsules

- Behavior:
  - Classes: Method implementation.
  - Capsules: Defined by state machines which run in response to the arrival of signals.

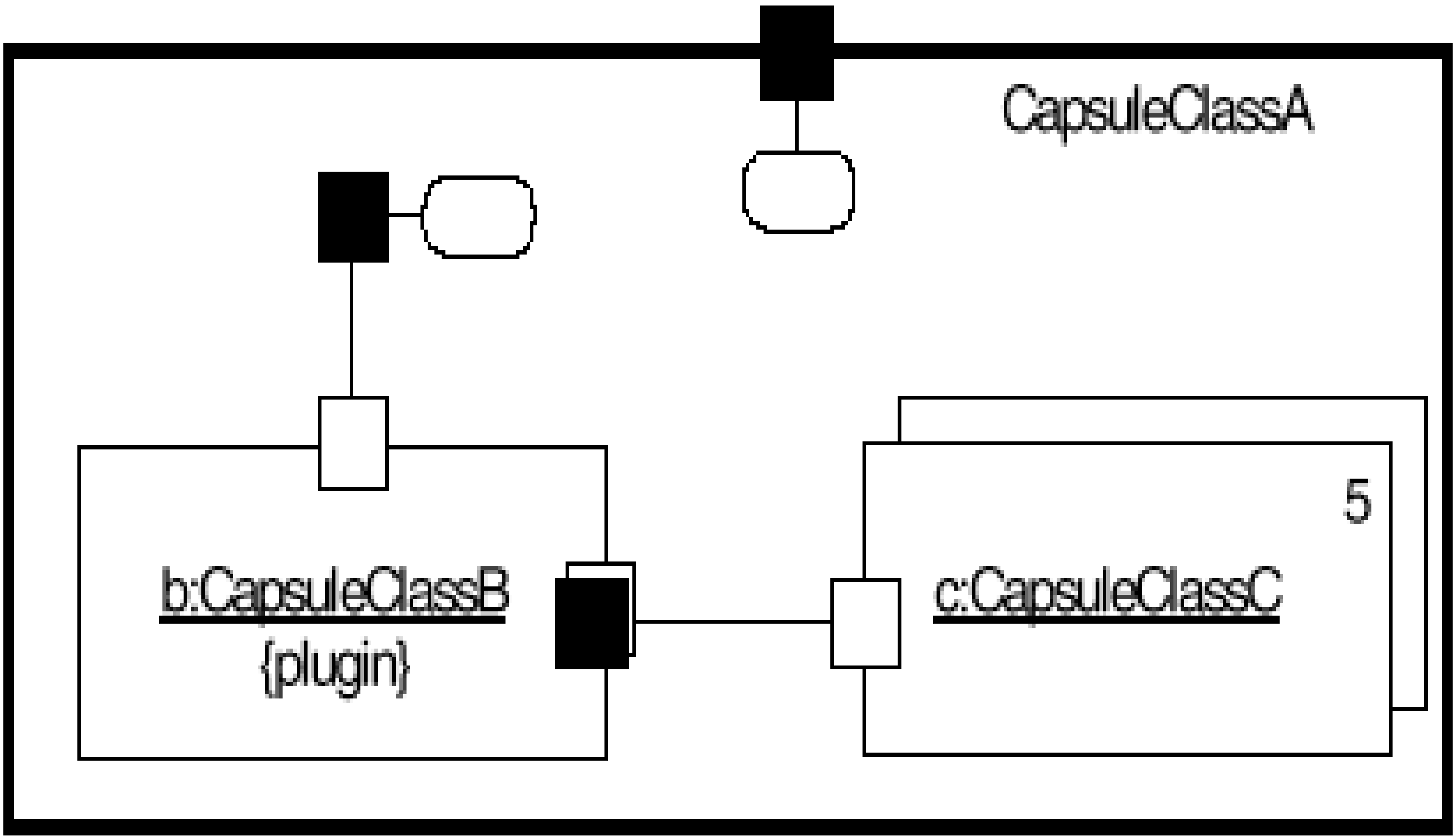
Simple vs. Complex Capsules

- The functionality of a **simple capsule** is realized by the state machine associated with it.
- **Complex capsules** combine a state machine with an internal network of collaborating sub-capsules.

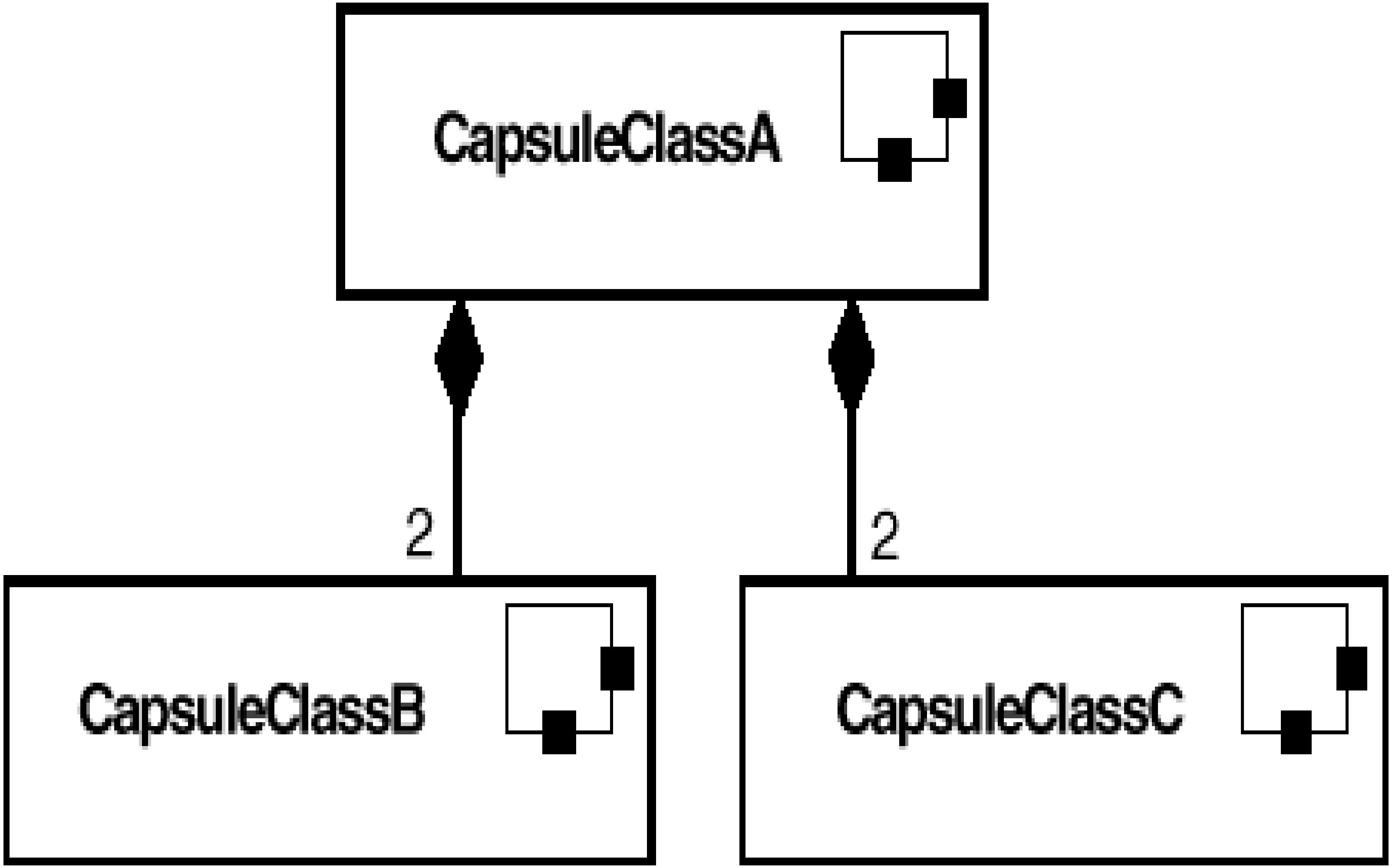
Class Diagram



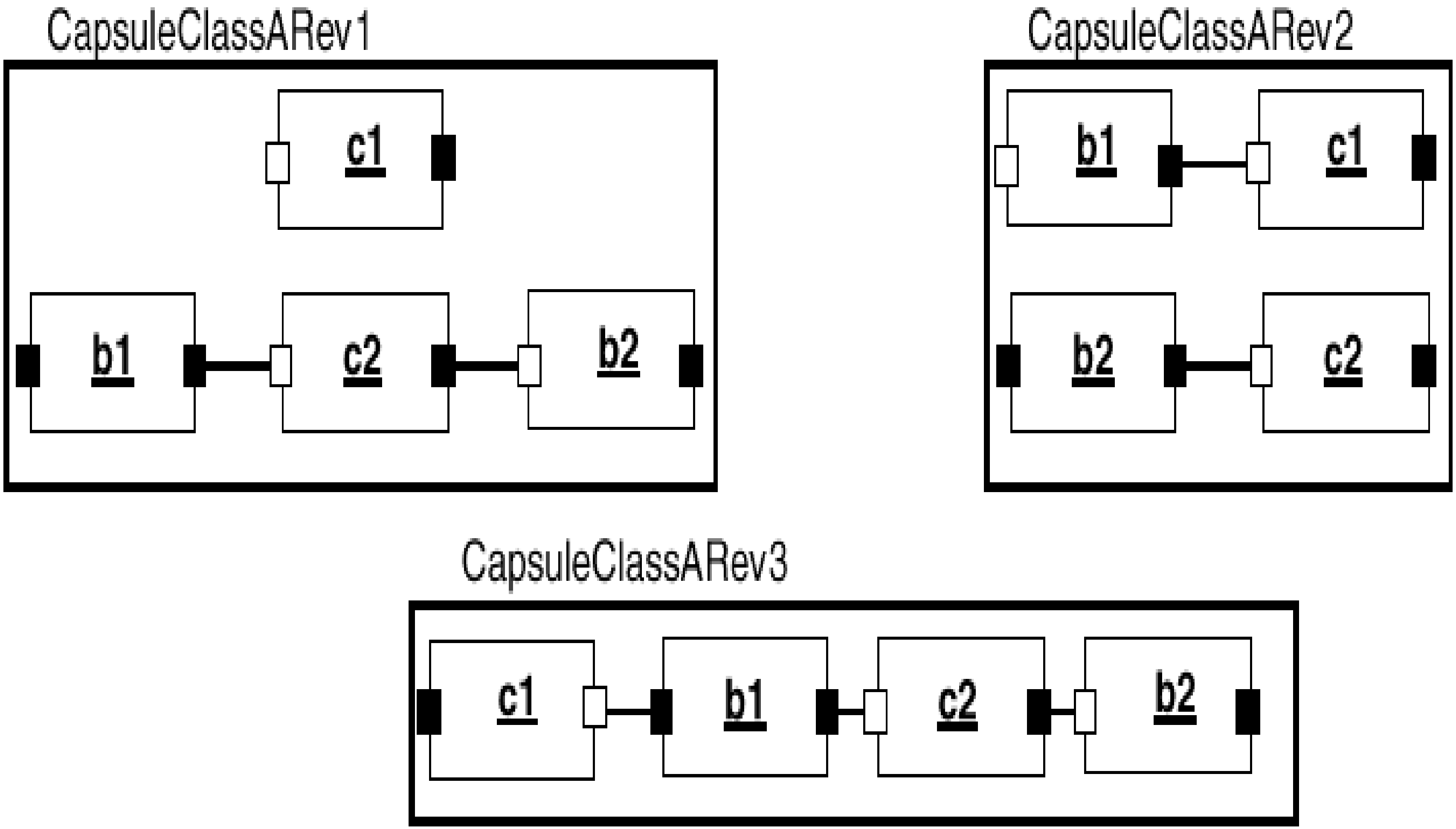
Capsule Collaboration Diagram



Example Class Diagram



Possible Collaboration Diagrams



Ports

- Ports are **boundary objects** for a capsule instance.
- Unlike an interface (which is a behavioral thing), a port includes both structure and behavior.
- Each port plays a specific role in a protocol.
- The protocol defines the valid flow of information (signals) between connected ports of capsules.

Types of Ports

- Viewed from the outside, ports present the same object interface, and they cannot be distinguished except by their identity and protocol role.
- Viewed from inside the capsule, they can be one of two kinds:
  - **relay ports** - connected to sub-capsules
  - **end ports** - connected to the capsule’s state machine

Relay Ports

- Relay ports are connected, through a connector, to a sub-capsule.
- Relay ports simply pass all signals through to/from the sub-capsule.
- Signals are exchanged without delay (zero overhead). If no connector is attached, the signal is lost.
- Relay ports have **public visibility** and can only appear on the boundary of a capsule.

End Ports

- End ports are boundary objects for the state machines of capsules.
- They are the ultimate sources and sinks of all signals sent by capsules.
- To send a signal, a state machine invokes a send or call operation on an end port.
- Since communication may be asynchronous, an end port has a queue to hold signals that have been received, but not yet processed.

Types of End Ports

- Like relay ports, **public end ports** appear on the boundary of a capsule with public visibility.
- **Protected end ports** appear inside of a capsule as part of the capsule’s internal implementation.
- Note: Port type is determined by a port’s internal connectivity and visibility outside of the capsule.

Connectors

- Capture key communication relationships between capsules.
- They identify which capsules can directly communicate with each other.

Protocol

- A **protocol** is a specification of desired behavior to take place over a connector.
- It is pure behavior and does not specify any structural properties.
- **Binary protocols**, involving only two participants, are the most common. For binary protocols, only one role, called the **base role**, needs to be specified. The conjugate role can be derived.

Protocol Roles and Protocols

- A protocol role is modeled in UML by the **<< protocolRole >>** stereotype.
- A protocol is modeled in UML by the **<< protocol >>** stereotype of collaboration with a composition relationship to each of its protocol roles.

Protocol Role

- A class model is a high-level generalization.
- An instance model is a low-level, specialized view of the system.
- A role model is an intermediate-level view of a system. It captures properties that all instances display.

Protocol

<<protocol>>
BinaryProtocolA

Incoming

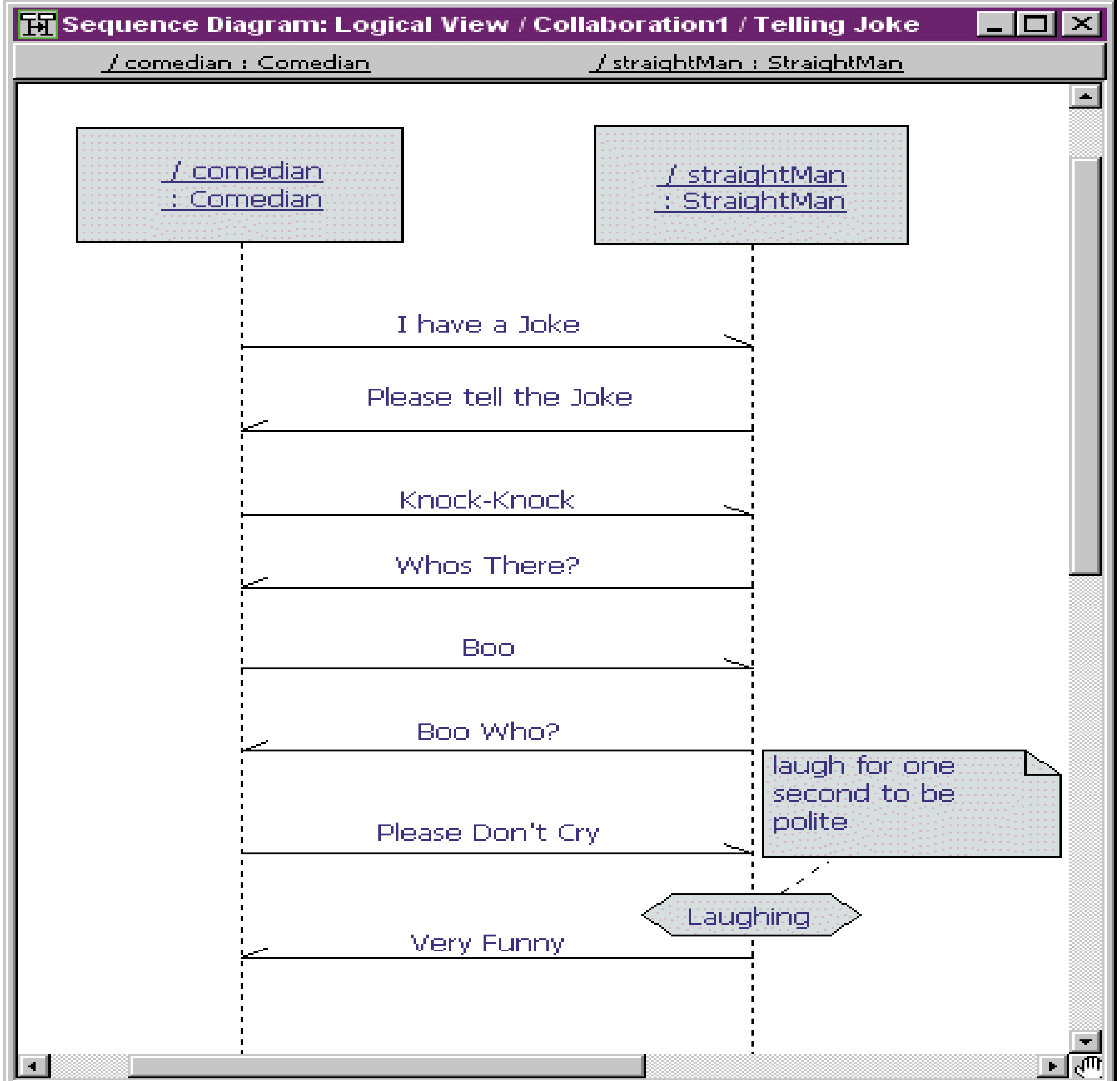
signal1
signal2
signal3

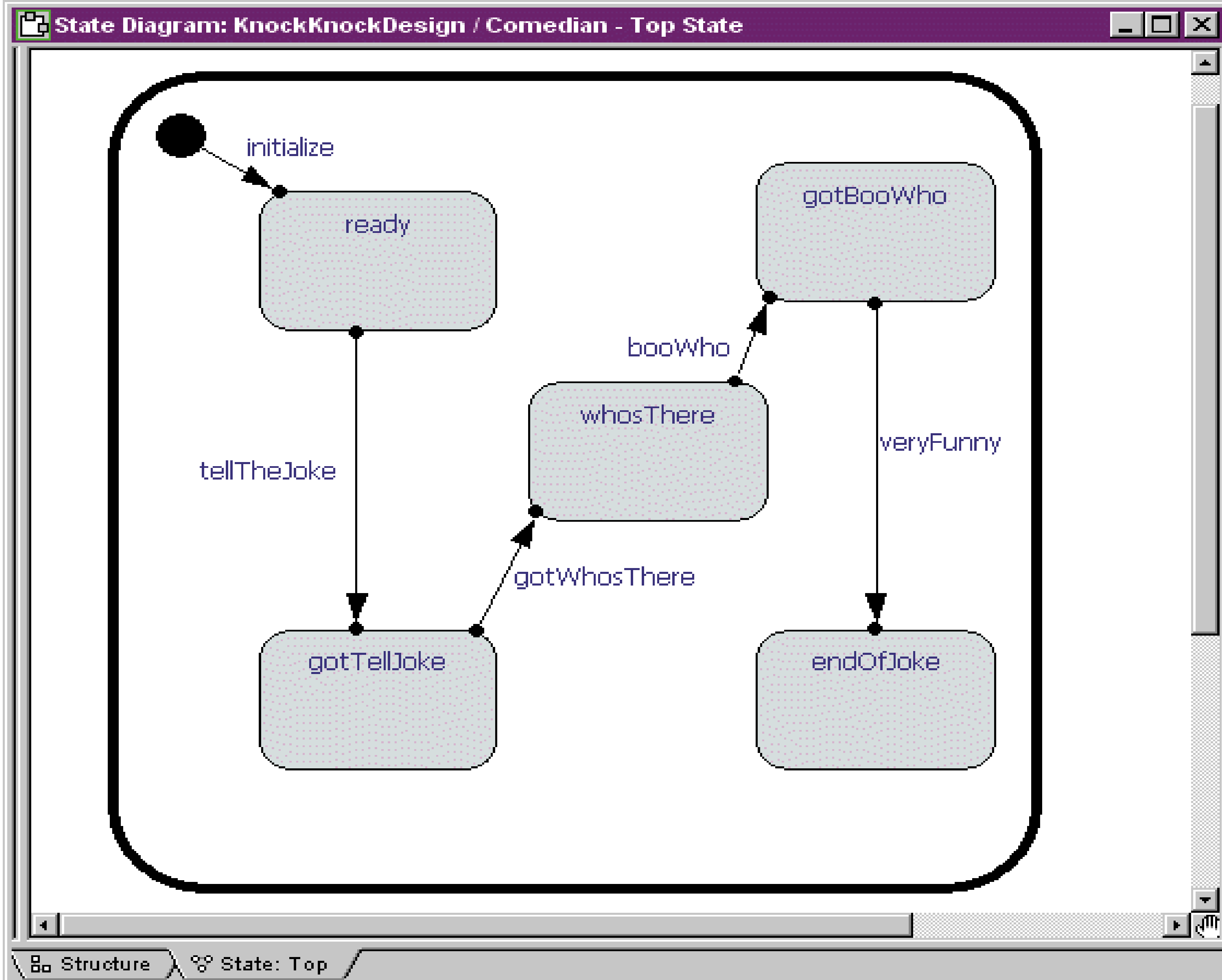
Outgoing

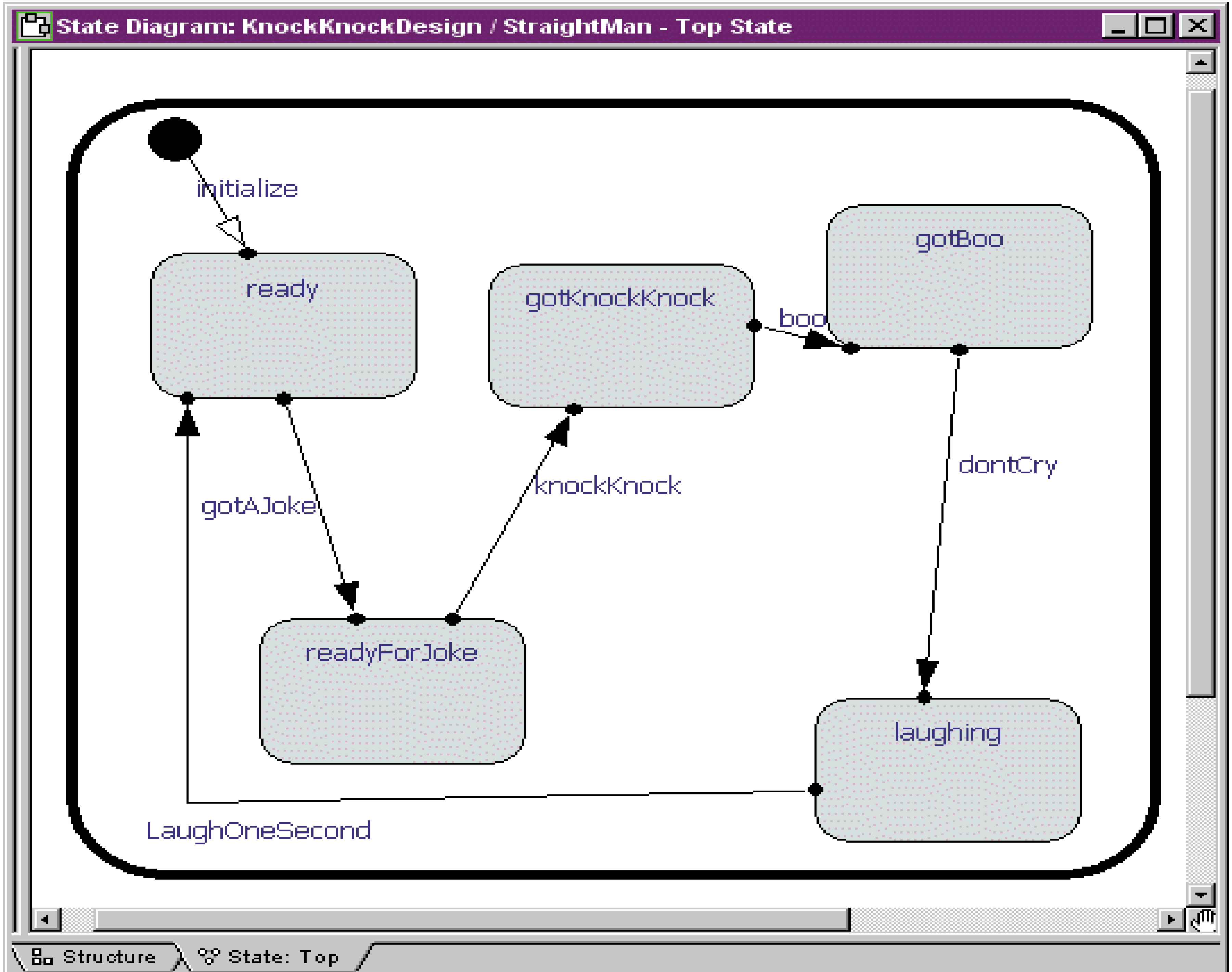
signal1
signal4
signal5

Example [M. Drahzal]

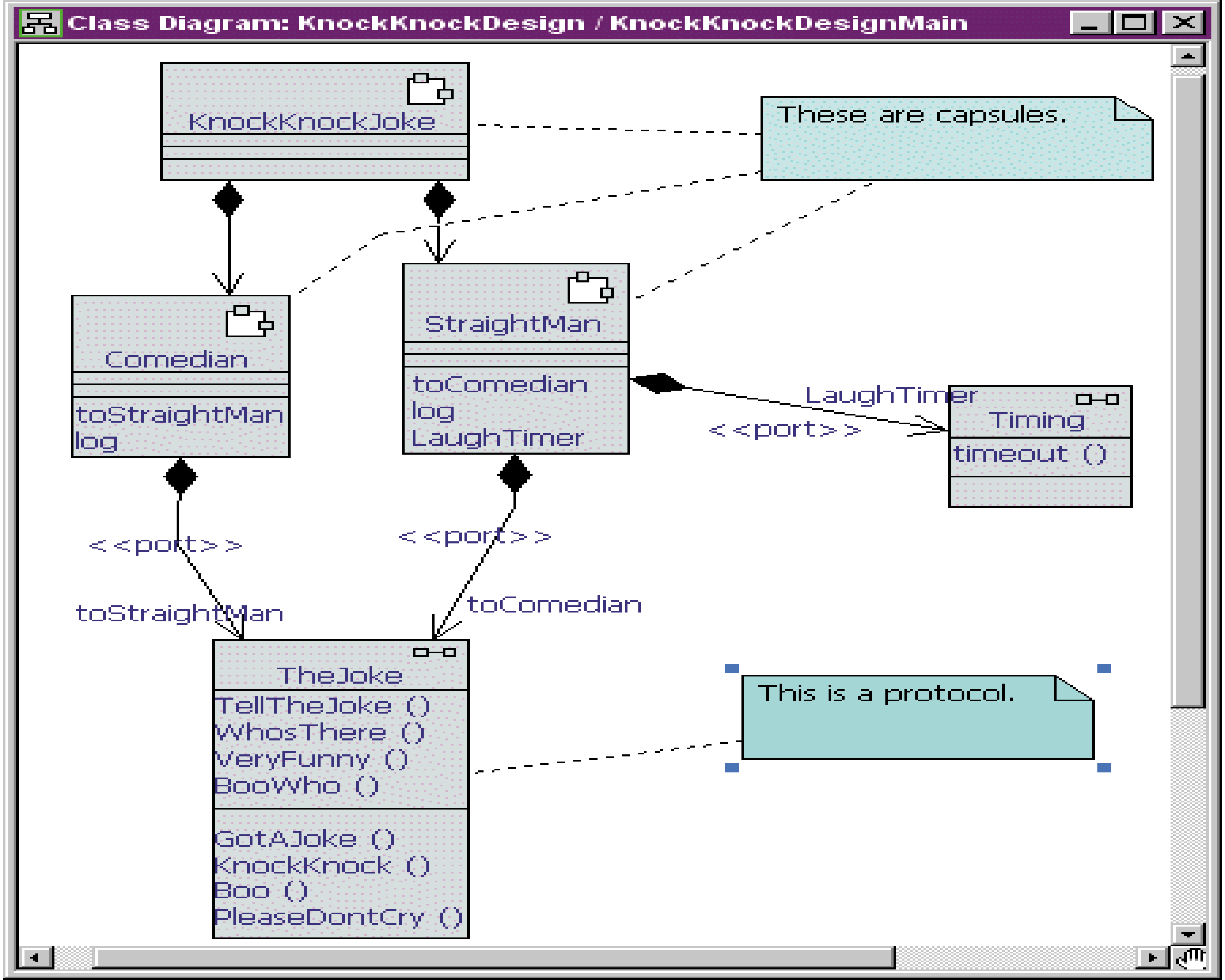
- Comedian: "Knock-Knock"
- Straight Man: "Who's there?"
- Comedian: "Boo”
- Straight Man: "Boo who?"
- Comedian: "Please don't cry!" (punchline)
- Straight Man: Laughs


Sequence Diagram: Logical View / Collaboration1 / Telling Joke
/ comedian : Comedian
/ straightMan : StraightMan
/ comedian
: Comedian
/ straightMan
: StraightMan
I have a Joke
Please tell the Joke
Knock-Knock
Whos There?
Boo
Boo Who?
laugh for one second to be polite
Please Don't Cry
Laughing
Very Funny


State Diagram: KnockKnockDesign / Comedian - Top State
initialize
ready
tellTheJoke
gotTellJoke
gotWhosThere
whosThere
booWho
gotBooWho
veryFunny
endOfJoke
Structure
State: Top



State Diagram: KnockKnockDesign / StraightMan - Top State
initialize
ready
gotKnockKnock
boo
gotBoo
gotAJoke
knockKnock
dontCry
readyForJoke
laughing
LaughOneSecond
Structure
State: Top


Class Diagram: KnockKnockDesign / KnockKnockDesignMain
KnockKnockJoke
These are capsules.
Comedian
toStraightMan
log
StraightMan
toComedian
log
LaughTimer
LaughTimer
<<port>>
Timing
timeout ()
<<port>>
<<port>>
toStraightMan
toComedian
TheJoke
TellTheJoke ()
WhosThere ()
VeryFunny ()
BooWho ()
GotAJoke ()
KnockKnock ()
Boo ()
PleaseDontCry ()
This is a protocol.

Time Service

- A **time service** is used to trigger events after an interval of time has expired.
- It is accessed through a standard port (service access point).
- A state machine can request to be notified with a "timeout" event a particular time of day is reached or when a particular time interval has elapsed.