

Course "Softwaretechnik"

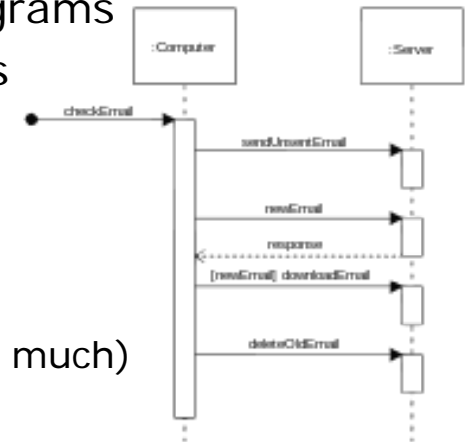
Dynamic Modeling, Req's Document

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit

Freie Universität Berlin, Institut für Informatik

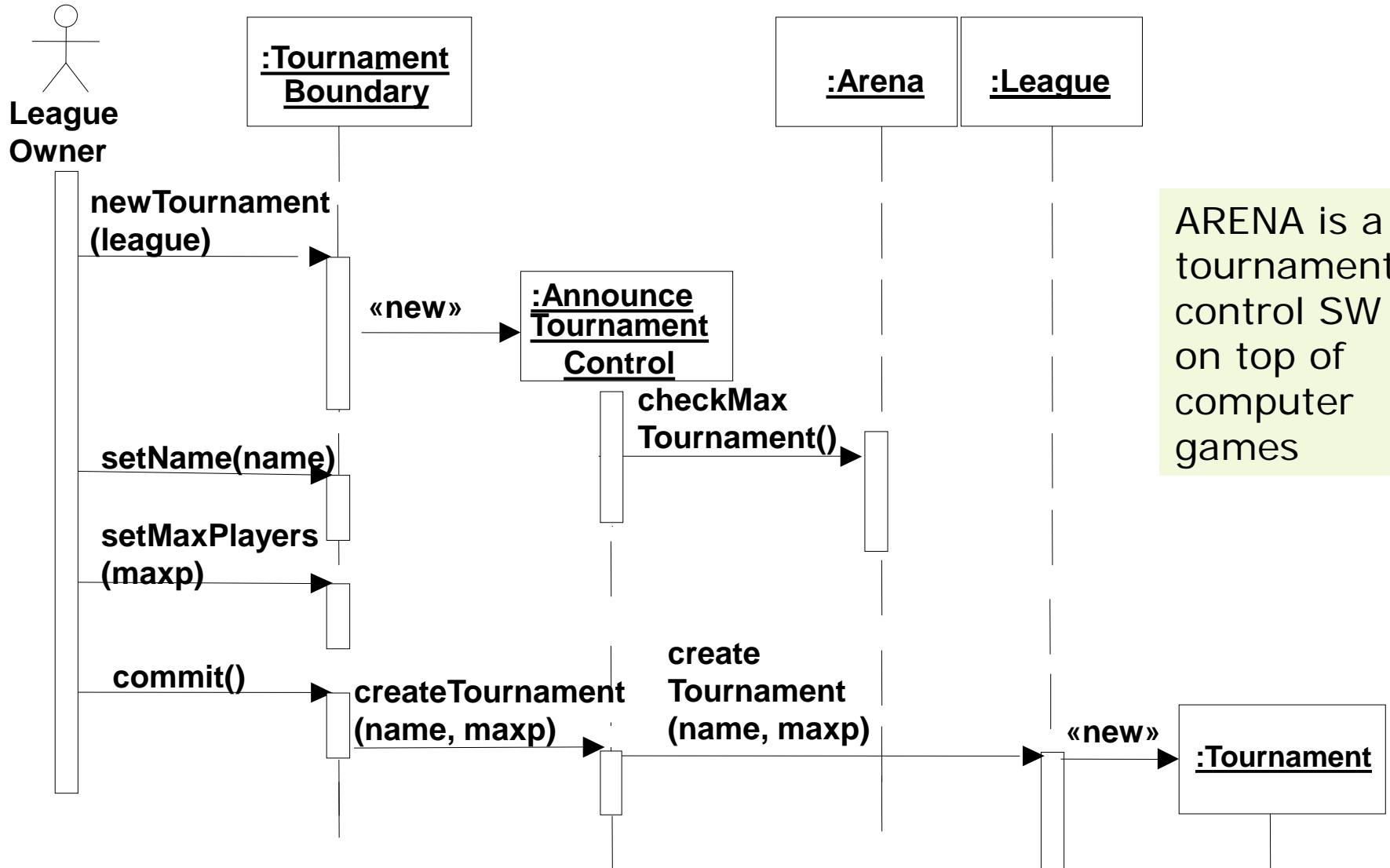
- Dynamic modeling
 - Sequence diagrams
 - ARENA
 - State machine diagrams
- Using dynamic modeling for the design of user interfaces
- Requirements analysis document template
- Conventional vs. agile
- Requirements analysis model validation

- Definition of dynamic model:
 - A collection of multiple behavior diagrams
 - such as state machine, activity, and sequence diagrams
 - usually at least one regarding each important class with important dynamic behavior
- Purpose:
 - Understand behavioral requirements (not discussed much)
 - 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 behavior of a single object → state machine diagram



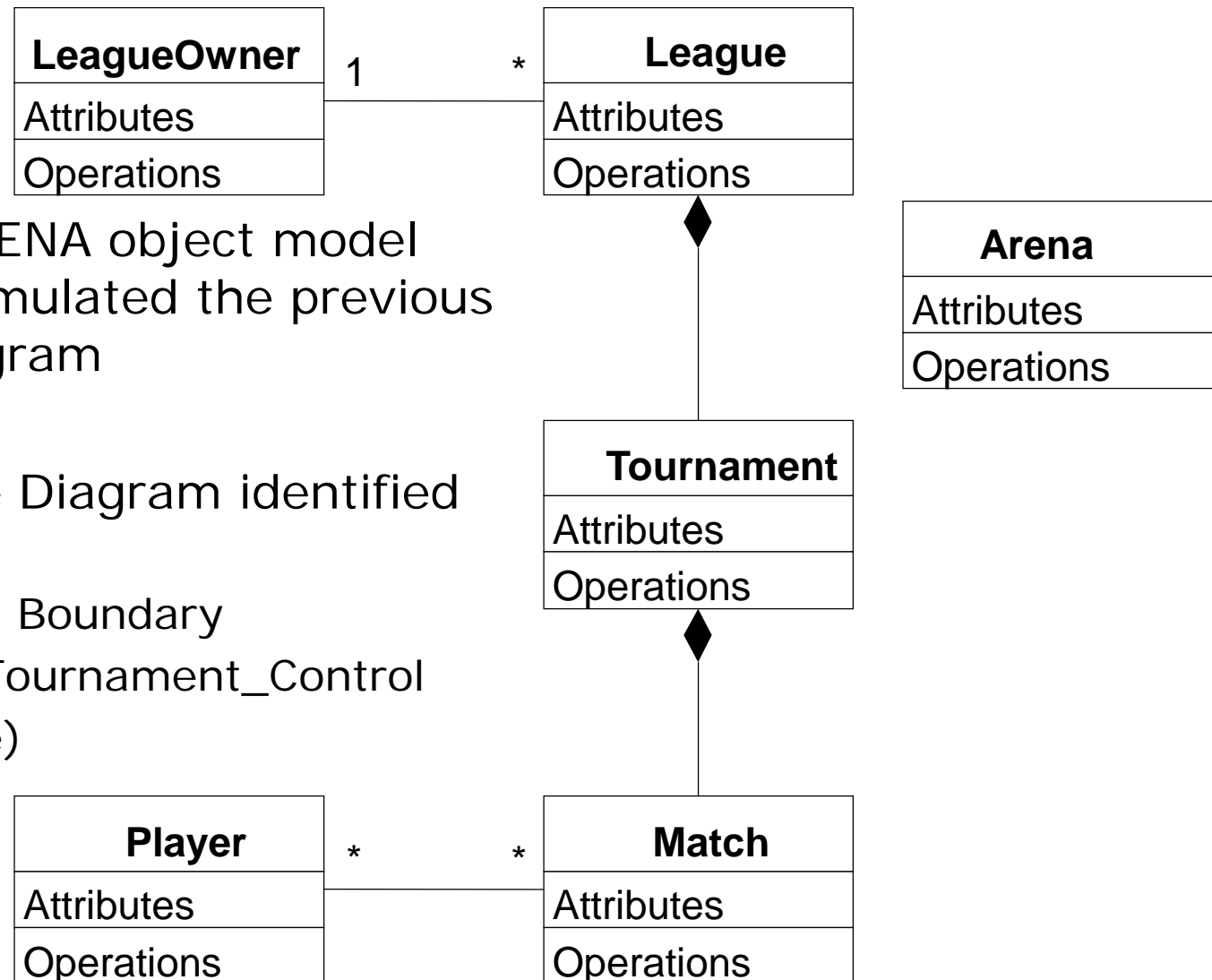
- A typical layout:
 - 1st column: The *actor* who initiated the use case
 - 2nd column: A *boundary object* (in design, perhaps not in analysis)
 - 3rd column: Perhaps a *control object* managing the use case
 - further columns: the other participating objects
- Creation:
 - Control objects are often 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

An ARENA sequence diagram: create tournament



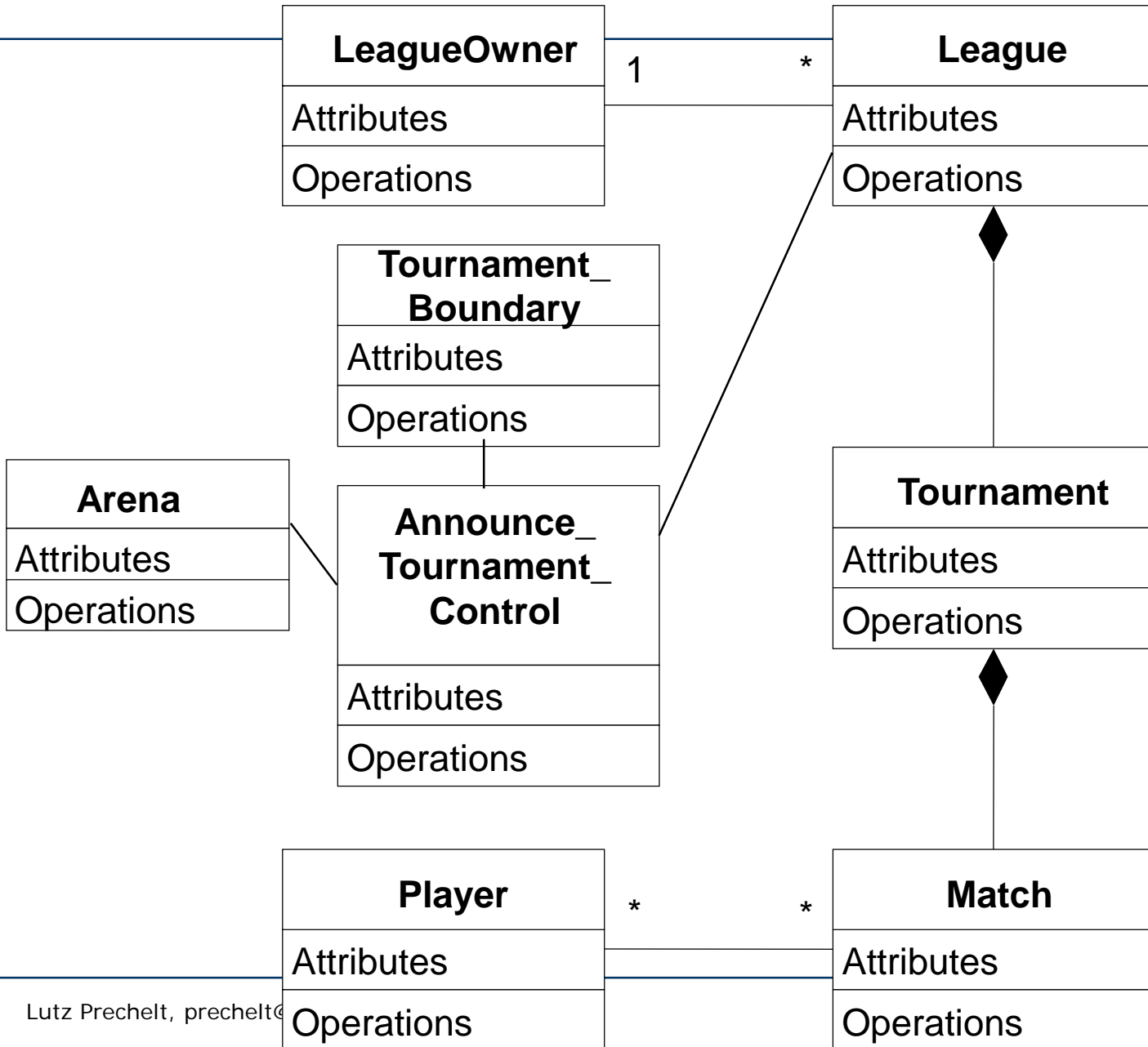
ARENA is a tournament control SW on top of computer games

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)



- 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
 - e.g. create_tournament() in AnnounceTournament_Control

- Sequence diagrams are derived from use cases
 - We therefore see the structure of the use cases
- The structure of the sequence diagram suggests how decentralized the resulting system structure might be
- We distinguish two basic structures of sequence diagrams (Ivar Jacobson):
 - Fork-style diagrams (central control)
 - Stair-style diagrams (distributed control)

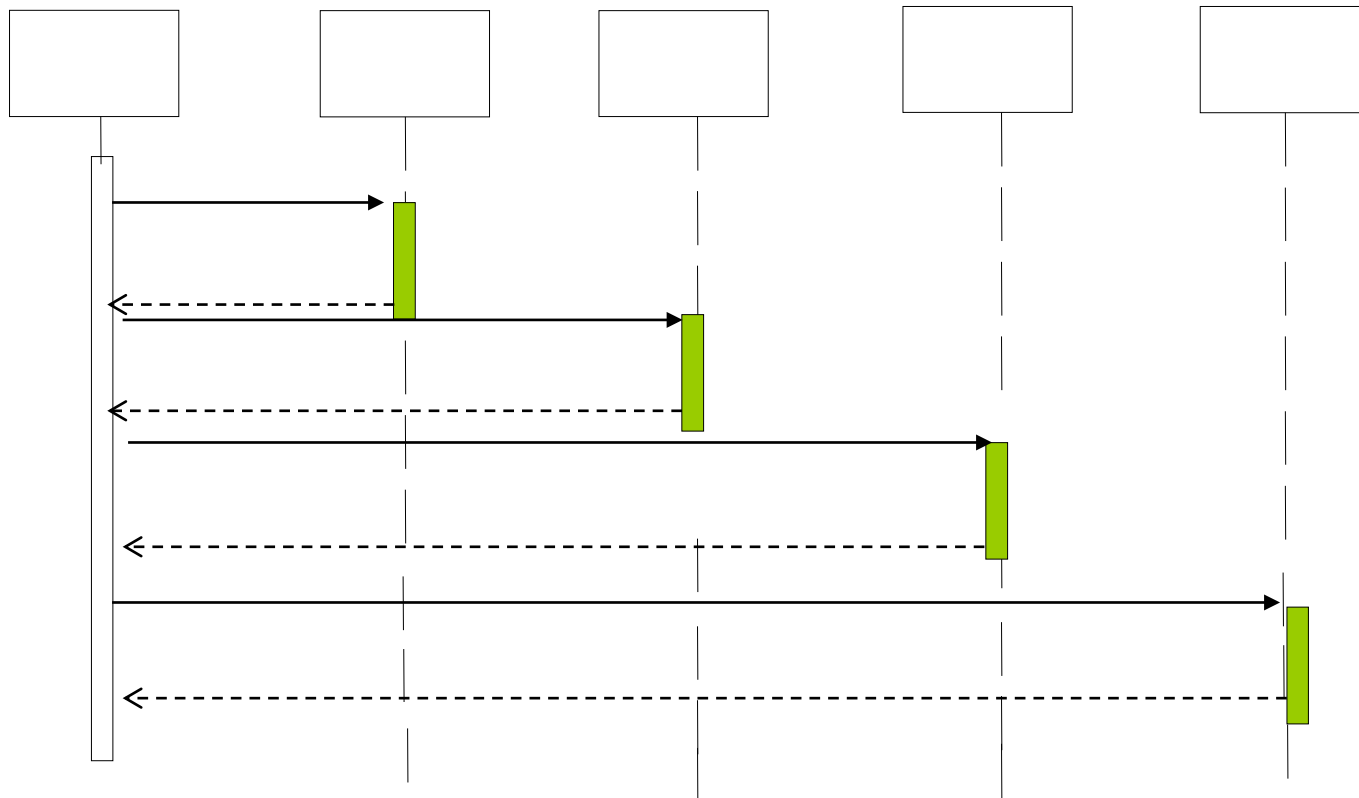
(see next slides)

UML stereotype symbols:



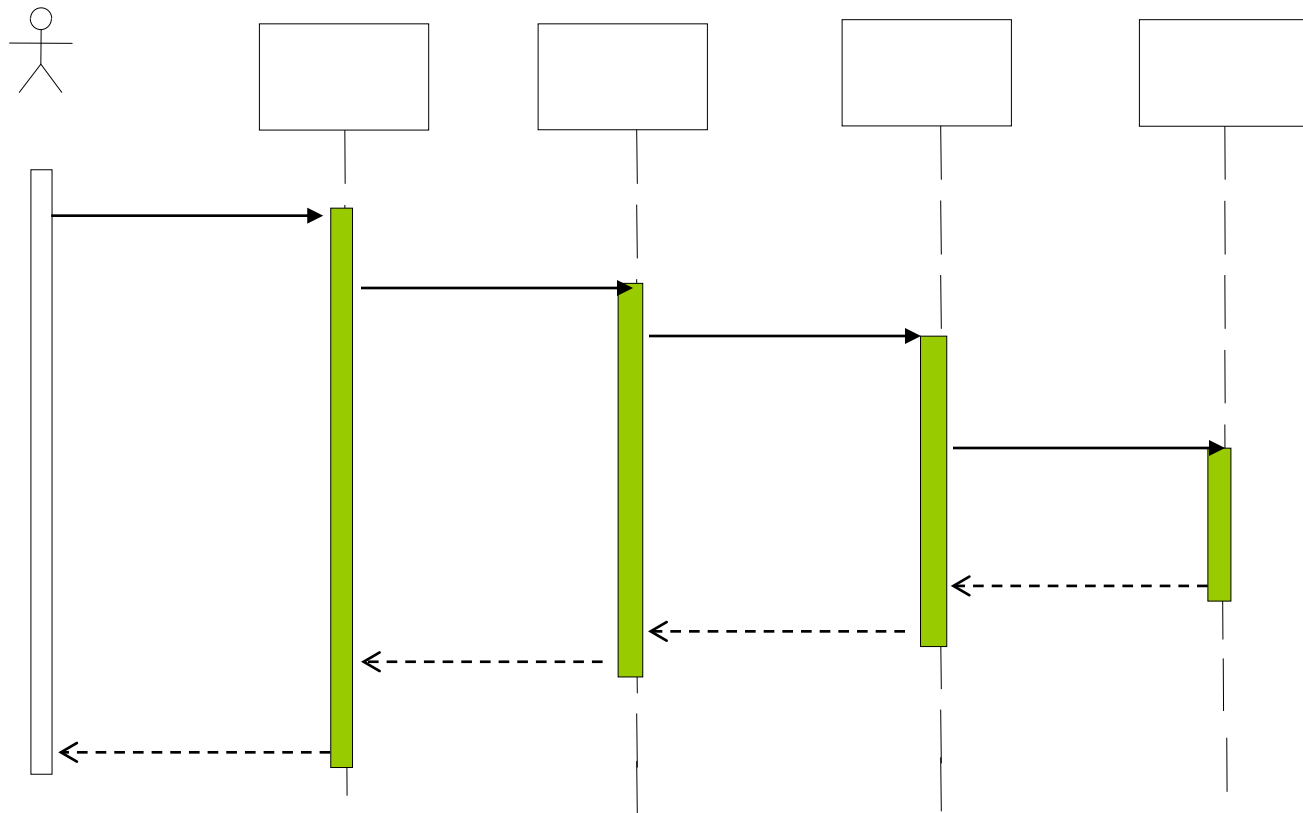
Central control: Fork diagram

- 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

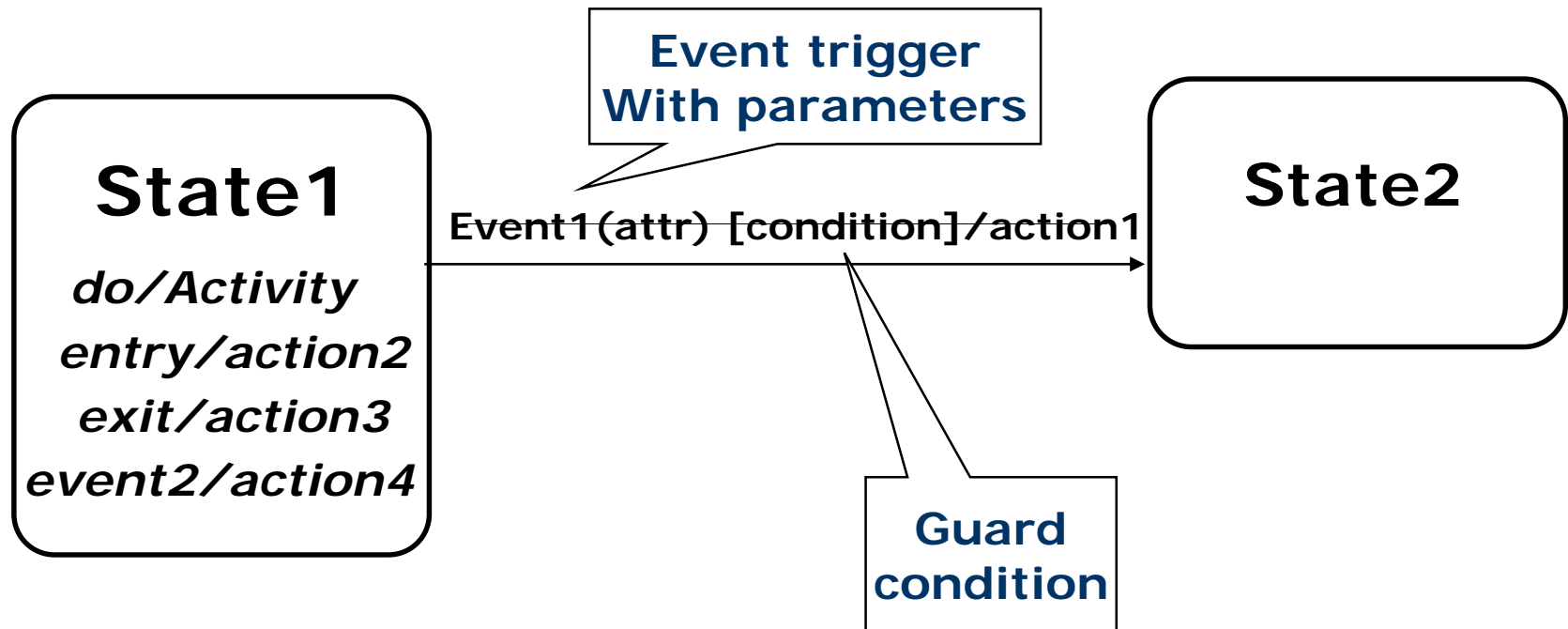


Which of these diagram types should be chosen?

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



- Notation based on work by Harel ("statecharts")
 - UML adds a few object-oriented modifications
- A UML state machine diagram (statechart diagram, state chart diagram) can be mapped to a finite state machine

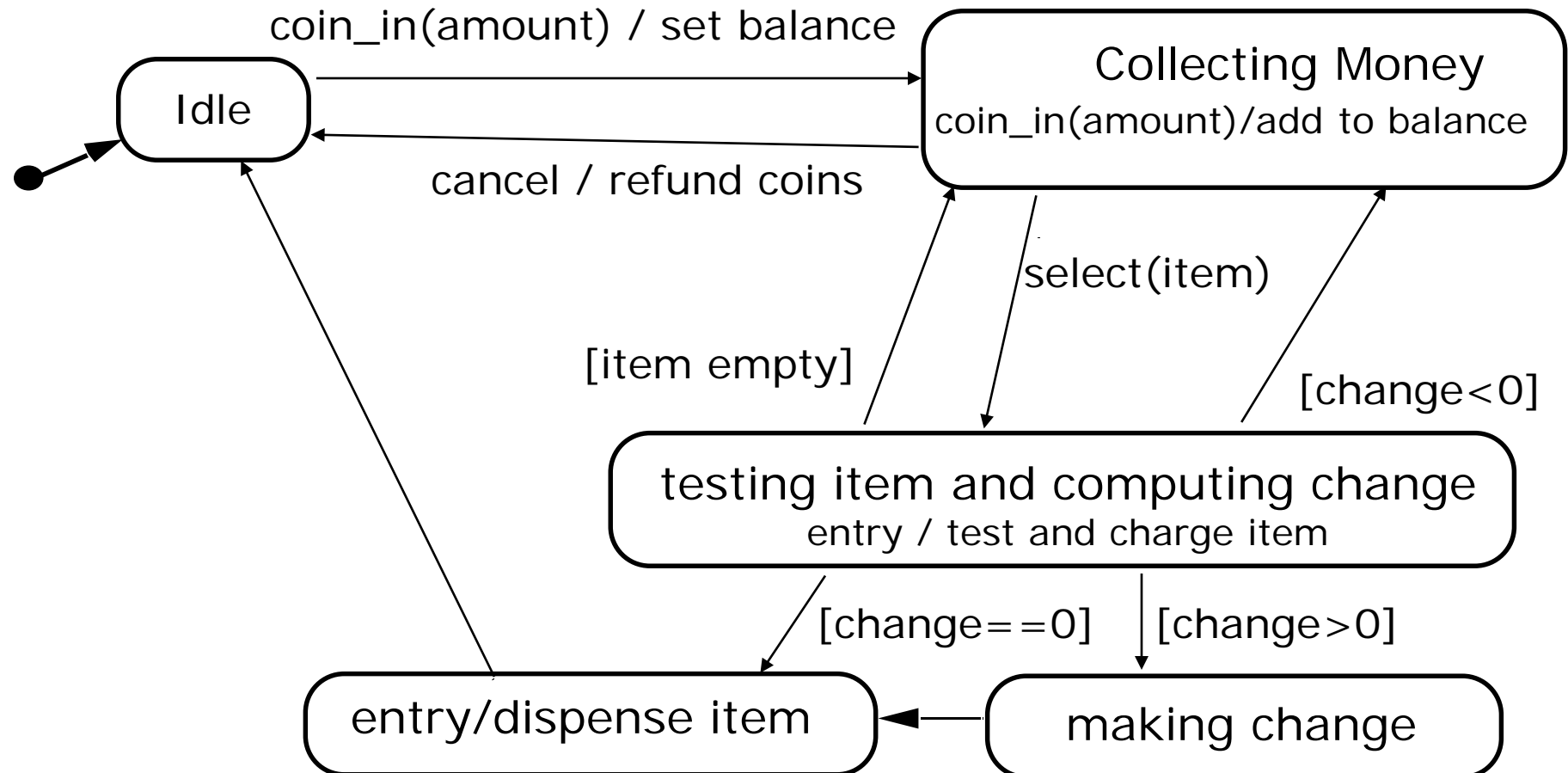
State machine diagrams

- Graph whose nodes are states and whose directed edges are transitions labeled by event names
- We distinguish between two types of executable nodes in a state machine:
 - Activity: Compound operation
 - can be described by its own Activity diagram
 - Action: Elementary operation
 - May in fact have structure, too, but the present state machine ignores it
- A state machine diagram relates events and states for one class
 - An object model with a set of objects can have a corresponding set of state machine diagrams

- 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 (in fact 7701 different states!)
 - What is appropriate depends on our current goal
- State has duration

Example of a state machine diagram

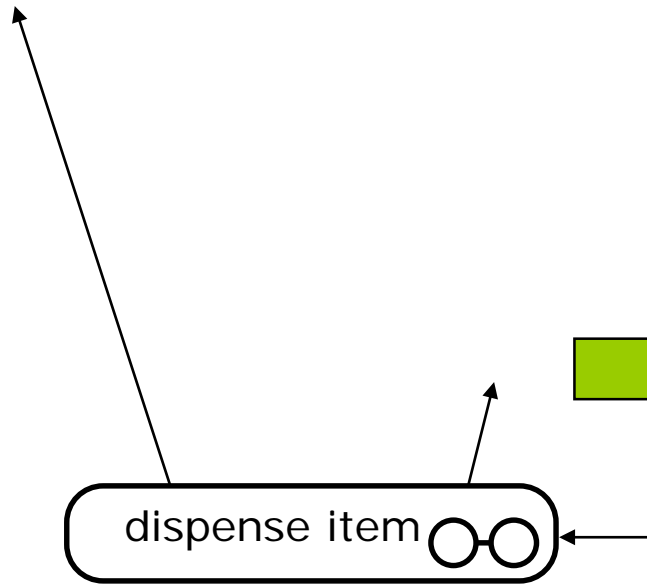
A vending machine example:



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

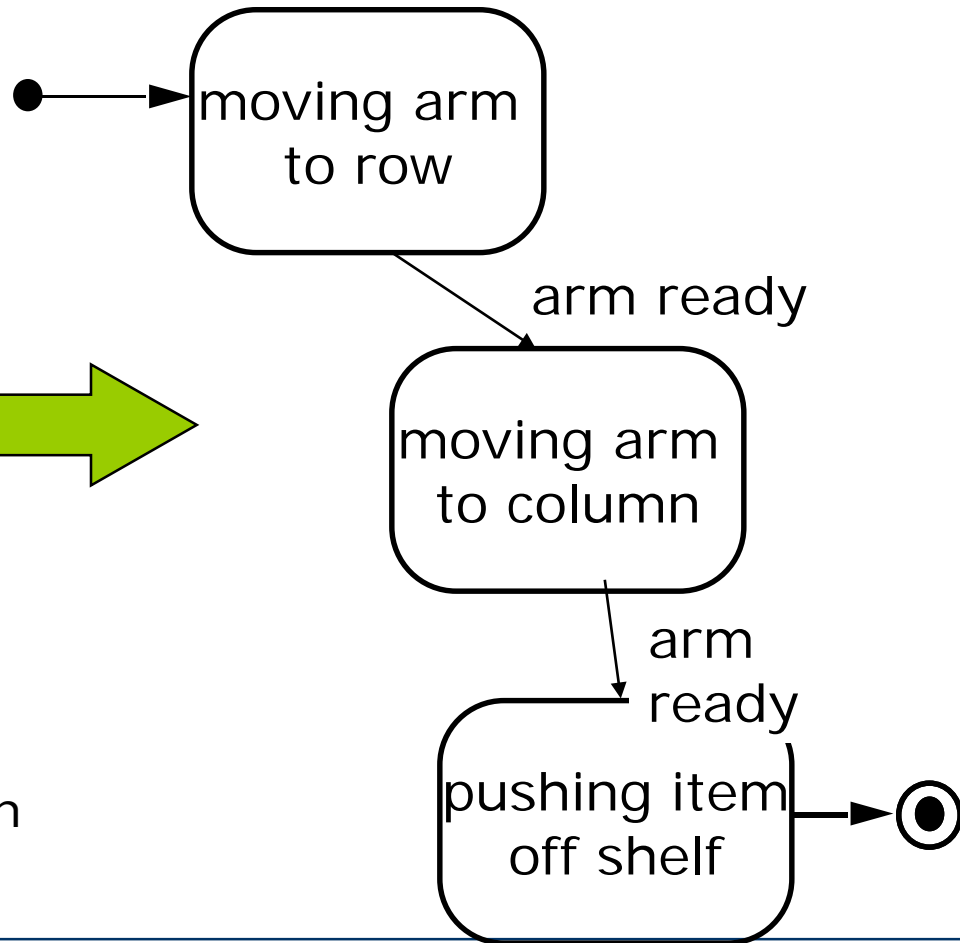
Nested states: Example

'Dispense item' as an atomic activity:

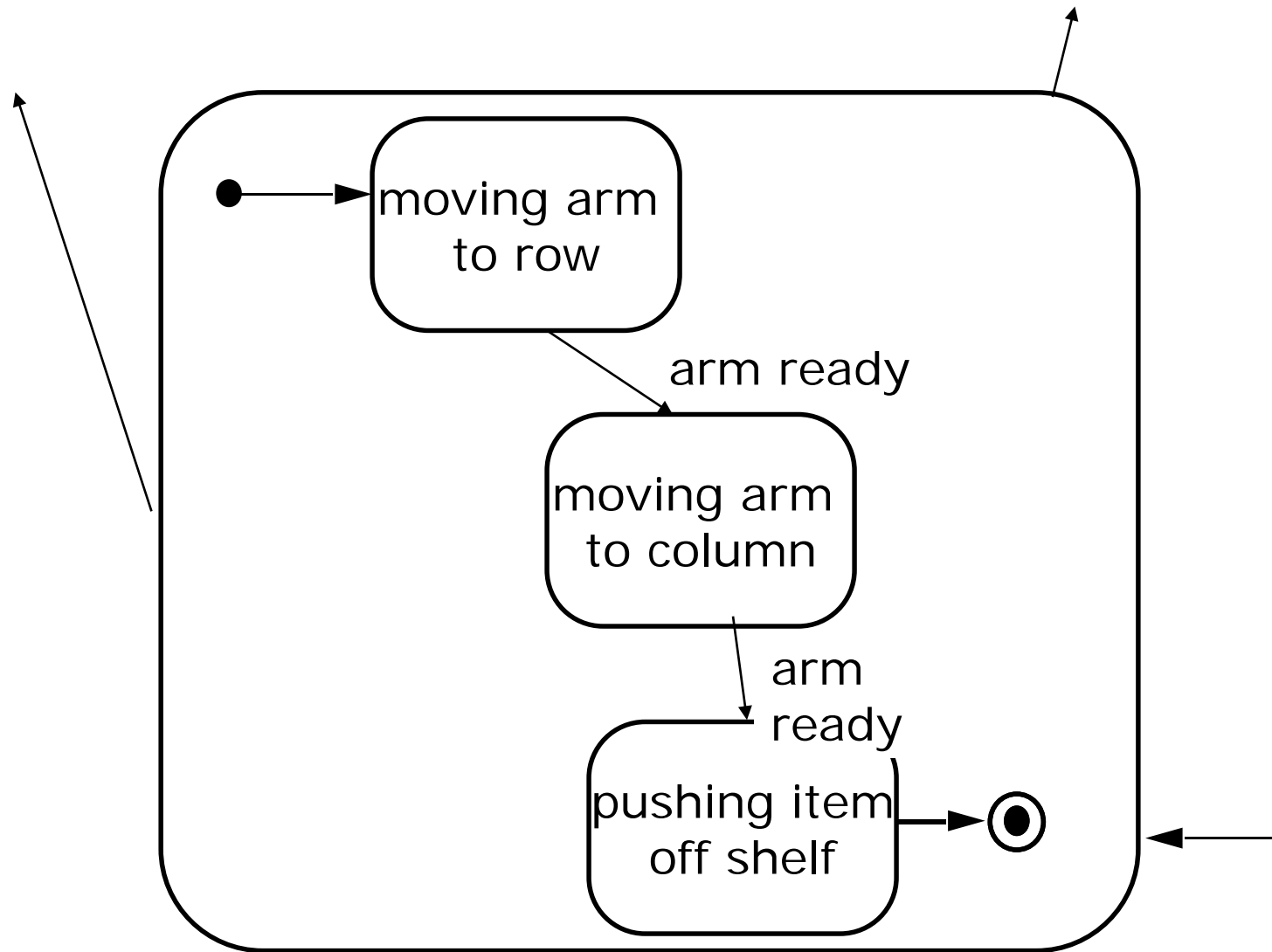


The little glasses indicate that there are sub-activities hidden in this composite activity

'Dispense item' as a composite activity:



Composite State

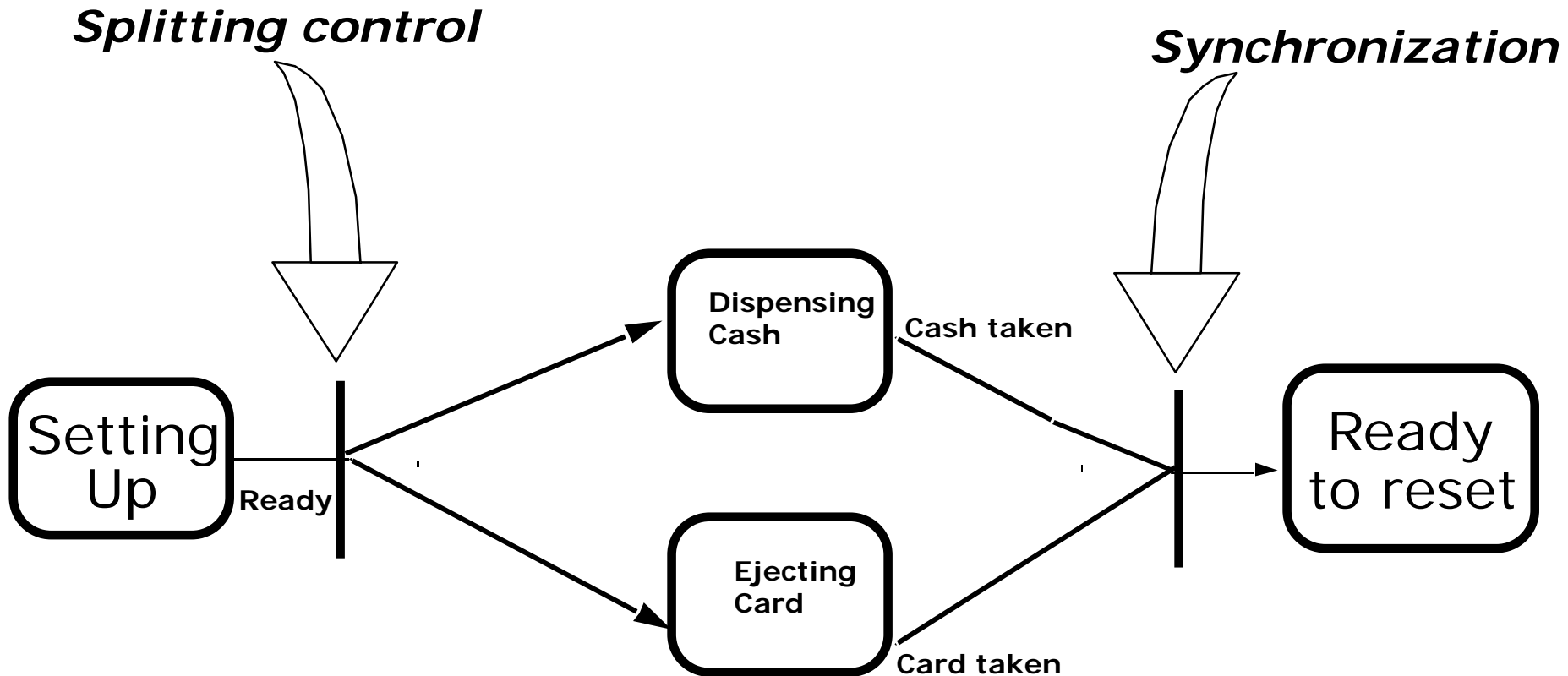


Two types of concurrency in UML:

- 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 may be divided into *regions* by dotted lines

This course mostly ignores concurrency!

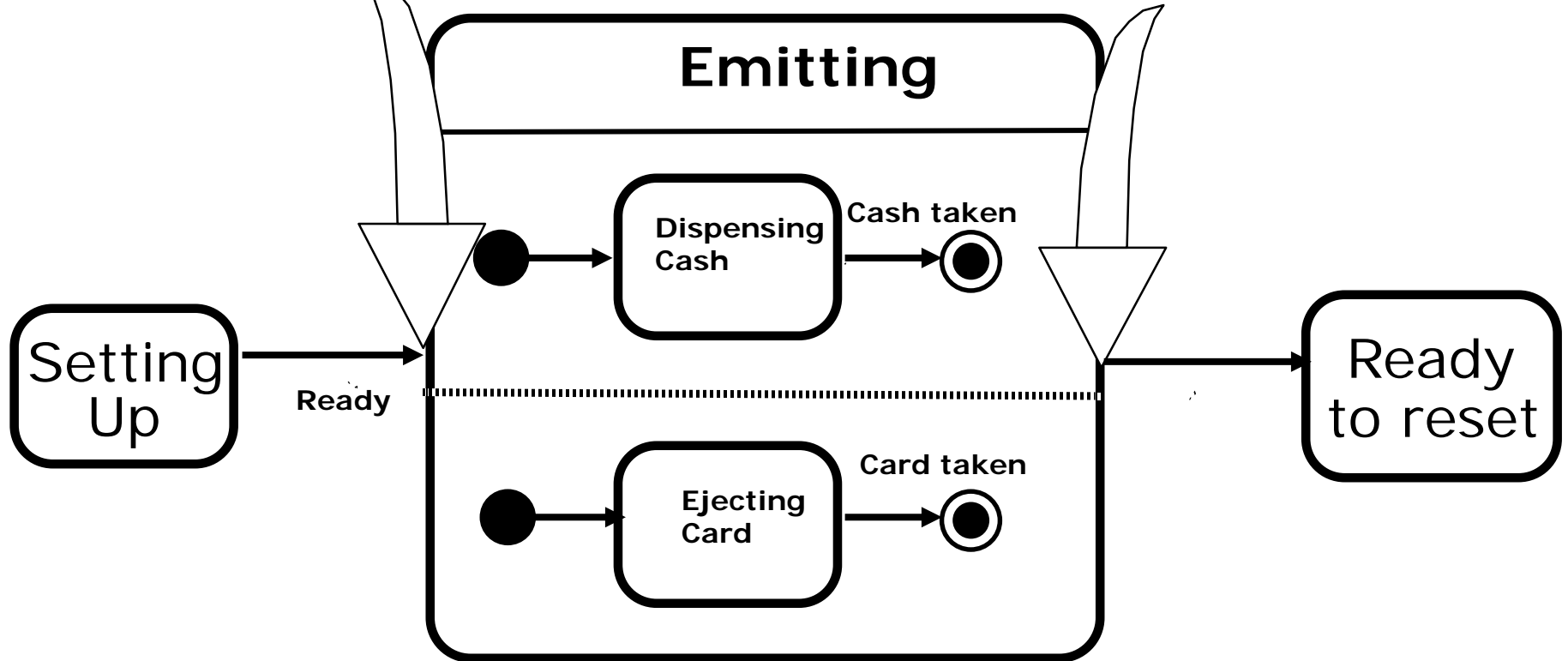
Example of concurrency within an object



Activity diagram style

Splitting control

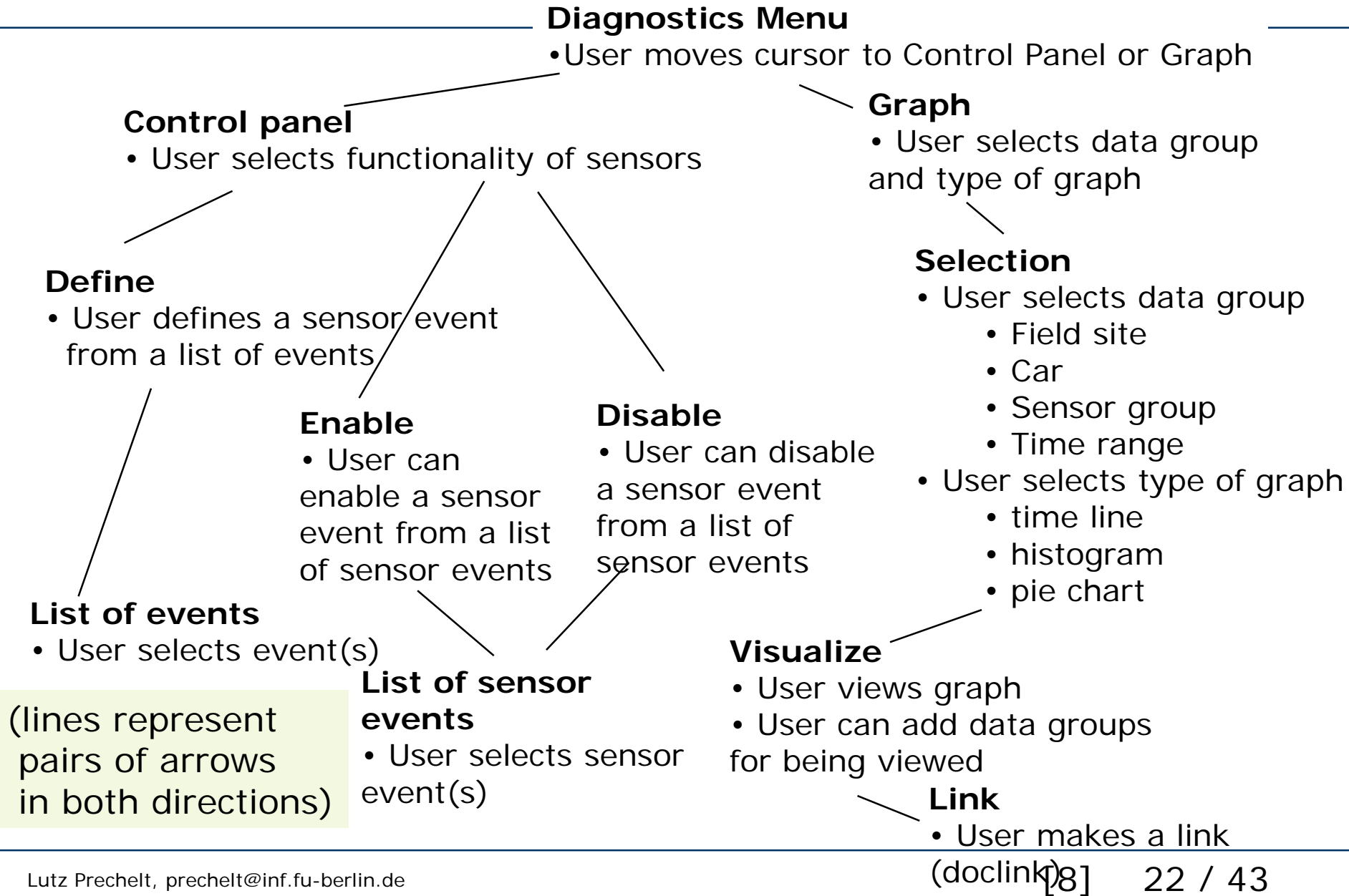
Synchronization



Statechart style

- 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



1. Construct dynamic models only for classes with significant (complex/important) dynamic behavior
 - To avoid "analysis paralysis"
 - Exception: If state diagrams suffice for code generation
 - e.g. for control logic in telecommunications systems
2. Consider only relevant attributes when defining states
 - Use abstraction heavily
3. Stick to a sensible granularity of actions and activities
 - This is still analysis, not design!
4. 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

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 machine diagrams*

Only for the dynamically interesting objects

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 easily described by use cases
 - The dynamic model (in particular using statecharts) is dominant

Requirements document

A possible requirements analysis document template

1. Introduction
2. Current system
3. Proposed system
 - 3.1 Overview
 - 3.2 Functional requirements [avoid overlap with 3.5.2]
 - 3.3 Nonfunctional requirements
 - 3.4 Constraints ("Pseudo requirements") *see the following slides on 3.5 (short), 3.3, 3.4*
 - 3.5 Analysis Model
 - 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

Remember this is only *conceptually* a single document.

It may *actually* be a variety of separate things, some not even written up at all.

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 (explain data: formats and terms)
 - 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

- 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?

- 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?

- 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, ...

- 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 disaster recovery plan?

This course mostly ignores operation!

Section 3.4

Constraints (pseudo requirements)

- Constraint:
 - Any client restriction on the solution domain
- Examples:
 - Common in areas with long-lived systems or strong regulation
 - Often highly specialized territory
 - The target platform must be an IBM iSeries
 - The implementation language must be PL/SQL
 - The documentation standard X must be used
 - ActiveX must not be used
 - The system must interface to an IBM 1621 papertape reader

Concluding remarks:

1. Conventional vs. Agile



Requirements documents (req docs) in Conventional vs. Agile processes

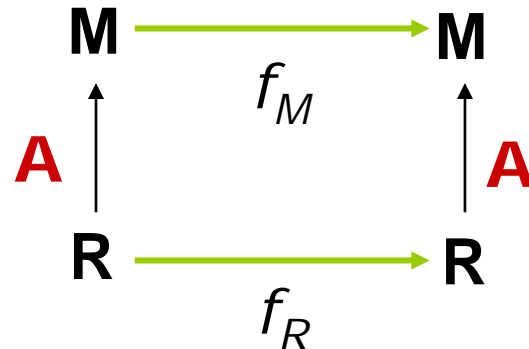
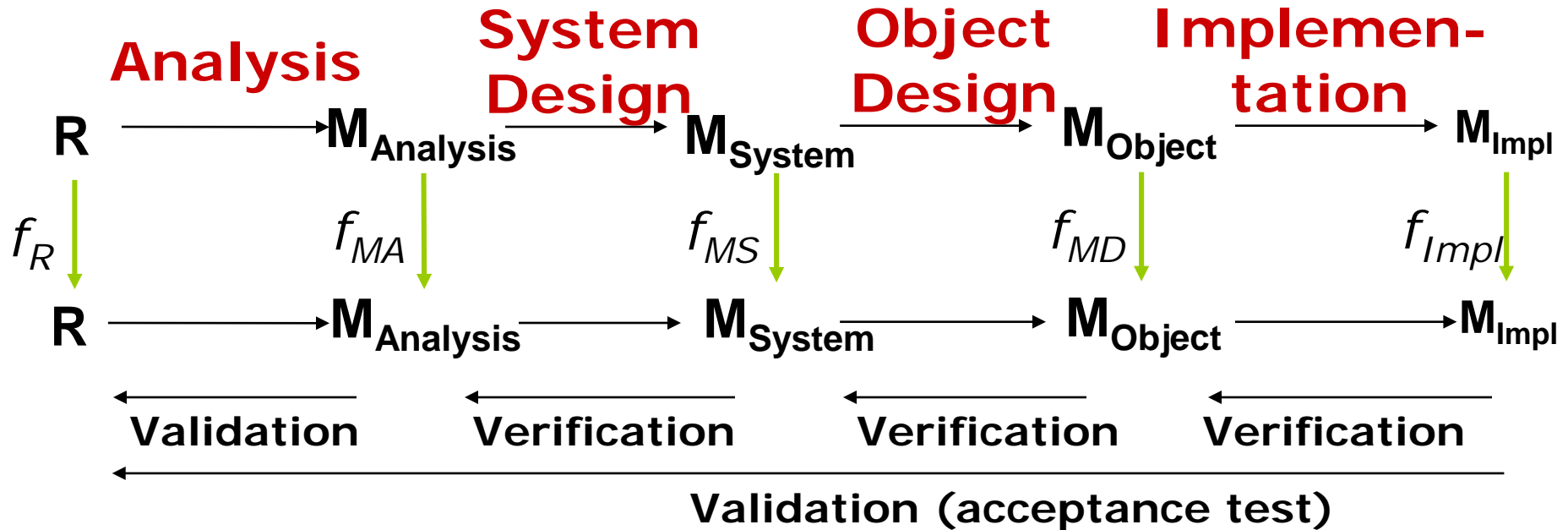
- Having a detailed, formal requirements document is the hallmark of a conventional project
- Often a core element of a fixed-price contract for a hired project
- Despite their size, req doc quality is often dubious
- Initially called Lightweight processes, because they shied away from many conventional process elements
- Meant-to-be Agile projects are often run in fixed-price style as well. Huh??
- Iff an "agile req doc" is less detailed, quality *may* be better

To understand a software organization,
looking at their req docs is helpful.

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 a later 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

2. Verification and validation of models



M = Model

R = Reality

f = Behavior/relationships

A = abstraction/modelling

- 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

- 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 sufficiently unambiguous (i.e. not vague)?
 - All plausible interpretations will be acceptable solutions
- Is the model realistic?
 - The model can be implemented with acceptable effort

3. 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:
 - requirements priorities
 - the criteria that will be used to accept or reject the system
 - a revision process
 - a schedule, and probably a budget
- In a phased development model, this is a single event, in modern iterative development, it will be several
 - and can even be a continuous process

- Sequence diagrams and State Machine diagrams can help in identifying classes and (in particular) operations
- The requirements document contains far more than just functional requirements as use cases
- True agile processes will not have such a document

Thank you!