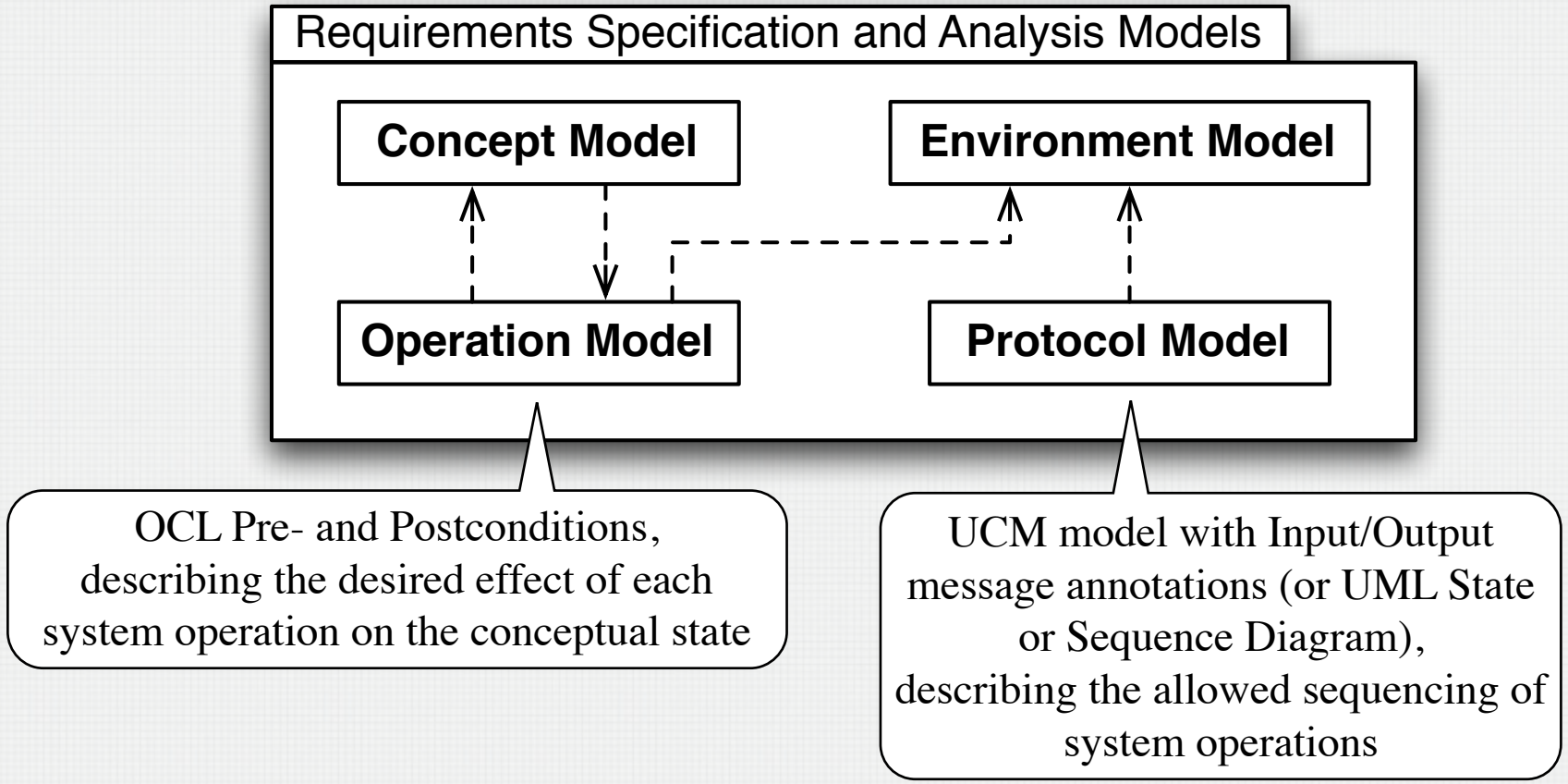
Operation Model

Jörg Kienzle & Alfred Strohmeier

Behavioural Requirements Overview

- Operation Model
  - System Operation
  - Operation Schema
  - Messages
  - Pre- and postconditions
- Protocol Model
  - User Requirements Notation
  - State diagrams
  - Sequence diagrams
- 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 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 (i.e. create an association instance);
  - Remove a link form an association (i.e. delete an association instance);
  - Send a message to an actor.

System Operation (2)

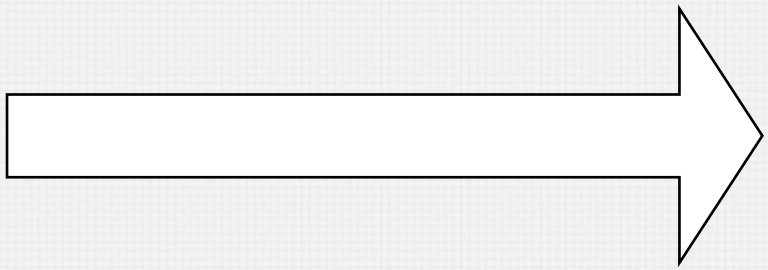
- The operations are specified by preconditions and postconditions, i.e. logical predicates.
- The precondition characterizes the valid initial states of the system when the operation is invoked.
  - If the precondition is not true, the effect of the operation is undefined.
- The result of the operation is expressed as a postcondition.
  - The postcondition describes the changes 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.



Precondition
must be true before

Postcondition
must be true after

Operation Schema (1)

- 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.
- Description: A concise natural language description of the purpose and effects of the operation.
- Use Cases: This clause provides cross-references to related use case(s).
- Scope: All classes and associations from the class model of the system 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.

Operation Schema (2)

- New: This clause provides declarations of names that designate objects to be "created" by the operation, i.e. the postcondition will state oclIsNew() for them.
- Aliases: This clause provides declarations of names that act like aliases. Aliases are name substitutions that are treated as an atom.
- Pre: The condition that must be met for the postcondition to be guaranteed. It is a boolean expression written in OCL, standing for a predicate.
- Post: The condition that will be met after the execution of the operation. It is a boolean expression written in OCL, standing for a predicate.

Operation Schema Example (1)

Message (type) declarations:

InsufficientFunds_e(); DispenseCash(amount: Money);

Operation: Bank::withdraw (acc: Account, request: Money);

Description: Request from an ATM of some amount to be taken from a given account. Cash is dispensed only if account has sufficient funds.

Scope: Account;

Messages: ATM::{InsufficientFunds_e; DispenseCash;};

Operation Schema Example (2)

Pre: true;
Post:
if (acc.balance@**pre** ≥ request) **then**
acc.balance = acc.balance@**pre** - request &
sender^dispenseCash(request)
else
sender^insufficientFunds_e()
endif;

Naming Conventions

- Generally names are case sensitive, and the following conventions should be used for better readability, and in order to avoid ambiguities:
  - Capital first letter for datatypes, e.g. String, classes, e.g., Person, and associations, e.g. Owns;
  - Lowercase first letter for data values and data constants, e.g. Color::blue, Gender::male, objects, e.g., john: Person, roles, e.g. wife, collections, e.g. company.employee, allEmployees, attributes, e.g. john.birthdate, and functions (methods), e.g. isUnique().
- Comments are written following two successive dashes (minus signs) and signify the rest of the line is a comment.

Tuples

- The Tuple notation of OCL is used for denoting the attribute values of an object and the values of a composite datatype. Tuples can also be used for denoting the actual parameters of a message.

  Tuple ::= “Tuple” “{" TupleItem ("," TupleItem)* “}"
  TupleItem ::= name [: TypeName] "=" value

- The name is the name of an object attribute, datatype field or message parameter.
- A tuple must always be complete, i.e. values must be defined for all attributes, fields, or parameters.

Tuple Examples

```
-- a tuple with a single component:
Tuple {amount = request}

-- a tuple with three components
Tuple {
    name : String = 'Josh Kronfield',
    birthdate : Date = Tuple {year = 1971, month = 6, day = 20},
    nationality : String = 'New Zealander'}

Context: p: Person, c: Company
p.all = Tuple { name : String = 'Josh Kronfield',
    birthdate = Tuple {year = 1971, month = 6, day = 20},
    nationality : String = 'New Zealander'}
c.all = Tuple { name = 'Microsoft',
    headquarters = 'Richmond',
    budget = 50.0E9}
```

Messages Reminder

- Messages are model elements. They have parameters. 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, i.e. we deal only with what's called Signal(s) in the UML.

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, referred to by **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 was sent to the actor (e.g. as a result of an operation.)

Message Declarations

MessageDeclaration ::=
  MessageName "(" ParameterList ")"
ParameterList ::= Parameter ("," Parameter)*
Parameter ::= EntityDeclaration

- Note that the "sender" is not explicitly declared. Indeed, any actor is allowed as a sender, and it cannot be constrained to a subtype. The sender instance is set implicitly when a message is delivered.

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.

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));
```

Operation Schema: Operation

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

- The SystemClassName is the server of the operation. It is also the context of the schema. self therefore refers to an instance of this class.
- 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

- Scope lists all conceptual system state from the Concept Model that the system operation accesses or modifies
- NameList is a list of class and association class names.
  - Note that "simple" associations are not listed, only association classes.

- 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 declaration designates an object or a collection of objects to be "created" by the operation, i.e. the postcondition will state **oclIsNew**() for it, or its elements, respectively.
  - Each name in an ObjectDeclaration declares a distinct object.
- Examples

  New:

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

Operation Schema: Aliases

"**Aliases**" ":" ItemList
ItemList ::= (Alias ";")*
Alias ::= EntityDeclaration ”=" Expression

- An alias is a name substitution that overrides precedence rules, i.e., treated as an atom, and not just as a macro expansion.
  - Everything declared in the Aliases clause is local to the schema.
- Examples

  Aliases:

  art: Person = **self**.person→**any**(p | p.firstName = "arthur");
  b: Integer = art.account.balance;

New and Aliases and Scope

- If a ClassName is used in an ObjectDeclaration or ObjectCollectionDeclaration of a New clause or in an EntityDeclaration of an Aliases clause, it must be in the scope of the schema, i.e. declared in the Scope clause.
- Similarly, if a ClassName is used in an Expression of an Aliases clause, it must be in the scope of the schema.

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;};

Operation Schema: Pre and Post (1)

"**Pre**:" Condition ";"
"**Post**:" Condition ";"
Condition ::=
  BooleanExpression ("&" BooleanExpression)*

- Condition is a boolean expression, the meaning of the sign “&” being that of a logical and.
- Expressions are written in OCL
- If the precondition is true, the operation terminates and the postcondition is true after the execution of the operation.
- The pre- and postcondition assertions constitute the contract model of the operation. If the precondition is met, then the operation will meet the postcondition, but if the precondition is not met, then nothing is guaranteed, i.e., the effect of the operation is undefined.

Operation Schema: Pre and Post (2)

- An empty precondition can be expressed by the constant condition true
- Pre and Post clauses refer only to entities declared in the Aliases and New clauses, to parameters of the operation, to self, to sender, or to entities navigated to from any of the previous ones.
- A Condition in a Post clause can use the **@pre** suffix to denote the value of a property before the triggering of the system operation.
- Note that the Pre and Post clause may make use of (OCL) functions, but it is not possible to refer to another Operation Schema within the postcondition of a schema.

Messages in Postconditions (1)

- Messages that are output by the system during the execution of an operation are specified in the respective schema by stating:
  - the type of the message and the destination actor type;
  - the condition(s) under which the message instance is sent;
  - the actual parameters of the message instance;
  - the destination actor instance;
  - asserting by special syntax that the message instance was sent to the actor instance.

Messages in Postconditions (2)

- Stating in a postcondition that a message instance was sent to some actor instance is done as follows:
  - destination actor followed by a **^** sign
  - followed by a message (type) name
  - followed within parentheses by its actual parameters.
- Note that the notation is such that a message instance cannot be sent twice (since there is no name for referring to it).

  ActorExpression “^” MessageName "(" ActualParameterList ")“
  ActualParameterList ::= [Actual (“,” Actual)*]
  Actual ::= ValueExpression I “**?**” “:” TypeExpression
- ActorExpression must denotate an actor instance.
- The question mark denotates an actual with unknown value or with a value of no importance.

Messages in Postconditions Example

- The examples could be part of a withdrawal operation performed by an ATM of a bank.

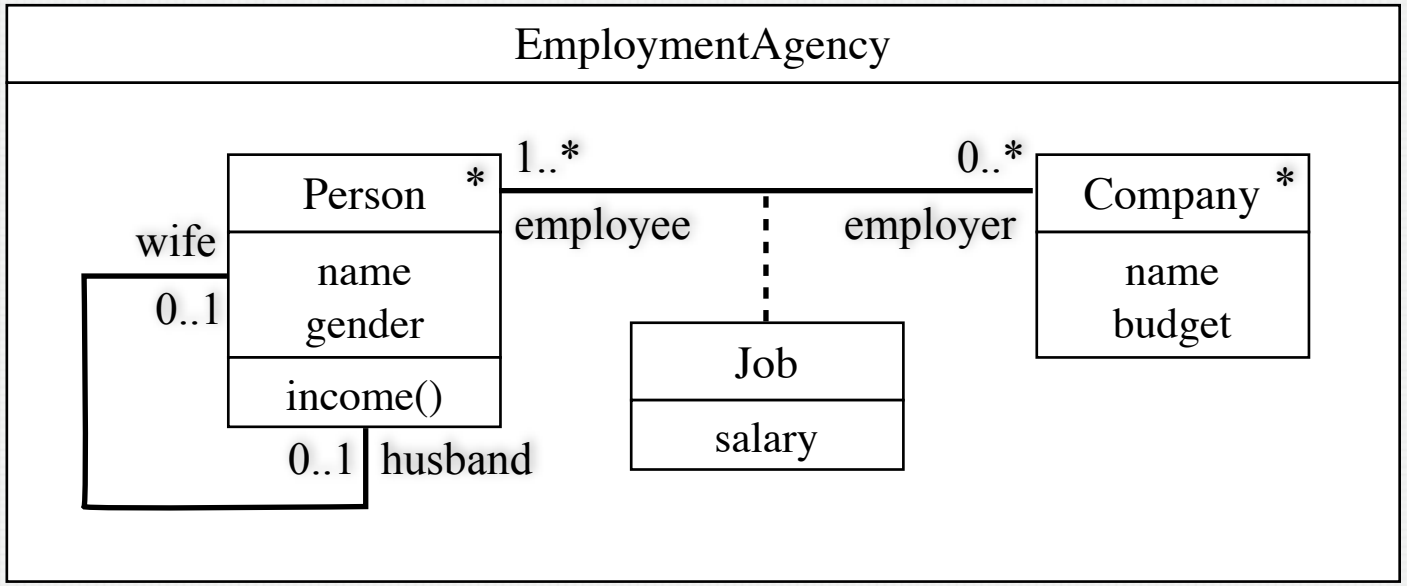
Message (type) declarations:
    InsufficientFunds_e ();
DispenseCash (amount: Money);
Report (t: Transaction);
type Transaction **is** TupleType
        {amount: Money, timestamp: Date, d: Direction};

Messages: ATM::{InsufficientFunds_e; DispenseCash; Report;};
Post:
sender^dispenseCash (request) &
sender^report (Tuple {amount = request,
    timestamp = Tuple {year = 2000, month = 2, day =14},
        d = Direction::debit});

Messages in Postconditions (3)

- It is possible to access and make assertions about all the instances of a given message type sent to an actor instance.
  - The notation is:
    ActorExpression “^^” MessageName "(" ActualParameterList ")“
- It denotates a Sequence of message instances of type MessageName.
- For example, if observer is an actor instance, and Update is a message type with two Integer parameters, it is possible to write:
  observer^^update(?: Integer, ?: Integer)
  - The type of the result is Sequence(Update)

Example UML Class Diagram



Operation Schema Example

Operation: EmploymentAgency::jobFilled
(worker: Person, comp: Company, amount: Money);
Description: Creates a job for a given person and company, where company must have a budget smaller than or equal to 10 million;
Scope: Person; Company; Job;
New: researchJob: Job;
Post:

researchJob.**oclIsNew**() &
researchJob.salary = amount &
researchJob.employee = worker &
researchJob.employer = comp;

Operation Schema Example (2)

message (type) declaration

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

Operation: Bank :: checkAssets ();
Description: Request issued by a manager. Lists the balances of all accounts, together with the owner's names. Sends multiple messages, one for each account!
Scope: Account; Customer; Owns;
Messages: Manager :: {Asset;};
Post:

```
self.account→forAll(a | sender^^asset(? : String, ? : AccNumber, ? : Money)
    →one (m : OclMessage |
        m.name = a.customer.name and m.number = a.number and
        m.amount = a.balance))
```

Operation Schema Example (3)

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

Operation: Bank :: checkAssets ();
Description: Request issued by a manager. Lists the balances of all accounts, together with the owner's names. Sends one message containing all assets.
Scope: Account; Customer; Owns;
Messages: Manager :: {CurrentAssets;};
Post: **let** seq: Sequence (LineItem) **in**
self.account→**forAll** (a | seq→**includes**
(**Tuple** {name = a.customer.name, number = a.number,
amount = a.balance})) **and sender^**currentAssets (seq)
endlet;

Parameterized Predicates

- A parameterized predicate is a boolean expression
- Useful to
  - Improve readability of post clauses
  - Reuse commonly occurring effects

Predicate: SystemName :: PredicateName ([ParameterList]);
Aliases: if needed.
Body: Condition;

Parameterized Predicates Example (1)

- Message type declaration

  type Line **is TupleType** {number: AccNumber, amount: Money};
  Statement(content: Set(Line));

- Predicate declaration

  Predicate: Bank :: StatementHasBeenSent(Customer c);
  Body: **let** currentStatement: Set(Line) **in**
  c.myAccounts→**forAll** (a |
  currentStatement→**includes**
  (**Tuple** {number = a.number,
  amount = a.balance})) **and**
  c.client^statement(currentStatement);

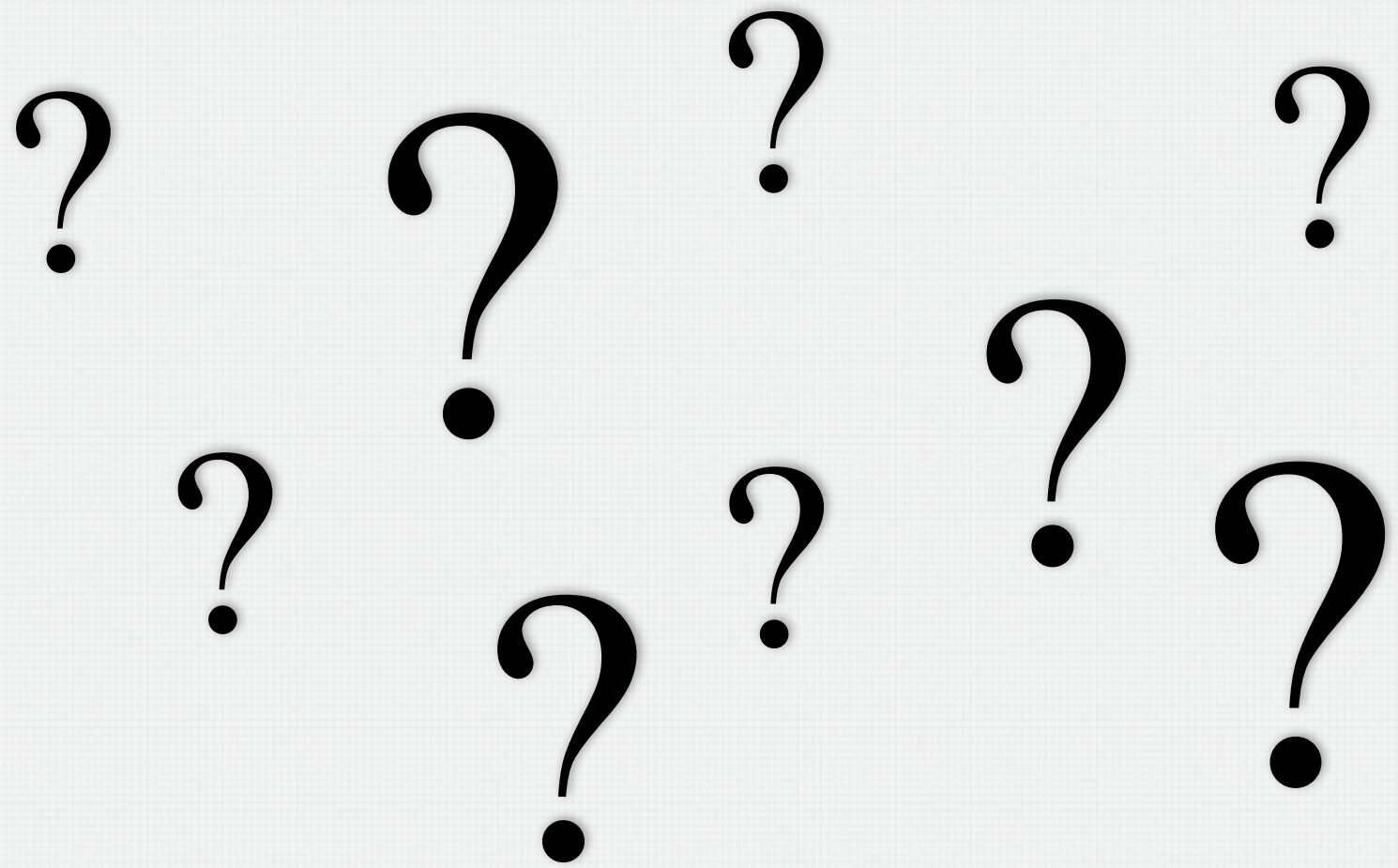
Parameterized Predicates Example (2)

- StatementHasBeenSent can now be used in several operation schemas

Operation: Bank :: MonthlyStatement(Customer c);
Post: self.StatementHasBeenSent(c);

Operation: Bank :: CloseAccount(Account a);
Post: **self**.account→**excludes**(a) **and** **self**.StatementHasBeenSent(a.owner);

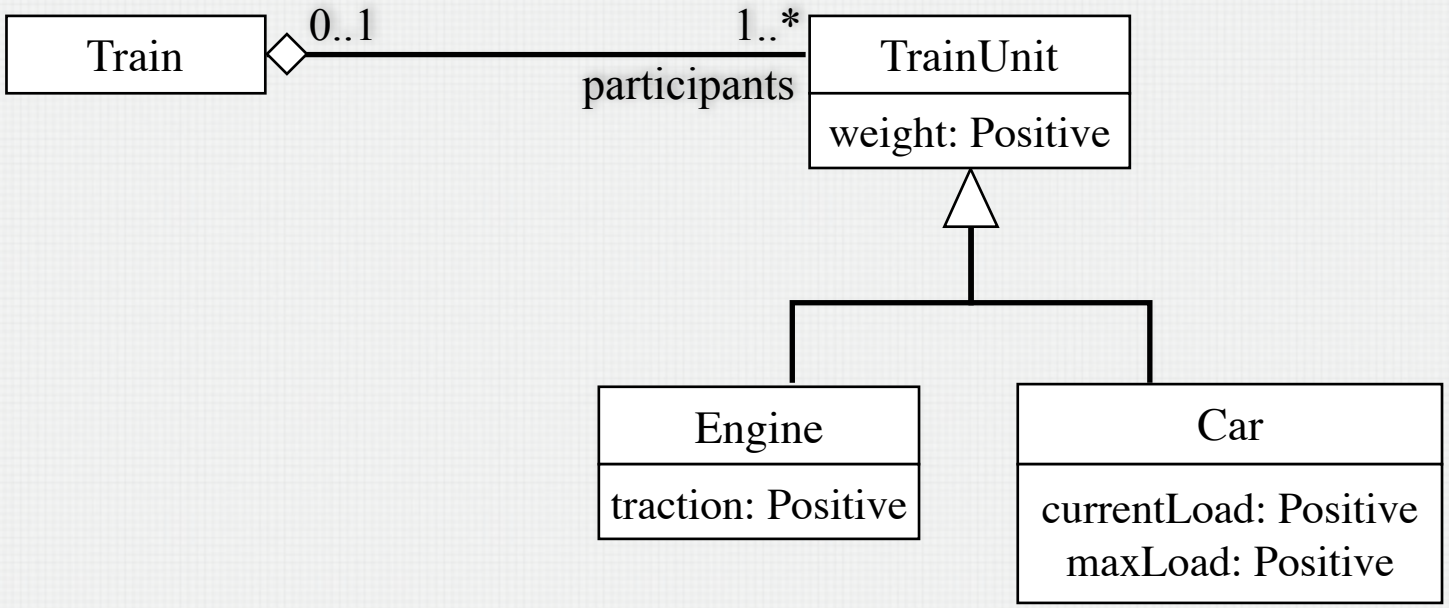
Questions?



Pre- and Post-condition Question

- Are the following clauses appropriate for preconditions and/or postconditions?
  1. No condition.
  2. The user actor has not yet pushed any button.
  3. The system invariant A, e.g. ”A worker is at least fourteen years old.”, is fulfilled.
  4. The protocol as defined by the Protocol Model is obeyed.
  5. First an unfilled order is sent to the packager, then the billing department is informed about the order, and finally the procurement department is asked to stock up the delivered good.

Train Depot Continued



Train Depot Questions

- 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?

- You can use the functions defined in the OCL lecture exercise, i.e. totalTraction, totalWeight, totalLoad.

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: Exit 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;

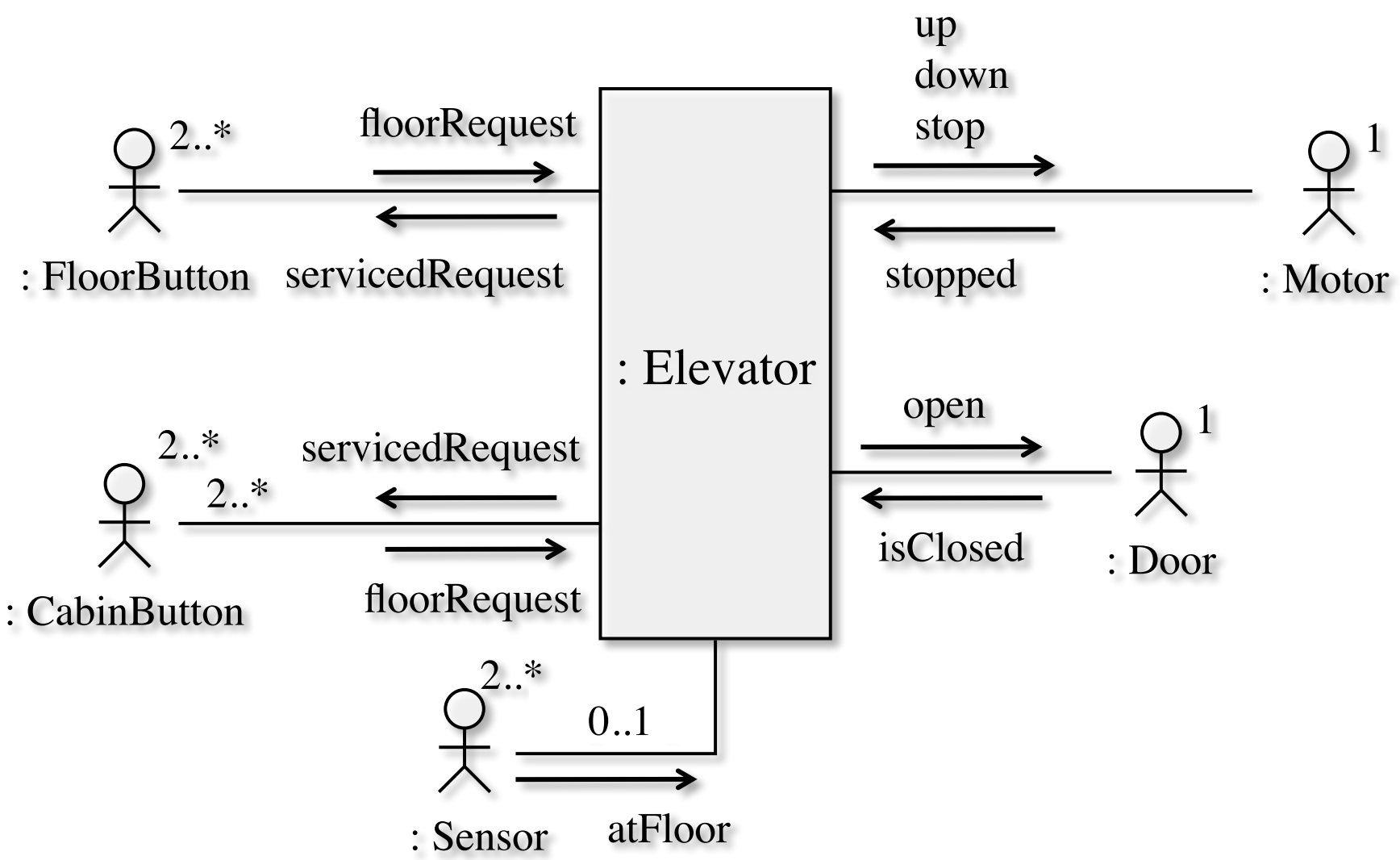
(3-5)lla.1 *System schedules the request*; use case continues at the same step.

4a. System determines that another request has priority.

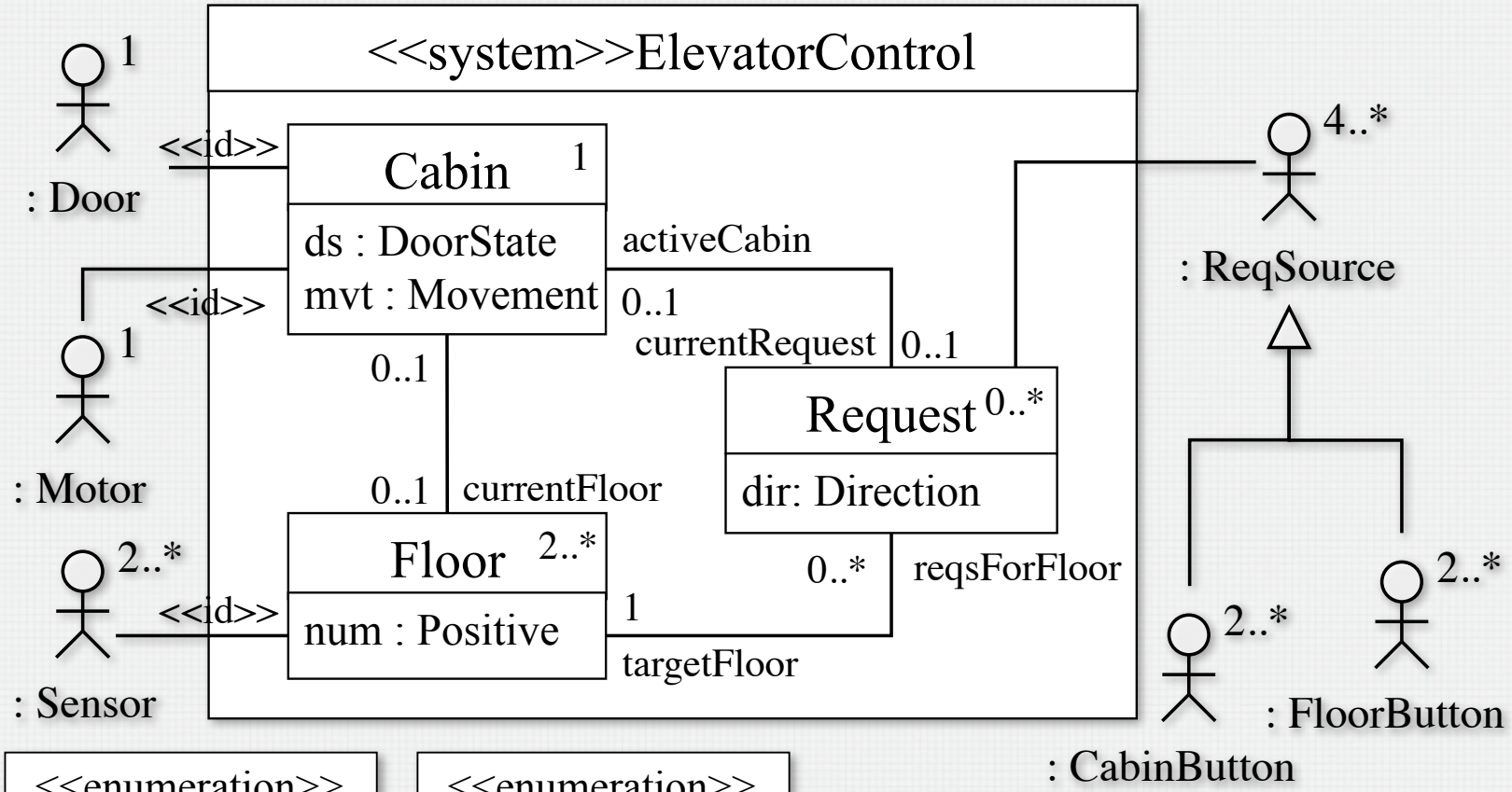
4a.1. *System schedules the request*; use case continues at step 4.

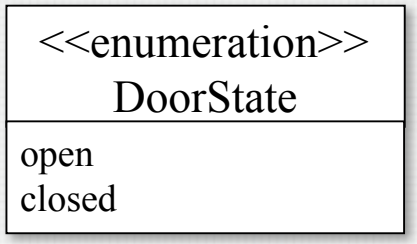
9a. System determines that there are additional requests pending; use case continues at step 3.

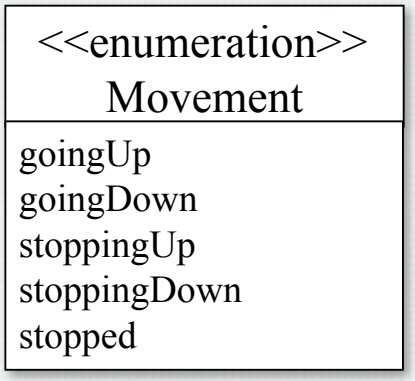
Elevator Environment Model

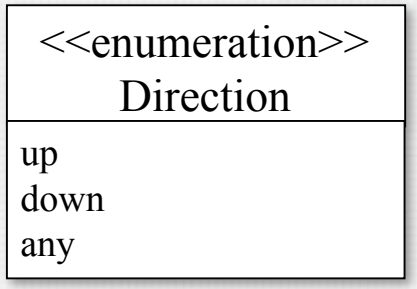


Elevator Concept Model









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).