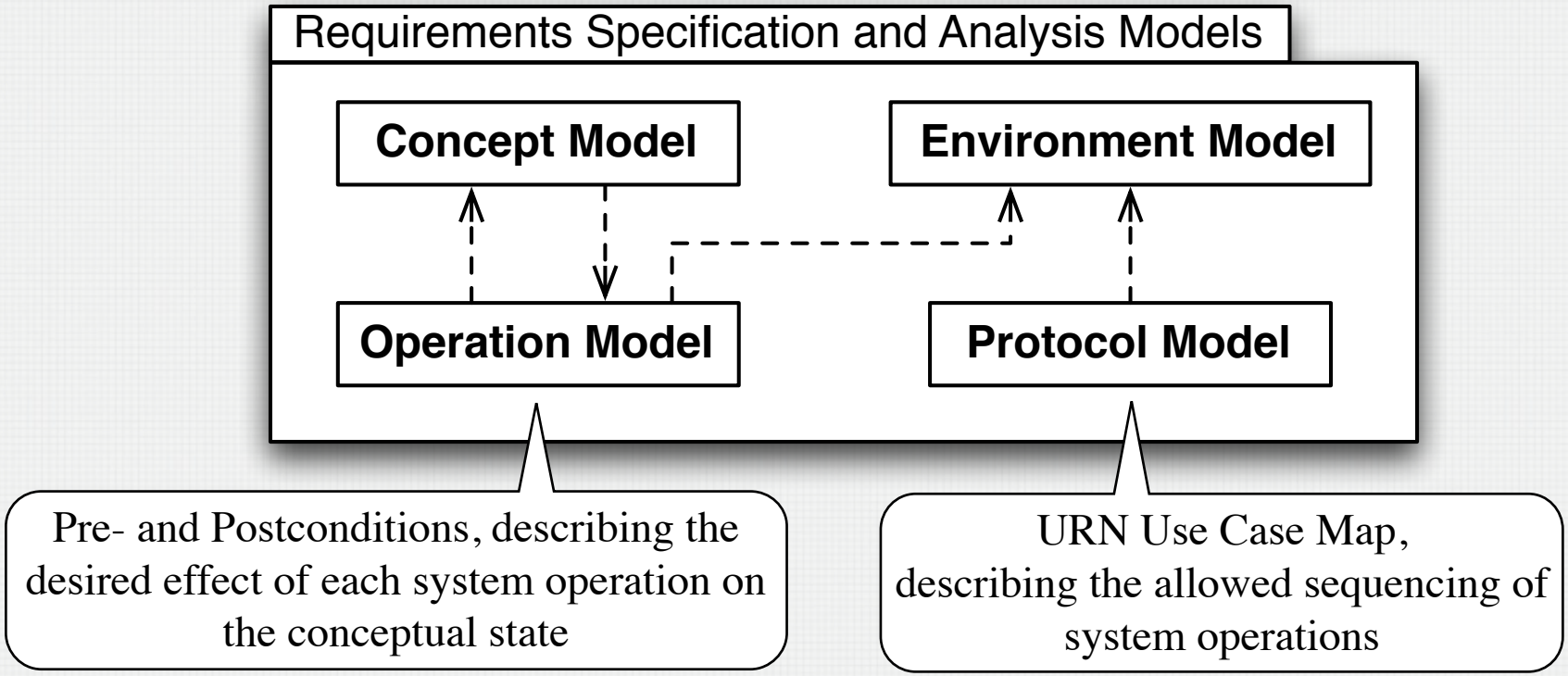
Specifying Behavioural Requirements

Jörg Kienzle & Alfred Strohmeier

Behavioural Requirements Overview

- Operation Model
  - System Operation
  - Operation Schema
  - Messages
  - Pre- and postconditions
- Protocol Model
  - Use Case Maps
  - From Use Cases to Protocol Model
- Requirements Specification Process Summary
- Checking Consistency of Requirements Models

Fondue Models: Requirements Spec.



Requirements Specification Phase

- Purpose
  - To produce a complete, consistent, and unambiguous description of
    - the problem domain and
    - the functional requirements of the system.
- Models are produced, which describe
  - Structural Models (see previous lecture)
  - Behaviour Models
    - Operation Model
      - Defines for each system operation the desired effect of its execution on the conceptual state
    - Protocol Model
      - Defines the system protocol, i.e., describes the allowed sequencing of system operations
  - The models concentrate on describing what a system does, rather than how it does it.

Operation Model (1)

- The Operation Model specifies each system operation declaratively by defining its effects in terms of (conceptual) system state changes and messages output by the system.
  - In the requirements specification / analysis phase, the conceptual state of a system is modelled as a set of objects that participate in associations.
  - The actual composition of the system state at any moment depends on what system operations have been invoked.

System Operation

- A system operation is considered to be a black box. No information is given about intermediate states when it is performed.
- A system operation may:
  - Create a new instance of a class;
  - Remove an object from the system state;
  - Change the value of an attribute of an existing object;
  - Add a link to an association;
  - Remove a link from an association;
  - Send a message to an actor (or multiple actors).

System Operation (2)

- The operations are typically specified by preconditions and postconditions, i.e. logical predicates.
- The precondition characterizes the valid initial states of the system when the operation is invoked. If the precondition is not true, the effect of the operation is undefined.
- The result of the operation is expressed as a postcondition. The postcondition describes the changes that were made to the state of the system and what messages have been sent to actors. It must determine the behaviour for all valid initial states (satisfiable schema).

System Contract

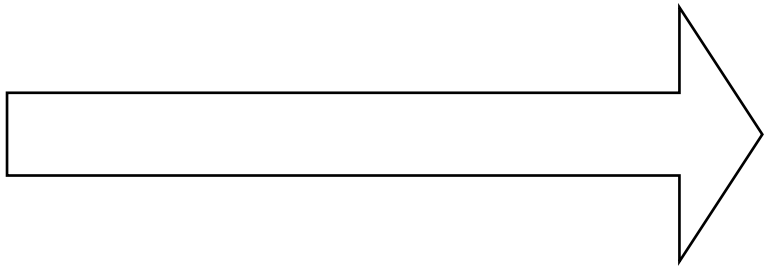
- The state of a system may also be subject to invariants, conditions that are true throughout its entire life cycle, i.e., especially before and after performing an operation.
- The precondition, the postcondition and the invariants define the contract for the service the system promises to provide (contract model of the system).

System Operations

- A system operation is considered to be a black box: no information about intermediate states.

Before

After



Precondition
must be true before

Postcondition
must be true after

Operation Schema

- Operation: The entity that services the operation (aka the name of the system), followed by the name of the operation and parameter list, and the type of the returned message, if any.
- Scope: All classes and associations from the Concept Model defining the name space of the operation.
- Messages: This clause declares all the message types that are output by the operation together with their destinations, i.e. the receiving actor classes.
- New: This clause declares names for objects that might be created by this system operation.
- Pre (optional): A concise natural language description of the preconditions that must be met in order for this operation to make sense.
- Post: A concise natural language description of the effects of the operation on the conceptual system state (i.e. the entities of the concept model).
- Use Cases: This clause provides cross-references to related use case(s).

Operation Schema Example (1)

Message (type) declarations:

InsufficientFunds_e(); DispenseCash(amount: Money);

Operation: Bank::withdraw (acc: Account, request: Money);
Scope: Account;
Messages: ATM::{InsufficientFunds_e; DispenseCash;};
Pre: true
Post: If there is enough money in the account, the effect of the operation is to decrease the balance of the account by the requested amount, as well as to send a “Dispense Cash” output message to the sender of the withdraw message, i.e. to the ATM machine requesting the withdraw. In case there is not enough money on the account, the operation outputs an “Insufficient Funds” message.

Messages (1)

- Messages are model elements. They have parameters. Messages are specifications of observable instances. A message type is quite similar to a class, and message instances are similar to objects of the class. Also, the parameters of a message type are similar to the attributes of a class.
- All communications between the system and the actors are through message instance delivery. The flow of information is in the same direction as the direction of the message; otherwise stated, all parameters are of mode in.
- All messages are asynchronous

Messages (2)

- Input message instances are incoming to the system and trigger input events that lead to the execution of system operations. The signature of the event corresponds to that of the message.
  - The parameters of the input message are the parameters of the input event which are the parameters of the system operation.
- Output message instances are outgoing from the system and are delivered to a destination actor.
- A message instance has an implicit reference to its sender. Only actors can act as senders.
- If an actor is able to deal with instances of a given message (type), then it is possible to state that a message instance is sent to the actor (e.g. as a result of an operation.)

Messages (3)

- Some message types are qualified as being exceptions. An instance of an exception signals an unusual outcome to the receiver, e.g., an overdraft of an account.
- We use a naming convention to differentiate exceptions from “regular” messages. The suffix “_e” is added to the name of an exception. The reason for this naming convention is to help specifiers visually differentiate between different kinds of messages.

Operation Schema: Operation

"**Operation**" ":" SystemClassName "::" OperationName "(" [ParameterList] ")"

- The SystemClassName is the server of the operation. It is also the context of the schema.
- The OperationName together with the ParameterList follows the syntax of a message declaration, since it corresponds to an input message sent to the system.
- All parameters in ParameterList are of mode in. This does not mean that the state of an object which is a parameter cannot be changed.
- Example

  Bank :: withdrawCash(acc: Account, amount: Money);

Operation Schema: Scope

"**Scope**" ":" NameList

NameList ::= (NameListElement ";")*

NameListElement ::= ClassName

- The Scope should list all conceptual system state (classes and association classes from the Concept Model) that the operation uses and affects.
  - Note that "simple" associations are not listed, only association classes.
- NameList is a list of class and association class names.

- Examples of NameListElements
  - Person
  - Account
  - Job

Operation Schema: New

"**New**: " ItemList
ItemList ::= (Item ";")*
Item ::= ObjectDeclaration | ObjectCollectionDeclaration

- The New clause declares names for objects or object collections to be created by the execution of the operation.
  - Each name in an ObjectDeclaration declares a distinct object.
- Examples

  New:
  acc1, acc2: Account;
  john: Person;
  bidders: Set (Person);

Operation Schema: Messages

"**Messages**" ":" ActorWithMessagesList
ActorWithMessagesList ::= (ActorWithMessages ";")*
ActorWithMessages ::=
ActorClassName "::" "{" (MessageName ";")* "}"

- ActorWithMessages shows which kinds of messages are sent to a given actor class.
- Examples

  Messages:

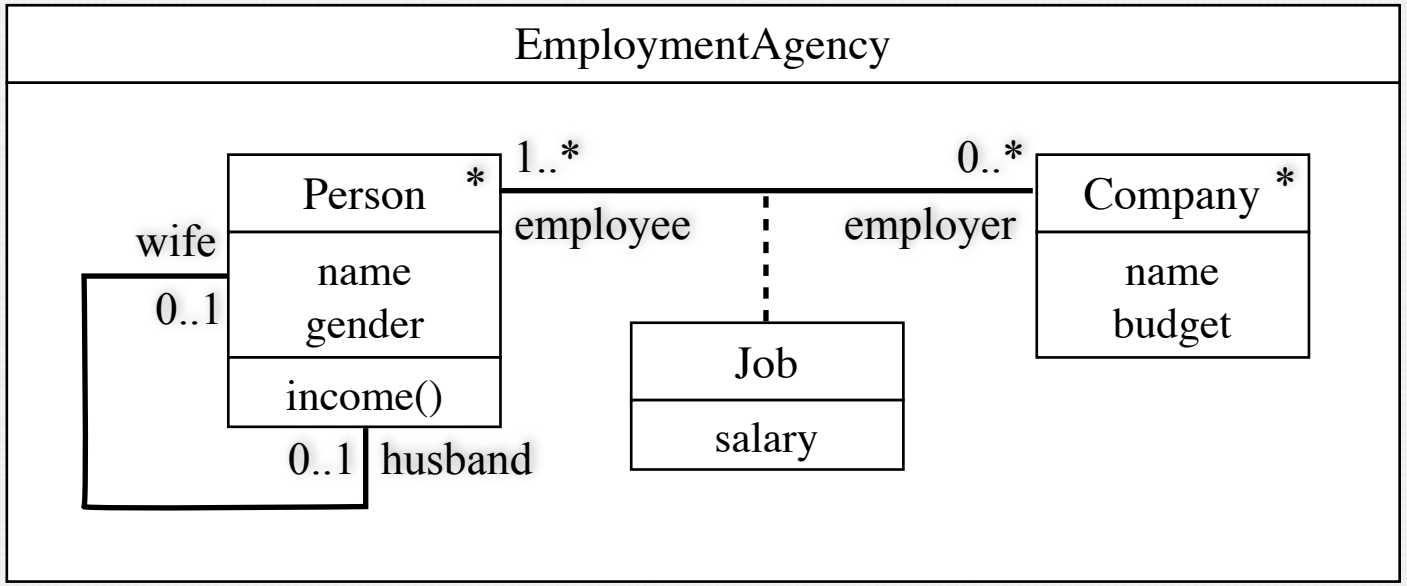
  ATM :: {DispenseCash; InsufficientFunds_e; Report;};
  Bank :: {Withdraw_r;};
  Clerk :: {AccountNumber;};

Message Declaration Examples

InsufficientFunds_e ();
DispenseCash (amount: Money);
DebitReport (amount: Money, timestamp: Date);

type Direction **is enum** {debit, credit};
type Transaction **is TupleType**
 {amount: Money, timestamp: Date, d: Direction};
Report(t: Transaction)
MonthlyReport(movements: Sequence
(Transaction));

Example UML Class Diagram



Operation Schema Example

Operation: EmploymentAgency::jobFilled
(worker: Person, comp: Company,
amount: Money);

Scope: Person; Company; Job;

New: newJob: Job;

Post: Creates a job for a given person and company, and initializes the salary attribute;

Operation Schema Example (2)

Message (type) declaration: Asset(name: String, number: AccNumber, amount: Money);

Operation:Bank::checkAssets ();
Scope: Account; Customer; Owns;
Messages: Manager::{Asset;};
Post: Sends multiple "Asset" output messages to the manager, one for each account in the bank system, providing the balance of the account together with the owner's name.

type LineItem **is TupleType** {name: String, number: AccNumber, amount: Money};
Message (type) declaration: CurrentAssets (contents: Sequence (LineItem));

Operation: Bank :: checkAssets ();
Scope: Account; Customer; Owns;
Messages: Manager :: {CurrentAssets;};
Post: Sends one output message to the manager, which lists the balances of all accounts, together with the owner's names.

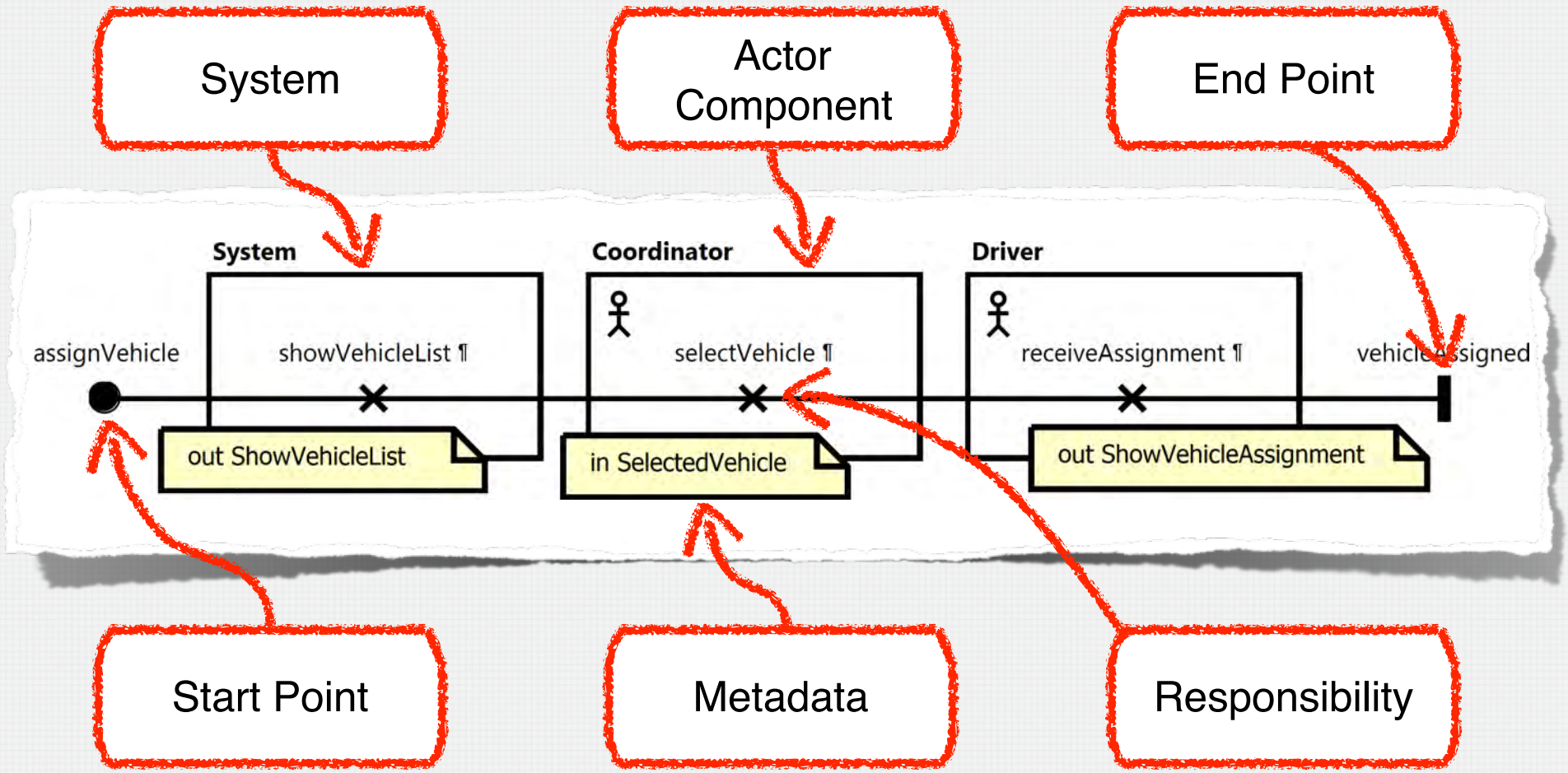
Protocol Model (1)

- The Protocol Model defines the allowable sequences of interactions that the system may have with its environment over its lifetime.
- If at any point the system receives an event, either time-triggered or triggered by a message, that is not allowed according to the Protocol Model, then the system ignores the event and leaves the state of the system unchanged.
  - Note: A dependable system, instead of ignoring the message, should inform the environment about the “interaction error”
- In this class, the Protocol Model is depicted by a use case map (UCM).

Use Case Maps

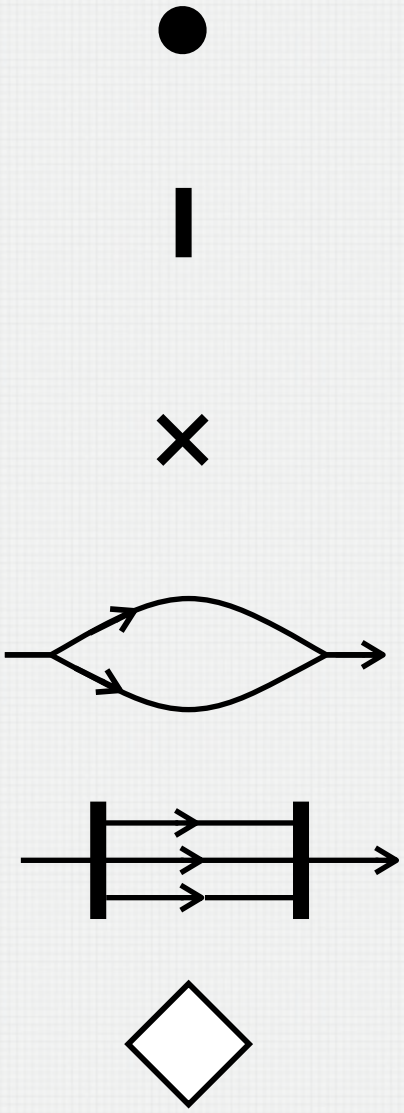
- Part of the User Requirements Notation (ITU Standard)
  - Not part of the UML
- Use Case Maps (UCMs) model scenario concepts
  - Causal relationships between responsibilities
  - Mainly for operational requirements, functional requirements, and business processes
  - For reasoning about scenario interactions, performance, and architecture
- Use Case Maps provide …
  - Visual description of behaviour superimposed over entities (from stakeholders and users to software architecture to hardware)
  - Easy graphical manipulation of scenario descriptions
  - Single scenario view
  - Combined system view
  - Connections to goal models (not covered in this class)
  - Connections to performance models and testing models (not covered in this class)

Use Case Map Example



Basic UCM Path Elements

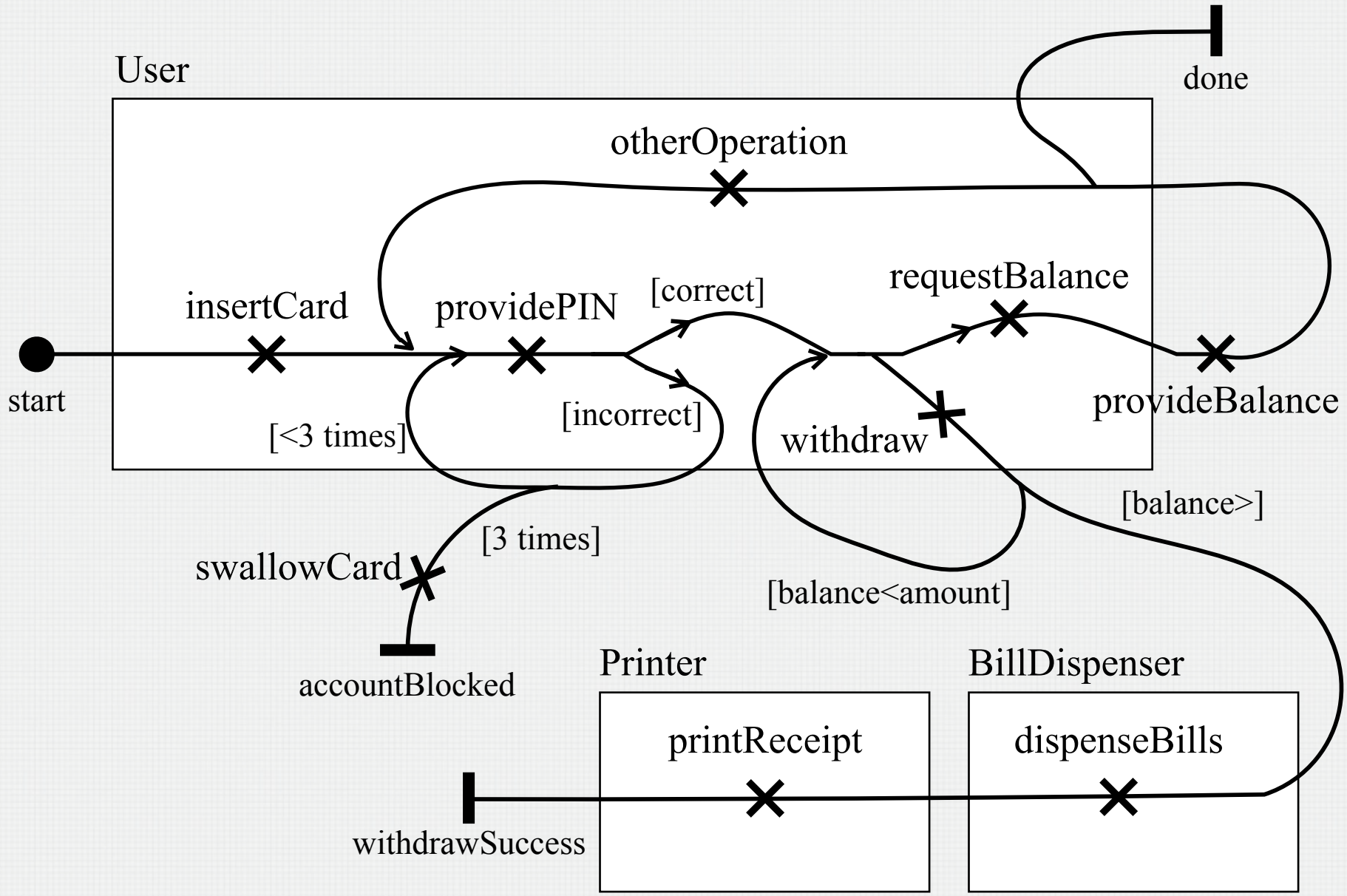
- Start Node
  - Flow of control begins here
  - There can be several start nodes
- End Node
  - Flow of control ends here
  - There can be several end nodes
- Responsibility
  - Represents an activity that is carried out by the system or an actor
- OR-fork/join
  - One path is taken, depending on a condition
- AND-fork/join
  - Fork: All paths are taken in parallel
  - Join: Wait until all paths are complete
- Stubs
  - Flow of control continues on another UCM



ATM Example

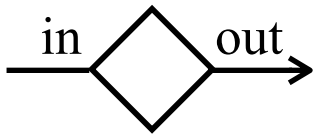
- An individual ATM works as follows:
  - The user starts a session by inserting her/his card. From then on, the user can abort the session whenever it pleases her/him. The session then ends immediately and the card is ejected, so the user can grab it.
  - Then the user must type in her/his personal identification number. Up to three trials are allowed, then the ATM swallows the card, and the session ends.
  - Once authorized, the user can either ask for account information or request a withdrawal.
  - If s/he requests a withdrawal, the specified amount is delivered, a receipt is printed, the card is ejected (so the user can grab it), and the session ends.
  - If the user asks for account information, the information is displayed. The user is then asked if s/he wants to perform another operation. If not, the session ends. If the user asks for another operation, for security reasons, s/he is asked to provide again her/his PIN, getting again up to three trials.

ATM Interaction UCM

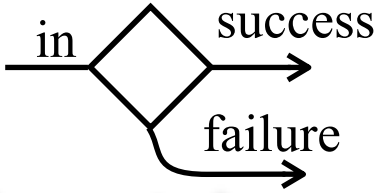


Advanced UCM: Stubs

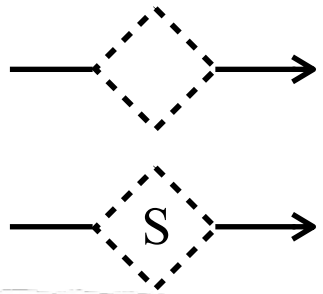
- A Stub is a placeholder for a (or several) UCM maps (called plug-in maps)
  - in paths are linked to start nodes, end nodes are linked to out-paths
- Static stub



- Static stub (multiple out-paths)

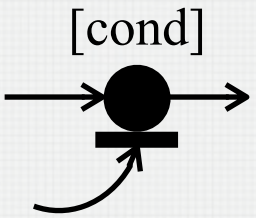


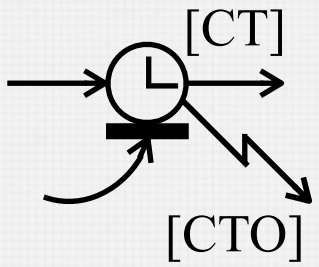
- Dynamic stub and dynamic synchronizing stub
  - Multiple UCMs are linked to a dynamic stub, and the ones that execute are chosen depending on a condition
  - For synchronizing stubs, the flow of control only continues once all executing UCMs have terminated



WAITING PLACES AND TIMERS

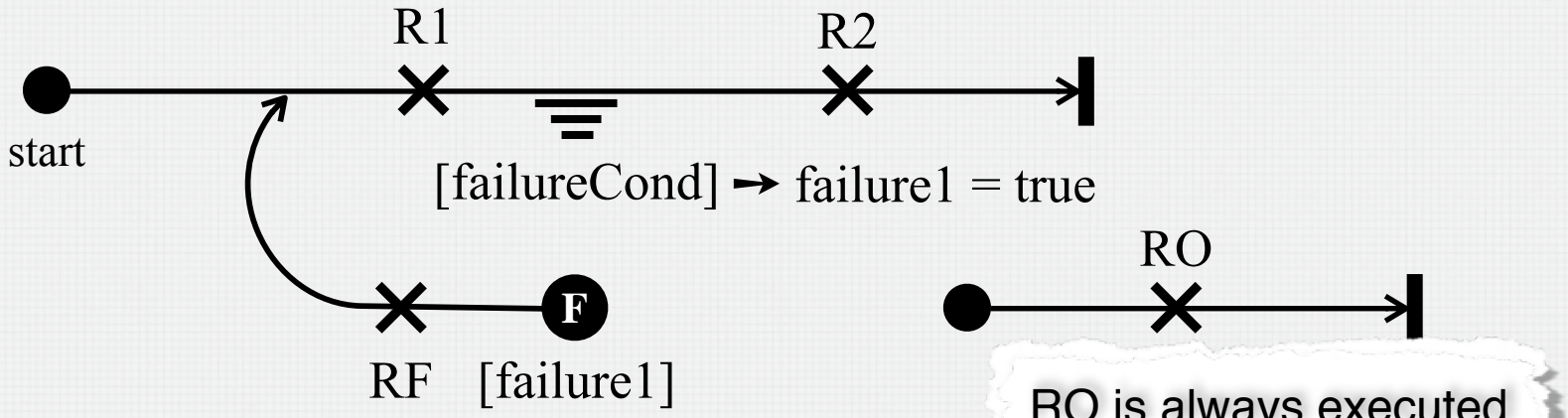
- Timer and Waiting Place
  - Transient = a trigger is counted only if a scenario is already waiting
  - Persistent = all triggers are counted (i.e., remembered)
- Waiting place
  - Scenario is allowed to continue if cond = true or the trigger counter > 0
- Timer
  - Regular path if CT = true
  - Regular path if CT = false and no timeout and CTO = false and trigger counter > 0
  - Timeout path if CT = false and CTO = true
  - Timeout path if CT = false and timeout occurred and CTO = false and trigger counter = 0





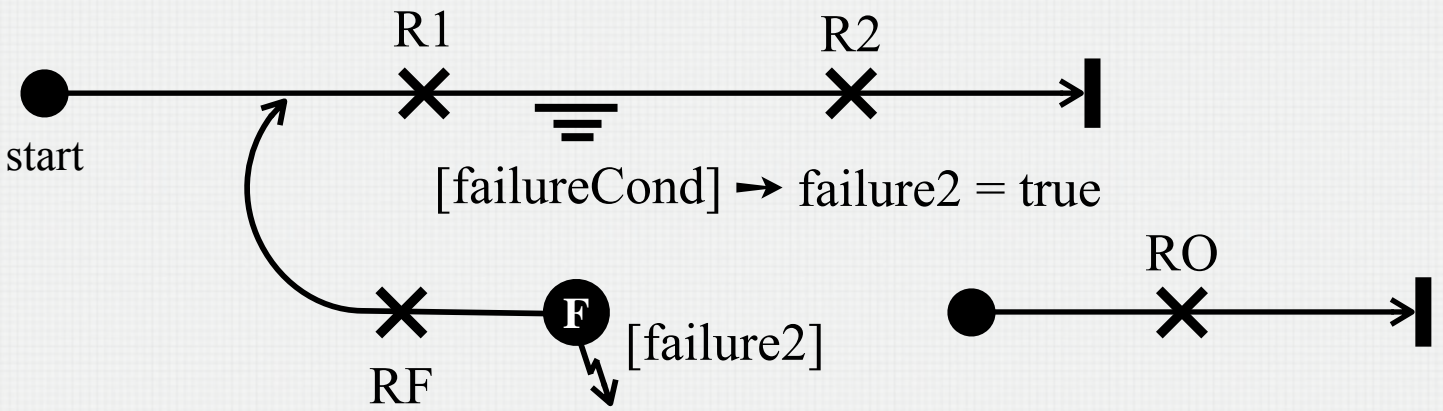
Failures and Aborts (1)

- Failure Start Point
  - Activated if condition evaluates to true
- Explicit approach with Failure Points
  - Indicates location of failure on scenario path
  - Failure condition sets failure variable to indicate which failure occurred
  - Control flow continues at failure point that corresponds to failure variable
    - Concurrent control flows remain unaffected



Failures and Aborts (2)

- Abort Start Points
  - Activated if condition evaluates to true
  - Aborts all other paths in the abort scope (all concurrent branches that are active on the same or lower level maps)



If failure2 is activated, control flow that executes RO is aborted

From Use Cases to UCMs

- The Use Case Model defines the interaction scenarios between the environment and the system informally
  - We use UCMs to formalize the interaction flow
- The Environment Model defines input and output messages used during the interaction
  - The UCMs describe the flow of control between the inputs and outputs
- Inputs are modelled as responsibilities marked with stereotype <<in>>
- Outputs are modelled as responsibilities marked with stereotype <<out>>

Parking Garage Example

This is an informal description of how an automobilist interacts with a parking garage control system (PGCS) when parking his car. The function of the PGCS is to control and supervise the entries and exits into and out of a parking garage. The system ensures that the number of cars in the garage does not exceed the number of available parking spaces.

The entrance to the garage consists of a gate, a state display showing whether any parking space is available, a ticket machine with a ticket request button and a ticket printer, and an induction loop (i.e., a device that can detect the presence or absence of a vehicle). To enter the garage, the driver receives a ticket indicating the arrival time upon his request. The gate opens after the driver takes the ticket. The driver then parks the car and leaves the parking garage. In case of problems, the PGCS notifies an attendant by means of an attendant call light.

EnterGarage UC High-Level (1)

Use Case: EnterGarage
Scope: PGCS
Level: User-Goal
Intention in Context: The Driver wants to enter the garage with his vehicle.
Multiplicity: Only one Driver can enter the garage at a given time per entry. If there are n entries, then n EnterGarage use cases can execute at the same time.
Primary Actor: Driver
Secondary Actors: Gate, Attendant
Main Success Scenario:

Driver drives the car to the entrance and stops.

1. *Driver* informs *System* that she is requesting entry.
2. *System* delivers ticket to *Driver*.
3. *System* is made aware that *Driver* took the ticket.
4. *System* instructs *Gate* to open.

Driver drives car passed the gate into the garage.

5. *System* is made aware that *Driver* has left the entry and passed the gate.
6. *Gate* informs *System* that it is closed.

EnterGarage UC High-Level (2)

Extensions:

2a. There are no more parking spots available.

2a.1 *System* informs *User* that there are no more parking spots available. Use case ends in failure.

3a. There is a problem with the ticket printer.

3a.1 *System* notifies *Attendant*. Use case ends in failure.

3b. Timeout: User has not taken the ticket.

3b.1 *System* notifies *Attendant*. Use case ends in failure.

5a. Timeout: User has not driven past the gate.

5a.1 *System* notifies *Attendant*. Use case ends in failure.

- Use case is written at a high level of abstraction. Not all secondary actors are mentioned.

➡ We need to identify all secondary actors, and clearly mark inputs and outputs

EnterGarage UC Low-Level (1)

Use Case: EnterGarage
Scope: PGCS
Level: User-Goal
Intention in Context: The Driver wants to enter the garage with his vehicle.
Multiplicity: Only one Driver can enter the garage at a given time per entry. If there are n entries, then n EnterGarage use cases can execute at the same time.
Primary Actor: Driver
Secondary Actors: TicketButton, TicketPrinter, Gate, InductionLoop, AttendantLight, Display
Main Success Scenario:

Driver drives the car to the entrance, stops and hits TicketButton.

1. *TicketButton* informs *System* that a Driver is requesting entry. «in»
2. *System* instructs *TicketPrinter* to print ticket for *Driver*. «out»
3. *TicketPrinter* informs *System* that *Driver* took the ticket. «in»
4. *System* instructs *Gate* to open. «out»

Driver drives car past the gate into the garage.

5. *InductionLoop* informs *System* that *Driver* has left the entry and passed the gate. «in»
6. *Gate* informs *System* that it is closed. «in»

EnterGarage UC Low-Level (2)

Extensions:

2a. There are no more parking spots available.

2a.1 *System* displays on *Display* that there are no more parking spots available. Use case ends in failure. «out»

3a. *TicketPrinter* informs System that there is a printing problem. «in»

3a.1 *System* turns on *AttendantLight*. Use case ends in failure. «out»

3b. Timeout: User has not taken the ticket.

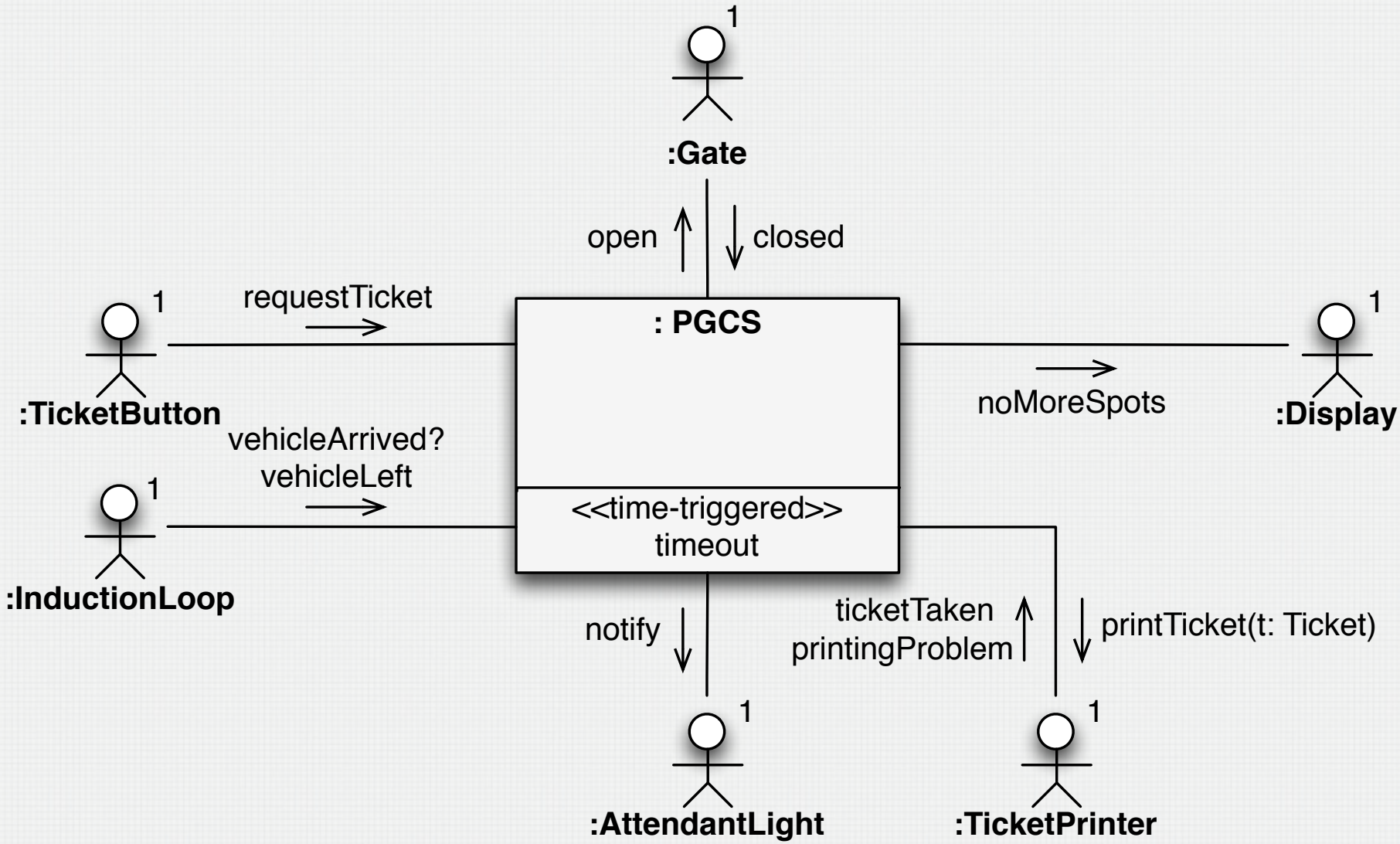
3b.1 *System* turns on *AttendantLight*. Use case ends in failure. «out»

5a. Timeout: User has not driven past the gate.

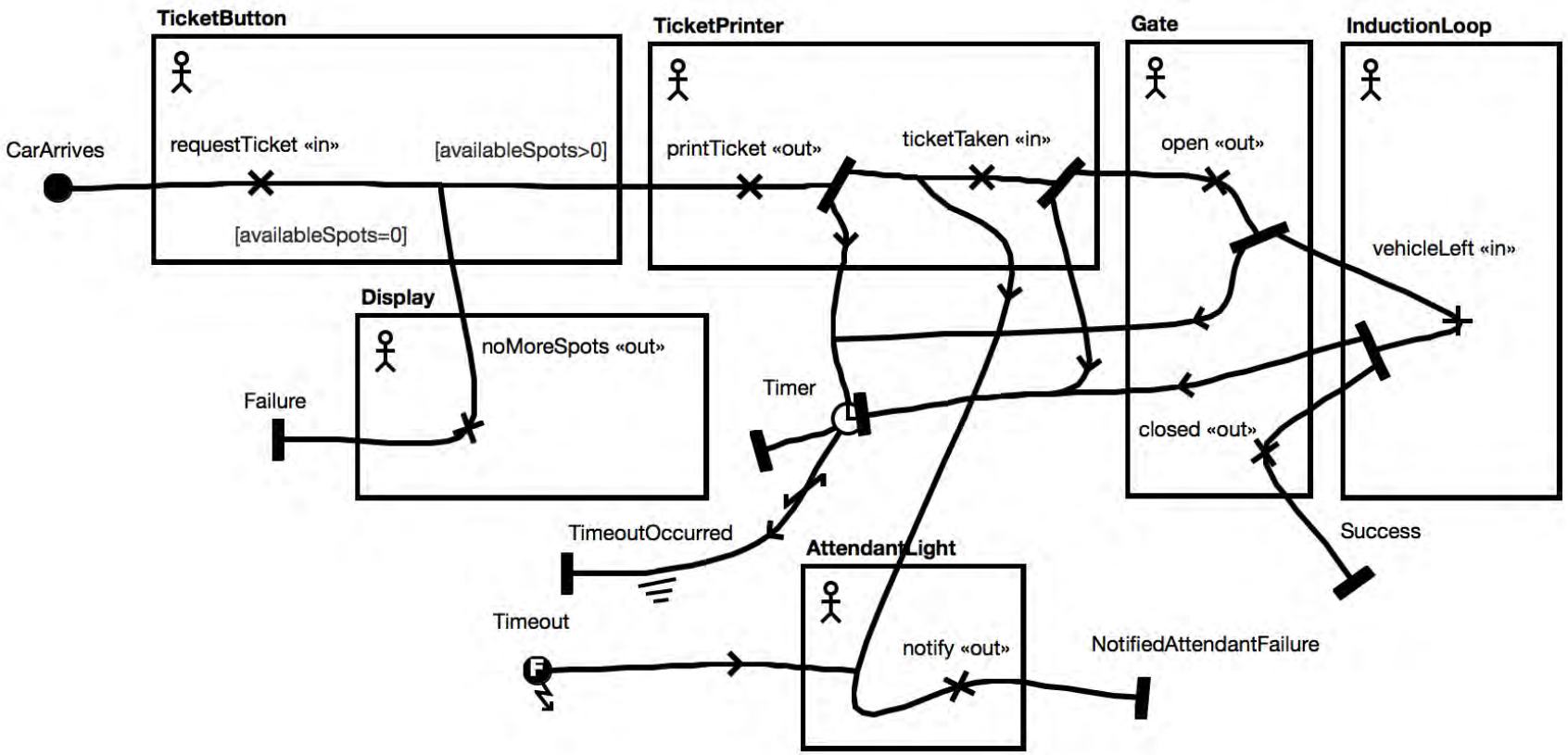
5a.1 *System* turns on *AttendantLight*. Use case ends in failure. «out»

- Now we can define message types for each communication, and derive the Environment Model

Parking Garage Environment Model



EnterGarage Protocol Model



Protocol Model and Operation Model (1)

- The behaviour of a system is defined by the Protocol Model and the Operation Model taken together.
- The Protocol Model determines the acceptability of an event and therefore of the corresponding triggering message.
- The precondition in the Operation Schema determines if the effect of an event is well behaved / defined.
- The Protocol Model takes precedence over the precondition, as shown by the following table:

	Precondition true	Precondition false
Protocol accepts	Operation invoked and effect defined	Operation invoked but effect undefined
Protocol rejects	Event ignored	Event ignored

Protocol Model and Operation Model (2)

- Rejecting/ignoring an input event means that the state of the system is unaffected.
  - However, analysis uses an abstract notion of state, and the implementation is free to respond to the erroneous event and its triggering message, for example, with a helpful error message.
- A system need not have a Protocol Model
  - All input events are then acceptable at any time.

Checking for Model Consistency

- The analysis models should be complete and consistent
  - A model is complete when it captures all the meaningful abstractions in the domain.
  - Models are consistent when they do not contradict each other.
    - A model can also be checked for internal consistency.

Requirements Specification Process (1)

- 1. Determine the system interface
  - 1.1 For establishing the system interface, analyze the scenarios in the Use Case Model. For each scenario:
    - Find the actors who are involved, and
    - The services they need.
  - 1.2 Develop the Environment Model: identify actors, output messages, and input messages (system operations).
  - 1.3 Produce the Concept Model by adding the boundary and actors to the Domain Model. Only actors having direct interaction with the system should be shown, and nothing else should appear outside of the boundary. Add roles to all association ends.

Requirements Specification Process (2)

- 2. Develop the Behaviour Model
  - 2.1 Develop the Protocol Model
    - Generalize the scenarios of the Use Case Model and define system states.
    - Combine system states to form the Protocol Model.
  - 2.2 Develop the Operation Model
    - 2.2.1 For each system operation, develop the precondition and description of effects:
      - Describe each aspect of the result with a sentence.
      - Use the Environment Model to find the messages that have to be output as a result.
    - 2.2.2 Derive Scope and Messages clauses from the description (incrementally with 2.2.1)
    - 2.2.3 Complete the message (type) declarations in the Environment Model

Requirements Specification Process (3)

- 3. Check the Analysis Models
  - 3.1 Check for completeness against the requirements:
    - All possible scenarios stated in the use cases are covered by the Protocol Model.
    - All required system services can be mapped onto system operations.
    - All static information is captured by the Concept Model.
    - Any other information, e.g. technical definitions and invariant constraints, are documented.
    - To check for completeness, compare the Use Case Model and the Operation Model
      - Inspect the use cases and define the state change that each should cause. Then "execute" the use cases, using the operation schemas. Check that resulting state conforms to what was expected.
      - If the use cases describe alternate flows, make sure that the Concept Model contains all the state needed to decide which scenario should execute.

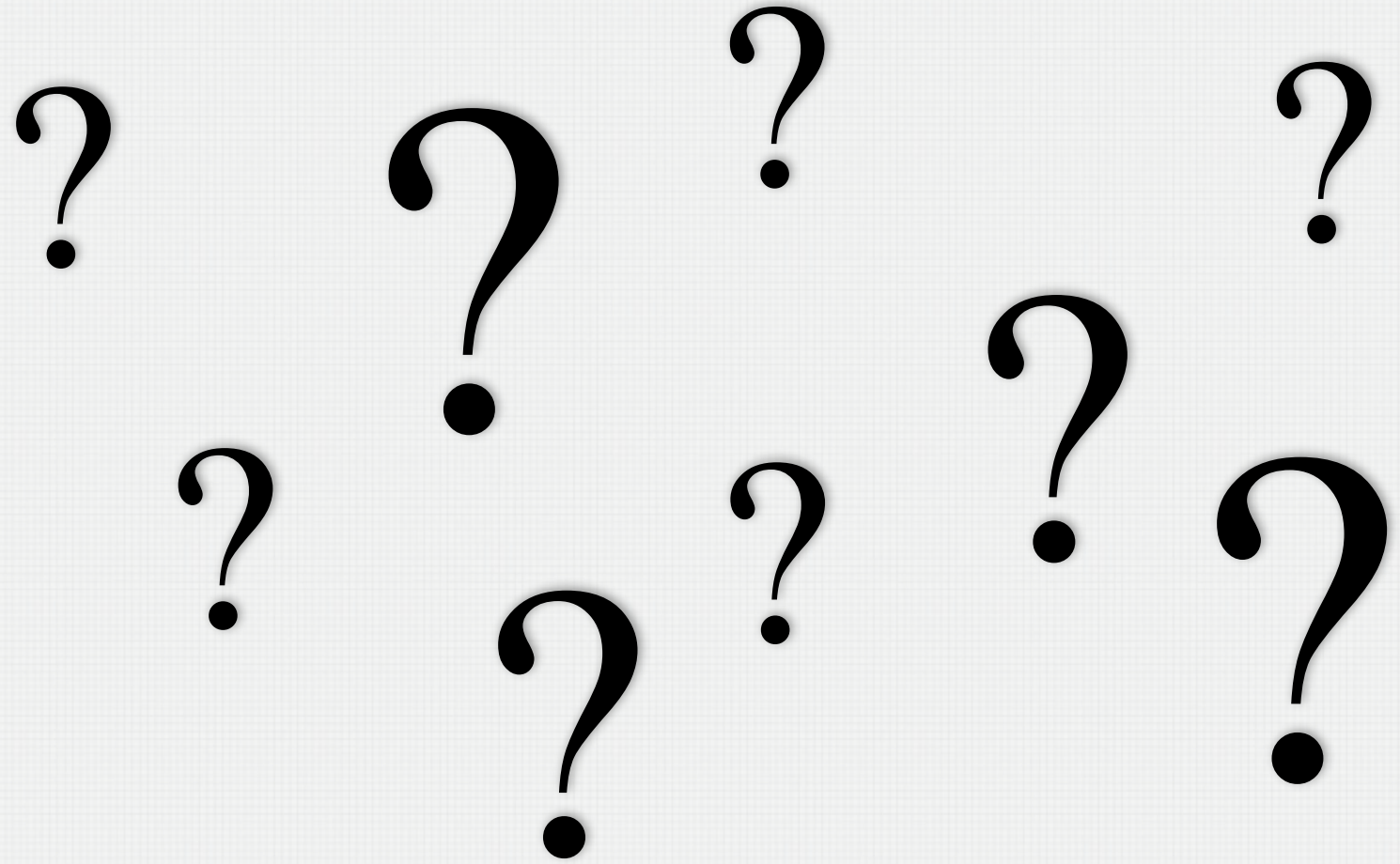
Requirements Specification Process (4)

- 3.2 Consistency between models:
  - Domain Model versus Concept Model:
    - All classes, relationships and attributes mentioned in the Domain Model appear in the Concept Model, or their absence can be justified and is documented.
  - Environment Model versus Concept Model:
    - The boundary of the Concept Model is consistent with the Environment Model.
  - Environment Model versus Protocol Model:
    - Every input message in the Environment Model appears in the Protocol Model as an event, and vice versa.
  - Concept Model versus Operation Model:
    - All classes, attributes and associations used in the descriptions of the Operation Model are part of the Concept Model, and there are no “useless” classes in the Concept Model.
    - The Operation Model must preserve Concept Model invariants.
  - Environment Model versus Operation Model:
    - An actor that appears in the Operation Model is part of the Environment Model.
    - All input messages in the Environment Model must trigger an operation modeled by an operation schema in the Operation Model, and all output messages in the Environment Model must be generated by a system operation.

Iterative Development

- It is usually necessary to go forth and back between the Environment Model, the Concept Model and the Operation Model to make them
  - Complete
  - Consistent
  - As simple as possible (by eliminating the unused elements)

Questions?



Train Depot Questions (1)

- Develop a Concept Model that models the following situation:
  - A train is composed of train engines and cars.
  - Train engines and cars have a certain weight (measured in steps of 1 kg).
  - A car has a current load and a maximum carrying capacity (also measured in steps of 1 kg).
  - Train engines can pull up to a certain amount of kg (traction).

Train Depot Question (2)

- Write Operation Schemas to

1. Change the load of a car (as a result of loading or unloading it). The new weight is a parameter of the operation.
2. Add an (existing) car to a train.
3. Transfer one train unit from one train to another one.
4. Compute the total load of a train and communicate it to the driver of the train. What are the consequences for the concept model?

Elevator Operation Model

- You are to devise the Operation Model for the elevator system based on the Environment Model and the Concept Model.
  - There is only one elevator cabin, which travels between the floors.
  - There is a single button on each floor to call the lift.
  - Inside the elevator cabin, there is a series of buttons, one for each floor.
  - Requests are definitive, i.e., they cannot be cancelled, and they persist; thus they should eventually be serviced.
  - The arrival of the cabin at a floor is detected by a sensor.
  - The system may ask the elevator to go up, go down or stop. In this example, we assume that the elevator's braking distance is negligible.
  - The system may ask the elevator to open its door. The system will receive a notification when the door is closed. This simulates the activity of letting people on and off at each floor.
  - The door closes automatically after a predefined amount of time. However, neither this function of the elevator nor the protection associated with the door closing (stopping it from squashing people) are part of the system to realize.

Take Lift Use Case (1)

Use Case: Take Lift
Scope: Elevator Control System
Level: User Goal
Intention in Context: The User intents to go from one floor to another.
Multiplicity: The System has a single lift cabin that may service many users at any one time.
Primary Actor: User
Main Success Scenario:
1. User enters lift.
2. User exits lift at destination floor.

Extensions:
1a. User fails to enter lift; use case ends in failure.

Enter Life Use Case (1)

Use Case: Enter Lift
Scope: Elevator Control System
Level: Subfunction
Intention in Context: The User intends to enter the cabin at a certain floor.
Primary Actor: User
Secondary Actors: Floor Sensor, Motor, Door
Main Success Scenario:
1. User requests System for lift;
2. System acknowledges request to User.
3. System requests Motor to go to source floor.

Step 4 is repeated until System determines that the source floor of the User has been reached

4. Floor Sensor informs System that lift has reached a certain floor.
5. System requests Motor to stop;
6. Motor informs System that lift is stopped.
7. System requests Door to open;

User enters lift at source floor.

Enter Life Use Case (2)

Extensions:

3a. System determines that another request has priority:

3a.1. System schedules the request; use case continues at step 2.

3b. System determines that the cabin is already at the requested floor. 3b.1a System determines that the door is open; use case ends in success.

3b.1b System determines that the door is closed; use case continues at step 7.

Exit Life Use Case (1)

Use Case: Enter Lift
Scope: Elevator Control System
Level: Subfunction
Intention in Context: The User intends to leave the cabin at a certain floor.
Primary Actor: User
Secondary Actors: Floor Sensor, Motor, Door
Main Success Scenario:
Steps 1 and 2 can happen in any order.
1. User requests System to go to a floor.
2. System acknowledges request to User.
3. Door informs System that it is closed.
4. System requests Motor to go to destination floor.

Step 5 is repeated until System determines that the destination floor of the User has been reached.

5. Floor Sensor informs System that lift has reached a certain floor.
6. System requests Motor to stop.
7. Motor informs System that lift is stopped.
8. System requests Door to open.
9. User exits lift at destination floor.

Exit Lift Use Case (2)

Extensions:

(3-5)lla. User requests System to go to a different floor;

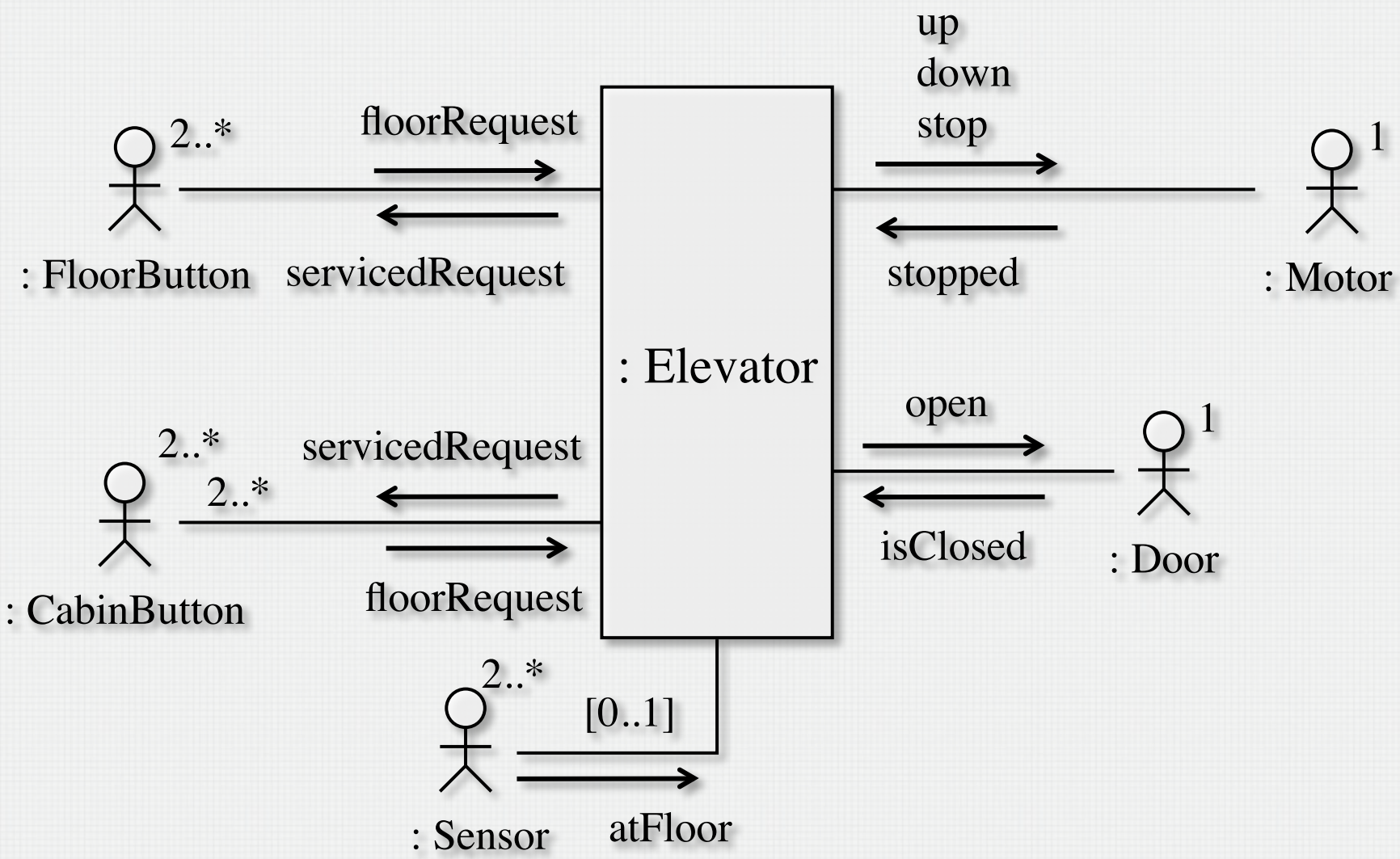
(3-5)lla.1 System schedules the request; use case continues at the same step.

4a. System determines that another request has priority.

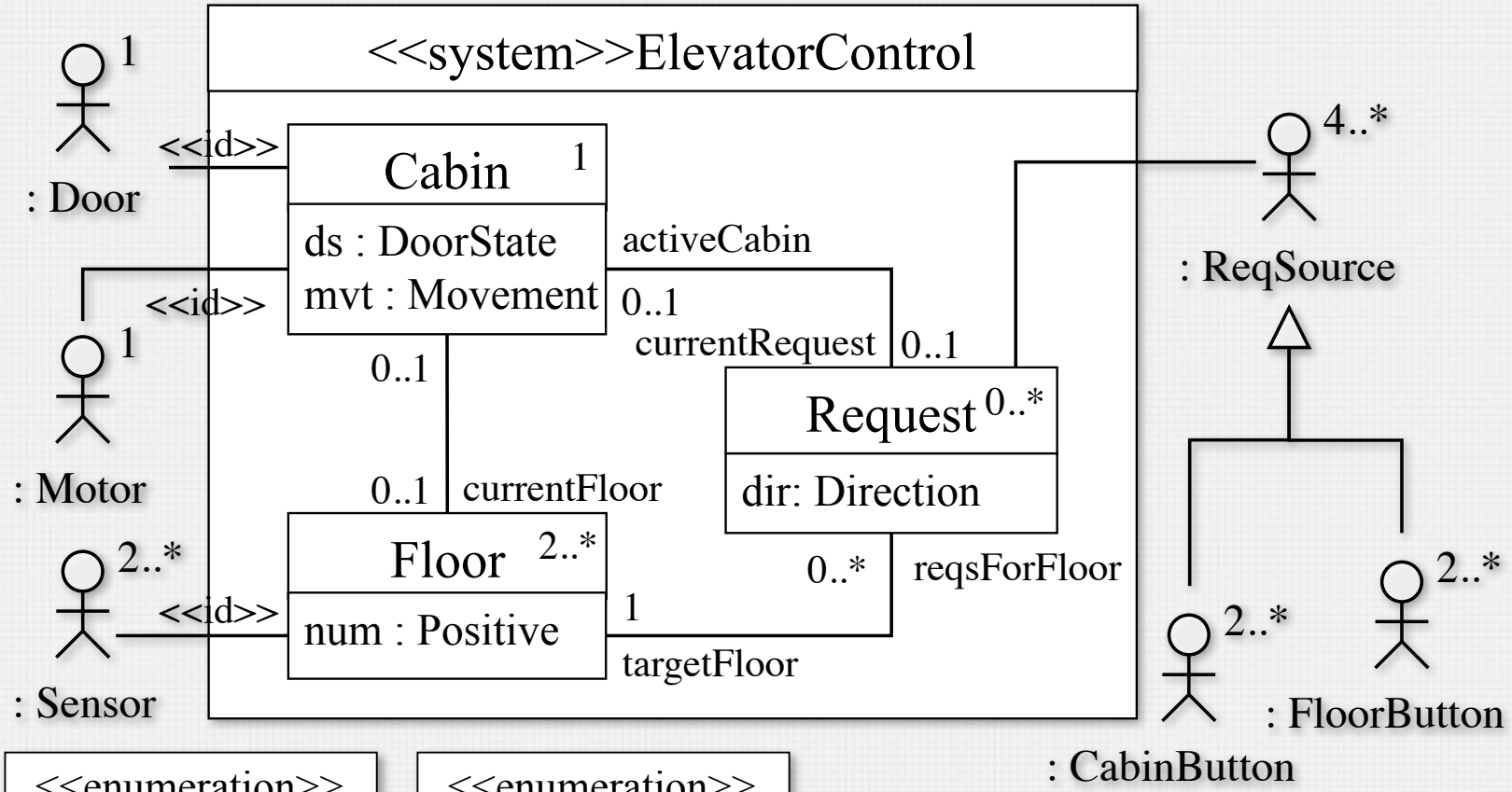
4a.1. System schedules the request; use case continues at step 4.

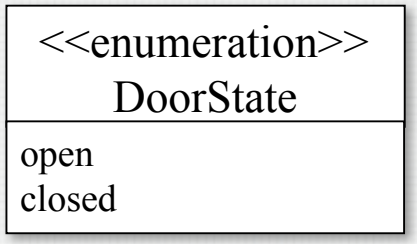
9a. System determines that there are additional requests pending; use case continues at step 3.

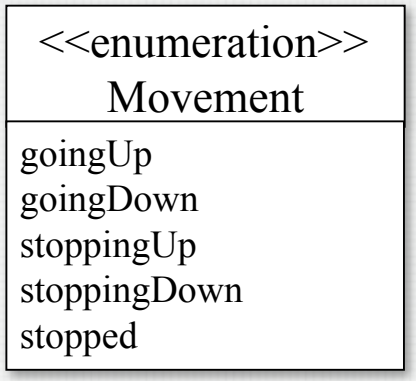
Elevator Environment Model

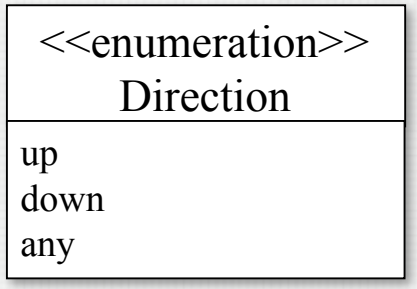


Elevator Concept Model









Elevator Operation Model Question

- You are to develop the Operation Model for the Elevator System based on the Environment Model and the Concept Model, i.e. you have to write the 4 operation schemas atFloor(f : Floor), stopped, isClosed, floorRequest(f : Floor).

Clinical Lab System Question

- The task is to develop a computerized data management system for a clinical test analyzer. An analyzer can carry out tests on body fluids such as blood, urine, and swab specimens. An analyzer is capable of carrying out tests on several samples simultaneously.
- The technician enters a batch of samples from a single patient by first entering the patient's identification and then indicating, one at a time, the tests that need to be performed on the samples. By a "batch end" message, s/he informs the system that there are no more samples for the current patient. When all the tests for a patient have been performed by the analyzer, they are collected together into a patient report, which is sent to the technician.

Clinical Lab System Question (2)

- The system can perform test requests for more than one patient at a time. The technician may ask for a report reflecting the current status of a patient's tests before they are all completed. The tests for a patient may also be aborted, in which case a patient report containing just the test results collected so far is generated and all further tests on samples from the same patient are ignored.
- Environment Model
  - Show by a Environment Model the interaction between the technician, the system and the analyzer.
  - Provide message declarations.
  - Write down some possible/forbidden message sequences; show both input and output messages. (Can also be answered based on the Protocol Model.)
- Protocol Model
  - Devise a Protocol Model for the clinical lab system

Drink Vending Machine (1)

A drink vending machine is a simple but non-trivial kind of reactive system. The physical machine consists of several components: drink shelves, a sensor that detects when a drink has been delivered, drink selector buttons, lights indicating drink availability, a display, a coin slot, a cancel button, and a money box that stores the coins inside the machine. Inside the machine, there is also a small terminal for use by the service person. Human interaction takes place between a customer or service person and the physical components when, for example, the customer selects a drink, or when the service person replenishes a shelf or changes the price of a drink. Finally, software coordinates the physical components, receiving messages from them and sending commands to them.

Drink Vending Machine (2)

During an informal talk with the vending company it has been determined that the interaction between a customer and the machine should be as follows: the customer first selects a drink, then inserts coins, and when the inserted amount reaches or exceeds the price for the drink, the drink is provided together with the change.

Also, due to the constraints of the hardware, some decisions had to be made. They are summarized below:

Drink Vending Machine (3)

The money box is actually composed of three different collectors: the first one keeps coins until the sale is complete or cancelled, the second one receives the coins from the first one once the drink has been distributed, and the third one contains coins used for change. The money box accepts 25 cents, 1 dollar and 2 dollar coins. It is assumed that the first collector is big enough to hold as many coins as needed to buy the most expensive drink and more. The second collector, although very big, has a limited capacity and must be emptied by the service person now and then. If the moneybox is full, the vending machine goes out of service. The third collector contains a fixed amount of 25 cent coins used for change. If there is no change left and a client does not insert the right amount, “no change” is displayed and the money is returned to the client.

Drink Vending Machine (4)

Drinks are stored on shelves. Each shelf is associated with a beverage kind and a price. Prices are all multiples of 25 cents. The number of shelves of the machine is fixed, and the capacity of each shelf is fixed as well. Initially, there are no drinks in the shelves.

If for some reason a selected drink can not be delivered, the sale is cancelled.

Only authorized personnel is allowed to service the machine.

Buy Drink Use Case (1)

Use Case: Buy Drink

Scope: Vending Machine

Level: User Goal

Intention in Context: The intention of the *Customer* is to buy a drink in exchange of money.

Multiplicity: There can always be only one *Customer* interacting with the system at a given time.

Primary Actor: Customer

Secondary Actors: Selector Button, Coin Slot, Shelf, Sensor, Money Box, Drink Light, Cancel Button, Display, Terminal

Precondition: The system is in service, filled with drinks and change, and the Money Box is not full.

Buy Drink Use Case (2)

Main Success Scenario:

Customer selects drink by pushing appropriate drink selector button.

1. *Button* notifies *System* of selected drink.
2. *System* displays the price of the selected drink on *Display*.

Customer inserts a coin into Coin Slot.

3. *Coin Slot* notifies *System*.
4. *System* recognizes the coin, and updates the remaining price on *Display*.

Steps 3 and 4 are repeated until the amount of inserted money reaches or exceeds the price of the drink.

5. *System* validates that there are sufficient funds for the selection and notifies *Shelf* to start dispensing the drink.
6. *Sensor* informs *System* that the drink has been dispensed.
7. *System* asks *Money Box* to collect the specified amount of money and, if necessary, provide the change.

Customer collects the drink and optionally the change.

Buy Drink Use Case (3)

Extensions:

2a. *System* ascertains that the selected drink is not available and flashes *Drink Lights*; use case ends in failure.

4a. *System* fails to identify the coin; *System* asks *Money Box* to eject coin; use case continues at step 3.

(3-4)a. *Customer* informs *System* to abort the sale by hitting the *Cancel* button;

(3-4)a.1 *System* asks *Money Box* to eject coins; use case ends in success.

(3-4)b. *System* times out.

(3-4)b.1 *System* asks *Money Box* to eject the inserted coins; use case ends in failure.

5a. *System* ascertains that the inserted money exceeds the price for the drink and that there is not enough change;

5a.1 *System* asks *Money Box* to eject inserted coins.

5a.2 *System* displays “no change” on *Display*; use case ends in failure.

7a ll. The *Money Box* is full.

7a ll.1 *System* displays “no service” on *Display* and goes out of service; use case ends successfully.

7b ll. The delivered drink was the last one of that kind.

7b ll.1 *System* turns on the appropriate *Drink Light*; use case ends successfully.

Service Machine Use Case (1)

Use Case: Service Machine

Scope: Vending Machine

Level: User Goal

Intention in Context: The intention of the Service Person is to maintain the machine by ensuring that there are drinks available, modifying drink pricing, and by collecting the money earned.

Multiplicity: There can be only one service person servicing the machine at a given time.

Primary Actor: Service Person

Precondition: No Customer is currently using the system.

Service Machine Use Case (2)

Main Success Scenario:

Service Person interacts with the system by using the Terminal.

1. *Service Person* identifies himself with *System*.

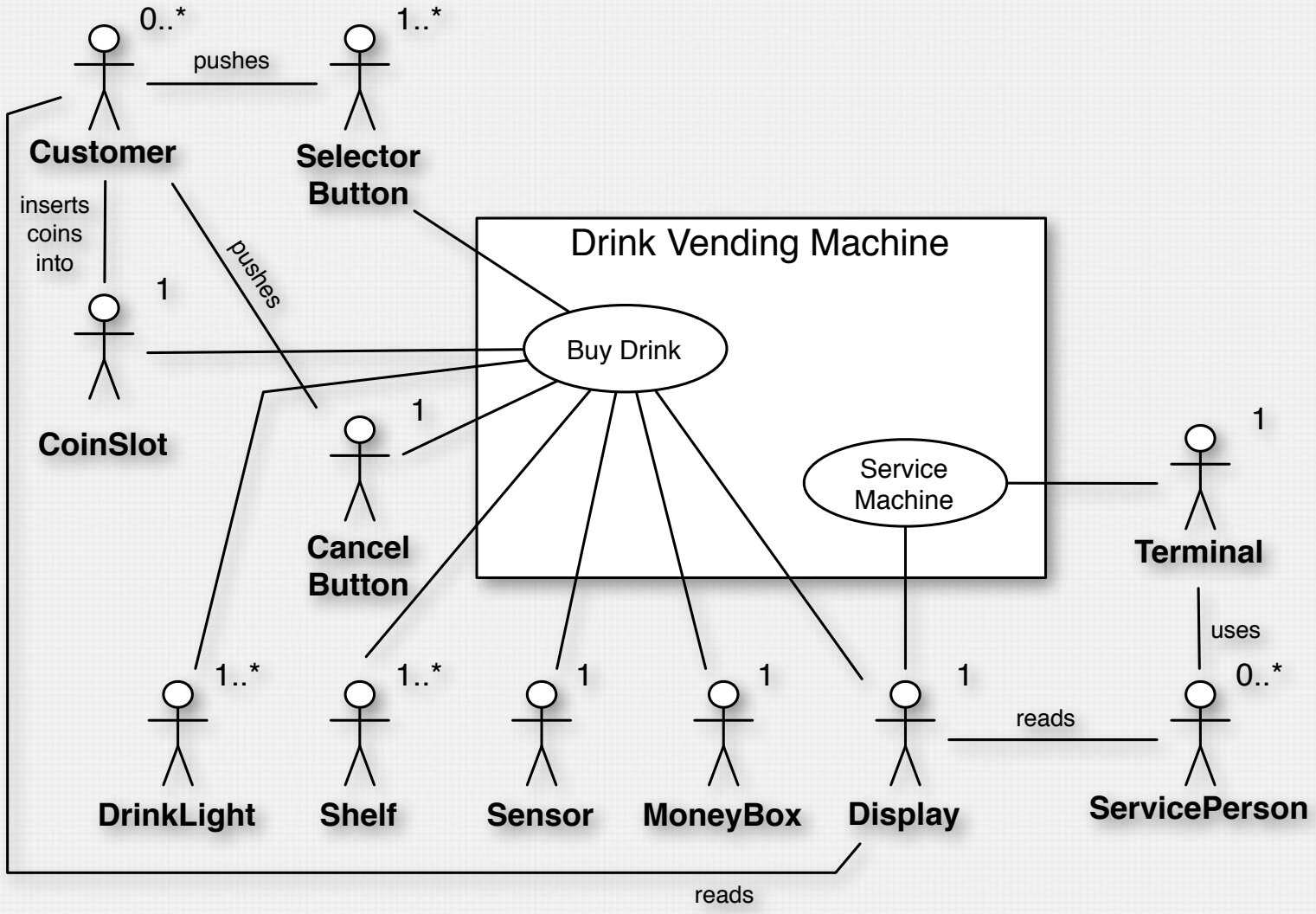
Steps 2-3 can be repeated for each shelf, in any order.

2. *Service Person* informs *System* of new price for a shelf.
3. *Service Person* replenishes a shelf and informs *System* of new number of drinks for that shelf.
4. *Service Person* empties the *Money Box*, replenishes the change and informs the *System*.
5. *Service Person* informs *System* that maintenance is over.

Extensions:

2a. *System* fails to identify the *Service Person*; use case ends in failure.

Vending Machine Use Case Diagram



Drink Vending Machine Questions

1. Create an Environment Model for the Drink Vending Machine
2. Create a protocol model for the Drink Vending Machine