Course "Softwaretechnik"
Book Chapter 5
Analysis: Dynamic Modeling

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit

Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Dynamic modeling
  - Sequence diagrams
  - State diagrams
- Using dynamic modeling for the design of user interfaces

And then:

- Requirements analysis document template
  - esp. non-functional req's.
- Requirements analysis model validation

How do you find classes?

- In previous lectures we have already established
  the following sources
  - Application domain analysis: Talk to client to identify abstractions
  - Application of general world knowledge and intuition
  - Scenarios
  - Use Cases
  - Textual analysis of problem statement (Abbott)

- Today we show how to identify classes and their operations
  and attributes from dynamic models
  - Activity lines in sequence diagrams identify candidates for classes
  - Messages in sequence diagrams may turn up as operations in classes
  - Actions and activities in statecharts or activity diagrams are candidates for public operations in classes

Dynamic Modeling with UML

- Diagrams for dynamic modeling
  - Interaction diagrams describe behavior examples
  - Statecharts specify a single object

- Interaction diagrams
  - Sequence diagram:
    - Dynamic behavior of a set of objects in time sequence
    - Good for real-time specifications and complex scenarios
  - Collaboration diagram:
    - Different but roughly equivalent diagram type, rare
- Statechart diagram:
  - A state machine of the responses of an object to the receipt of outside stimuli (Events)
  - Activity Diagram: A special type of statechart diagram, where all states are action states
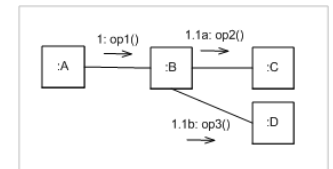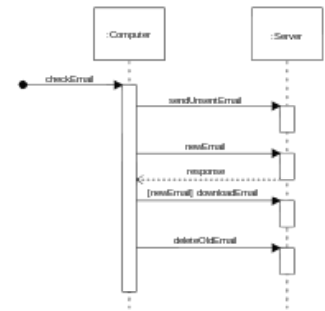
- Definition of dynamic model:
  - A collection of multiple behavior diagrams (such as statechart, activity, and sequence diagrams),
    - usually at least one regarding each important class
    with important dynamic behavior

- Purpose:
  - Understand behavioral requirements
  - Detect and supply methods for the object model

- How do we do this?
  - Start with use case or scenario, plus identification of classes
  - Model interaction between objects ➔ sequence diagram
  - Model dynamic behavior of a single object ➔ statechart diagram

Heuristics for sequence diagrams

- A typical layout:
  - 1st column: The *actor* who initiated the use case
  - 2nd column: A *boundary object* (perhaps missing in analysis)
  - 3rd column: The *control object* managing the rest of the use case
  - further columns: the other participating objects

- Creation:
  - Control objects are often created at the initiation of a use case
  - Additional boundary objects may be created by control objects

- Access:
  - Entity objects are accessed by control and boundary objects
  - Entity objects should never call boundary or control objects:
    - This makes it easier to share entity objects across use cases and
    - makes entity objects resilient against technology-induced changes in boundary objects

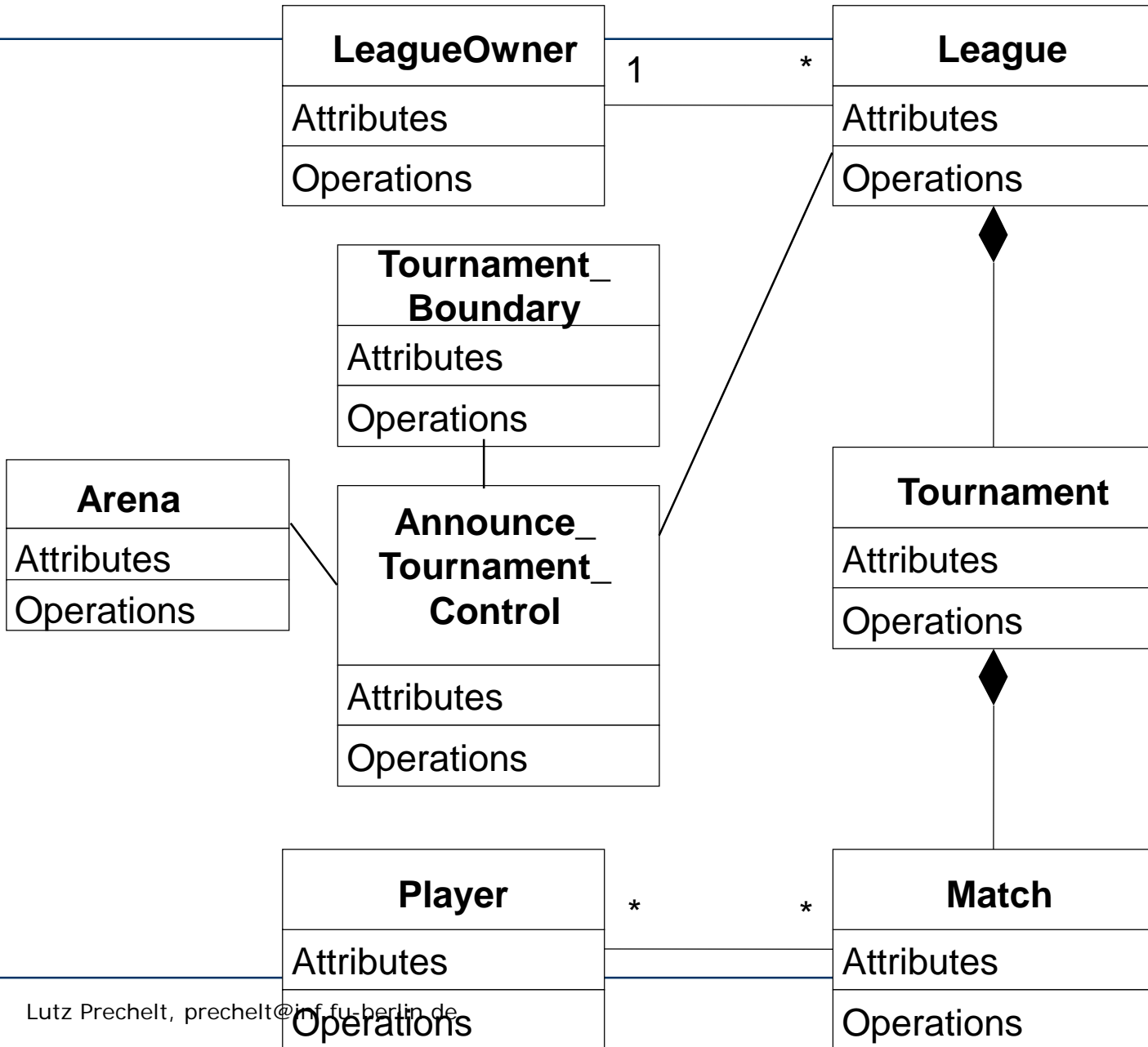# An ARENA sequence diagram: create tournament
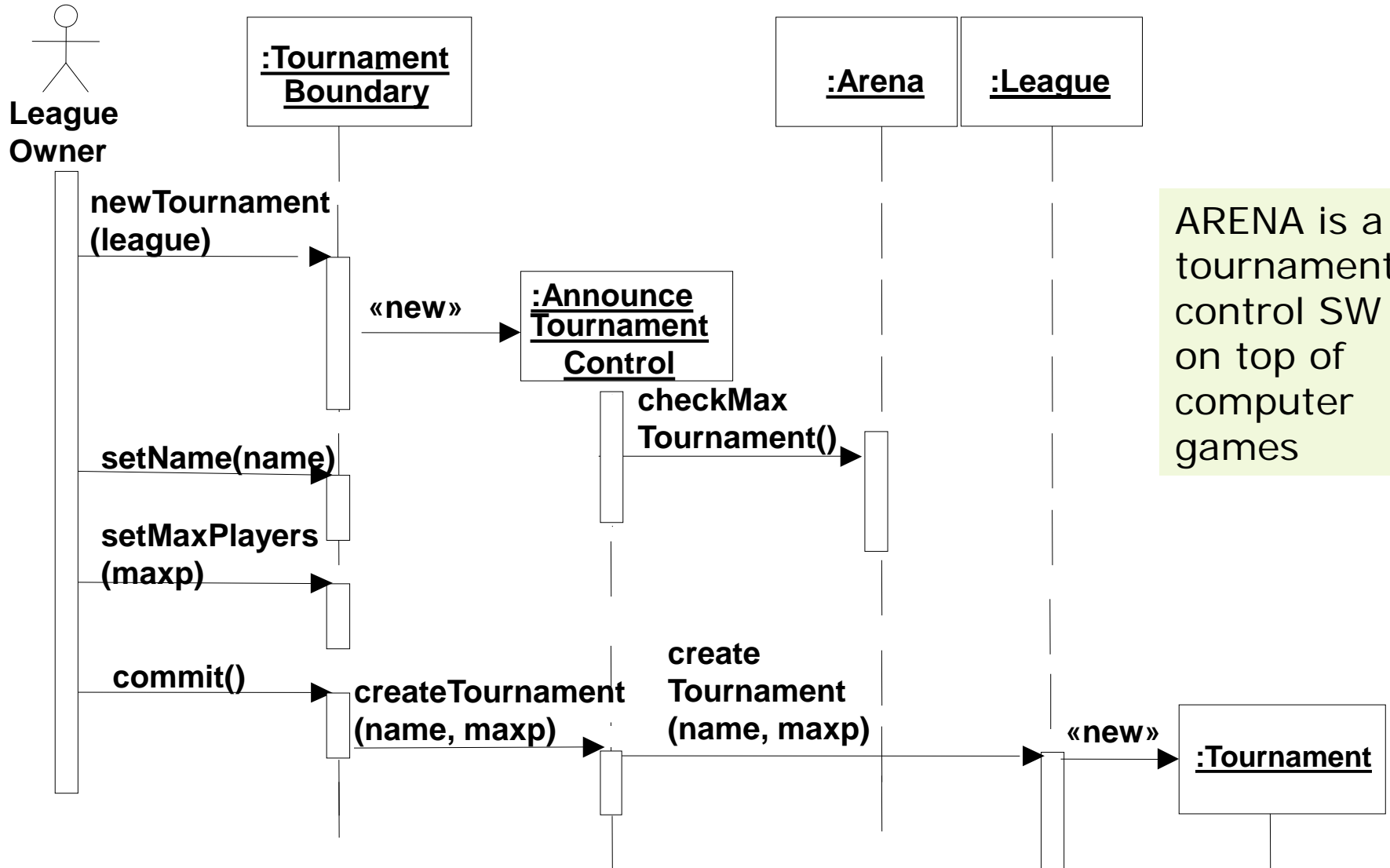
ARENA is a tournament control SW on top of computer games

ARENA's Object Model (before)
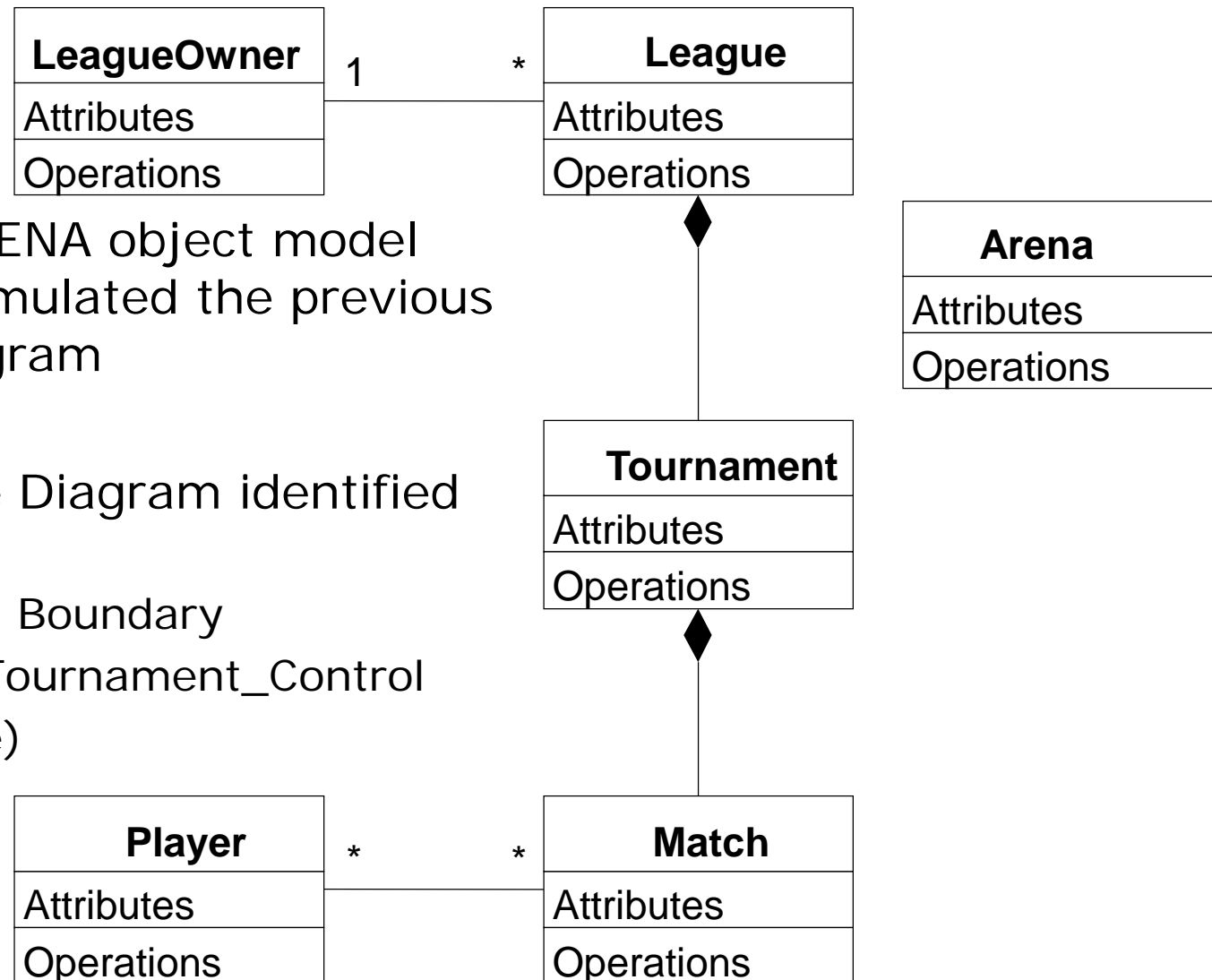
- This is the ARENA object model before we formulated the previous sequence diagram

- The Sequence Diagram identified new classes
  - Tournament Boundary
  - Announce_Tournament_Control

  (see next slide)

ARENA's Object Model (new)

LeagueOwner
Attributes
Operations

League
Attributes
Operations

1   *

Tournament_ Boundary
Attributes
Operations

Arena
Attributes
Operations

Announce_ Tournament_ Control
Attributes
Operations

Tournament
Attributes
Operations

Player
Attributes
Operations

Match
Attributes
Operations

*   *

Freie Universität Berlin

- The Sequence Diagram also supplied us several new events
  - newTournament(league)
  - setName(name)
  - setMaxPlayers(maxp)
  - commit()
  - checkMaxTournaments()
  - createTournament(name, maxp)

Who "owns" these events?

- For each object that receives an event there is a public operation in the associated class
  - The name of the operation is usually the name of the event

Example from the sequence diagram

League Owner

:Tournament Boundary

:Arena

:League

createTournament is a (public) operation in Announce_Tournament_Control

and so:

newTournament (league)

«new»

:Announce Tournament Control

checkMax Tournament()

setName(name)

setMaxPlayers (maxp)

commit()

createTournament (name, maxp)

create Tournament (name, maxp)

«new»

:Tournament

What else
can we get out of sequence diagrams?
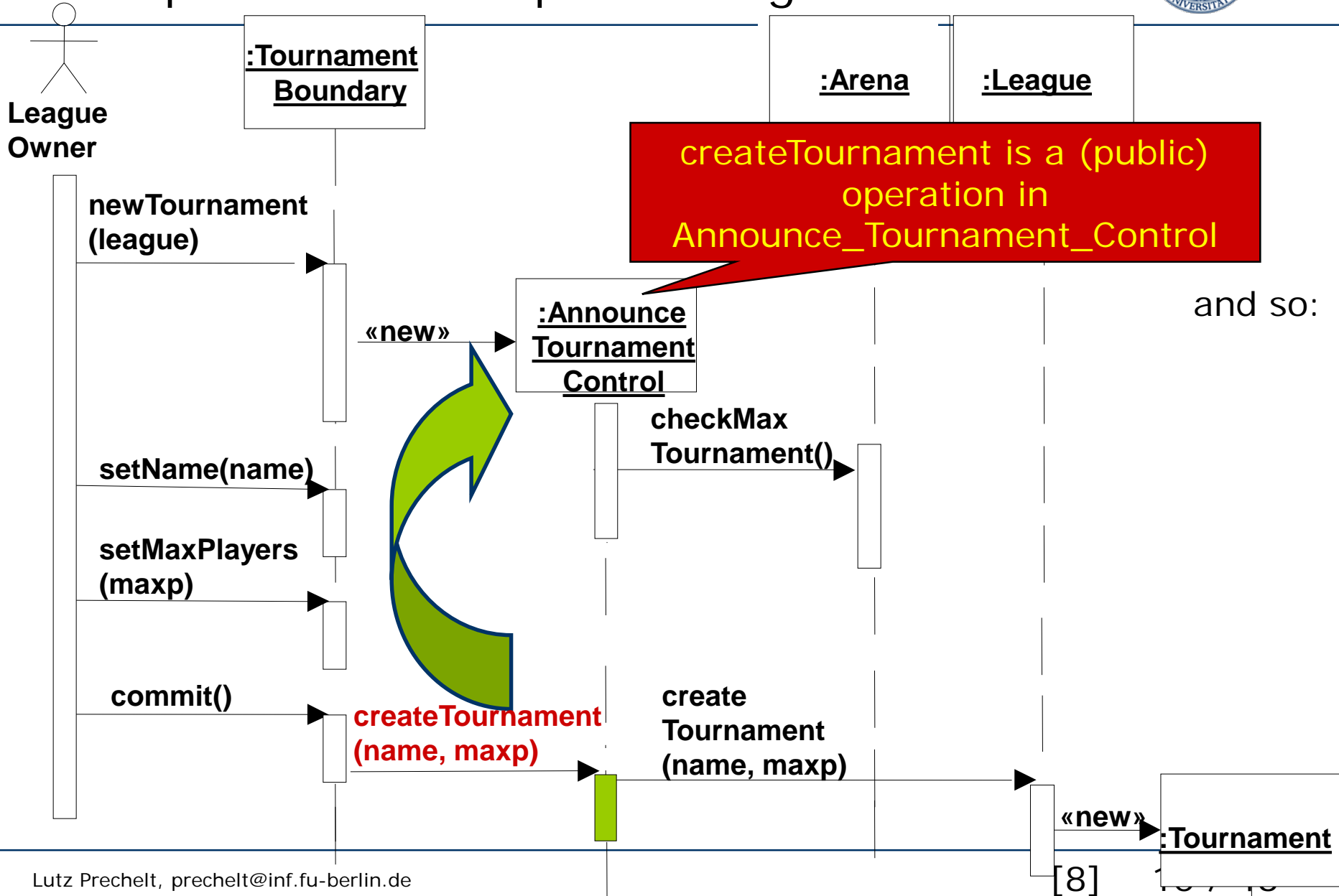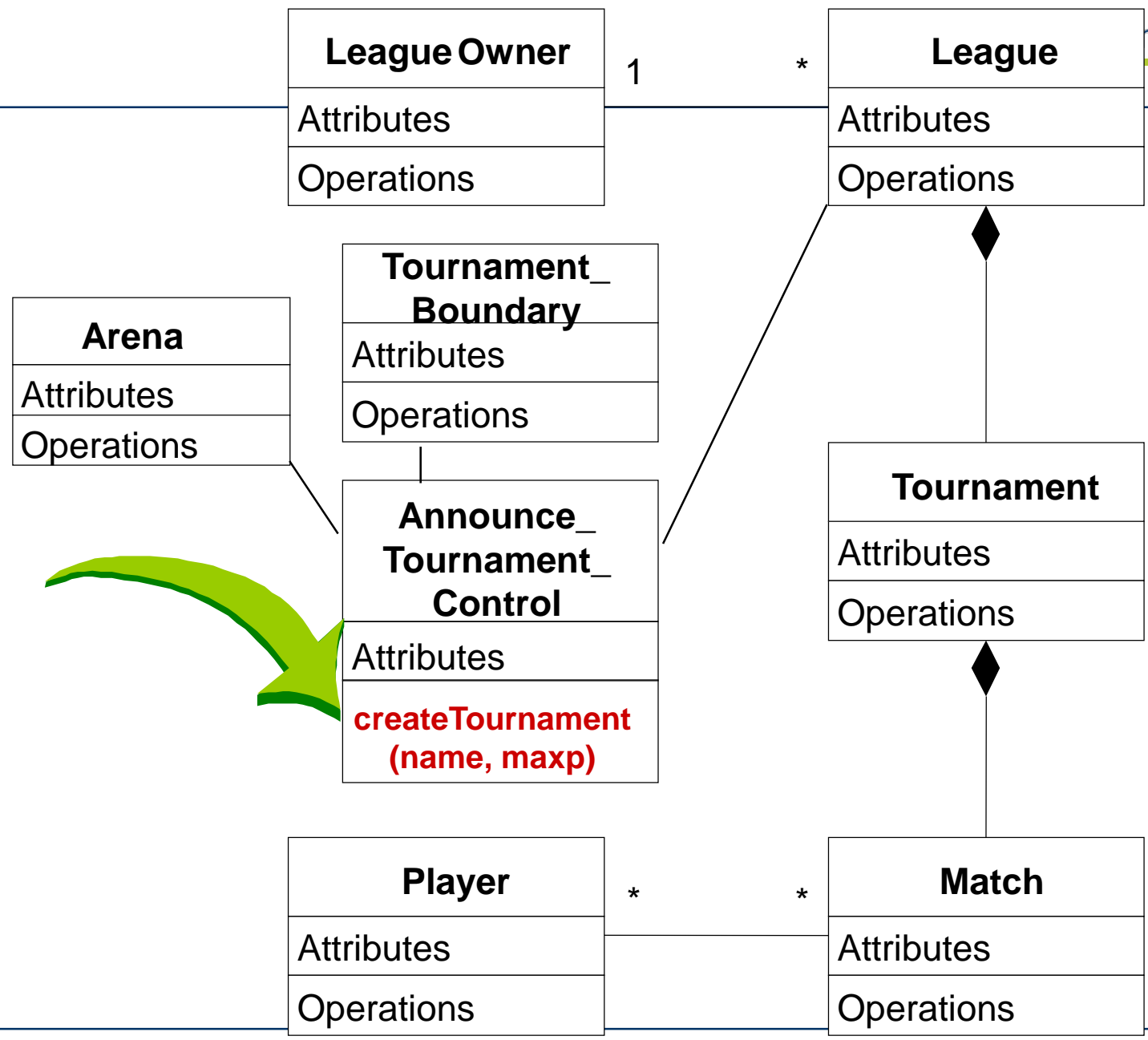
- Sequence diagrams are derived from use cases
  - We therefore see the structure of the use cases

- The structure of the sequence diagram helps us to determine how decentralized the system should be

- We distinguish two basic structures of sequence diagrams (Ivar Jacobson):
  - Fork-style diagrams (central control)
  - Stair-style diagrams (distributed control)

  (see next slides)

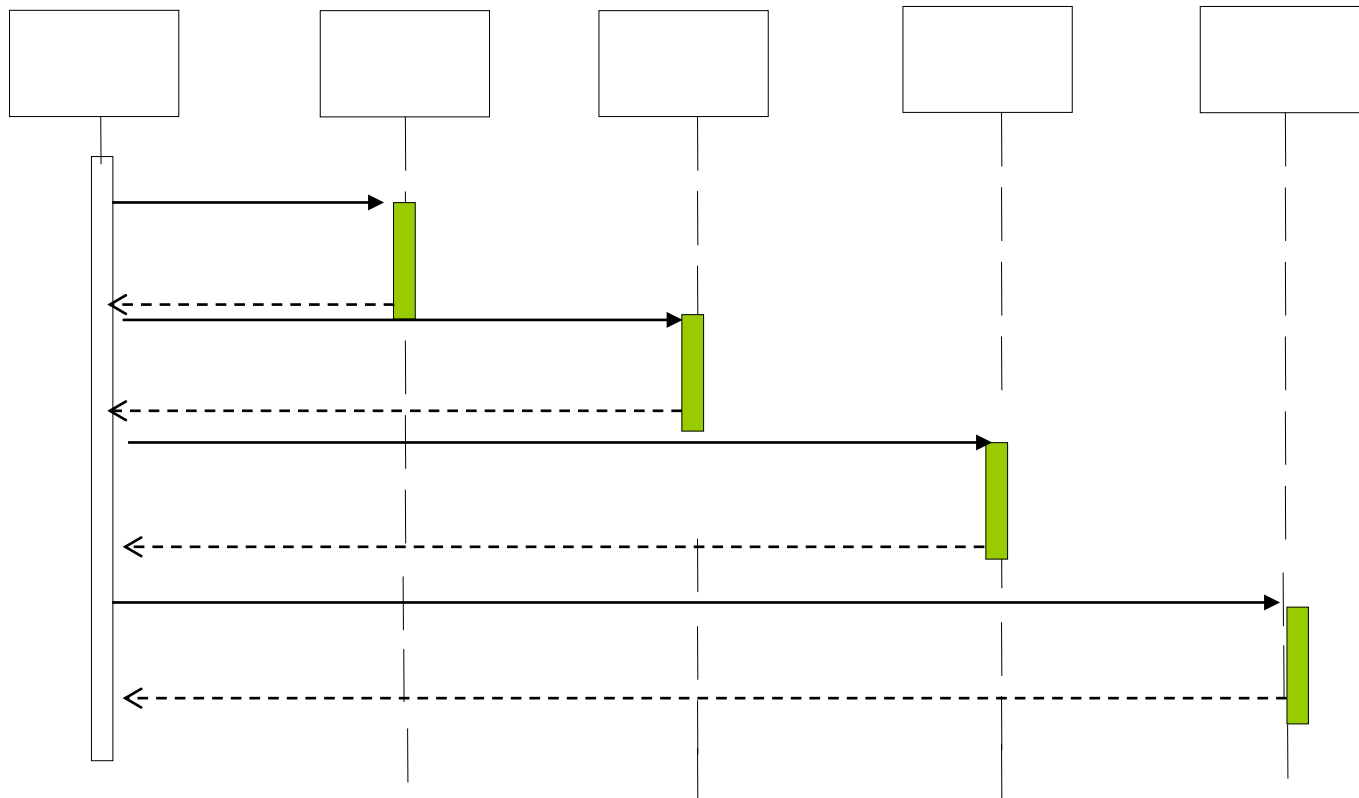btw:

Actor   Control   Boundary   Entity

Central control: Fork diagram

- Much of the dynamic behavior is placed in a single object, usually the control object
  - It knows all the other objects and uses them for direct questions and commands

Decentralized control: Stair diagram

- The dynamic behavior is distributed.
  Each object delegates some responsibility to other objects
  - Each object knows only a few of the other objects and knows which objects can help with a specific behavior

Fork or Stair?
(Design thinking, not analysis thinking!)
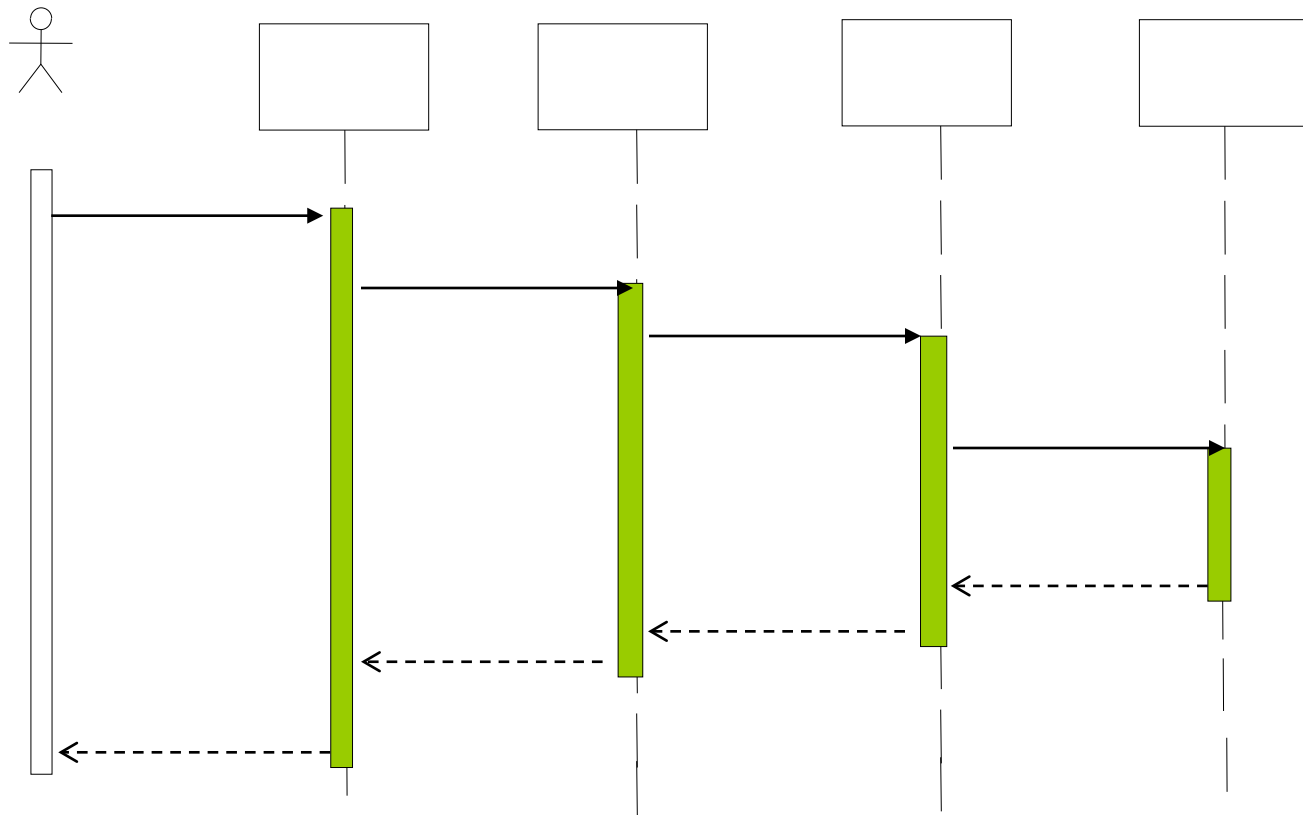
Which of these diagram types should be chosen?

- Object-oriented fans claim the stair structure is better
  - "The more the responsibility is spread out, the better"
- However, this is not always true
  - One should usually have a "suitable" mix of both forms
  - (see also design patterns "Mediator", "Façade")

Considerations:

- Decentralized control structure is locally simple:
  - Objects do not get overly complex
  - Responsibilities are easy to understand
- Centralized control structure better supports change:
  - The operations can easily change order
  - New operations can easily be inserted for new requirements

UML statechart diagram notation

Event trigger With parameters

State1

do/Activity
entry/action2
exit/action3
event2/action4

Event1(attr) [condition]/action1
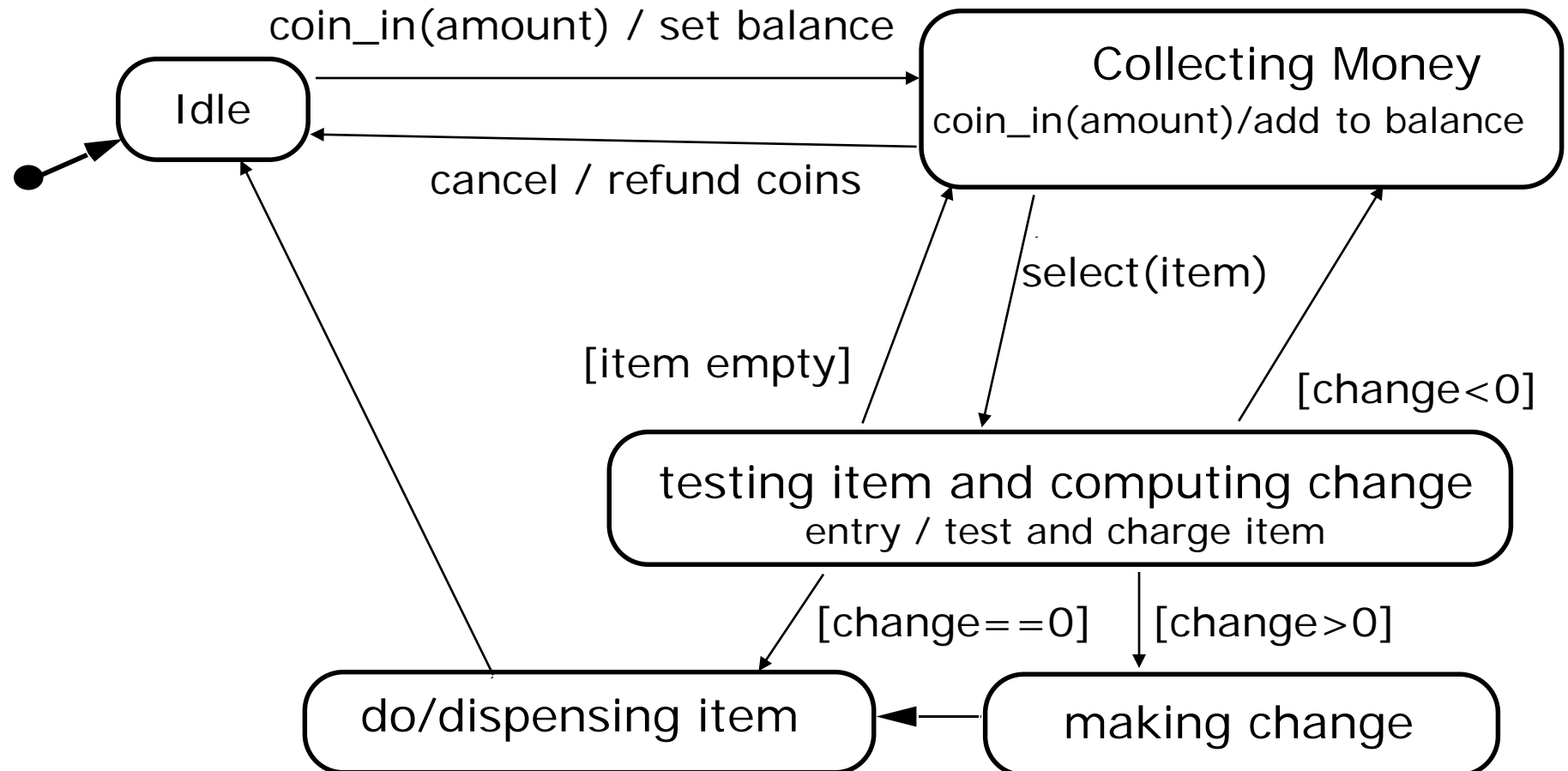
State2

Guard condition
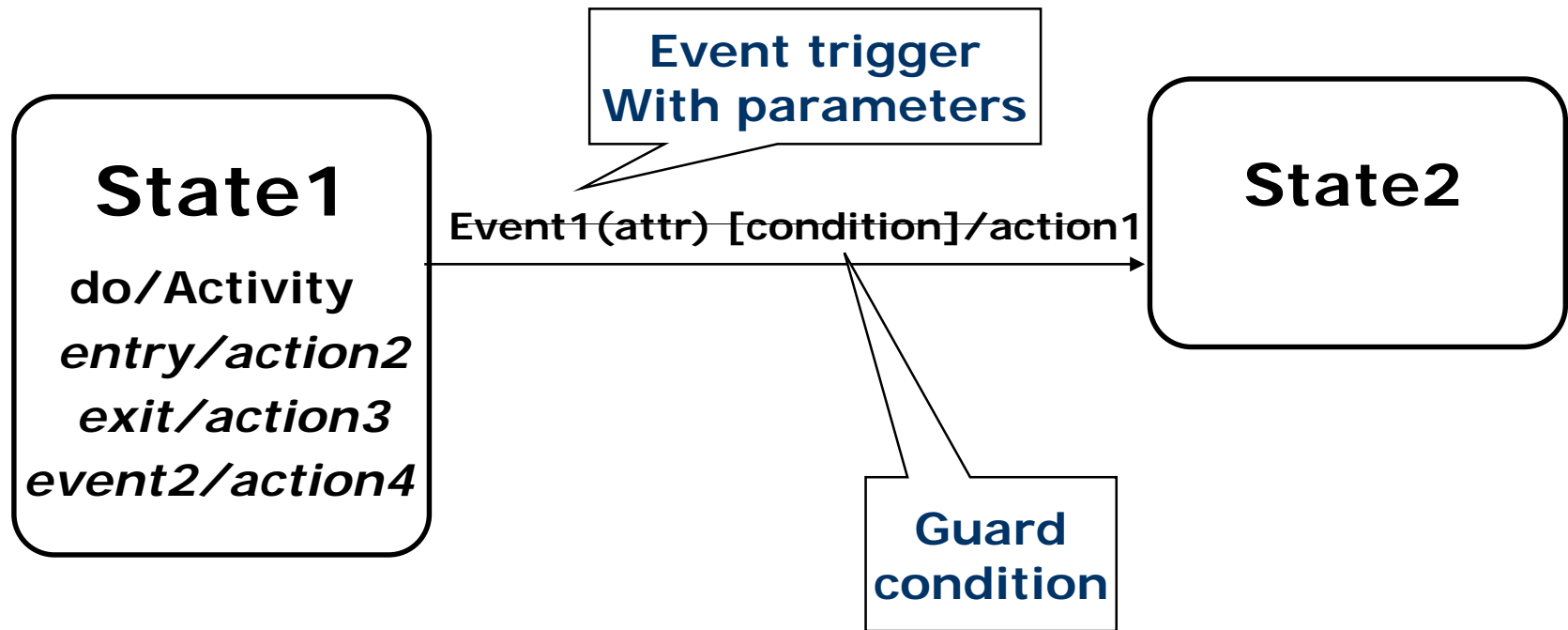
- Notation based on work by Harel
  - UML adds a few object-oriented modifications
- A UML statechart diagram can be mapped into a finite state machine

Statechart diagrams

- Graph whose nodes are states and whose directed arcs are transitions labeled by event names
- We distinguish between two types of elements in statecharts:
  - Activity: Compound operation that takes time to complete
    - associated with states
    - (in UML:) can be described by its own Activity diagram
  - Action: Elementary, "instantaneous" operation
    - associated with events
    - associated with states (reduces drawing complexity): Entry, Exit, Internal Action
    - (May in fact have structure, too, but the present statechart ignores it)
- A statechart diagram relates events and states for one class
  - An object model with a set of objects can have a corresponding set of state diagrams

State

- An abstraction of the attribute values of a later implementation class
  - A state describes a certain set of configurations of attribute values in an object (instance)

- Basically an "appropriate" equivalence class of attribute value configurations that need not be distinguished
  - example: the state *"in_active_region"* may mean
    - x in 0..150 & y in 100...150
  - What is appropriate depends on our current goal

- State has duration

Example of a statechart diagram

Idle

coin_in(amount) / set balance

Collecting Money
coin_in(amount)/add to balance

cancel / refund coins

select(item)

[item empty]

[change<0]

testing item and computing change
entry / test and charge item

[change==0]   [change>0]

do/dispensing item

making change

Note some states do not have (nor need) a name,
but need further details

Nested state diagram

- Activities in states are composite items denoting other lower-level state diagrams
  - which may be spelled out or not
- A lower-level state diagram corresponds to lower-level states and events that are invisible at the higher level
- The set of substates in such a nested state diagram denotes a *composite state*
  - enclosed by a large rounded box, also called region
- Transitions from other states to the composite state enter the initial substate of the composite state
  - Much like the entry point of a subroutine
- Transitions to other states from a composite state are inherited by all the substates (state inheritance)
  - Much like a runtime exception whose occurrence can terminate a method at many points
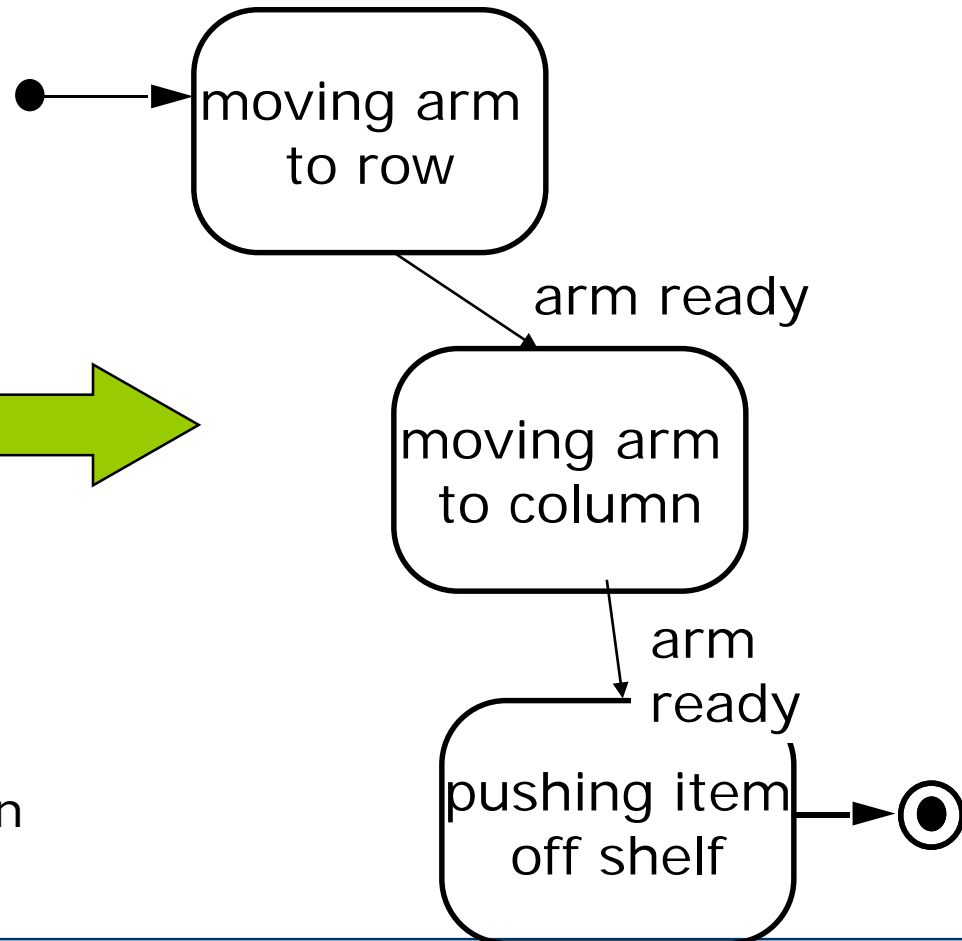
Example
of a nested statechart diagram

'Dispense item' as
an atomic activity:

'Dispense item' as
a composite activity:
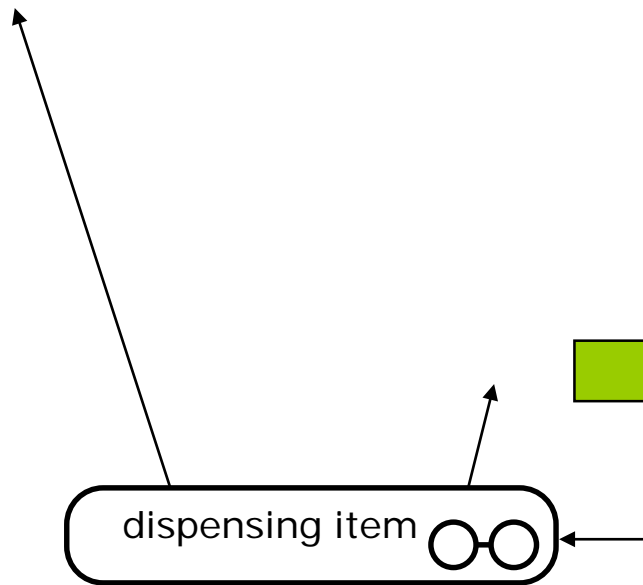
The little glasses indicate that
there are sub-activities hidden in
this composite activity
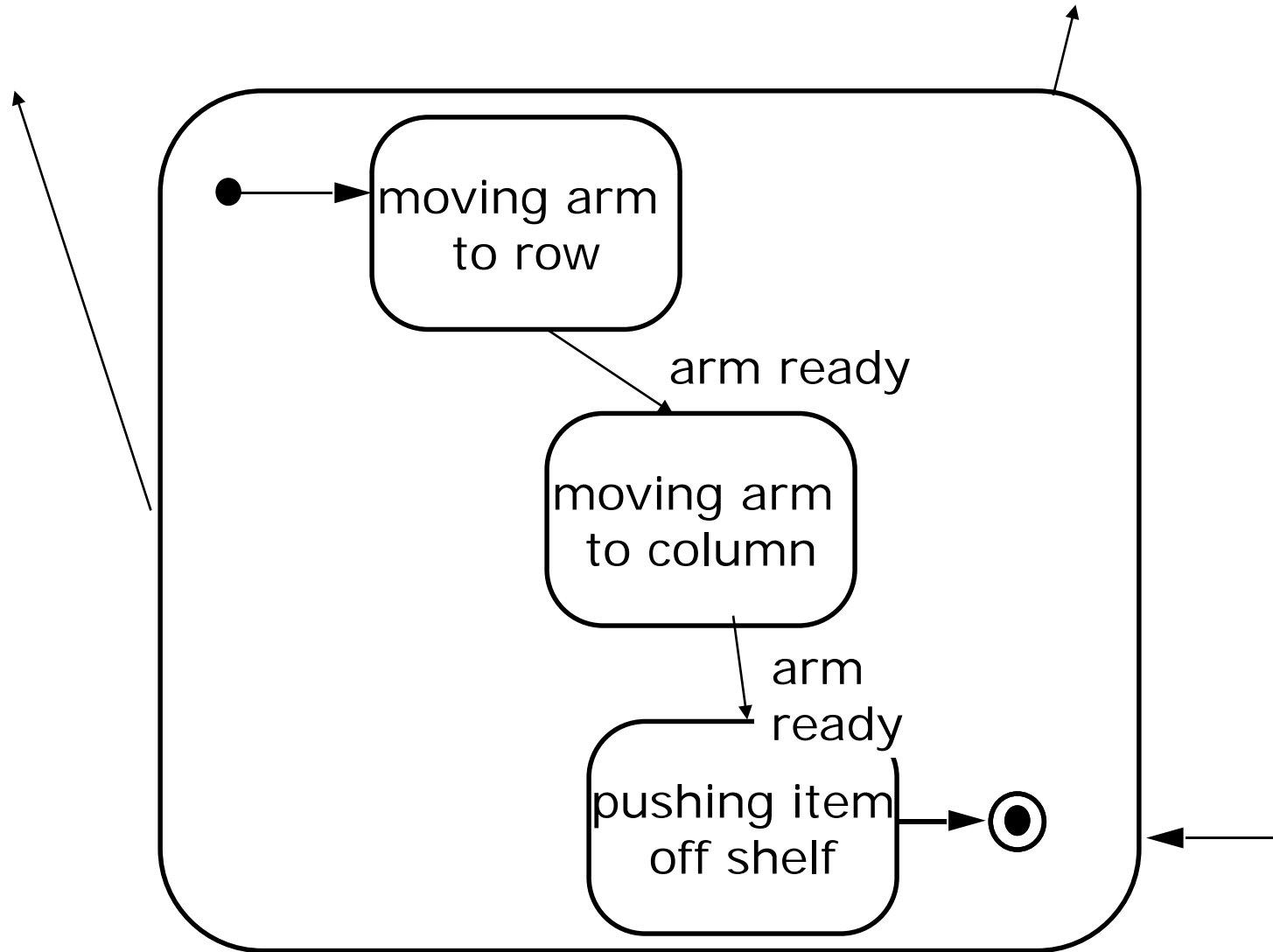
Two types of concurrency:

- 1. System concurrency (across objects)
  - State of overall system as the aggregation of state machines, one for each object
  - Note that one state diagram (for a class) may result in many state machines (one per instance of the class)
  - Each state machine is conceptually executing concurrently with all others

- 2. Object concurrency (within objects)
  - An object can be partitioned into subsets of states (attributes and links) such that each subset has its own subdiagram
  - The state of the object consists of a set of states: one state from each subdiagram
  - State diagrams (or composite states) are divided into *regions* by dotted lines

Example
of concurrency within an object

Splitting control

Synchronization

Setting Up

Ready

Dispensing Cash

Cash taken

Ejecting Card

Card taken

Ready to reset
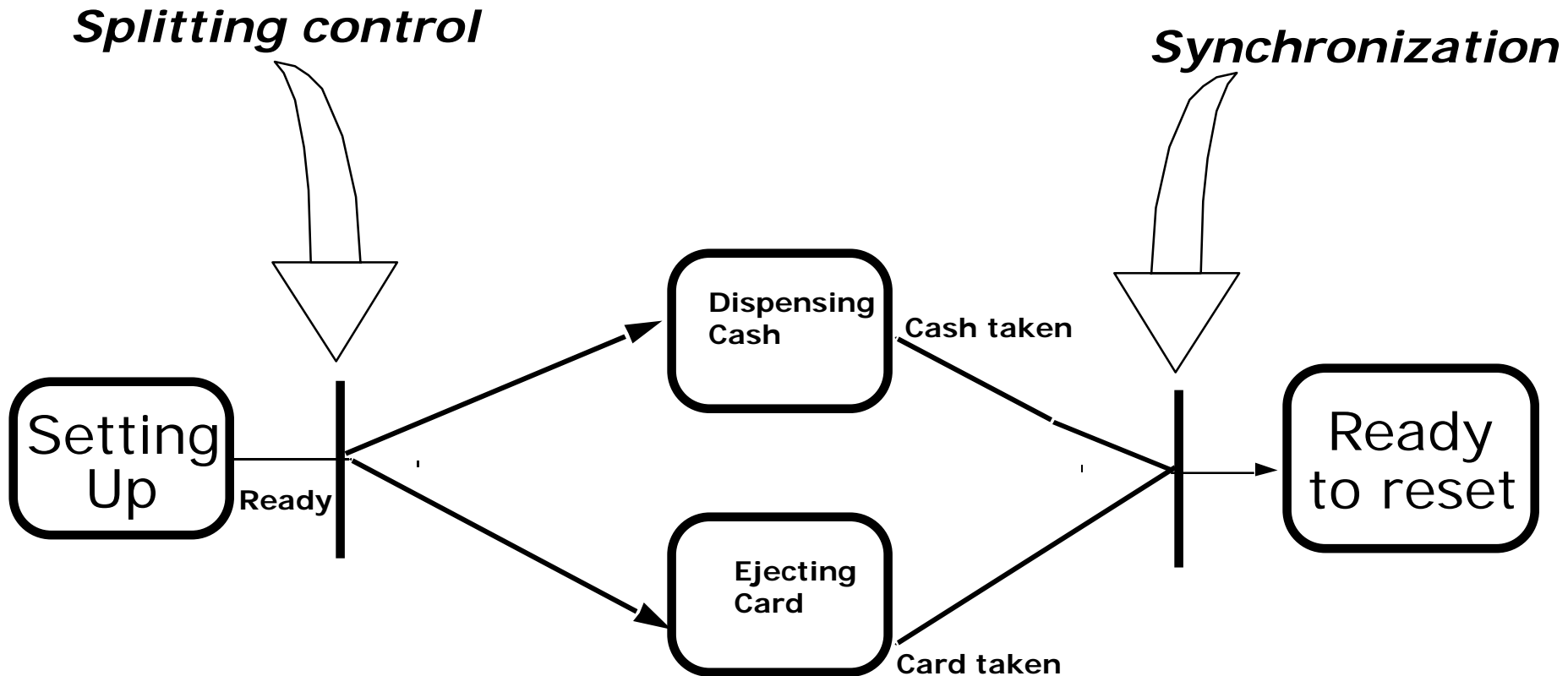
Activity diagram style

Splitting control

Synchronization

Emitting

Setting Up

Ready

Dispensing Cash — Cash taken

Ejecting Card — Card taken

Ready to reset

Statechart style

Dynamic modeling of user interfaces
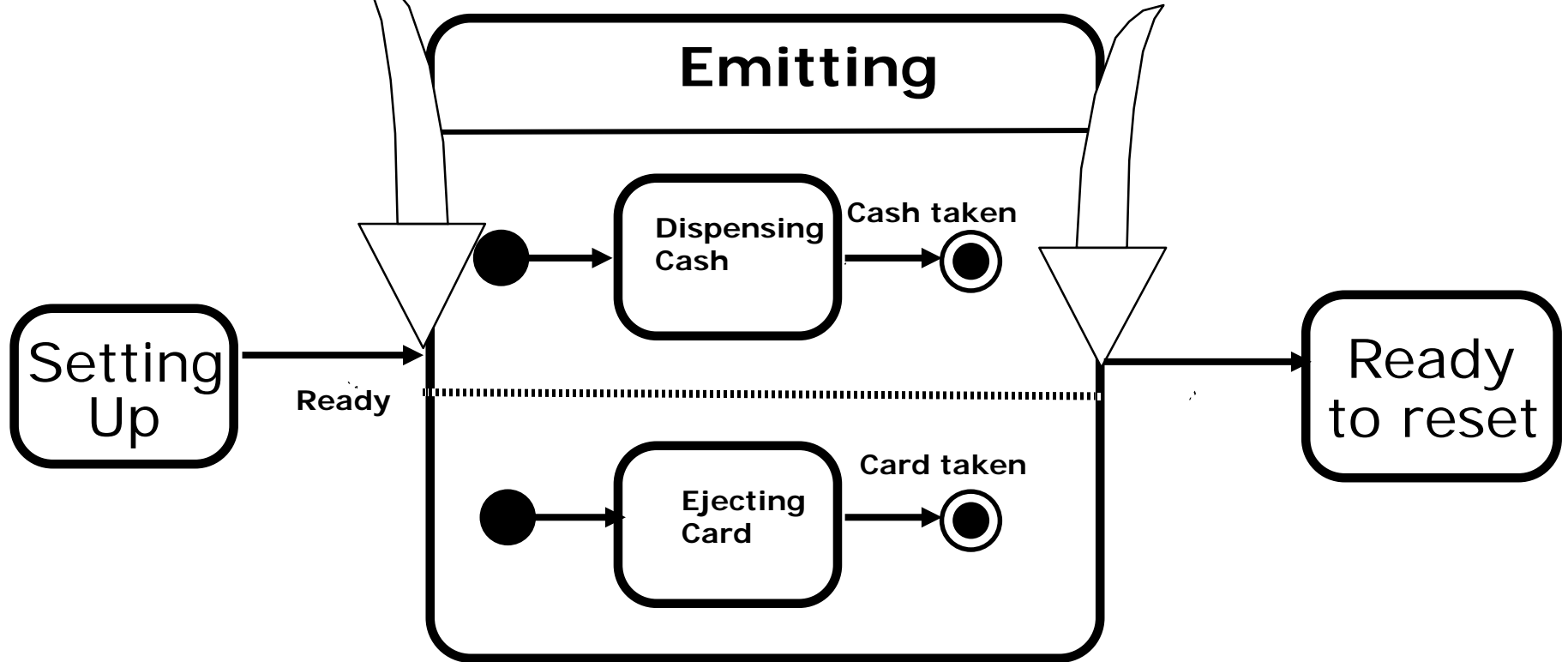
- Statechart diagrams can be used for the design of user interfaces
  - to represent the Navigation Path or Page Flow
- States: Name of screens
  - Graphical layout of the screens associated with the states helps when presenting the dynamic model of a user interface
- Activities/actions are shown as bullets under screen name
  - Often only the exit action is shown
- State transitions: Result of exit action
  - Button click
  - Menu selection
  - Cursor movements
- Good for web-based user interface design

Simplified navigation path example

Diagnostics Menu
• User moves cursor to Control Panel or Graph

Control panel
• User selects functionality of sensors

Graph
• User selects data group and type of graph

Define
• User defines a sensor event from a list of events

Selection
• User selects data group
  • Field site
  • Car
  • Sensor group
  • Time range
• User selects type of graph
  • time line
  • histogram
  • pie chart

Enable
• User can enable a sensor event from a list of sensor events

Disable
• User can disable a sensor event from a list of sensor events

List of events
• User selects event(s)

(lines represent pairs of arrows in both directions)

List of sensor events
• User selects sensor event(s)

Visualize
• User views graph
• User can add data groups for being viewed

Link
• User makes a link

Practical tips for state chart modeling

- Construct dynamic models only for classes with significant (complex/important) dynamic behavior
  - Avoid "analysis paralysis"
  - Exception: If state diagrams suffice for code generation
    - e.g. for control logic in telecommunications systems
- Consider only relevant attributes when defining states
  - Use abstraction heavily
- Stick to a sensible granularity of actions and activities
  - This is still analysis, not design!

- Reduce notational clutter
  - Try to put actions into state boxes (look for identical actions on events leading to the same state)

Summary: requirements analysis

1. What is the external behavior?   👉 **Functional Modeling**

    Create scenarios , use cases, use case diagrams

        Talk to client, observe, get historical records, do thought experiments

2. What is the structure of the system?   👉 **Object Modeling**

    Create *class diagrams*

        Identify objects, associations, attributes, operations

3. What is its behavior?   👉 **Dynamic Modeling**

    Create *sequence diagrams*
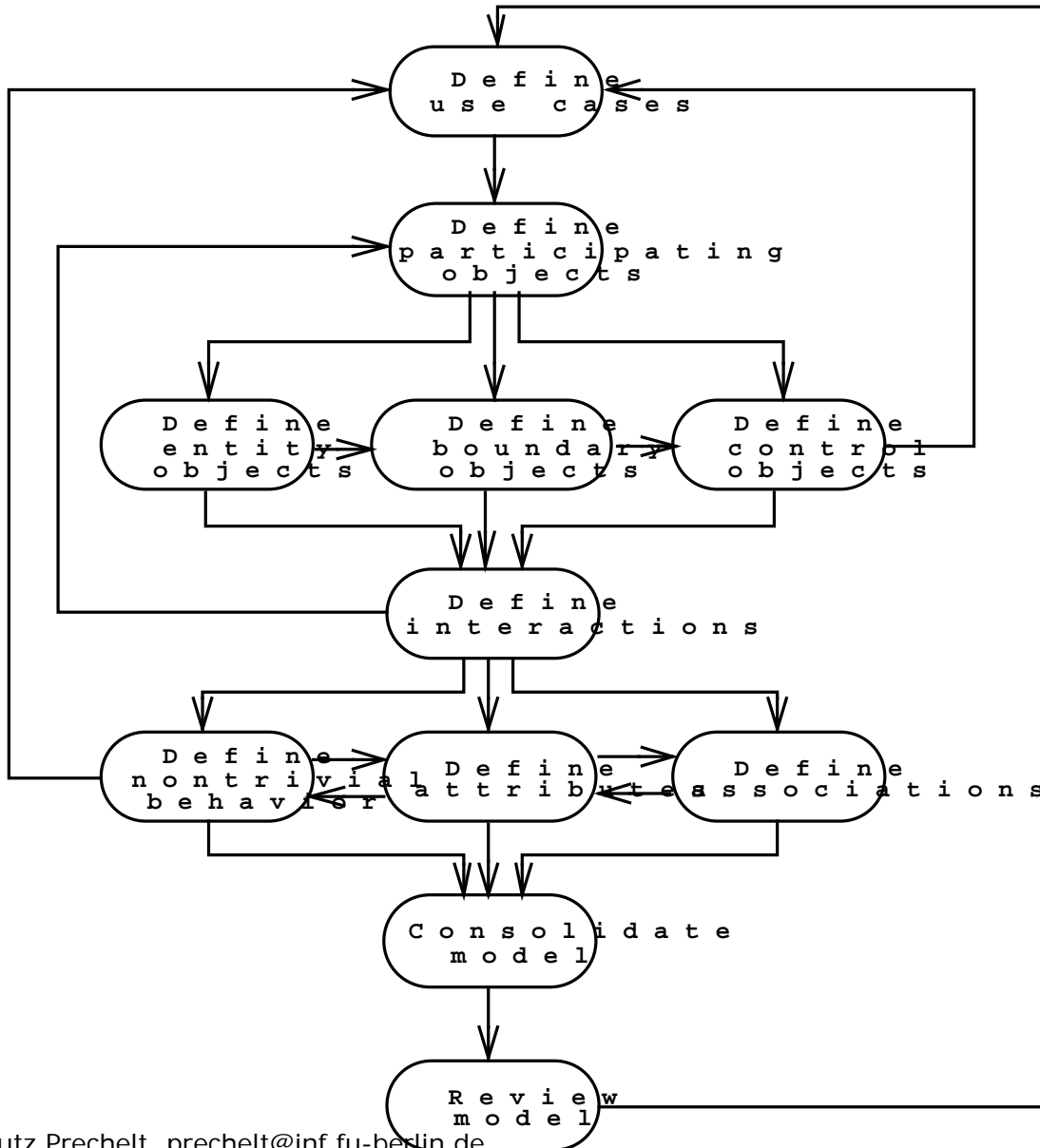
        Identify senders and receivers

        Show sequence of messages exchanged between objects

    Create *state diagrams*

        Only for the dynamically interesting objects

Freie Universität Berlin

Note that this diagram is rather vague, as the meaning of the arrows is not explained

Define use cases

Define participating objects

Define entity objects

Define boundary objects

Define control objects

Define interactions

Define nontrivial behavior

Define attributes

Define associations

Consolidate model

Review model

When is a model dominant?

- We call a model dominant if it contains a much larger fraction of the *interesting* information than the others

Examples:

- Simple database system:
  - Situation: The operations are straightforward (load, store), but there are complex data structures
  - Consequence: The static object model is dominant

- Telephone switching system:
  - Data structures do not tell us much and behavior is too complex to be fully described by use cases
  - The dynamic model (in particular using statecharts) is dominant

Requirements analysis document template

1. Introduction
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements   [keep this short! →3.5.2]
   - 3.3 Nonfunctional requirements
   - 3.4 Constraints ("Pseudo requirements")   *see the following slides*
   - 3.5   Analysis Model   *on 3.5 (short), 3.3, 3.4*
       - 3.5.1 Scenarios
       - 3.5.2 Use case model
       - 3.5.3 Object model
           - 3.5.3.1 Data dictionary
           - 3.5.3.2 Class diagrams
       - 3.5.4 Dynamic model
       - 3.5.5 User interface
4. Glossary

Section 3.5: system models

- 3.5.1 Scenarios
  - As-is scenarios, visionary scenarios

- 3.5.2 Use case model
  - Actors and use cases

- 3.5.3 Object model (this is still analysis!)
  - Data dictionary
  - Class diagrams (classes, associations, attributes and operations)

- 3.5.4 Dynamic model
  - State diagrams for classes with significant dynamic behavior
  - Sequence diagrams for collaborating objects (protocol)

- 3.5.5 User Interface
  - Navigational Paths, Screen mockups

Section 3.3:
nonfunctional requirements

- 3.3.1   User interface and human factors
- 3.3.2   Documentation
- 3.3.3   Hardware considerations
- 3.3.4   Performance characteristics
- 3.3.5   Error handling and extreme conditions
- 3.3.6   System interfacing
- 3.3.7   Quality issues
- 3.3.8   System modifications
- 3.3.9   Physical environment
- 3.3.10   Security issues
- 3.3.11   Resources and management issues

see the following slides

Nonfunctional requirements: trigger questions

- 3.3.1 User interface and human factors
  - What type of user will be using the system?
  - Will more than one type of user be using the system?
  - What sort of training will be required for each type of user?
  - Is it particularly important that the system be easy to learn?
  - Must users be particularly well protected from making errors?
  - What sort of UI input/output devices will be used?

- 3.3.2 Documentation
  - What kind of documentation is required?
  - What audience is to be addressed by each document?

- 3.3.3 Hardware considerations
  - What hardware is the proposed system to be used on?
  - What are the characteristics of the target hardware, including memory size and auxiliary storage space?

Nonfunctional requirements (2)

- 3.3.4 Performance characteristics
  - Are there any speed, throughput, or response time constraints on the system?
  - Are there size or capacity constraints on the data to be processed by the system?

- 3.3.5 Error handling and extreme conditions
  - How should the system respond to input errors?
  - How should the system respond to extreme conditions?

- 3.3.6 System interfacing
  - What input is coming from systems outside the proposed system?
  - What output is going to systems outside the proposed system?
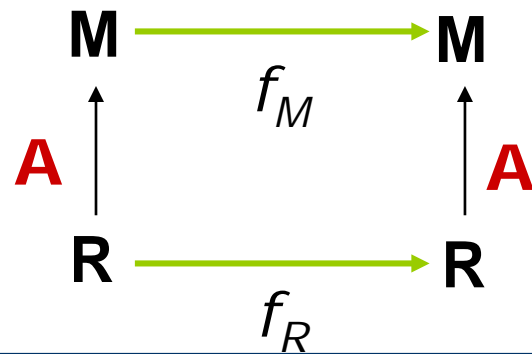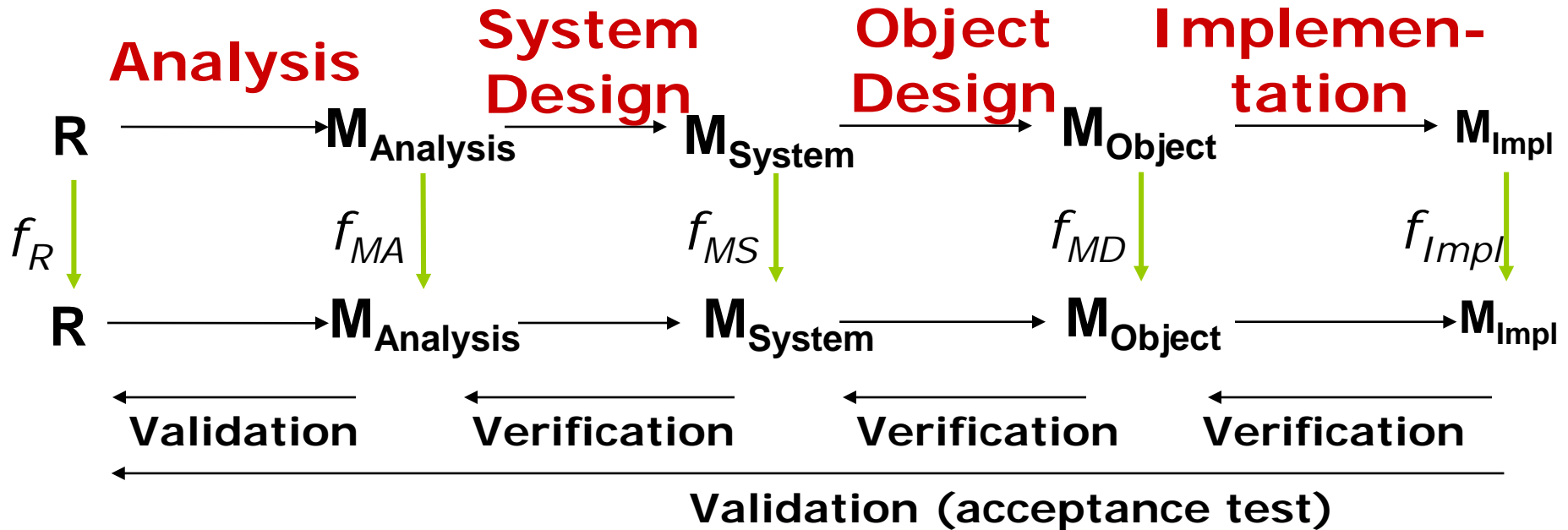
Freie Universität Berlin

- 3.3.7 Quality issues
  - What are the requirements for reliability?
  - Must the system trap faults?
  - How fast must the system restart after a failure?
  - What is the acceptable system downtime per day/month/year?
  - Is it important that the system be portable (able to move to different hardware or operating system environments)?

- 3.3.8 System Modifications
  - What parts of the system are likely candidates for later modification?
  - What sorts of modifications are expected?

- 3.3.9 Physical Environment
  - For example, unusual temperatures, humidity, vibrations, magnetic fields, …

Nonfunctional requirements (4)

- 3.3.10 Security Issues
  - Must access to any data or the system itself be controlled?
  - Is physical security an issue?

- 3.3.11 Resources and Management Issues
  - How often will the system be backed up?
  - Who will be responsible for
    - system installation?
    - daily operation and configuration?
    - back up? When? How often?
    - maintenance?
  - What is the desaster recovery plan?

Freie Universität Berlin

- Constraint:
  - Any client restriction on the solution domain

- Examples:
  - The target platform must be an IBM iSeries
  - The implementation language must be COBOL
  - The documentation standard X must be used
  - A dataglove must be used
  - ActiveX must not be used
  - The system must interface to a papertape reader

Analysis **System Design** **Object Design** **Implemen- tation**

$R \longrightarrow M_{Analysis} \longrightarrow M_{System} \longrightarrow M_{Object} \longrightarrow M_{Impl}$

f_R f_{MA} f_{MS} f_{MD} f_{Impl}

$R \longrightarrow M_{Analysis} \longrightarrow M_{System} \longrightarrow M_{Object} \longrightarrow M_{Impl}$

Validation   **Verification**   **Verification**   **Verification**

Validation (acceptance test)

$M \xrightarrow{f_M} M$

A A

$R \xrightarrow{f_R} R$

M = Model

R = Reality

f = Behavior/relationships

A = abstraction/modelling

Correctness, completeness and consistency

- Verification is an equivalence check between two related models:
  - The second was derived from the first by transformation.
    Is the transformation correct?
- Validation is different. We don't have two models, we need to compare one model with reality
  - "Reality" can also be an artificial system, like a legacy system

- Requirements and implementations should be validated with the client and the user
  - Techniques for requirements: Formal and informal reviews (Meetings, requirements review)
  - Techniques for implementations: Acceptance testing, system use
- Requirements validation involves the checks for
  - Correctness, Completeness, Ambiguity, Realism

Modeling checklist for the review

- Is the model correct?
  - A model is correct if it represents the client's view of the the system: Everything in the model represents an aspect of reality
- Is the model complete?
  - Every relevant scenario, including exceptions, is described
- Is the model consistent?
  - The model does not have components that contradict each other (for example, deliver contradicting results)
- Is the model unambiguous?
  - The model describes one target reality, not many
- Is the model realistic?
  - The model can be implemented with acceptable effort

At the end of analysis:
Project agreement

- The project agreement represents the acceptance of (parts of) the analysis model (as documented by the requirements analysis document) by the client

- The client and the developers converge on a single idea and agree about the functions and features that the system will have. In addition, they agree on:
  - a list of prioritized requirements
  - a revision process
  - a list of criteria that will be used to accept or reject the system
  - a schedule, and probably a budget

Prioritizing requirements

- High priority ("Core requirements")
  - Must be addressed during *analysis, design, and implementation*
  - A high-priority feature must be demonstrated successfully during client acceptance

- Medium priority ("Optional requirements")
  - Must be addressed during *analysis and design*
  - Often implemented and demonstrated in the second iteration of the system development

- Low priority ("Fancy requirements")
  - Must be addressed during *analysis* ("very visionary scenarios")
  - Illustrates how the system may be going to be used in the future
    - e.g. once not-yet-available technology becomes available

Summary

- In this lecture, we reviewed the construction of the dynamic model from use case and object models.
  - In particular: Sequence and Statechart diagrams for identifying new classes and operations

- In addition, we described the requirements analysis document and its components

Thank you!