Politecnico
di Milano

Modeling requirements

Modeling requirements

- What is a model
- What to model in RE
- Tools for modeling

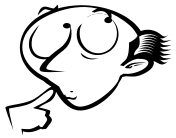
The Central Role of Requirements Models

stakeholders

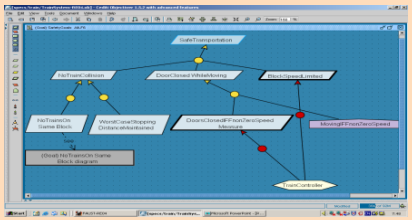
existing systems

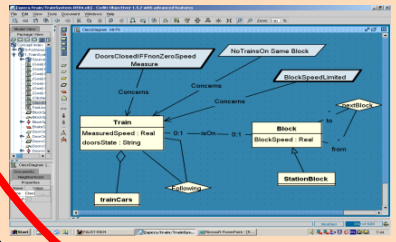
documents

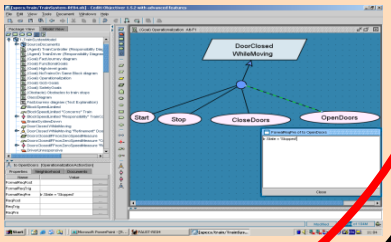
elicitation
& **modelling**



Requirements Models

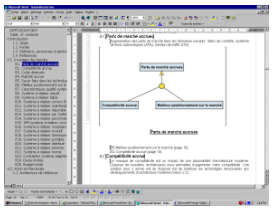






analysis
& validation

generation of
RE deliverables

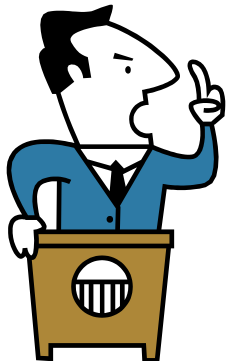


requirements document

Model: A definition

"A model is a representation in a certain medium of something in the same or another medium.
The model captures the important aspects of the thing being modeled and simplifies or omits the rest"

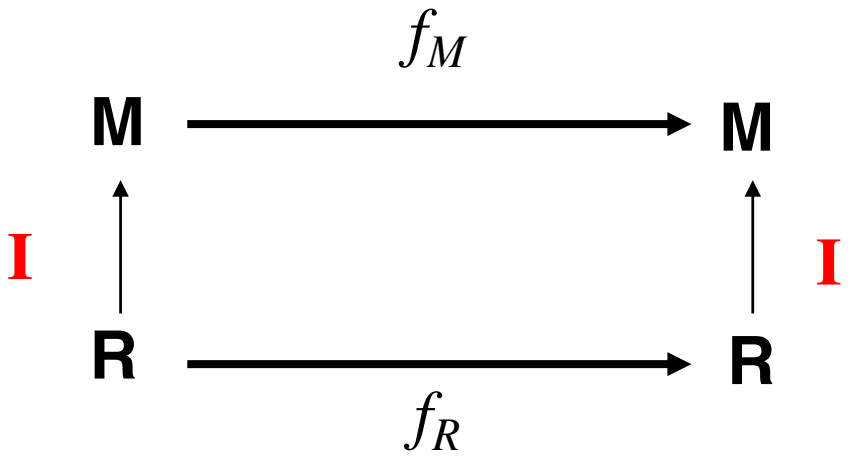
Grady Booch



Reality and Model

- Reality R: Real Things, People, Processes, Relationship
- Model M: Abstractions from (really existing or only thought of) things, people, processes and relationships between these abstractions

What is a "good" model?



- I is the mapping of real things in reality R to abstractions in the model M
  - also called **Interpretation**
- Relationships, which are valid in reality R, are also valid in model M
  - fR: relationship between real things in R
  - fM: relationship between abstractions in M

Why models?

- We use models
  - To abstract away from details in the reality, so we can draw complicated conclusions in the reality with simple steps in the model
  - To get insights into the past or presence
  - To make predictions about the future

Models of software systems

- A model of a software system is a representation of the system from a specific point of view
  - expressed in a modelling language
  - has semantics and notation
  - easier to use for a specific purpose than the final system
- Modeling language
  - artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules.

What are software models for

- Capture and precisely state requirements and domain knowledge
- Think about the design of a software system
- Generate usable work products
- Give a simplified view of complex systems
- Evaluate and simulate a complex system
- Generate potential configurations of systems
  - all consistent configurations should be possible
  - not always possible to represent all constraints in the model (model is an abstraction !)

Modelling issues

- Coherence
  - different views of the system must be coherent
- Variations in interpretation and ambiguity
  - define where different interpretations of the model are acceptable

What should we model?

- The objects and people that are of interest for the give problem
  - E.g. the aircraft and the sensors and actuators relevant to the braking system
- The relevant phenomena
  - Weels_turning, Reverse_enabled, …
- The goals, requirements, and domain assumptions

Which tools can we use for modeling?

- Any language (e.g., Italian, English, …)
  - Pros: simplicity of use
  - Cons:
    - high level of ambiguity,
    - it is easy to forget to include relevant information
- A formal language (e.g., first order logic, Alloy, Z, …)
  - Pros:
    - possibility to use some tool to support analysis and validation
    - the approach forces the user in specifying all relevant details
  - Cons: you need to be expert in the use of the language

Which tools can we use for modeling?

- A semi-formal language like UML
  - Pros:
    - simpler than a formal language
    - imposes some kind of structure in the models
  - Cons:
    - not amenable for automated analysis
    - some level of ambiguity
- A mixed approach
  - Use a semi-formal language for the basics
  - Comment and complement the semi-formal models with explanatory informal text
  - Use a formal language for the most critical parts

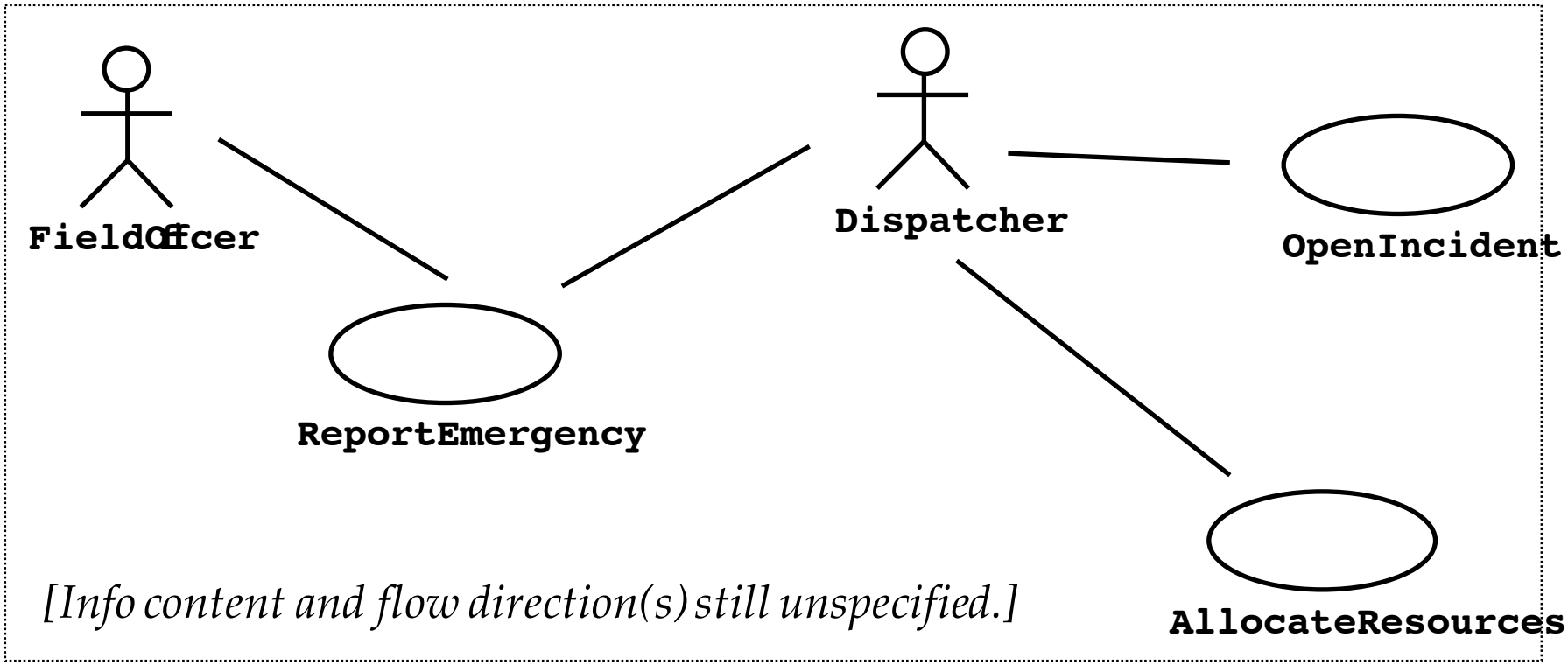
UML and requirement engineering

Use Cases

- A use case is a flow of events in the system, including interaction with actors
- It is initiated by an actor
- Each use case has a name
- Each use case has a termination condition

Use Case Model: The set of all use cases specifying the complete functionality of the system

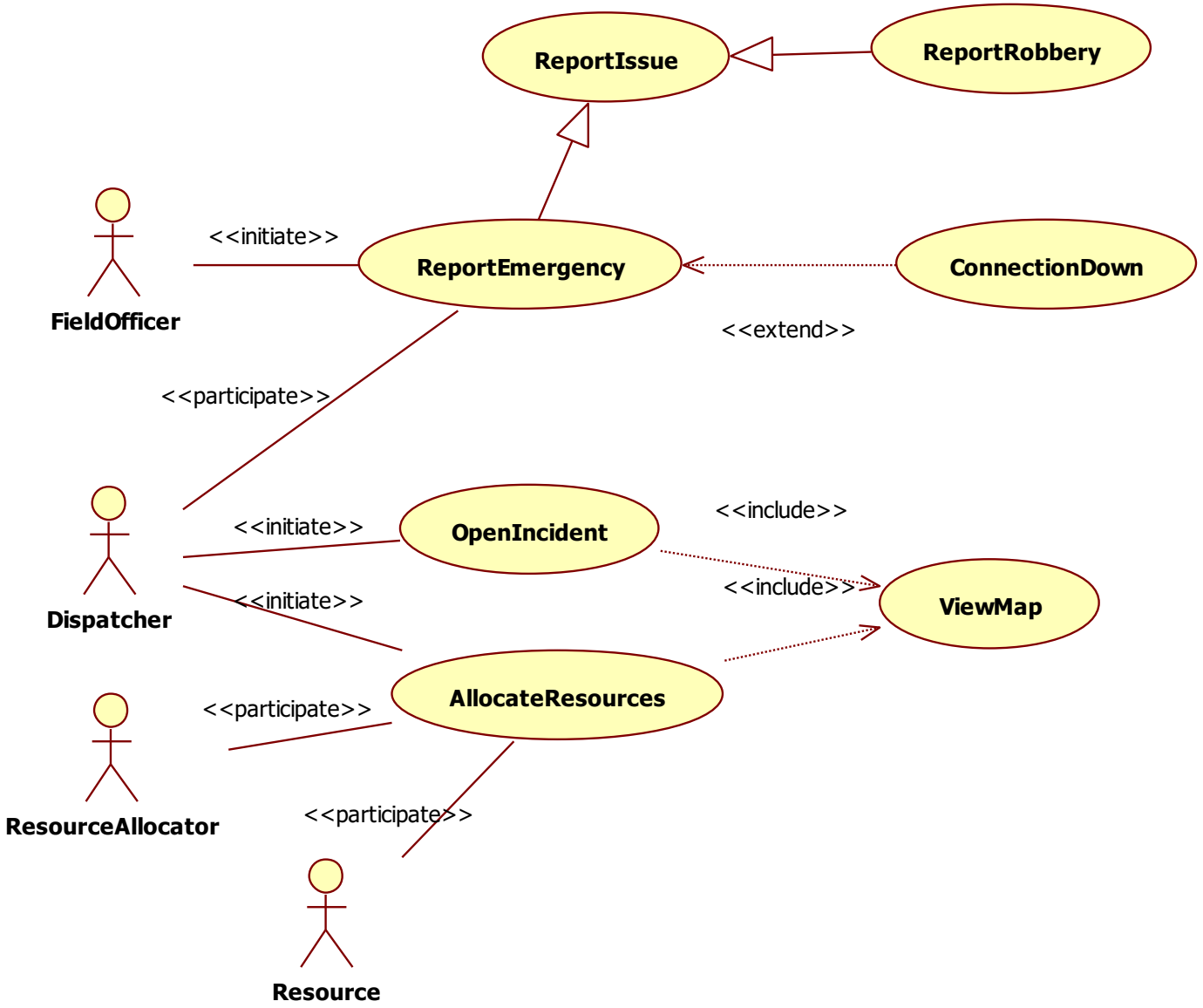
Example: Use Case Model for Incident Management



Use Case Associations

- A use case association is a relationship between use cases
- Important types of use case associations
  - Include
    - A use case uses another use case ("functional decomposition")
  - Extends
    - A use case extends another use case
  - Generalization
    - An abstract use case has several different specializations

Examples


ReportIssue
ReportRobbery
FieldOfficer
<<initiate>>
ReportEmergency
ConnectionDown
<<extend>>
<<participate>>
Dispatcher
<<initiate>>
OpenIncident
<<include>>
<<include>>
ViewMap
<<initiate>>
AllocateResources
<<participate>>
ResourceAllocator
<<participate>>
Resource

Requirements-level class diagrams

- are conceptual models for the application domain
  - different from OO software design models
- may model objects that will not be represented in the software-to-be
  - because this is not known at the start of the requirements modelling effort
- do not attach operations (methods) to objects
  - it's best to postpone this kind of decisions until software design

How do you find objects and classes?

- Analyze any description of the problem and application domain you may have
- Analyze your scenarios and user cases descriptions
- Finding objects is the central piece in object modeling
- A possible tool to use in the analysis
  - Abbott Textual Analysis, 1983, also called noun-verb analysis
    - Nouns are good candidates for classes
    - Verbs are good candidates for operations

Finding Participating Objects in Use Cases

- Pick a use case and look at its flow of events
  - Find terms that developers or users need to clarify in order to understand the flow of events
  - Look for recurring nouns (e.g., Incident),
  - Identify real world entities that the system needs to keep track of (e.g., FieldOfficer, Dispatcher, Resource),
  - Identify real world procedures that the system needs to keep track of (e.g., EmergencyOperationsPlan),
  - Identify data sources or sinks (e.g., Printer)
  - Identify interface artifacts (e.g., PoliceStation) [e.g., telecomm link?]
- Be prepared that some objects are still missing and need to be found
- Always use the user's terms

Object modeling: an Example

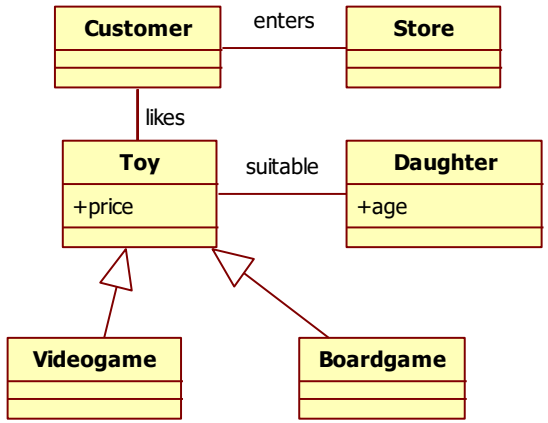
- Consider this text
  - *The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.*
  - *An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buys the game and leaves the store*

Textual Analysis using Abbot's technique of Natural Language Analysis [OOSE 5.4.1]

Example	*Grammatical construct*	*UML Component*
"Monopoly"	Concrete Person, Thing	Object
"toy"	noun	class
"3 years old"	Adjective	Attribute
"enters"	verb	Operation/Association
"depends on…."	Intransitive verb	Association
"is a" ,"either..or", "kind of…"	Classifying verb	Inheritance
"Has a ", "consists of"	Possessive Verb	Aggregation
"must be", "less than…"	modal Verb	Constraint

Generation of the class diagram for the example

- The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.
- An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buys the game and leaves the store



Object modeling and the Report Emergency use case

- Nouns are in red
- Verbs are in green

- The FieldOfficer activates the "Report Emergency" function of her terminal. FRIEND responds by presenting a form to the officer.
- The FieldOfficer fills the form, by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point, the Dispatcher is notified.
- The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report.
- The FieldOfficer receives the acknowledgment and the selected response.

Textual analysis

Term	Grammatical construct	UML Component
FieldOfficer	Noun	Class
FRIEND	Noun	This is the S2B object of our analysis. It does not have a corresponding Class.
Emergency	Noun	Class
Level, type, location	Noun	Attributes
Brief description	Noun + Adjective	Attribute
Response	Noun	Class
Dispatcher	Noun	Class
Incident	Noun synonym of emergency	No need for a new class
Emergency report	Noun	Class containing the attributes listed above

Textual analysis

Term	Grammatical construct	UML Component
Activates	Verb	Association/Operation
Responds	Verb	
Fills	Verb	Association/Operation
Describes	Verb	Association/Operation
Submits	Verb	Association/Operation
Is notified	Passive verb	
Creates	Verb	Association/Operation
Selects	Verb	Association/Operation
Acknowledges	Verb	Association/Operation
Receives	Verb	Event

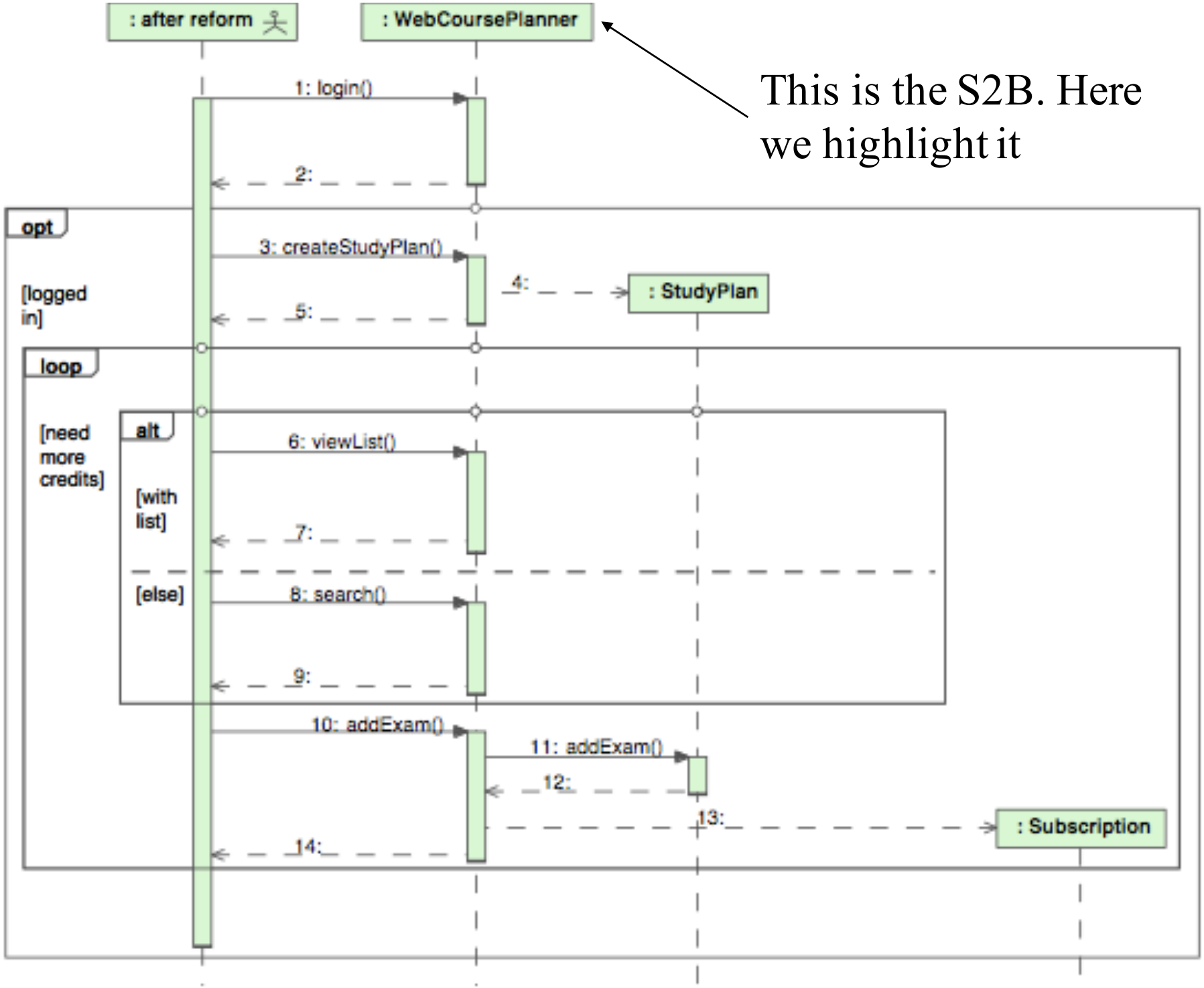
Now you can try to define the class diagram yourself

Dynamic Modeling

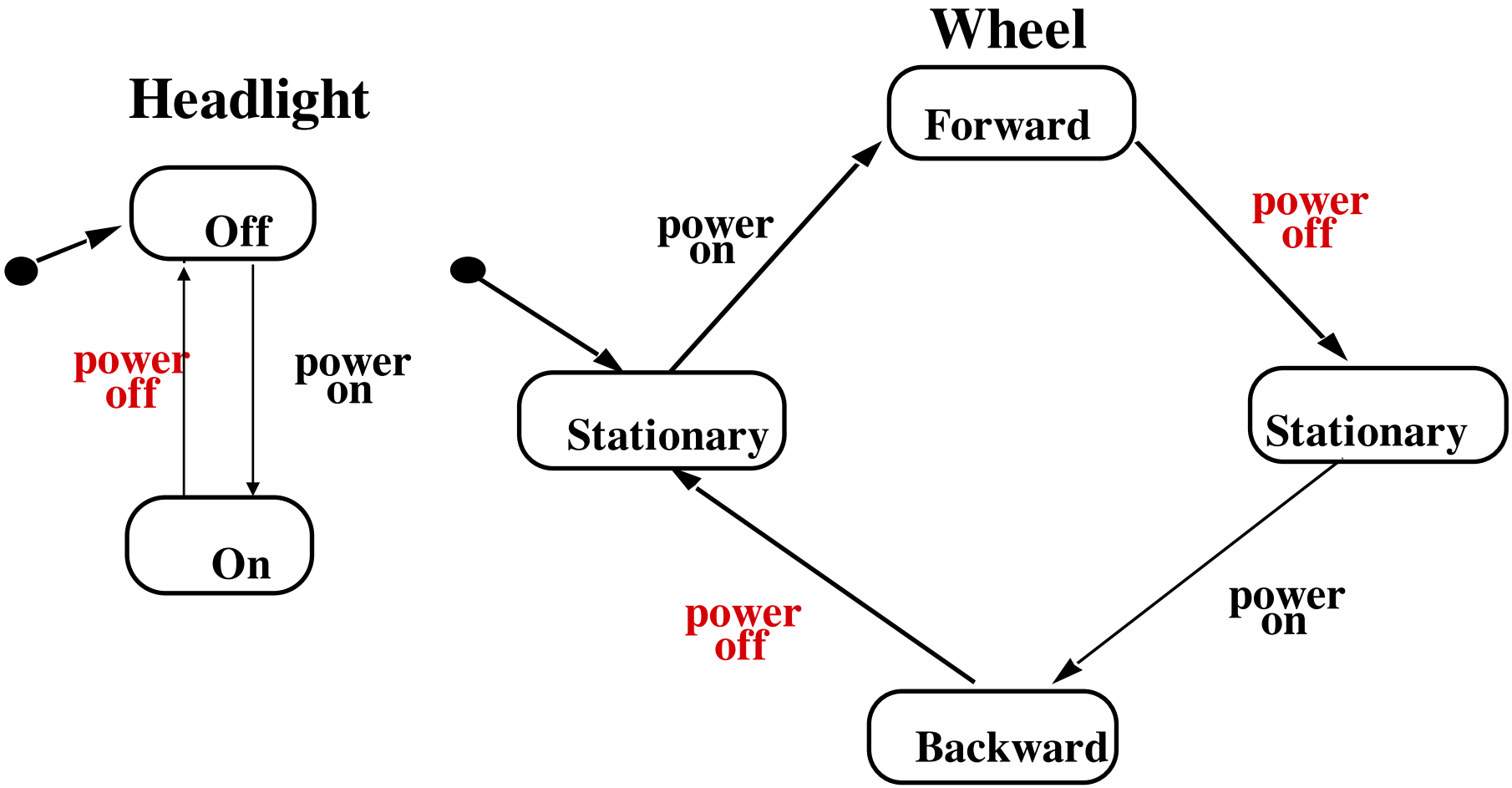
- Purpose:
  - Supply methods to model interactions, behaviors of participants and workflow
- How do we do this?
  - Start with use case or scenario
  - Model interaction between objects => sequence diagram
  - Model dynamic behavior of a single object => statechart diagram
  - Model workflow => activity diagram

Sequence Diagram

- From the flow of events in the use case or scenario, proceed to the sequence diagram
- A sequence diagram is a graphical description of objects participating in a use case or scenario using a DAG (directed acyclic graph) notation
- Relation to object identification:
  - Objects/classes have already been identified during object modeling
  - Other objects are identified as a result of dynamic modeling
- Heuristic:
  - A event always has a sender and a receiver.
  - The representation of the event is sometimes called a message
  - Find sender and receiver for each event => These are the objects participating in the use case


: after reform
: WebCoursePlanner
This is the S2B. Here we highlight it
1: login()
2:
opt
[logged in]
3: createStudyPlan()
4:
: StudyPlan
5:
loop
[need more credits]
alt
[with list]
6: viewList()
7:
[else]
8: search()
9:
10: addExam()
11: addExam()
12:
13:
: Subscription
14:

An example of statechart: toy car



State Chart Diagram vs Sequence Diagram

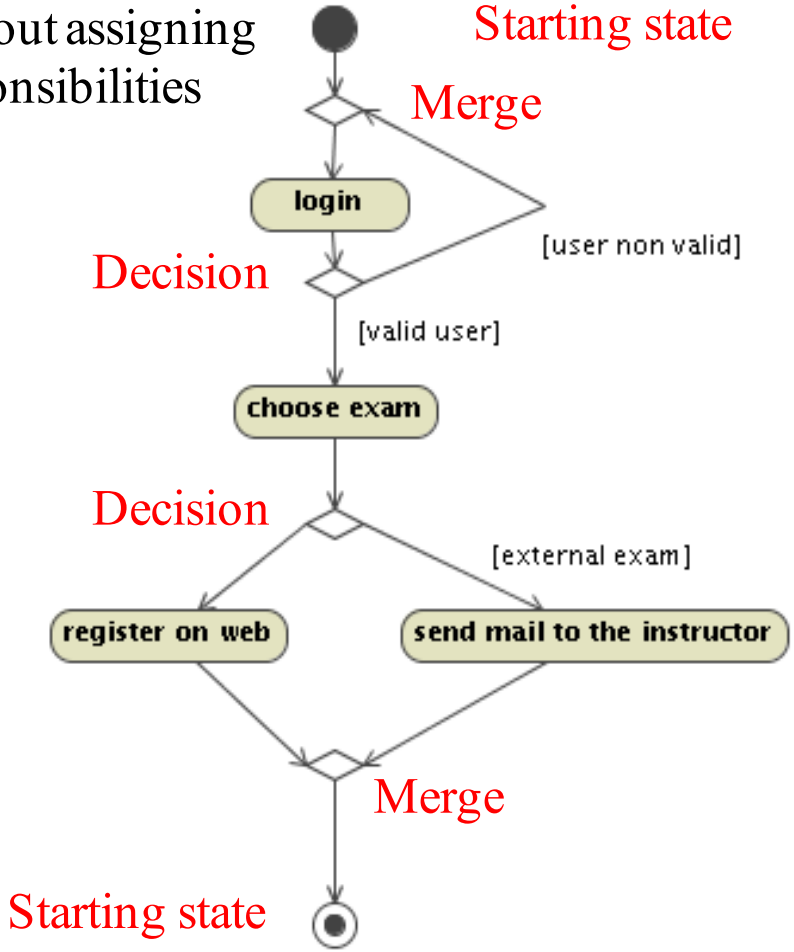
- State chart diagrams help to identify:
  - Changes in an individual object over time
- Sequence diagrams help to identify
  - The temporal relationship of between objects over time
  - Sequence of operations as a response to one or more events
- State diagrams are class-level documentation; sequence diagrams are instance-level documents.
  - We can infer many possible event-order-dependent behaviors from an STD;
  - A sequence diagram shows one behavioral history, for one specific ordering of input events.

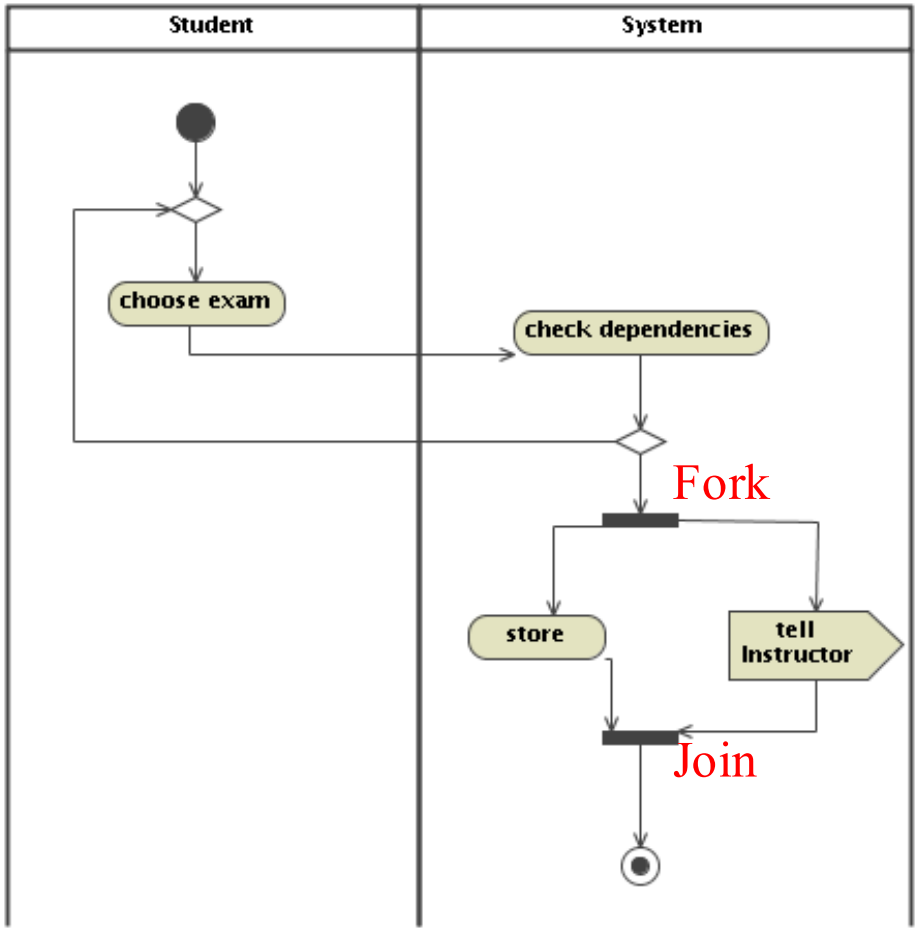
Statechart vs sequence diagram

Statechart	Sequence diagram
Focus on changes in an individual object over time	Focus on the interaction between objects over time
Class-level documentation	Instance-level documentation
We can infer many possible event and order-dependent behaviors	Shows one specific case or a subset of cases

Activity diagrams

Describes an activity without assigning responsibilities





Describes "register on the web" by highlighting the responsibilities of the student and of the system

Sequence diagram vs activity diagram

Sequence diagram	Activity diagram
Focus on object interaction	Focus on activities and flow of activities
Suited for describing an interaction protocol	Suited for describing a process

Practical Tips for Dynamic Modeling

- Construct dynamic models only for classes with significant dynamic behavior
  - Avoid “analysis paralysis”
- Consider only relevant attributes
  - Use abstraction if necessary [Partition the data state space into control states (equivalence classes of data state values that have the same dynamic behavior)]
- Look at the granularity of the application when deciding on actions and activities
- Reduce notational clutter
  - Try to put actions into state boxes (look for identical actions on events leading to the same state)

Summary: Requirements Analysis

- What are the transformations? → **Functional Modeling**
  - Create scenarios and use case diagrams
    - Talk to client, observe, get historical records, do thought experiments
- What is the structure of the world? → **Object Modeling**
  - Create class diagrams [static information models]
    - Identify objects.
    - What are the associations between them? What is their multiplicity?
    - What are the attributes of the objects?
    - What operations are defined on the objects?
  - Is there any state change in an object that is to be defined explicitly? If yes, create some statecharts [Dynamic class behavior models]

Summary: Requirements Analysis (cont)

- How is the expected interaction between the S2B and the environment?

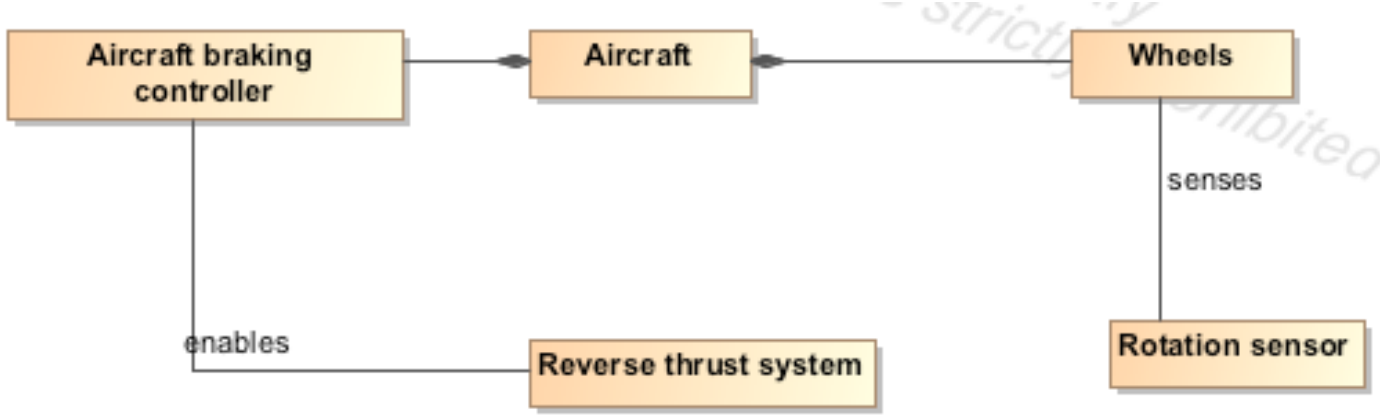
Dynamic Modeling

  - Create sequence diagrams from use cases [Dynamic object behavior instance examples]
    - Identify event senders and receivers
    - Show sequence of events exchanged between objects. Identify event dependencies and event concurrency.
  - Create activity diagrams when you want to highlight important processes [workflow]

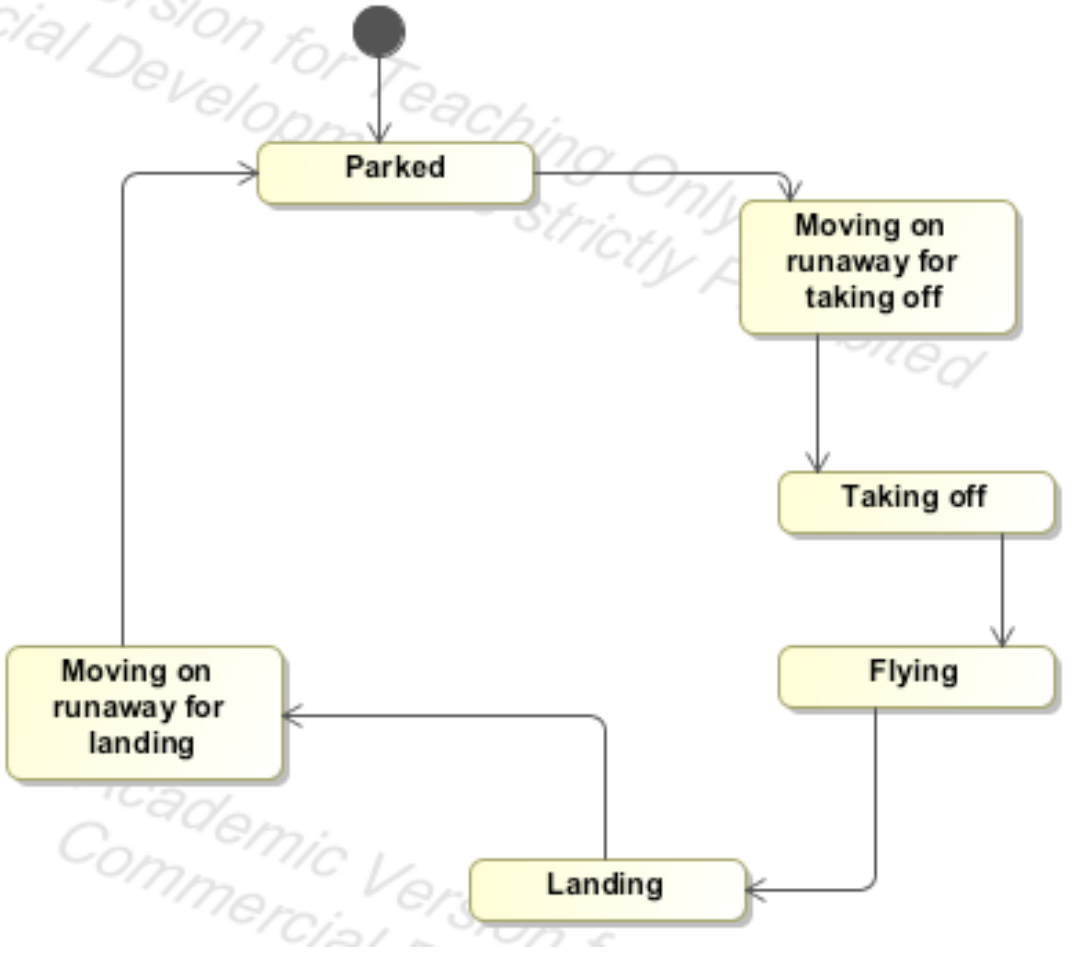
Modeling the Airbus braking logic with UML

What can we model?
What not?

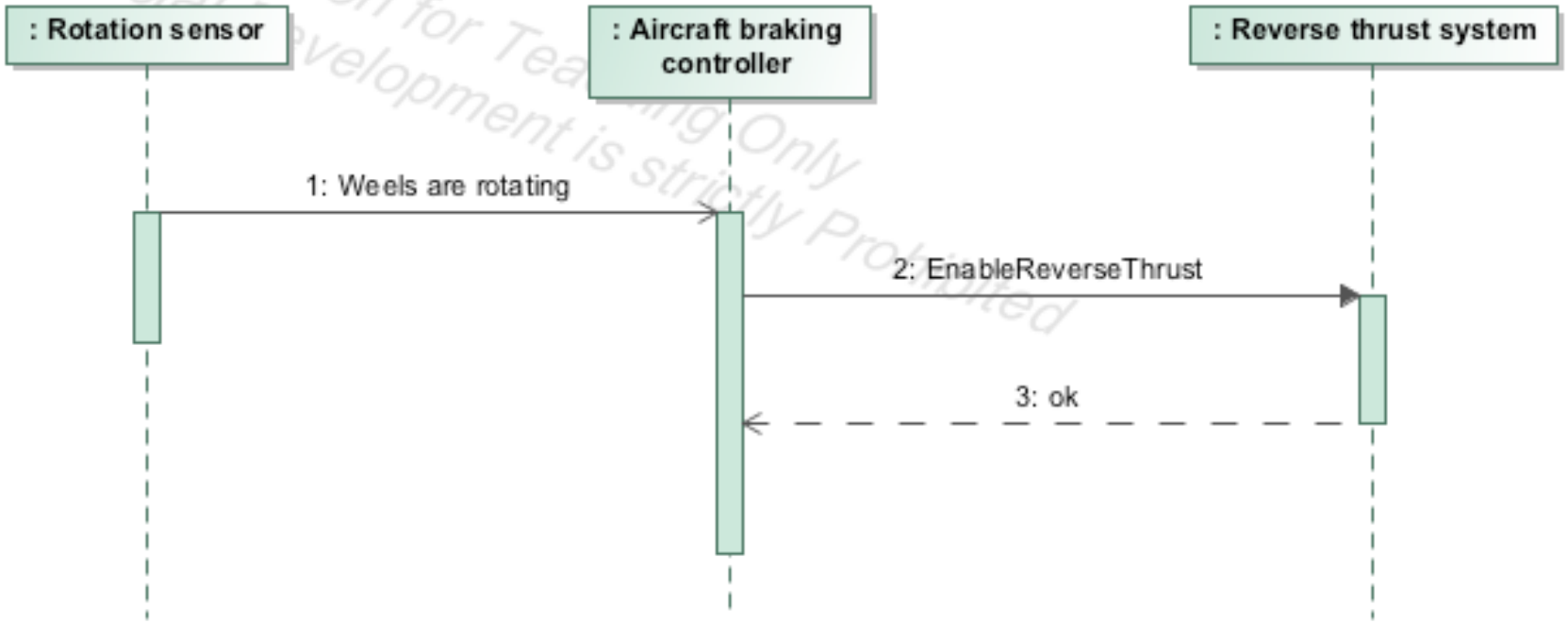
Domain modeling: main entities and relationships



Domain modeling: states of an aircraft



Dynamic behavior: enabling reverse thrust when wheels are rotating



It the UML spec complete?

- We have described all phenomena
  - Aircraft, wheels, sensor, reverse thrust system
- … and a use case EnablingReverseThrust
- Are we missing something?
- Are we representing goals, domain properties and requirements?
  - Goal
    - Reverse_enabled ⇔ Moving_on_runway
  - Domain properties
    - Wheel_pulses_on ⇔ Wheels_turning
    - Wheels_turning ⇔ Moving_on_runway
  - Requirements
    - Reverse_enabled ⇔ Wheels_pulses_on

It the UML spec complete?

- Pure UML does not help us in expressing assertions
- We can complement its usage with
  - Some formal or informal description of these assertions

Politecnico
di Milano

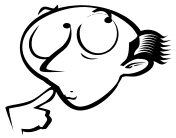
Requirements Analysis and Specification Document (RASD)

RASD

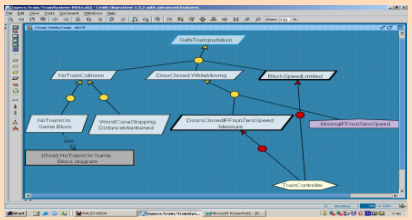
existing systems

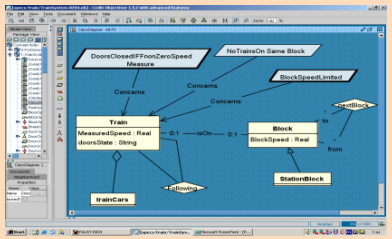
documents

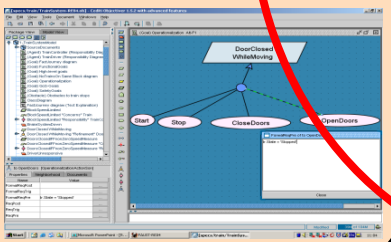
elicitation & modelling



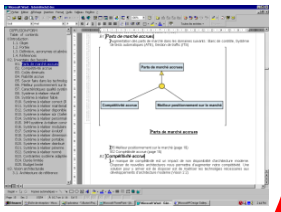
Requirements Models







generation of RE deliverables



requirements document

analysis & validation

Purposes of the RASD

- Communicates an understanding of the requirements
  - explains both the application domain and the system to be developed
- Contractual
  - may be legally binding!
- Baseline for project planning and estimation (size, cost, schedule)
- Baseline for software evaluation
  - supports system testing, verification and validation activities
  - should contain enough information to verify whether the delivered system meets requirements
- Baseline for change control
  - requirements change, software evolves

(slide: Steve Easterbrook)

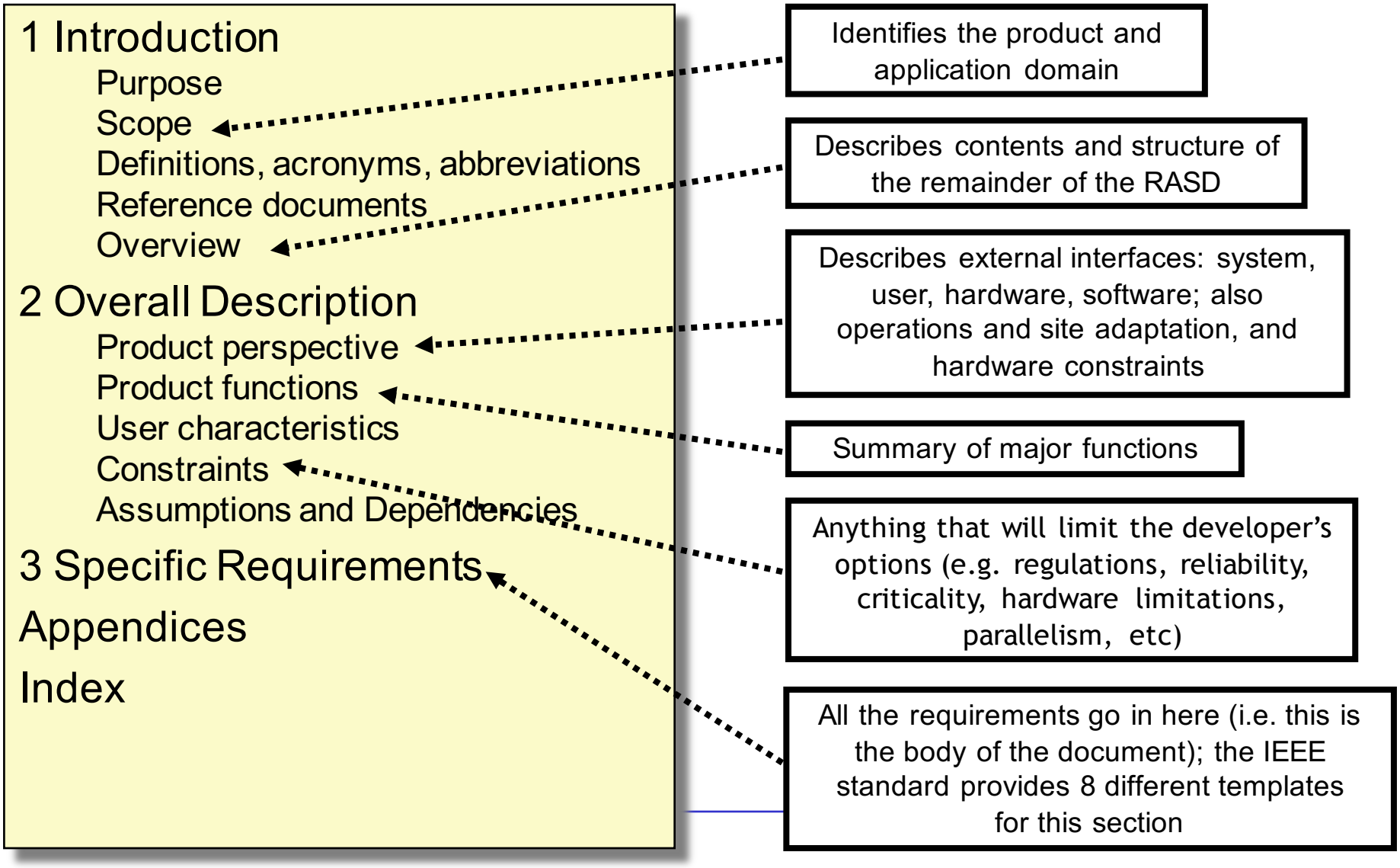
Audience of the RASD

- Costumers & Users
  - most interested in validating system goals and high-level description of functionalities
  - not generally interested in detailed software requirements
- Systems Analysts, Requirements Analysts
  - write various specifications of other systems that inter-relate
- Developers, Programmers
  - have to implement the requirements
- Testers
  - determine that the requirements have been met
- Project Managers
  - measure and control the analysis and development processes

(slide: Steve Easterbrook)

IEEE Standard for RASD

Source: *Adapted from ISO/IEC/IEEE 29148 dated Dec 2011*



(slide: Steve Easterbrook)

IEEE STD Section 3 (example)

Source: *Adapted from ISO/IEC/IEEE 29148 dated Dec 2011*

3.1 External Interface Requirements
- 3.1.1 User Interfaces
- 3.1.2 Hardware Interfaces
- 3.1.3 Software Interfaces
- 3.1.4 Communication Interfaces

3.2 Functional Requirements

this section organized by mode, user class, feature, etc. For example:

3.2.1 User Class 1

3.2.1.1 Functional Requirement 1.1

…

3.2.2 User Class 2

3.2.1.1 Functional Requirement 1.1

…

…

3.3 Performance Requirements

3.4 Design Constraints

3.4.1 Standards compliance

3.4.2 Hardware limitations

etc.

3.5 Software System Attributes
- 3.5.1 Reliability
- 3.5.2 Availability
- 3.5.3 Security
- 3.5.4 Maintainability
- 3.5.5 Portability

3.6 Other Requirements

(slide: Steve Easterbrook)

Target qualities for a RASD (1)

- Completeness
  - w.r.t. goals: the requirements are sufficient to satisfy the goals under given domain assumptions

    Req and Dom |= Goals
    - all Goals have been correctly identified, including all relevant quality goals
    - Dom represent valid assumptions; incidental and malicious behaviours have been anticipated
  - w.r.t. inputs: the required software behaviour is specified for all possible inputs
  - Structural completeness: no TBDs

(slide: adapted from A. van Lamsweerde)

Target qualities for a RASD (2)

- Pertinence
  - each requirement or domain assumption is needed for the satisfaction of some goal
  - each goal is truly needed by the stakeholders
  - the RASD does not contain items that are unrelated to the definition of requirements (e.g. design or implementation decisions)
- Consistency
  - no contradiction in formulation of goals, requirements, and assumptions

(slide: adapted from A. van Lamsweerde)

Target qualities for a RASD (3)

- Unambiguity
  - unambiguous vocabulary: every term is defined and used consistently
  - unambiguous assertions: goals, requirements and assumption must be stated clearly in a way that precludes different interpretations
  - verifiability: a process exists to test satisfaction of each requirement
  - unambiguous responsibilities: the split of responsibilities between the software-to-be and its environment must be clearly indicated

(slide: adapted from A. van Lamsweerde)

Target qualities for a RASD (4)

- Feasibility
  - the goals and requirements must be realisable within the assigned budget and schedules
- Comprehensibility
  - must be comprehensible by all in the target audience
- Good Structuring
  - e.g. highlights links between goals, reqts and assumptions
  - every item must be defined before it is used
- Modifiability
  - must be easy to adapt, extend or contract through local modifications
  - impact of modifying an item should be easy to assess

(slide: adapted from A. van Lamsweerde)

Target qualities for a RASD (5)

- Traceability
  - must indicate sources of goals, requirements and assumptions
  - must link requirements and assumptions to underlying goals
  - facilitates referencing of requirements in future documentation (design, test cases, etc.)

(slide: adapted from A. van Lamsweerde)

RASD: In which sections do we include all we have learnt about requirements?

- The RASD does not necessarily follow the order of our mental process to the requirements
- Section 1
  - Purpose part -> goals
  - Scope part -> analysis of the world and of the shared phenomena
- Section 2
  - Product perspective -> Further details on the shared phenomena and a domain model (class diagrams, and statecharts)
  - Product functions -> Requirements
  - User characteristics -> anything that is relevant to clarify their needs
  - Assumptions and dependencies -> Domain assumptions

RASD: In which sections do we include all we have learnt about requirements?

- Section 3
  - More details on all aspects in Section 2 if they can be useful for the development team
  - Section 3.2 -> Definition of use case diagrams, use cases and associated sequence/activity diagrams

A note on traceability

- Use cases are related to some requirements. Keep track of this relationship through proper identifiers
  - E.g. RE.3 is associated to UC.3.1 and UC.3.2
- We may also have use cases that refer to multiple requirements
  - E.g., UC.3.1 may refer also to RE.2 (even thought the main relationship is with RE.3)
  - Make this explicit in the presentation
    - E.g., you could build a traceability matrix

Traceability matrix

Raw ID	Goal ID	Req ID	Use Case ID	Comments
r1	G.1	RE.3	UC.3.1	
r2	G.1	RE.2	UC.3.1	

- This may grow during the development process, example:

Raw ID	Goal ID	Req ID	Use Case ID	Test case ID	Comments
r1	G.1	RE.3	UC.3.1	TC.3.1.1	
r2	G.1	RE.2	UC.3.1		

References

- M. Jackson, P. Zave, "Deriving Specifications from Requirements: An Example", Proceedings of ICSE 95, 1995
- M. Jackson, P. Zave, "Four Dark Corners of Requirements Engineering",TOSEM, 1997
- B. Nuseibeh, S. Easterbrook, "Requirements Engineering: A Roadmap", Proceedings ICSE 2000
- A. van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley and Sons, 2009
- M. Jackson, Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices, ACM Press Books, 1995
- S. Robertson and J. Robertson, Mastering the Requirements Process, Addison Wesley, 1999
- Requirements Engineering Specialist Group of the British Computer Society http://www.resg.org.uk/
- B. Bruegge & A.H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns, and Java, 2nd Edition, Prentice Hall, Upper Saddle River, NJ, September 25, 2003
- Slides by Emmanuel Letier UCL.