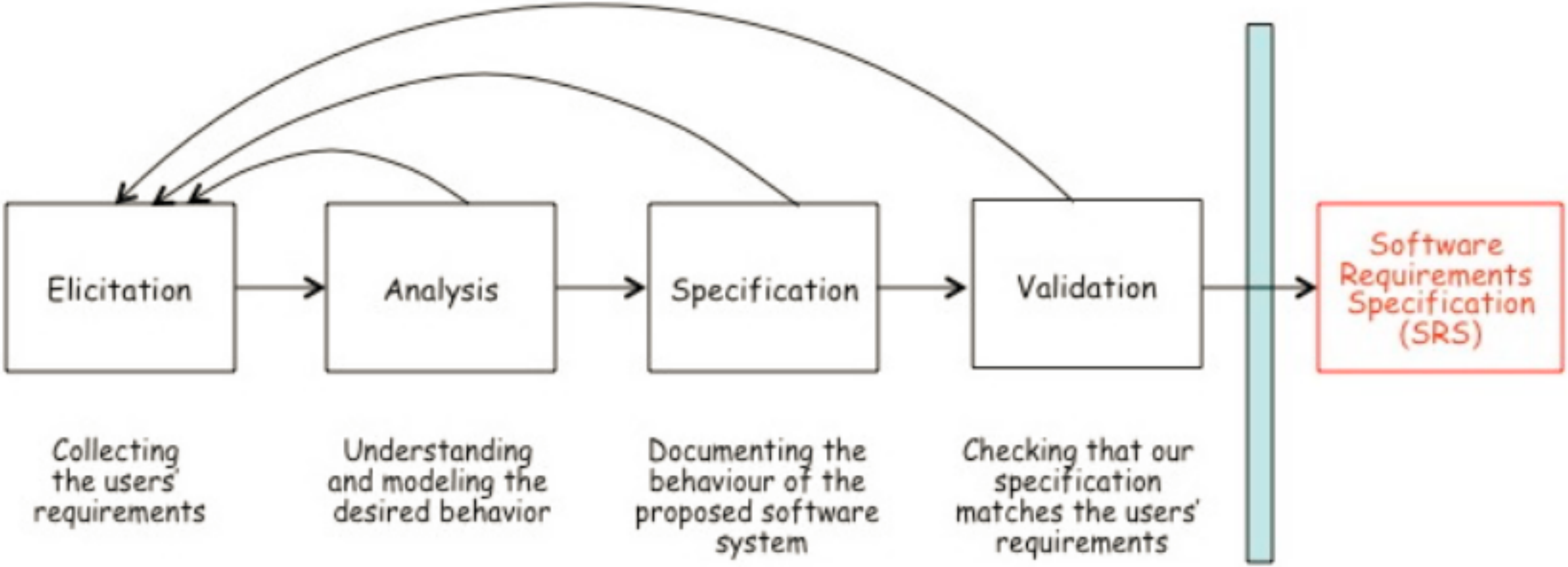
CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Documentation

Requirements Engineering Process


Elicitation
Analysis
Specification
Validation
Software Requirements Specification (SRS)
Collecting the users' requirements
Understanding and modeling the desired behavior
Documenting the behaviour of the proposed software system
Checking that our specification matches the users' requirements

SRS – Software Requirements Specification

- States the functions and capabilities a software system must provide, its characteristics, and the constraints it must respect.

- Should describe as thoroughly as necessary the system's behaviours under various conditions, as well as desired system qualities such as performance, security, and usability.

- Is the basis for subsequent project planning, design, and coding and the foundation for system testing and user documentation.

- It should **not** contain design, construction, testing, or project management details other than known design and implementation constraints.

What is it useful for?

- Customers, the marketing department, and sales staff need to know what product they can expect to be delivered.
- Project managers base their estimates of schedule, effort, and resources on the requirements.
- Software development teams need to know what to build.
- Testers use it to develop requirements-based tests, test plans, and test procedures.
- Maintenance and support staff use it to understand what each part of the product is supposed to do.

What is it useful for?

- Documentation writers base user manuals and help screens on the SRS and the user interface design.
- Training personnel use the SRS and user documentation to develop educational materials.
- Legal staff ensures that the requirements comply with applicable laws and regulations.
- Subcontractors base their work on, and can be legally held to, the specified requirements.

Organize, Organize, Organize

- Use an appropriate template to organize all the necessary information.
- Label and style sections, subsections, and individual requirements consistently.
- Use visual emphasis (bold, underline, italics, colour, and fonts) consistently and judiciously. Remember that colour highlighting might not be visible to people with colour blindness or when printed in grayscale.
- Create a table of contents to help readers find the necessary information.
- Number all figures and tables, give them captions and refer to them by number.

Organize, Organize, Organize

- If you are storing requirements in a document, define your word processor’s cross-reference facility rather than a hard-coded page or section numbers to refer to other locations within a document.
- If you use documents, define hyperlinks to let the reader jump to related sections in the SRS or other files.
- If you store requirements in a tool, use links to let the reader navigate related information.
- Include visual representations of information when possible to facilitate understanding.
- Enlist a skilled editor to ensure the document is coherent and uses a consistent vocabulary and layout.

A Software Requirements Specification Template

- 1. Introduction
  - 1.1.Purpose
  - 1.2.Scope
  - 1.3.Definitions, Acronyms and Abbreviations
  - 1.4.References
  - 1.5.Overview
- 2. Overall Description
  - 2.1.Product Perspective
  - 2.2.Product Functions
  - 2.3.User Characteristics
  - 2.4.Constraints
  - 2.5.Assumptions
  - 2.6.Apportioning of Requirements
- 3. Specific Requirements
  - 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.System Features
    - 3.2.1.[Feature 1]
      - 3.2.1.1.Purpose
      - 3.2.1.2.Response Sequence
      - 3.2.1.3.Functional Requirements
  - 3.3.Performance Requirements
  - 3.4.Design Constraints
  - 3.5.Software System Attributes
  - 3.6.Other Requirements
- Appendix
- Index

Note

All examples are related to the same system.

The set of examples in each section is not complete. Make sure you write a complete set of requirements for each section.

1. Introduction

The introduction presents an overview to help the reader understand how the SRS is organized and how to use it.

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms and Abbreviations

1.4 References

1.5 Overview

1.1 Purpose

- Identify the product or application whose requirements are specified in this document, including the revision or release number.
- If this SRS pertains to only part of a complex system, identify that portion or subsystem.
- Describe the different types of readers that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers.

Example

1.1 Purpose

This SRS describes the functional and non-functional requirements for software release 1.0 of the web-based e-catalog for cataloging physical and virtual items that are to be filed or stored away. This document is intended to be used by the project team members who will implement and verify the correct functioning of the system. All requirements listed are committed for release 1.0.

1.2 Scope

- Provide a short description of the software being specified and its purpose.
- Relate the software to the user or corporate goals, business objectives, and strategies.
- If a separate vision and scope or similar document are available, refer to it rather than duplicate its contents here.
- An SRS that specifies an incremental release of an evolving product should contain its scope statement as a subset of the long-term strategic product vision.
- You might provide a high-level summary of the release's major features or the significant functions it performs.

Example

1.2 Project Scope

The e-catalog will permit client administrators to build a catalogue of physical and virtual items for registered users to discover and check in/out. Physical items must be checked out from one of the item processing facilities, and once it is time to check in, the item can be mailed in or dropped off at the facility.

The e-catalog uses electronic IDs as barcodes to identify and monitor item status. Administrators can sort items into categories, and registered users can associate customizable labels with items and containers for easy identification. This web-based e-catalog, which can be used on desktop and mobile, provides an efficient and convenient way to retrieve and catalogue documents and items. A detailed project description is available in the E-catalog Product Vision and Scope Document [1].

1.3 Definitions, Acronyms and Abbreviations

- Describe any standards or typographical conventions used, including the meaning of specific text styles, highlighting, or notations.
- If you are manually labelling requirements, you might specify the format here for anyone who needs to add one later.
- Define any specialized terms that a reader needs to know to understand the SRS, including acronyms and abbreviations.
- Spell out each acronym and provide its definition.
- Consider building a reusable enterprise-level dictionary that spans multiple projects and incorporates by reference any terms that pertain to this project.
- Each SRS would then define only those terms specific to an individual project that do not appear in the enterprise-level dictionary.

Example

1.3 Definitions, Acronyms and Abbreviations

This document uses the following conventions:

1. Features will be documented with a description, a priority, and a set of functional requirements.

2. New features or features with new requirements will be denoted using the notation: (NEW).

3. “User” in System Features refers to the administrator and registered users.

4. If a data type is stated as String (alpha), this means only strings consisting of alphabet characters are permitted.

Example

5. If a data type is stated as a String (alphanumeric), this means strings consisting of the alphabet and numeric characters are permitted.

6. If a data type is stated as String, this means all string values are permitted unless there are additional constraints specified.

7. If no length is permitted for a data type, there is no length restriction.

8. In the State Machine Diagram Descriptions in the Appendix, events are bolded and grouped by large grouping states, and conditions are italicized.

1.4 References

- List any documents or other resources to which this SRS refers. Include hyperlinks to them if they are in a persistent location.
- These might include user interface style guides, contracts, standards, system requirements specifications, interface specifications, or the SRS for a related product.
- Provide enough information so the reader can access each reference, including its title, author, version number, date, source, storage location, or URL.

Example:

1.4 References

1. Wiegers, Karl. *Cafeteria Ordering System Vision and Scope Document*, *www.processimpact.com/projects/COS/COS Vision and Scope.docx*

2. Beatty, Joy. *Process Impact Intranet Development Standard, Version 1.3*, *www.processimpact.com/corporate/standards/PI Intranet Development Standard.pdf*

3. Rath, Andrew. *Process Impact Internet Application User Interface Standard, Version 2.0, www.processimpact.com/corporate/standards/PI Internet UI Standard.pdf*

1.5 Overview

This subsection should

a) Describe what the rest of the SRS contains;

b) Explain how the SRS is organized.

Example

1.5 Overview

The remainder of this SRS describes the system features and requirements. This SRS document is organized as follows. Section 2 includes…

2. Overall Description

This section presents a high-level overview of the product, the environment in which it will be used, the anticipated users, and known constraints, assumptions, and dependencies.

2.1 Product Perspective

2.2 Product Functions

2.3 User Characteristics

2.4 Constraints

2.5 Assumptions and Dependencies

2.6 Apportioning of Requirements

2.1 Product Perspective

- Describe the product’s context and origin. Is it the next member of a growing product line, the next version of a mature system, a replacement for an existing application, or an entirely new product?
- If this SRS defines a component of a larger system, state how this software relates to the overall system and identify major interfaces between the two.
- Consider including visual models such as a context diagram to show the product’s relationship to other systems.

Example

2.1 Product Perspective

The e-catalog system is a new, self-contained web-based database system implemented for the client company. The client company will sell it to other companies to replace the latter’s manual task of cataloging physical and virtual items. The context diagram in Figure 2.1 illustrates the external entities and system interfaces for release 1.0. The system is expected to evolve over several releases, ultimately allowing one to check the location of an item via virtual reality before going there.

Example

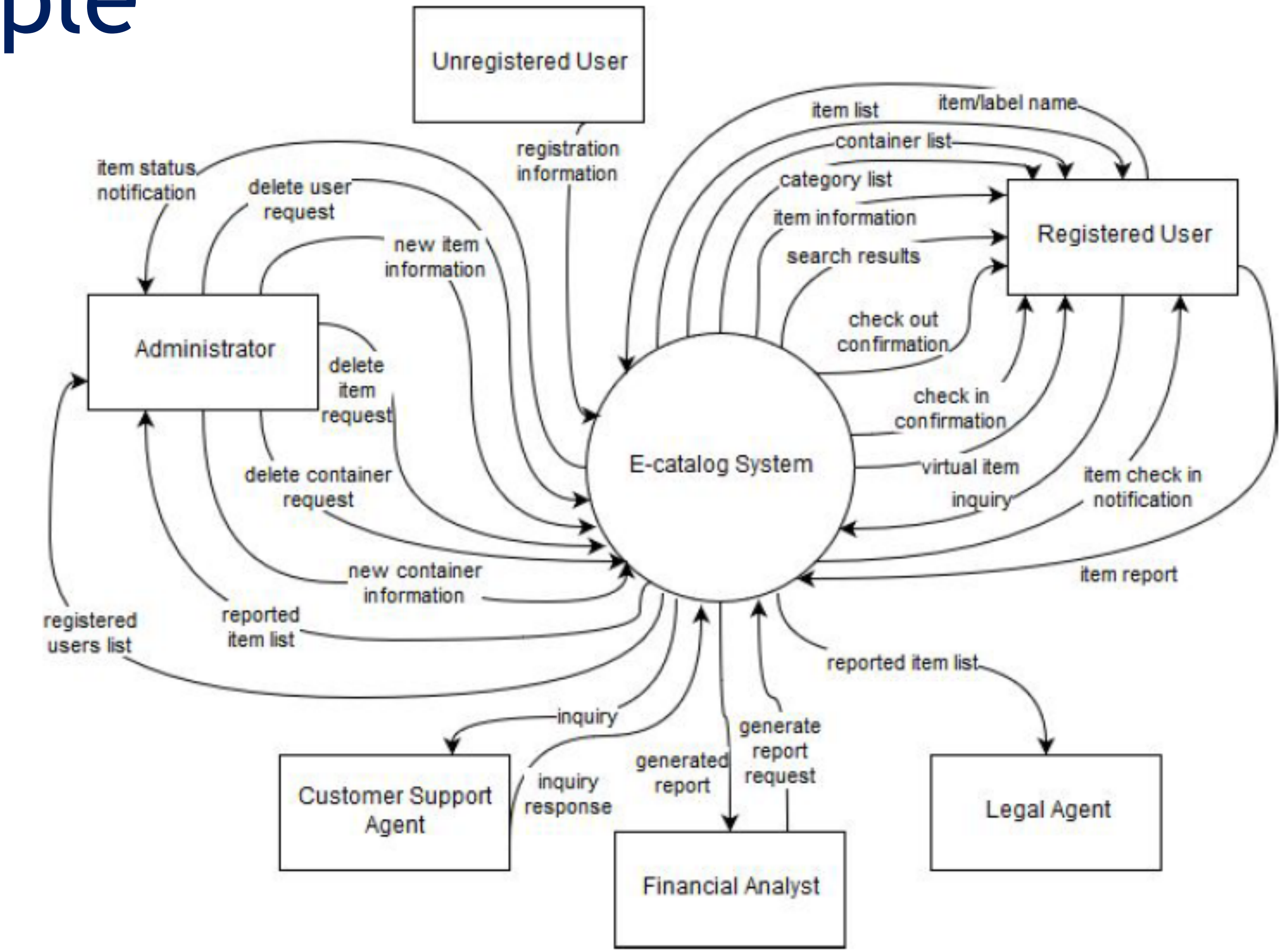


Figure 2.1: Context diagram for release 1.0 of e-catalog

2.2 Product Functions

This subsection of the SRS should provide a summary of the major functions that the software will perform.

Example

2.2 Product Functions

The software provides customer account maintenance and invoice preparation. In addition,…

2.3 User Characteristics

- Identify the various user classes you anticipate will use this product and describe their pertinent characteristics.
- Some requirements might pertain only to specific user classes. Identify the favoured user classes. User classes represent a subset of the stakeholders described in the vision and scope document.
- User class descriptions are a reusable resource.
- If a master user class catalogue is available, you can incorporate user class descriptions by simply pointing to them in the catalogue instead of duplicating information here.

Example

2.3 User Characteristics

Registered users are the target audience of this system, but the system can be targeted toward other users as well. Table 2.2 below discusses the user classes of the e-catalog system and their characteristics.

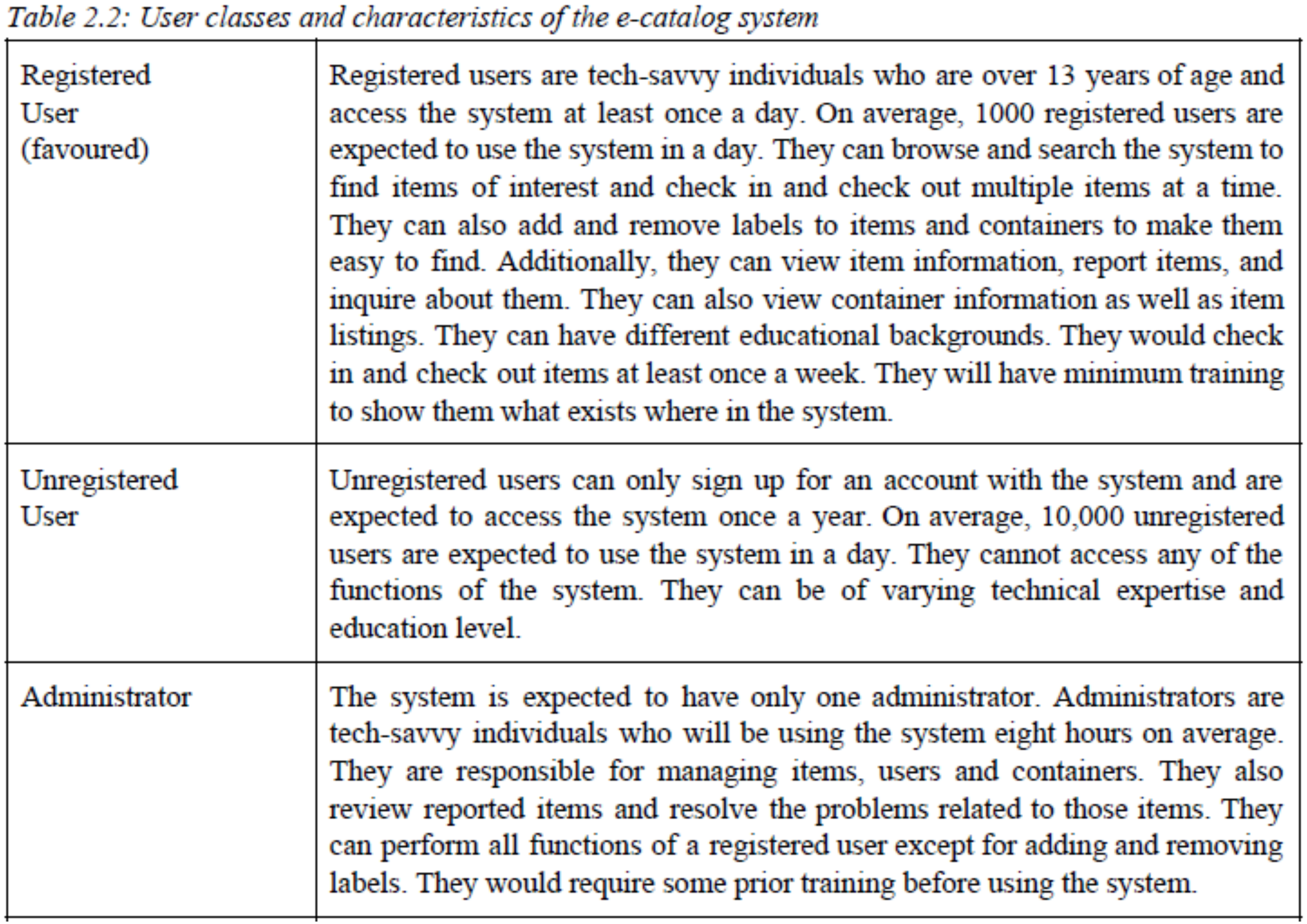
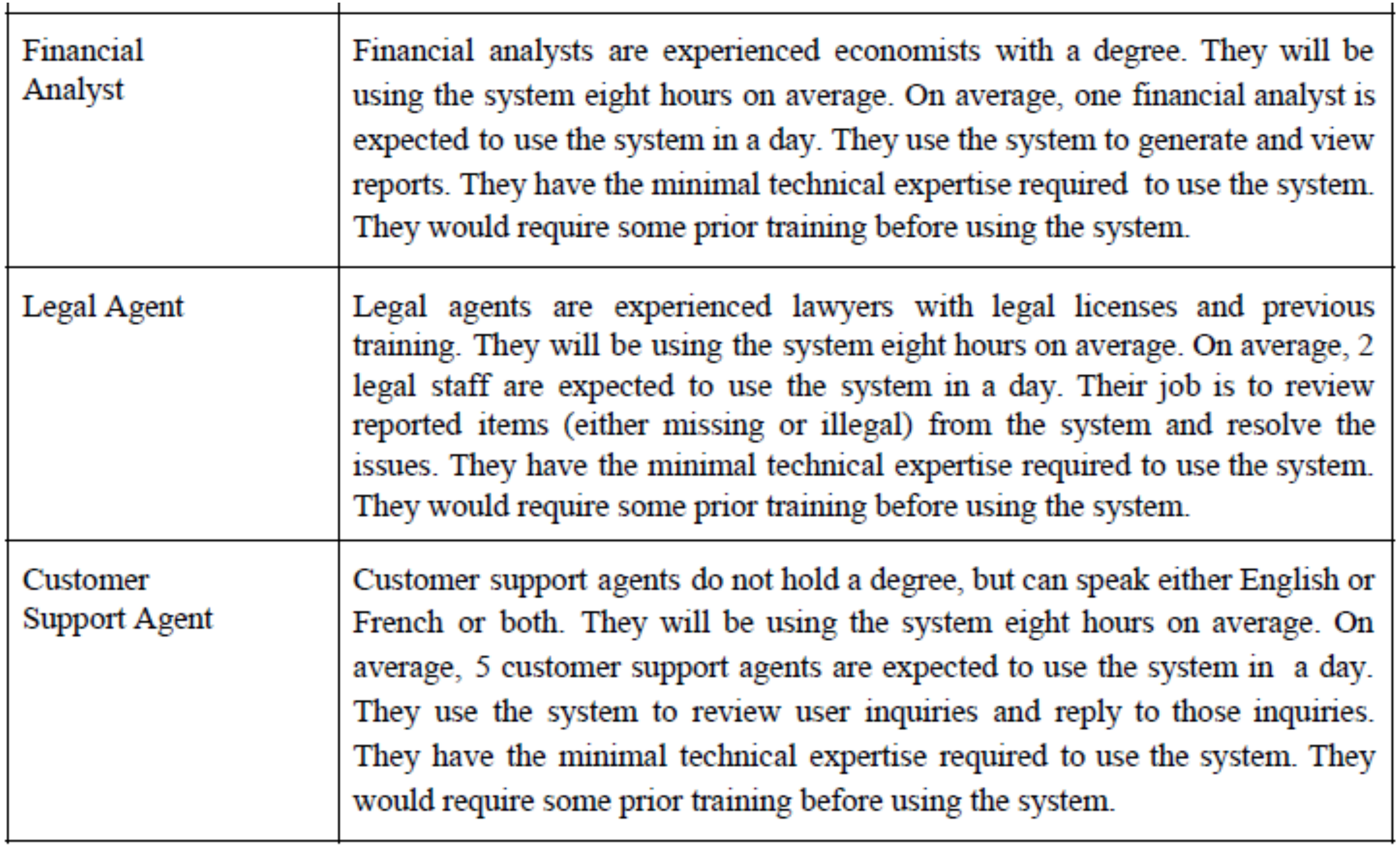


Table 2.2: User classes and characteristics of the e-catalog system

Registered User (favoured)	Registered users are tech-savvy individuals who are over 13 years of age and access the system at least once a day. On average, 1000 registered users are expected to use the system in a day. They can browse and search the system to find items of interest and check in and check out multiple items at a time. They can also add and remove labels to items and containers to make them easy to find. Additionally, they can view item information, report items, and inquire about them. They can also view container information as well as item listings. They can have different educational backgrounds. They would check in and check out items at least once a week. They will have minimum training to show them what exists where in the system.
Unregistered User	Unregistered users can only sign up for an account with the system and are expected to access the system once a year. On average, 10,000 unregistered users are expected to use the system in a day. They cannot access any of the functions of the system. They can be of varying technical expertise and education level.
Administrator	The system is expected to have only one administrator. Administrators are tech-savvy individuals who will be using the system eight hours on average. They are responsible for managing items, users and containers. They also review reported items and resolve the problems related to those items. They can perform all functions of a registered user except for adding and removing labels. They would require some prior training before using the system.



Financial Analyst	Financial analysts are experienced economists with a degree. They will be using the system eight hours on average. On average, one financial analyst is expected to use the system in a day. They use the system to generate and view reports. They have the minimal technical expertise required to use the system. They would require some prior training before using the system.
Legal Agent	Legal agents are experienced lawyers with legal licenses and previous training. They will be using the system eight hours on average. On average, 2 legal staff are expected to use the system in a day. Their job is to review reported items (either missing or illegal) from the system and resolve the issues. They have the minimal technical expertise required to use the system. They would require some prior training before using the system.
Customer Support Agent	Customer support agents do not hold a degree, but can speak either English or French or both. They will be using the system eight hours on average. On average, 5 customer support agents are expected to use the system in a day. They use the system to review user inquiries and reply to those inquiries. They have the minimal technical expertise required to use the system. They would require some prior training before using the system.

2.4 Constraints

This subsection of the SRS should provide a general description of any other items that will limit the developer’s options. These include:

a) Regulatory policies;

b) Hardware limitations (e.g., signal timing requirements)

c) Interface to other applications;

d) Parallel operation;

e) Audit functions;

f) Control functions;

g) Higher-order language requirements;

h) Signal handshake protocols (e.g., XON-XOFF, ACK-NACK);

i) Reliability requirements;

j) Criticality of the application;

k) Safety and security considerations.

Example

2.4 Constraints

The software must follow the regulatory policies listed below:

Because of the possible harm that the software can cause to human life, the following safety measures need to be incorporated at the development level:

2.5 Assumptions and Dependencies

- An *assumption* is a statement that is believed to be true in the absence of proof or definitive knowledge.
- Problems can arise if assumptions are incorrect, obsolete, not shared, or change so that certain assumptions will translate into project risks.
- One SRS reader might assume that the product will conform to a particular user interface convention, whereas another might assume something different.

- A developer might assume that a particular set of functions will be custom-written for this application. In contrast, the business analyst might think they will be reused from a previous project, and the project manager might expect to procure a commercial function library.
- The assumptions to include here are those related to system functionality; business-related assumptions appear in the vision and scope document
- Identify any *dependencies* the project or system being built has on external factors or components outside its control. For instance, if Microsoft .NET Framework 4.5 or a more recent version must be installed before your product can run, that’s a dependency.

Example

2.5 Assumptions and Dependencies

- AS-1: When registered users select to check in an item, either by drop-off or delivery, they do so immediately.
- AS-2: Permissions are only checked when a registered user attempts to check out an item.
- AS-3: Registered users have to check in an item to the facility they had checked it out from.
- DE-1: The check in and check out operations of the e-catalog system is dependent on barcode readers.

2.6 Apportioning of Requirements

This subsection of the SRS should identify requirements that may be delayed until future versions of the system.

Example

2.6 Apportioning of Requirements

The software is planned to include a major user viewing functionality after months of its release.

Payment using biometric confirmation is an expected functionality for future releases of the software.

3. Specific Requirements

- The SRS template discussed shows functional requirements organized by system feature, which is just one possible way to arrange them.
- Other organizational options include arranging functional requirements by functional area, process flow, use case, mode of operation, user class, stimulus, and response.
- Hierarchical combinations of these elements, such as use cases within user classes, are also possible.
- There is no single correct choice; select a method of organization that makes it easy for readers to understand the product’s intended capabilities.

3.1 External Interface Requirements

- This section provides information to ensure that the system communicates appropriately with users and external hardware or software elements. Reaching agreement on external and internal system interfaces has been identified as a software industry best practice (Brown 1996).

- 3.1.1 User Interfaces
- 3.1.2 Hardware Interfaces
- 3.1.3 Software Interfaces
- 3.1.4 Communications Interfaces

3.1.1 User Interfaces

Describe the logical characteristics of each user interface that the system needs. Some possible items to address here are:

- References to user interface standards or product line style guides that are to be followed
- Standards for fonts, icons, button labels, images, colour schemes, field tabbing sequences, commonly used controls, branding graphics, copyright and privacy notices, and the like
- Screen size, layout, or resolution constraints
- Standard buttons, functions, or navigation links that will appear on every screen, such as a help button

3.1.1 User Interfaces

- Shortcut keys
- Message display and phrasing conventions
- Data validation guidelines (such as input value restrictions and when to validate field contents)
- Layout standards to facilitate software localization
- Accommodations for users who are visually impaired, colour blind, or have other limitations

Example

3.1.1 User Interfaces

UI-1: The user interface shall contain the following accessibility requirements:

UI-1.1: The system provides a wheelchair accessibility flag to indicate that a location is wheelchair accessible.

UI-1.2: All keyboard shortcut keys are permitted for use throughout the whole system.

UI-2: The user interface shall contain the following resolution requirements:

UI-2.1: The desktop web application runs ideally in 1080p.

UI-2.2: The mobile web application runs ideally in 750p.

3.1.2 Hardware Interfaces

- Describe the characteristics of each interface between the system's software and hardware components, if any.
- This description might include the supported device types, the data and control interactions between the software and the hardware, and the communication protocols to be used.
- List the inputs and outputs, their formats, their valid values or ranges, and any timing issues developers need to be aware of.
- If this information is extensive, consider creating a separate interface specification document.

Example

3.1.2 Hardware Interfaces

No hardware interface requirements have been identified.

3.1.3 Software Interfaces

- Describe the connections between this product and other software components (identified by name and version), including other applications, databases, operating systems, tools, libraries, websites, and integrated commercial components.
- State the purpose, formats, and contents of the messages, data, and control values exchanged between the software components.
- Specify the mappings of input and output data between the systems and any translations needed for the data to get from one system to the other.

3.1.3 Software Interfaces

- Describe the services needed by or from external software components and the nature of the inter-component communications.
- Identify data that will be exchanged between or shared across software components.
- Specify nonfunctional interface requirements, such as service levels for response times and frequencies or security controls and restrictions.

Example

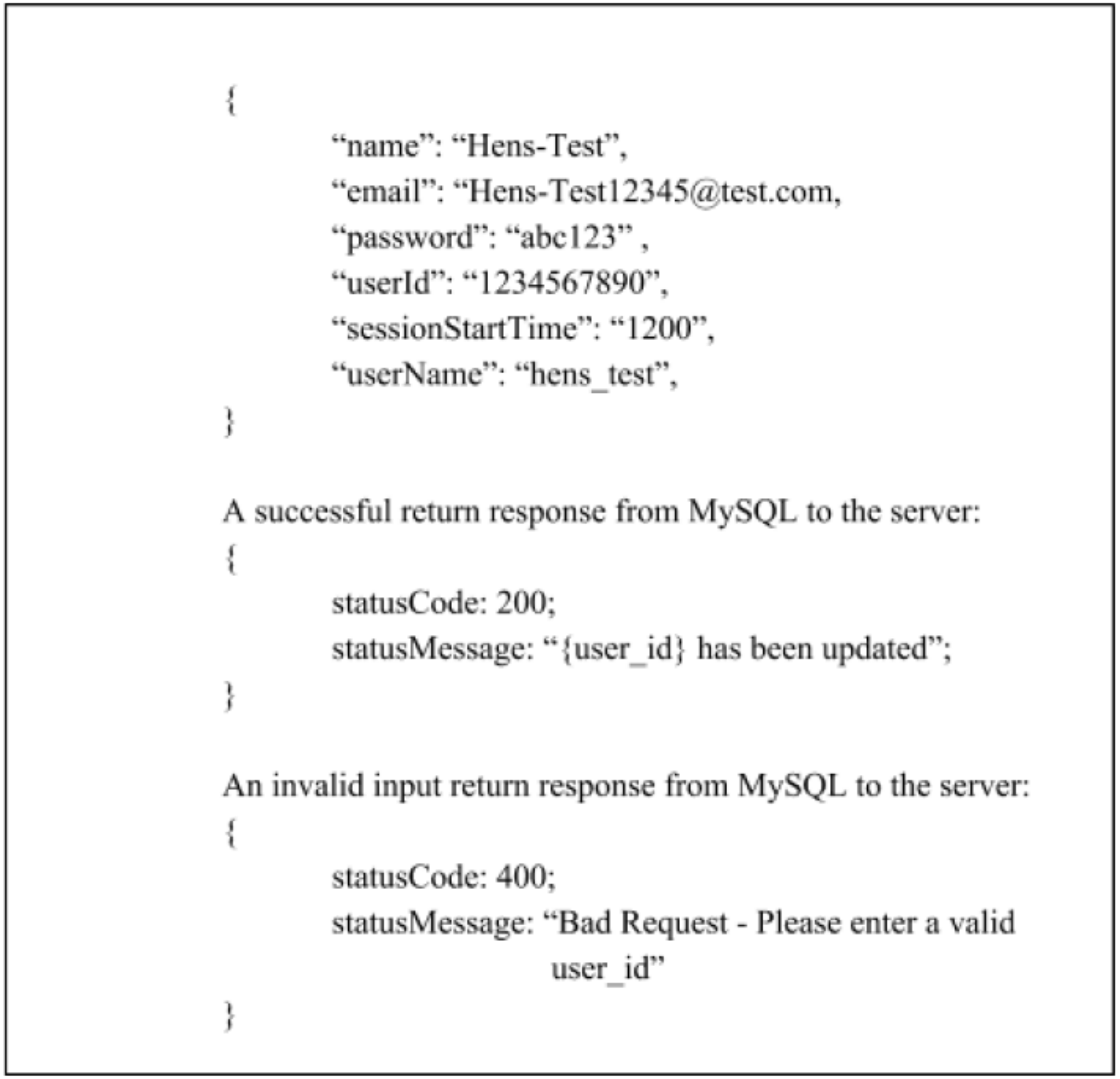
3.1.3 Software Interfaces

The system will contain the following interactions with the listed external software:

SI-1: MySQL

- SI-1.1: The system will interact with MySQL, a SQL based database relational database management system (DBMS) which will backup information once a month or prior to a major upgrade.
- SI-1.2: The system will interact with MySQL by following the listed protocol
  - SI-1.2.1: The user creates a request with the application.
  - SI-1.2.2: The application's external schema acknowledges the request and forwards the request to the conceptual schema.
  - SI-1.2.3: The application's conceptual schema acknowledges the request and forwards the request to the physical schema.
  - SI-1.2.4: The application's physical schema acknowledges the request and forwards the request to the MySQL server.
- SI-1.3: The system sends a payload to the MySQL server similar to the one listed below. Figure 5.1 below is an example of the payload when a new user is saved into the database:

Example



```
{
        “name”: “Hens-Test”,
        “email”: “Hens-Test12345@test.com,
        “password”: “abc123” ,
        “userId”: “1234567890”,
        “sessionStartTime”: “1200”,
        “userName”: “hens_test”,
}
```

A successful return response from MySQL to the server:

```
{
        statusCode: 200;
        statusMessage: “{user_id} has been updated”;
}
```

An invalid input return response from MySQL to the server:

```
{
        statusCode: 400;
        statusMessage: “Bad Request - Please enter a valid
                        user_id”
}
```

Figure 5.1: Sample payload sent to MySQL to insert a new user into the database

3.1.4 Communications Interfaces

- State the requirements for any communication functions the product will use, including email, web browser, network protocols, and electronic forms.
- Define any pertinent message formatting.
- Specify communication security and encryption issues, data transfer rates, handshaking, and synchronization mechanisms.
- State any constraints around these interfaces, such as whether certain types of email attachments are acceptable.

Example

3.1.4 Communications Interfaces

CI-4: The system will contain the following requirements in regard to notifications:

- CI-4.1: The system will send notifications to users 24 hours prior to the check in date.
- CI-4.2: Notifications will list the items and the time that the item must be returned by.

CI-5: The system will send an internal email to the customer support agent when a user files a request.

- CI-5.1: This email will allow attachments of no more than 10MB.

CI-6: The system will send an internal email to the legal agent when an item is reported.

- CI-6.1: This email will allow attachments of no more than 10MB.

3.2 System Features

- 3.2.1 [Feature 1]
  - 3.2.1.1 Purpose
  - 3.2.1.2 Response Sequence
  - 3.2.1.3 Functional Requirements

Example

3.2 System Features

3.2.1 Sign Up for an Account

3.2.1.1 Purpose

An unregistered user of the e-catalog system may select to sign up for an account, which will give them access to the main features of the e-catalog as described below. Priority = High.

3.2.1.2 Response Sequence

3.1.2 Functional Requirements

R-1: Only an unregistered user with a valid email address should be able to sign up for an account.

R-2: An unregistered user must choose a unique username when signing up for an account.

Example

3.2.2 Add a Label

3.2.2.1 Purpose

A registered user may select to add a label to an item or a container. Priority = Medium.

3.2.2.2 Response Sequence

3.2.2.3 Functional Requirements

R-1: A registered user should only be able to view the labels that they have added to an item or container.

R-2: A registered user should be able to add a unique label to an item.

R-3: A registered user should be able to add a label to a container (NEW).

R-4: Only a registered user should be able to add a label to an item or container.

3.3 Performance Requirements

- This subsection should specify both the static and the dynamic numerical requirements placed on the soft- ware or on human interaction with the software as a whole.

Example

3.3 Performance Requirements

PR1 The system should return the required information in at most 10 milliseconds.

PR2 95% of the transactions shall be processed in less than 1s.

3.4 Design Constraints

- This should specify design constraints that can be imposed by other standards, hardware limitations, etc.

Example

3.4 Design Constraints

DC1 - E-mail based solutions are forbidden.

DC2 - The software cost cannot exceed a million dollars.

3.5 Software System Attributes

- This section specifies nonfunctional requirements other than constraints, which are recorded in Section 2.4, and external interface requirements, which appear in Section 3.1.
- These quality requirements should be specific, quantitative, and verifiable. Indicate the relative priorities of various attributes, such as ease of use over ease of learning or security over performance.
- A rich specification notation clarifies the needed levels of each quality much better than can simple descriptive statements.

3.5 Softare System Attributes

3.5.1 Usability

3.5.2 Security

3.5.3 Safety

3.5.4 Availability

3.5.5 Integrity

3.5.6 Reliability

3.5.7 Efficiency

3.5.8 Robustness

3.5.9 Modifiability

3.5.10 Reusability

3.5.11 Scalability

3.5.12 Verifiability

Example

3.5 Software System Attributes

More than 50% of users should consider the software easy to use.

The software should run with at most a 30 millisecond delay when more than 10,000 users are connected simultaneously.

3.6 Other Requirements

- Define any other requirements that are not covered elsewhere in the SRS.
- Examples are legal, regulatory, or financial compliance and standards requirements; requirements for product installation, configuration, startup, and shutdown; and logging, monitoring, and audit trail requirements.
- Instead of just combining these all under “Other”, add any new sections to the template that are pertinent to your project. Omit this section if all your requirements are accommodated in other sections.
- Transition requirements necessary for migrating from a previous system to a new one could be included here if they involve software being written (as for data conversion programs) or in the project management plan if they do not (as for training development or delivery).

Example

3.6 Other Requirements

3.6.1 Legal Requirements

L-2: Items that are returned defective or damaged in any manner may result in the user account being charged for full replacements, which is handled beyond the scope of the system.

3.6.2 Regulatory Requirements

R-1: The system shall provide the following accessibility features to abide by Canadian law.

3.6.3 Financial Requirements

3.6.4 Transition Requirements

T-1: The system must follow the transition requirements in the order stated below:

T-1.1: Prior to system updates, the production code must be pushed onto a test server in which a quality analysis is performed initially by the QA team.

Example

3.6.5 Installation Requirements

3.6.6 Logs

3.6.6.1 Logging

L-1: All activities of users interacting with the system shall be logged into respective text files containing the following information:

L-1.1: Name, username…

3.6.6.2 Monitoring

M-2: System logs shall be monitored every month by the QA team except for immediate security issues.

3.6.6.3 Auditing

A-2: System logs shall be audited by the admin.

Appendix

- This optional section includes or points to pertinent analysis models such as data flow diagrams, feature trees or entity-relationship diagrams.
- Often, it is more helpful for the reader to incorporate specific models into the relevant sections of the specification instead of collecting them at the end.
- Include any additional table, diagrams, etc that you created for the final SRS.

Example

Appendix

Details about the decision of lowering the security of the system - …

Explanation of AI models - …

Characteristics of Excellent Requirements

- Characteristics of requirement statements
- Characteristics of requirements collections

Characteristics of Requirement Statements

- Complete
- Correct
- Feasible
- Necessary
- Prioritized
- Unambiguous
- Verifiable

Characteristics of Requirements Collections

- Complete
- Consistent
- Modifiable
- Traceable

Tips

- **Clarity and conciseness.** Write requirements in complete sentences using proper grammar, spelling, and punctuation. Keep sentences and paragraphs short and direct. Write requirements in simple language appropriate to the user domain, avoiding jargon. Define specialized terms in a glossary.
- **The keyword "shall".** A traditional convention uses the keyword "shall" to describe some system capability.
- **Active voice.** Write in the active voice to make it clear what entity is taking action described.
- **Individual requirements.** Avoid writing long narrative paragraphs that contain multiple requirements.

Tips

- **Appropriate detail.** An important part of requirements analysis is decomposing a high-level requirement into sufficient detail to clarify and flesh it out. There’s no single correct answer to the commonly asked question “How detailed should the requirements be?”. Provide enough specifics to minimize the risk of misunderstanding based on the development team’s knowledge and experience.
- **Representation techniques.** Readers’ eyes glaze over when confronting a dense mass of turgid text or a long list of similar-looking requirements. Consider the most effective way to communicate each requirement to the intended audience. Some alternatives to the natural language requirements we’re used to are lists, tables, visual analysis models, charts, mathematical formulas, photographs, sound clips, and video clips.

Ambiguity

- Avoid.
- Fuzzy words.

Ambiguity

Ambiguous Terms	Ways To Improve Them
acceptable, adequate	Define what constitutes acceptability and how the system can judge this.
and/or	Specify whether you mean "and", "or", or "any combination of" so the reader does not have to guess.
as much as practicable	Do not leave it up to the developers to determine what is practicable. Make it a "to be decided" and set a date to find out.
at least, at a minimum, not more than, not to exceed	Specify the minimum and maximum acceptable values.
best, greatest, most	State what level of achievement is desired and the minimum acceptable level of achievement.
between, from X to Y	Define whether the end points are included in the range.

Ambiguity

Ambiguous Terms	Ways To Improve Them
depends on	Describe the nature of the dependency. Does another system provide input to this system, must other software be installed before your software can run, or does your system rely on another to perform some calculations or provide other services?
efficient	Define how efficiently the system uses resources, how quickly it performs specific operations, or how quickly users can perform tasks with the system.
fast, quick, rapid	Specify the minimum acceptable time in which the system performs some action.
flexible, versatile	Describe the ways in which the system must be able to adapt to changing operating conditions, platoforms, or business needs.
improved, better, faster, superior, higher quality	Quantify how much better or faster constitutes adequate improvement in a specific functional area or quality aspect.

Ambiguity

Ambiguous Terms	Ways To Improve Them
including, including but not limited to, and so on, etc., such as, for instance	List all possible values or functions, not just examples, or refer the reader to the location of the full list. Otherwise, different readers might have different interpretations of what the whole set of items being referred to contains or where the list stops.
in most cases, generally, usually, almost always	Clarify when the stated conditions or scenarios do not apply and what happens then. Describe how either the user or the system cand distinguish one case from the other
match, equals, agree, the same	Define whether a text comparison is case sensitive and whether it means the phrase "contains", "starts with", or is "exact". For real numbers, specify the degree of precision in the comparison.
maximize, minimize, optimize	State the maximum and minimum acceptable values or some parameter.
normally, ideally	Identify abnormal or non-ideal conditions and describe how the system should behave in those situations.

Ambiguity

Ambiguous Terms	Ways To Improve Them
optionally	Clarify whether this means a developer choice, a system choice, or a user choice.
probably, ought to, should	Will it or will it not?
reasoable when necessary, where appropriate, if possible, as applicable	Explain how either the developer or the user can make this judgement.
robust	Define how the system is to handle exceptions and respond to the unexpected operating conditions.
seamless, transparent, graceful	What does "seamless" or "graceful" mean to the user? Translate the user's expectations into specific observable product characteristics.
several,some, many, few, multiple, numerous	State how many, or provide the minimum and maximum bounds of a range.

Ambiguity

Ambiguous Terms	Ways To Improve Them
shouldn't, won't	Try to state requirements as positives, describing what the system will do.
state-of-the-art	Define what this phrase means to the stakeholder.
sufficient	Specify how much of something constitutes sufficiency.
support, enable	Define exactly what functions the system will perform that constitute "supporting" some capability.
user-friendly, simple, easy	Describe system characteristics that will satify the customer's usage needs and usability expectations.

Ambiguity

- **The A/B construct.** Many requirements specifications include expressions in the form “A/B”, in which two related (or synonymous, or opposite) terms are combined with a slash.
- **Boundary values.** Many ambiguities occur at the boundaries of numerical ranges in both requirements and business rules.
- **Negative requirements.** People sometimes write requirements that say what the system will not do rather than what it will do. How do you implement a don’t-do-this requirement? Double and triple negatives are particularly tricky to decipher. Try to rephrase negative requirements into a positive sense that clearly describes the restricting behaviour.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

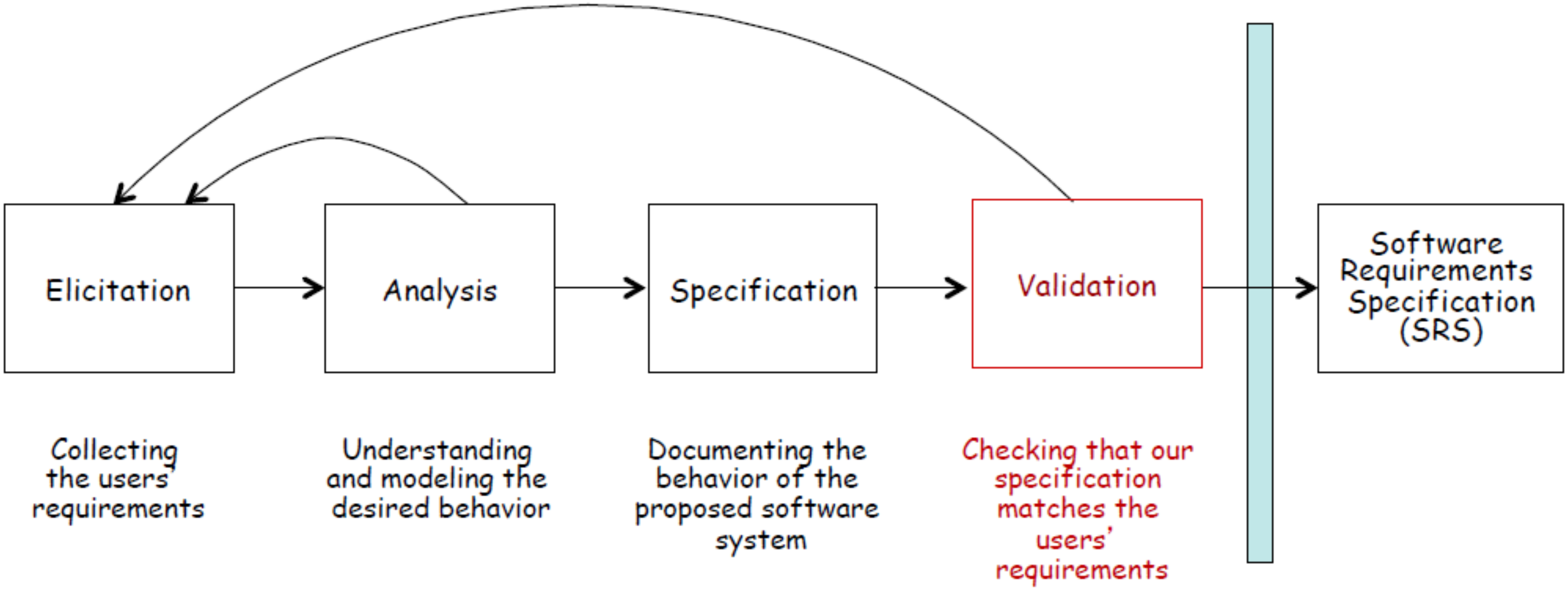
Documentation

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Validation and Verification

Requirements Engineering Process

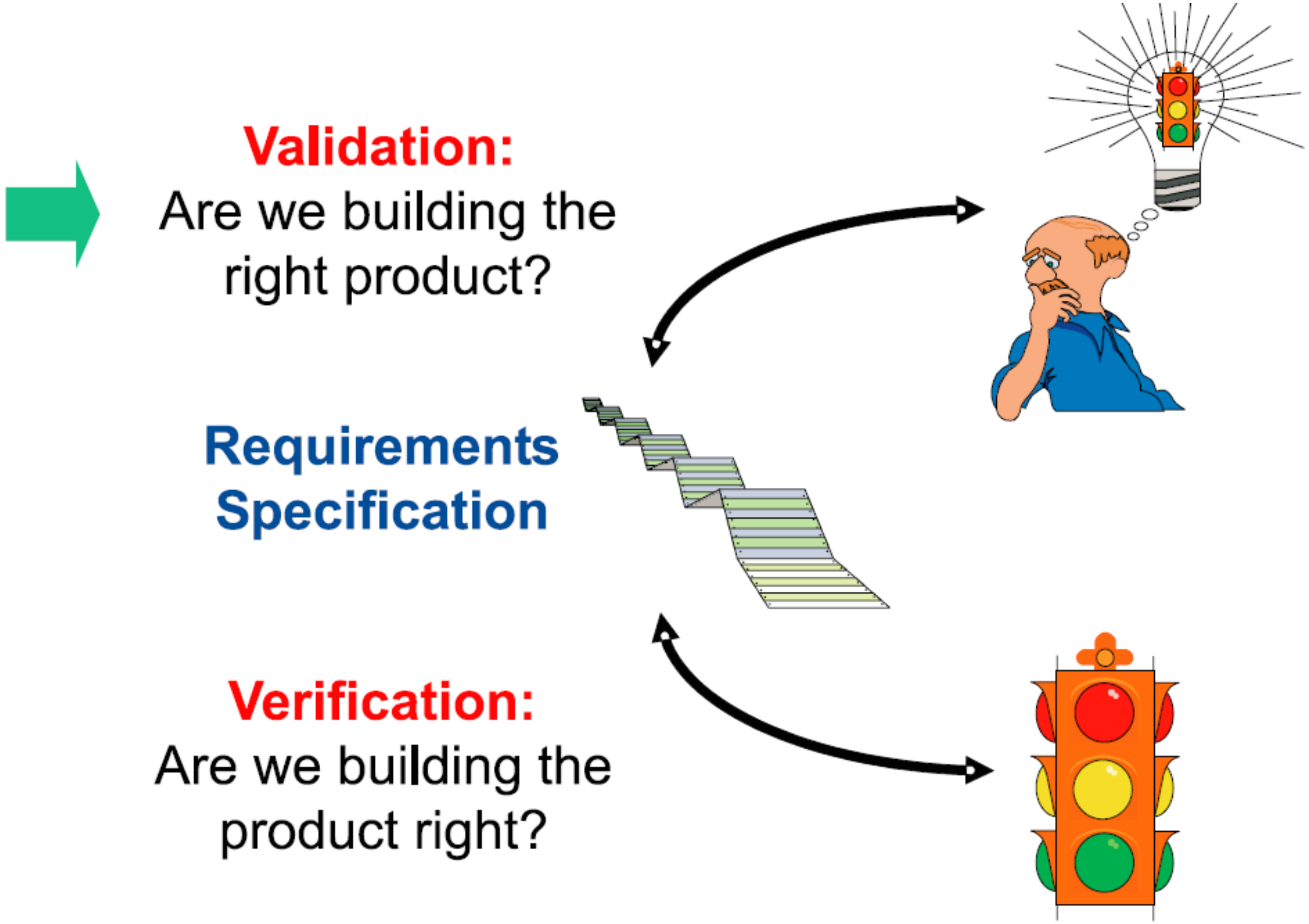


Elicitation
Analysis
Specification
Validation
Software Requirements Specification (SRS)
Collecting the users' requirements
Understanding and modeling the desired behavior
Documenting the behavior of the proposed software system
Checking that our specification matches the users' requirements

1. Validation vs. Verification

$$\boxed{\text{Spec, Domain} \models \text{Req}}$$

$$\boxed{\begin{array}{c}\text{Design} \models \text{Spec}\\ \text{Code} \models \text{Design}\\ \text{TestCases} \models \text{Spec}\end{array}}$$

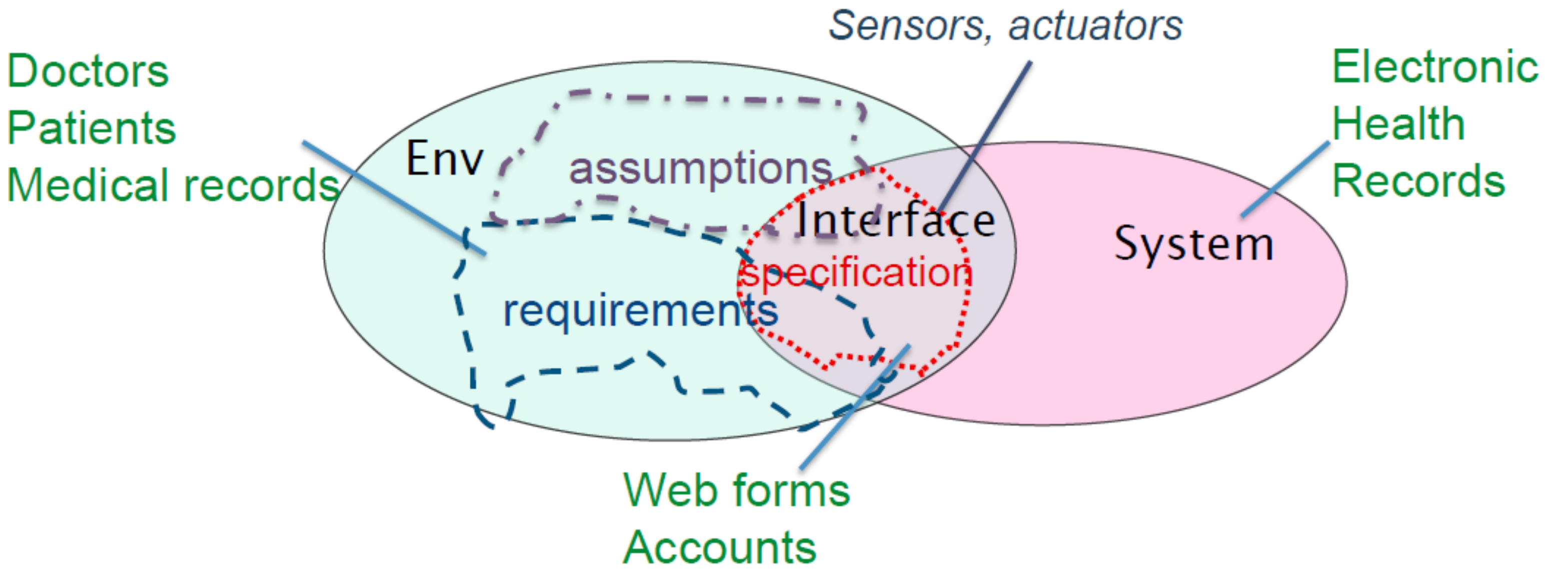


Validation:
Are we building the right product?

Requirements Specification

Verification:
Are we building the product right?

Requirements Engineering Reference Model

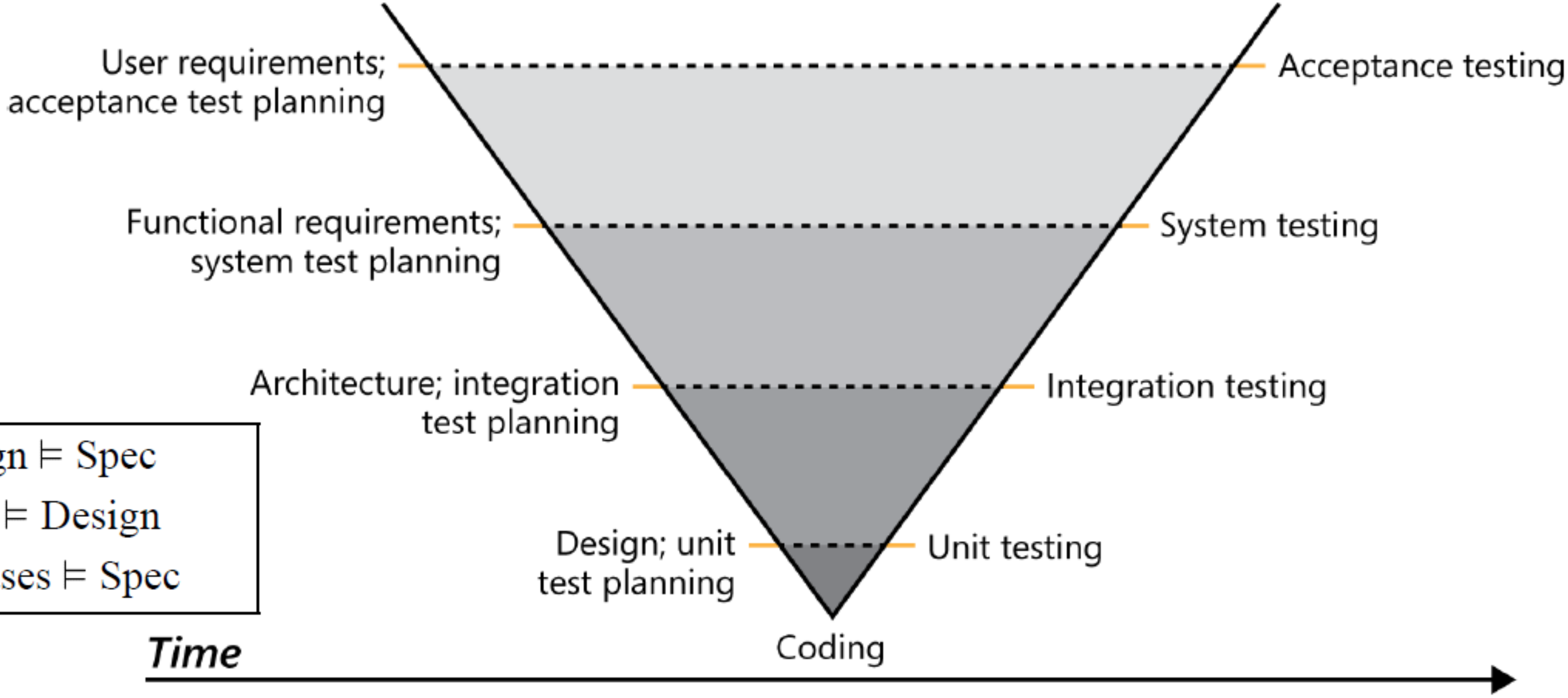


- A requirement is a condition or capability that must be achieved
- A specification is a description of the proposed system / solution
- A domain assumption records how the world (ought to) behave

Validation
Req are correct, complete
Domain are correct, complete

Requirements Engineering and Testing

$\text{Spec}, \text{Domain} \models \text{Req}$



User requirements; acceptance test planning

Acceptance testing

Functional requirements; system test planning

System testing

Architecture; integration test planning

Integration testing

Design; unit test planning

Unit testing

Coding

Time

$\text{Design} \models \text{Spec}$

$\text{Code} \models \text{Design}$

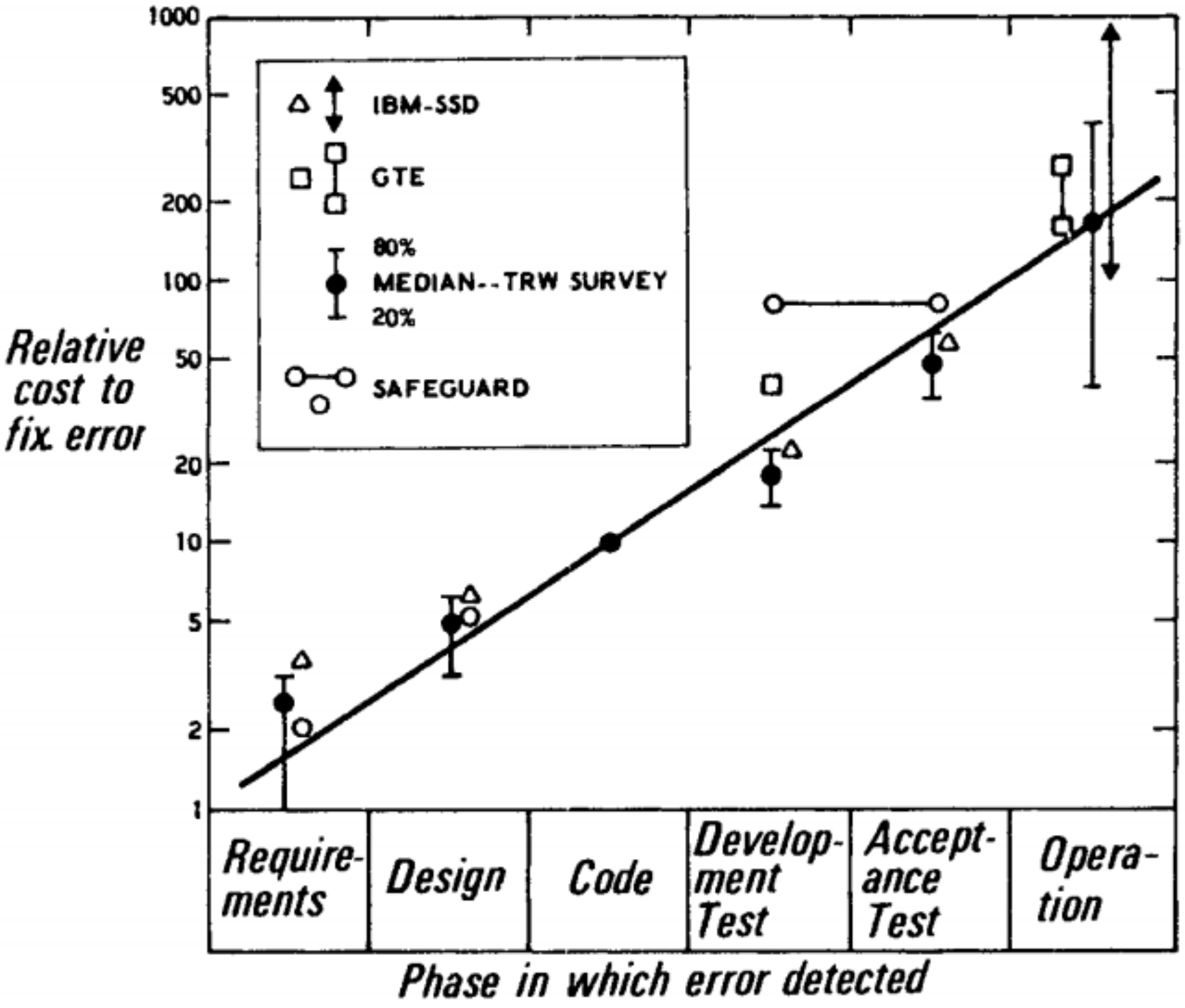
$\text{TestCases} \models \text{Spec}$

Requirements Validation

Requirements validation activities attempt to ensure that:

- The software requirements accurately describe the intended system capabilities and properties that will satisfy the various stakeholders' needs.
- The software requirements are correctly derived from the business requirements, system requirements, business rules, and other sources.
- The requirements are complete, feasible, and verifiable.
- All requirements are necessary, and the entire set is sufficient to meet the business objectives.
- All requirements representations are consistent with each other.
- The requirements provide an adequate basis to proceed with design and construction.

Cost of Fix Errors



2. Reviewing Requirements

- Reviewing requirements is a powerful technique for identifying ambiguous or unverifiable requirements, requirements that are not defined clearly enough for the design to begin, and other problems.
- Different kinds of peer reviews:
  - A *peer desk check*, in which you ask one colleague to look over your work product.
  - A *pass around*, in which you invite several colleagues to examine a deliverable concurrently.
  - A *walkthrough*, during which the author describes a deliverable and solicits comments.

2.1 The Inspection Process

- Michael Fagan developed the inspection process at IBM (Fagan 1976; Radice 2002).
- Others have extended or modified his method (Gilb and Graham 1993; Wiegers 2002).
- Inspection has been recognized as a software industry best practice (Brown 1996).
- Any software work product, including requirements, design documents, source code, test documentation, and project plans, can be inspected.
- It involves a small team of participants carefully examining a work product for defects and improvement opportunities.
- Inspections serve as a quality gate through which project deliverables must pass before they are baselined.

2.1.1 Participants

- The author of the work product and perhaps peers of the author.
  - Business analyst who wrote the requirement.
  - Another experienced business analyst to check for errors
- People who are the information sources fed into the item being inspected.
  - Actual users, customers, the author of the predecessor specification.
- People who will do work based on the item being inspected.
  - A developer, a tester, a project manager, and a user documentation writer.
- People who are responsible for interfacing systems that will be affected by the item being inspected.
  - Will check the external interfaces requirements

2.1.2 Inspection Roles

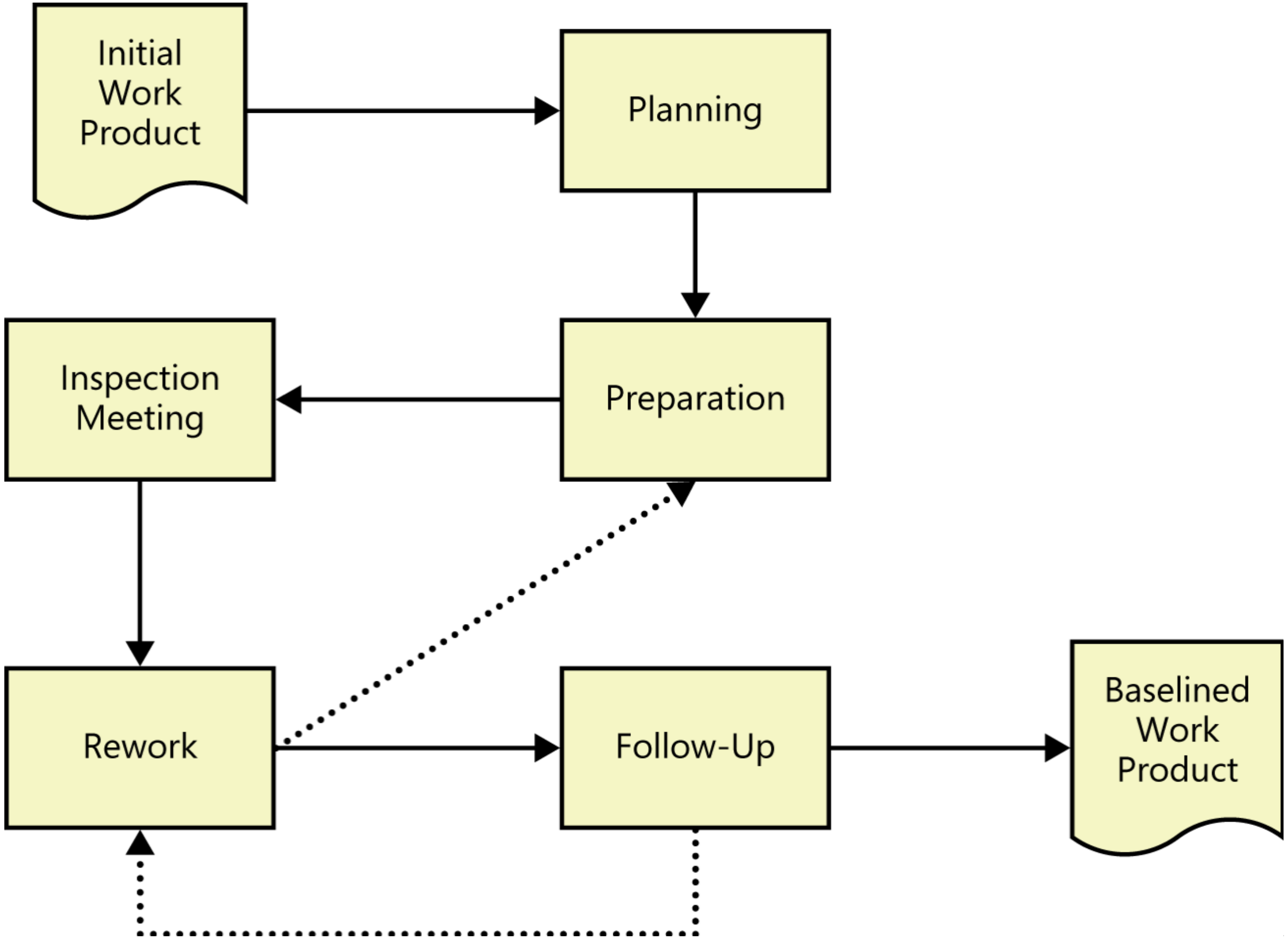
All participants in an inspection, including the author, look for defects and improvement opportunities. Some of the inspection team members perform the following specific roles during the inspection:

- Author
- Moderator
- Reader
- Recorder

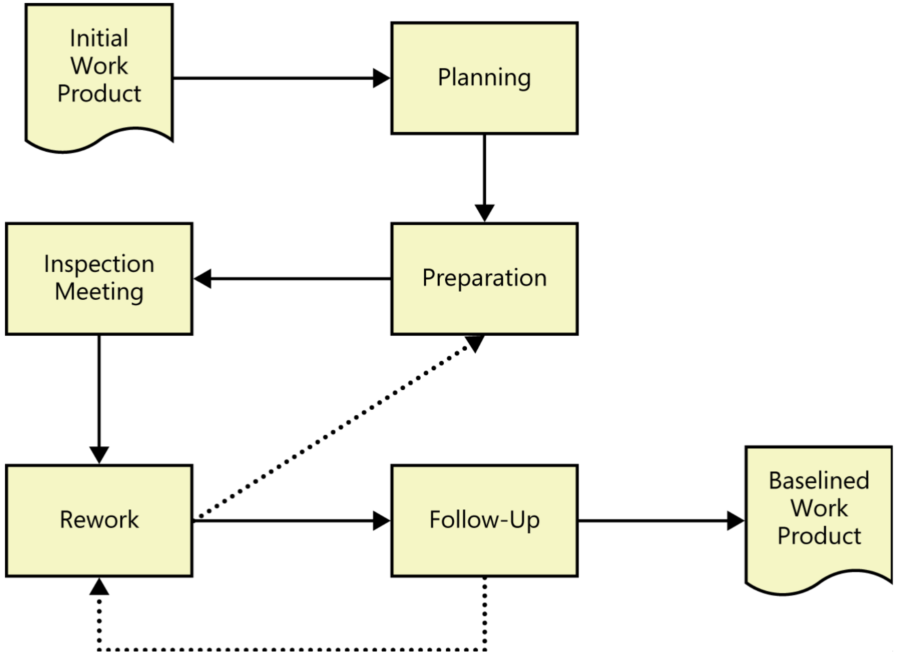
2.1.3 Entry Criteria

- Set clear expectations for authors to follow while preparing for an inspection.
- Keep the inspection team from spending time on issues that should be resolved before the inspection.
- The moderator uses the entry criteria as a checklist before inspecting.
- Examples:
  - The document conforms to the standard template and does not have obvious spelling, grammatical, or formatting issues.
  - Line numbers or other unique identifiers are printed on the document to facilitate referring to specific locations.
  - All open issues are marked as TBD (to be determined) or accessible in an issue-tracking tool.
  - The moderator did not find more than three significant defects in a ten-minute examination of a representative sample of the document.

2.1.4 Inspection Stages



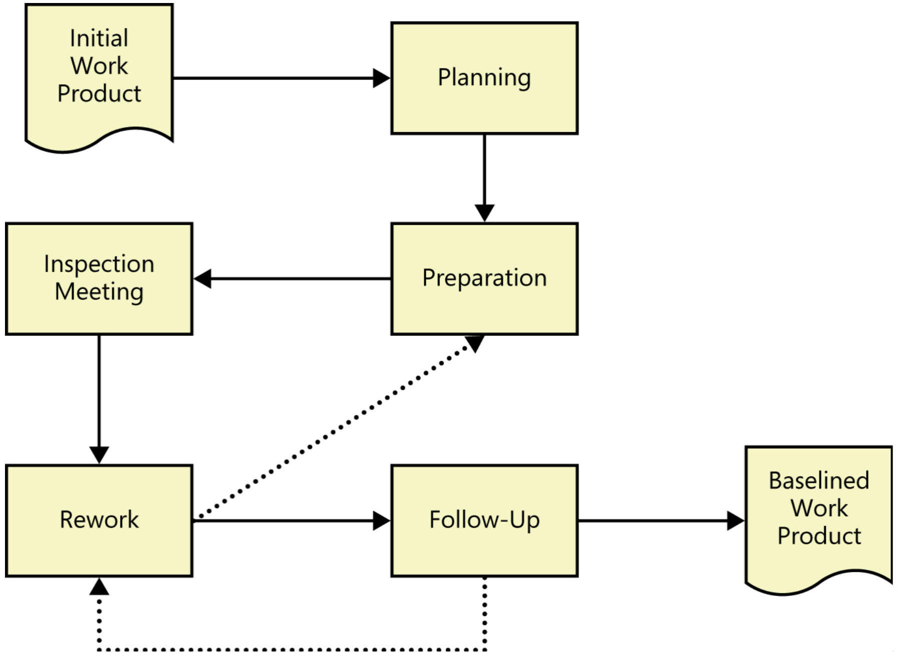
2.1.4 Inspection Stages



Planning

- The author and moderator plan the inspection together.
- They determine
  - who should participate,
  - what materials should the inspectors receive prior to the inspection meeting,
  - the total meeting time needed to cover the material, and
  - when the inspection should be scheduled.

2.1.4 Inspection Stages



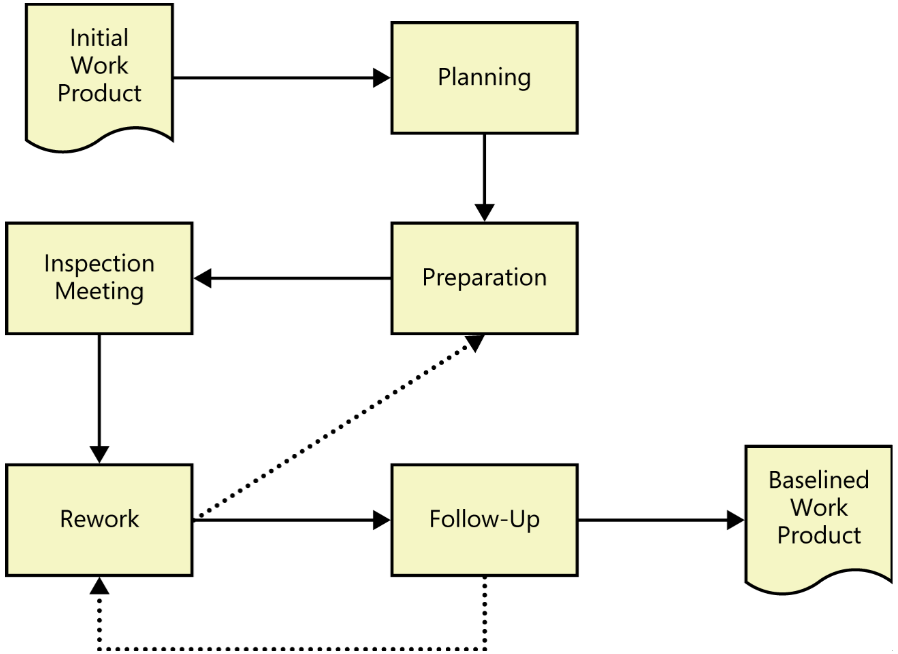
Preparation

- The author should share background information with inspectors so they understand the context of the items being inspected and know the author’s objectives for the inspection.

- Each inspector then examines the product to identify possible defects and issues, using the checklist of typical requirements defects described later.

- Up to 75 percent of the defects found by an inspection are discovered during preparation, so do not omit this step

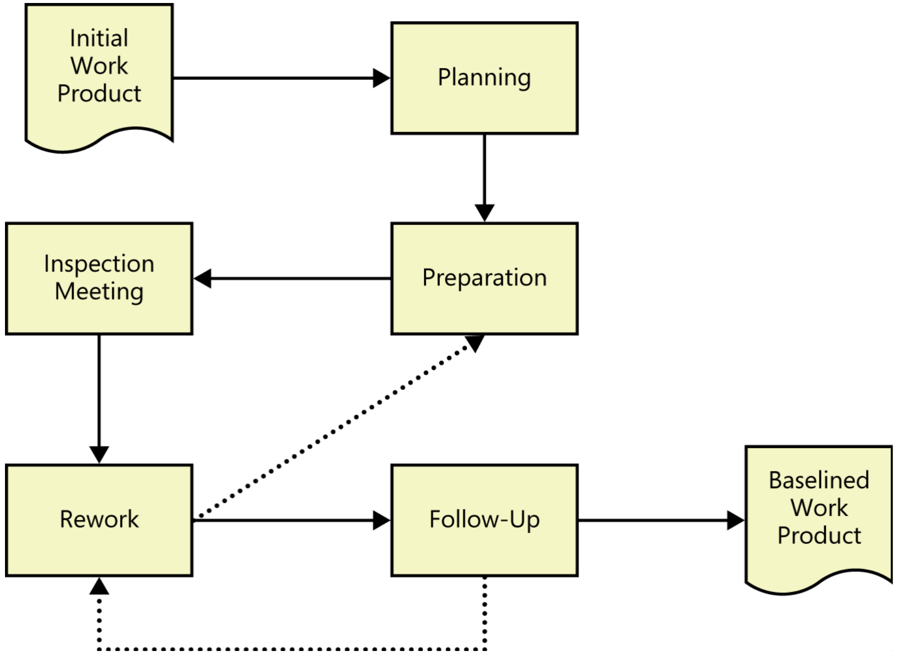
2.1.4 Inspection Stages



Inspection meeting

- The reader leads the other inspectors through the document describes one requirement at a time in his own words.
- As inspectors bring up possible defects and other issues, the recorder captures them in the action item list for the author of the requirement.
- The meeting aims to identify as many significant defects as possible.
- The inspection meeting should not last more than two hours; tired people are not effective inspectors. Schedule additional meetings if you need more time to cover all the material.

2.1.4 Inspection Stages



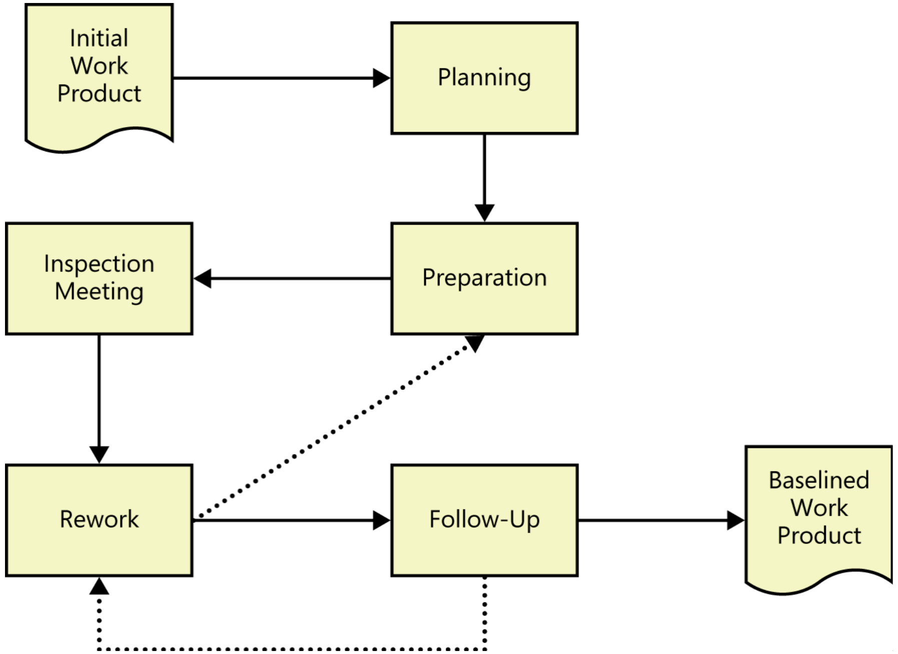
Rework

- Nearly every quality control activity reveals some defects.

- The author should plan to spend some time reworking the requirements following the inspection meeting.

- Uncorrected requirement defects will be expensive to fix down the road, so this is the time to resolve the ambiguities, eliminate the fuzziness, and lay the foundation for a successful development project.

2.1.4 Inspection Stages



Follow-up

- The moderator or a designated individual works with the author to ensure that all open issues were resolved, and errors were corrected properly.

- Follow-up brings closure to the inspection process and enables the moderator to determine whether the inspection's exit criteria have been satisfied.

- The follow-up step might reveal that some of the modifications made were incomplete or not performed correctly, leading to additional rework

2.1.5 Exit Criteria

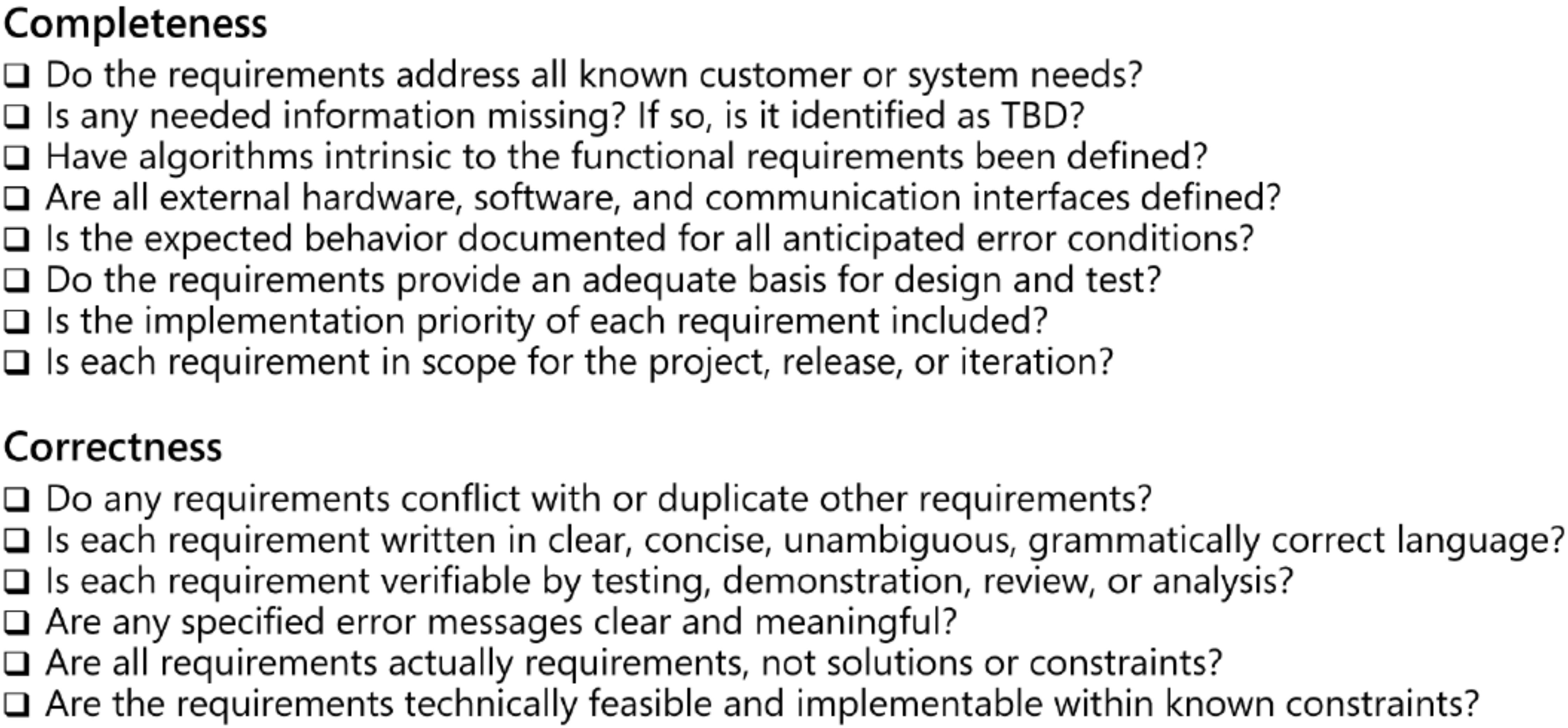
- The inspection process should define the exit criteria that must be satisfied before the moderator declares the entire inspection process (not just the meeting) complete.

- Here are some possible exit criteria for requirements inspections:
  - All issues raised during the inspection have been addressed.
  - Any changes in the requirements and related work products were made correctly.
  - All open issues have been resolved, or each open issue’s resolution process, target date, and owner have been documented.

2.2 Defect Checklist

- To help reviewers look for typical errors in the products they review, develop a defect checklist for each type of requirements document your projects create.
- Such checklists call the reviewers' attention to historically frequent requirement problems.
- Checklists serve as reminders.
- Over time, people will internalize the items and look for the right issues in each review out of habit.

2.2 Defect Checklist



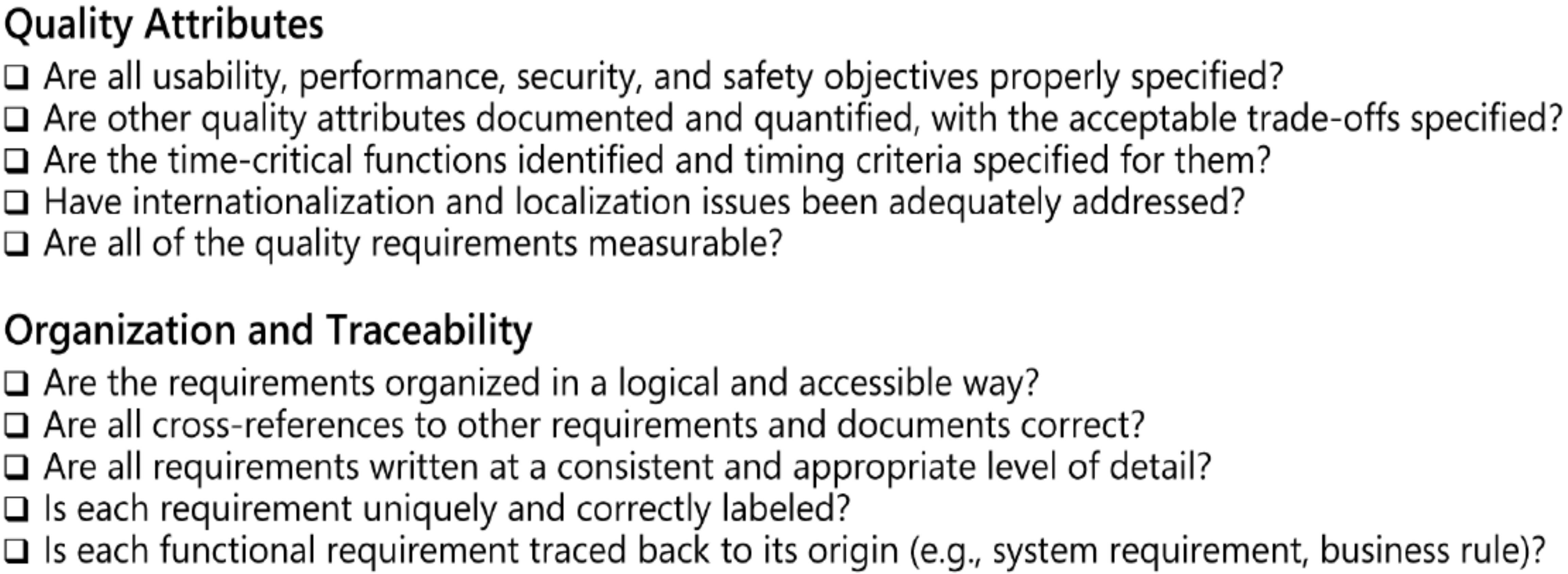
Completeness

- □ Do the requirements address all known customer or system needs?
- □ Is any needed information missing? If so, is it identified as TBD?
- □ Have algorithms intrinsic to the functional requirements been defined?
- □ Are all external hardware, software, and communication interfaces defined?
- □ Is the expected behavior documented for all anticipated error conditions?
- □ Do the requirements provide an adequate basis for design and test?
- □ Is the implementation priority of each requirement included?
- □ Is each requirement in scope for the project, release, or iteration?

Correctness

- □ Do any requirements conflict with or duplicate other requirements?
- □ Is each requirement written in clear, concise, unambiguous, grammatically correct language?
- □ Is each requirement verifiable by testing, demonstration, review, or analysis?
- □ Are any specified error messages clear and meaningful?
- □ Are all requirements actually requirements, not solutions or constraints?
- □ Are the requirements technically feasible and implementable within known constraints?

2.2 Defect Checklist



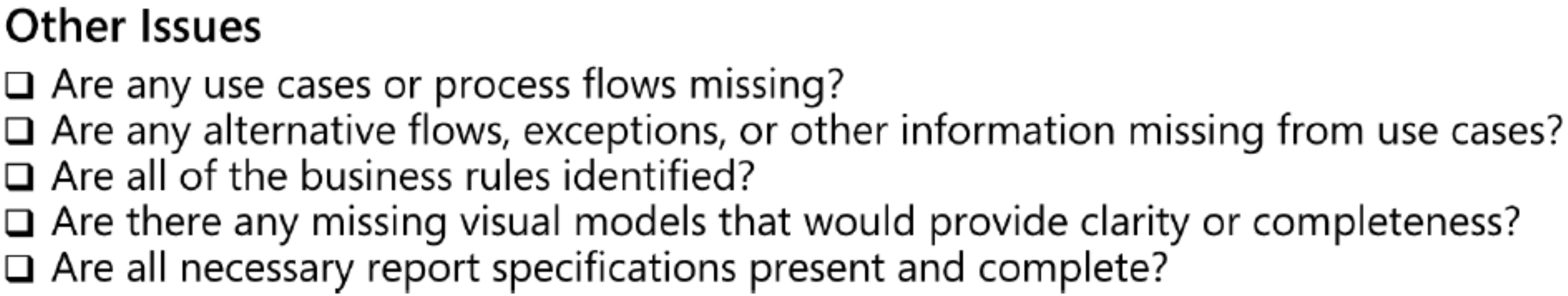
Quality Attributes

□ Are all usability, performance, security, and safety objectives properly specified?
□ Are other quality attributes documented and quantified, with the acceptable trade-offs specified?
□ Are the time-critical functions identified and timing criteria specified for them?
□ Have internationalization and localization issues been adequately addressed?
□ Are all of the quality requirements measurable?

Organization and Traceability

□ Are the requirements organized in a logical and accessible way?
□ Are all cross-references to other requirements and documents correct?
□ Are all requirements written at a consistent and appropriate level of detail?
□ Is each requirement uniquely and correctly labeled?
□ Is each functional requirement traced back to its origin (e.g., system requirement, business rule)?

2.2 Defect Checklist



Other Issues

□ Are any use cases or process flows missing?
□ Are any alternative flows, exceptions, or other information missing from use cases?
□ Are all of the business rules identified?
□ Are there any missing visual models that would provide clarity or completeness?
□ Are all necessary report specifications present and complete?

2.3 Requirements Review Tips

- **Plan the examination** by inviting certain reviewers to focus on specific sections of documents.
- **Start early,** when there are perhaps only 10 percent complete, not when you think they are “done”.
- **Allocate sufficient time** to perform the reviews in terms of actual hours to review (effort) and calendar time.
- **Provide context** for the document and the project if they are all working on different projects.

2.3 Requirements Review Tips

- **Set the review scope** by telling the reviewers what material to examine, where to focus their attention, and what issues to look for.
- **Limit re-reviewing** the same material more than three times. If you need someone to review it multiple times, highlight the changes so he can focus on them.
- **Prioritize review areas** of high risk or functionality that will be used frequently. Also, look for areas of the requirements with few issues logged already.

2.4 Requirements Review Challenges

- Large requirements documents
- Large inspection teams
- Geographically separated reviewers
- Unprepared reviewers

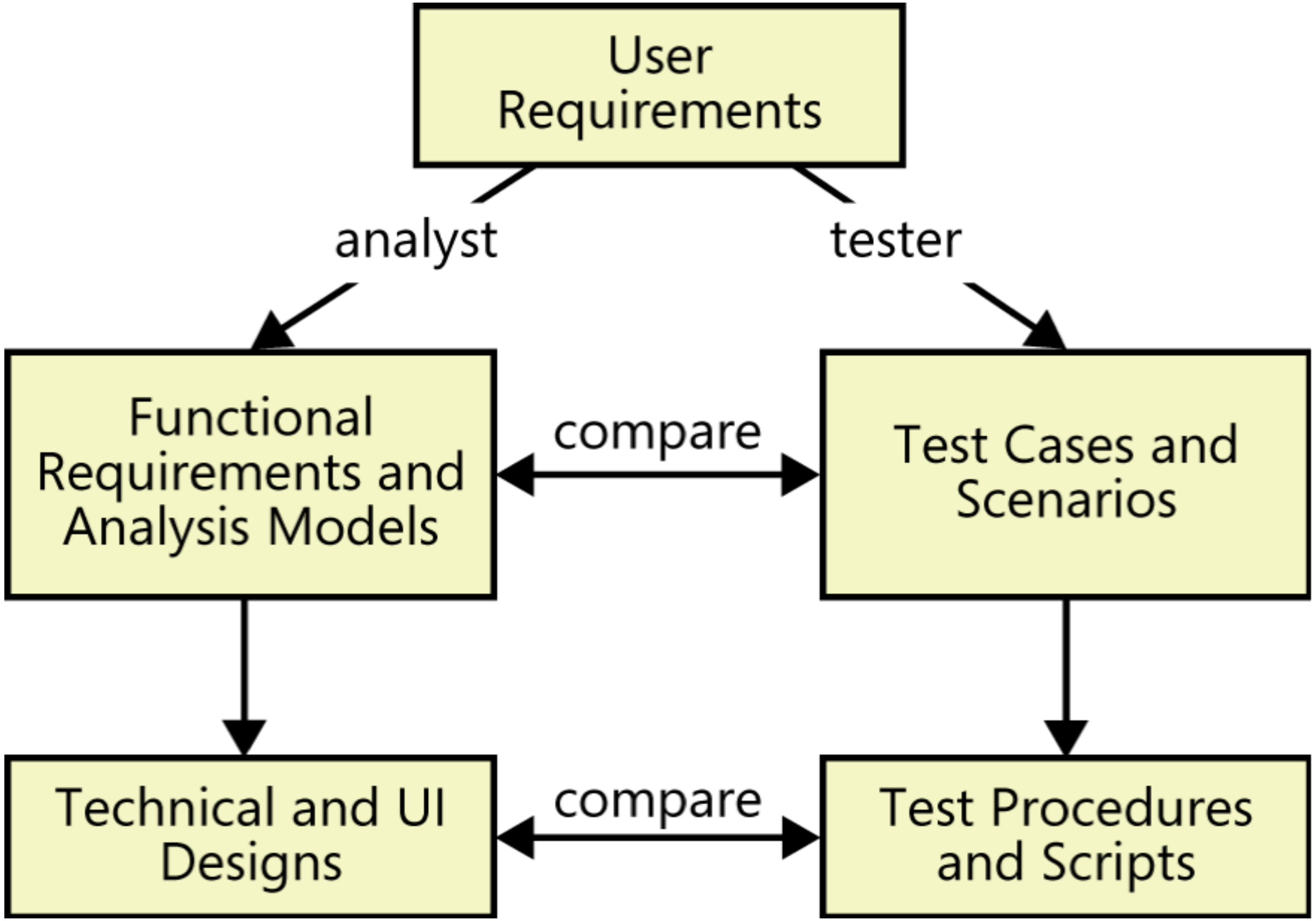
3. Prototyping Requirements

- Prototypes are validation tools that make the requirements real.
- All prototypes allow you to find missing requirements before more expensive activities like development and testing occur.
- Something as simple as a paper mock-up can be used to walk through use cases, processes, or functions to detect omitted or erroneous requirements.
- Prototypes also help confirm that stakeholders have a shared understanding of the requirements.
- Proof-of-concept prototypes can demonstrate that the requirements are feasible.
- Evolutionary prototypes allow the users to see how the requirements would work when implemented to validate that the result is what they expect.
- Additional levels of sophistication in prototypes, such as simulations, allow more precise validation of the requirements. However, building more sophisticated prototypes will also take more time.

4. Testing The Requirements

- The simple act of designing tests will reveal many problems with the requirements long before you can execute those tests on running software.
- Writing functional tests crystallizes your vision of how the system should behave under certain conditions.
- Vague and ambiguous requirements will jump out at you because you will not be able to describe the expected system response.
- Watch out for testers who claim they cannot begin their work until the requirements are done and testers who claim they do not need requirements to test the software. Testing and requirements have a synergistic relationship; they represent complementary views of the system.

4. Testing The Requirements



5. Validating Requirements With Acceptance Criteria

- Customers need to assess whether a system satisfies its predefined acceptance criteria.
- Acceptance criteria, and hence acceptance testing, should evaluate whether the product satisfies its documented requirements and whether it is fit for use in the intended operating environment

5.1 Acceptance Criteria

- Working with customers to develop acceptance criteria provides a way to validate both the requirements and the solution.
- Thinking about acceptance criteria offers a shift in perspective from the elicitation question of “What do you need to do with the system?” to “How would you judge whether the solution meets your needs?”
- Encourage users to use the SMART mnemonic (Specific, Measurable, Attainable, Relevant, and Time-sensitive) when defining acceptance criteria.

5.1 Acceptance Criteria

- Defining acceptance criteria is more than just saying that all the requirements are implemented or all the tests passed.
- Specific high-priority functionality must be present and operating correctly before the product can be accepted and used.
- Essential nonfunctional criteria or quality metrics that must be satisfied.
- Remaining open issues and defects.
- Specific legal, regulatory, or contractual conditions. (These must be fully satisfied before the product is considered acceptable.)
- Supporting transition, infrastructure, or other project (not product) requirements. (Perhaps training materials must be available and data conversions completed before the solution can be released.)

5.2 Acceptance Tests

- Acceptance tests constitute the most significant portion of the acceptance criteria.
- Creators of acceptance tests should consider the most commonly performed and essential usage scenarios when deciding how to evaluate the software’s acceptability.
- Automate the execution of acceptance tests whenever possible. This makes it easier to repeat the tests when changes are made and functionality is added in future iterations or releases.
- Acceptance tests must also address nonfunctional requirements.
- They should ensure that performance goals are achieved, that the system complies with usability standards, and that security expectations are fulfilled.
- Do not expect user acceptance testing to replace comprehensive requirements-based system testing, which covers all the standard and exception paths and a wide variety of data combinations, boundary values, and other places where defects might lurk.

Final Words

- Writing requirements is not enough.
- You need to ensure they are the right requirements and good enough to serve as a foundation for design, construction, testing, and project management.
- Acceptance test planning, informal peer reviews, inspections, and requirements testing techniques will help you build higher-quality systems faster and more inexpensively than ever.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Validation and Verification

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Temporal Logic

Background

- We learned about prescriptive specifications that describe how a system behaves from one point to another.
- System behaviour is decomposed into states, and the specification is described for each state
  - what input the system is ready to react to in that state and
  - what the system's response to the input event will be.
- What if we want to know about longer-term system behaviour?

Example

- Specification: if a car approaches the intersection, the light in its direction will eventually be green.
- If you use what we learned, you must draw several state diagrams covering each case in which a car approaches the intersection.
- Another approach is to use a notation designed for expressing system-wide properties such as temporal logic.

Logic

- Propositional logic expresses properties about fixed-valued variables.
- Predicate logic expresses properties about variables that change value. A logic formula is evaluated concerning a particular assignment of values to variables.
- In temporal logic, a formula may be evaluated over variables that change value over time.

1. Set of typed variables
    - Booleans, Integers, Sets.
    - If your system is Object-Oriented, you may have variables for object instantiations, attributes, etc.

Logic

2. Functions on typed variables:
   - Integers: +, -, *, /
   - Sets: ∪, ∩
   - Booleans: ∧, ∨, ¬
3. Predicates
   - Integers: <, >
   - Sets: ⊂, ∈
4. Equality
   - = (comparing two values of the same type)

Logic

5. Connectives
   - ¬, ∧, ∨, →
   - cond1 → cond2 ≡ ¬cond1 ∨ cond2

   if cond1 then cond2 else cond3 ≡

   (cond1 → cond2) ∧ (¬cond1 → cond3)

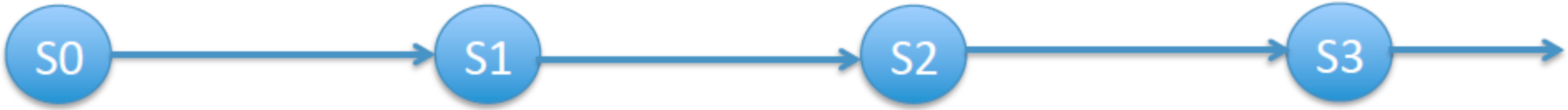
Logic

6. Quantifiers
   - $\forall\, x \in T : f(x)$
     For all $t \in T$: the interpretation of f with t substituted for x evaluates to true.
   - $\exists\, x \in T : f(x)$
     There exists $t \in T$: the interpretation of f with t substituted for x evaluates to true.
   - Scope of a quantifier: the extent to which the quantifier applies in the given formula.
     Without brackets, we assume that the scope extends to the right end of the formula.

Executions

In an executing system, variables change value over time.

- a particular execution of the system is represented by a sequence of states: $\sigma = S_0, S_1, S_2, \ldots$



S0	S1	S2	S3
$locked$	$locked$	$\neg locked$	$\neg locked$
$\neg coin$	$coin$	$\neg coin$	$\neg coin$
$\neg push$	$\neg push$	$\neg push$	$push$
$numCoins=0$	$numCoins=1$	$numCoins=1$	$numCoins=1$
$numEntries=0$	$numEntries=0$	$numEntries=0$	$numEntries=0$

Time-Dependent Logic (Timed logic)

- Many properties describe behaviour over time
- We can think of variables as time functions whose value depends on time.

Turnstile Example

coin: *time* → *boolean*

locked: *time* → *boolean*

push: *time* → *boolean*

enter: *time* → *boolean*

rotating: *time* → *boolean*

numEntries: *time* → *integer*

numCoins: *time* → *integer*

When writing formulas, every variable used in the formula specifies the time that the variable's value is referenced.

$$numCoins(0) = 0$$

$$coin(1) \rightarrow \neg locked(2)$$

Turnstile Example

It is hard to write specifications in terms of what the variable values will be at a particular point in time.

More often, one is interested in expressing the relationships between variable values.

For example, the barrier will be unlocked when a coin is inserted

$$\forall t \in Time : (coin(t) \rightarrow \neg locked(t+1))$$

$$\forall t_1 \in Time : (coin(t_1) \rightarrow \exists t_2 \in Time : (t_1 < t_2 \wedge \neg locked(t_2)))$$

Turnstile Example

It is always the case that the number of entries into the park is less than or equal to the number of coins received.

$$\forall t \in Time : (numEntries(t) \leq numCoins(t))$$

Turnstile Example

If a visitor pushes the turnstile and the turnstile is unlocked, then eventually the visitor will enter the park.

$$\forall t \in Time : ((push(t) \land \neg locked(t)) \rightarrow \exists t_1 \in Time : (t_1 > t \land enter(t_1)))$$

Turnstile Example

If a visitor pushes the turnstile when the turnstile is unlocked, then the turnstile rotates until the visitor enters the park.

$$\begin{aligned}
&\forall t \in Time : ((push(t) \wedge \neg locked(t)) \rightarrow \\
&\quad \exists t_1 \in Time : (t_1 > t \wedge enter(t_1) \wedge \\
&\qquad \forall t_2 \in Time : (t < t_2 < t_1 \rightarrow rotating(t_2))))
\end{aligned}$$

Explicit vs. Implicit Time

Notice that we often do not care about the values of variables at specific points in time. With the possible exception of time $t = 0$, when we might care about the initial values of the variables.

Mostly, we care about the <u>temporal ordering of events and variable values</u>. We want to express constraints on variable values regarding when they change value.

- If a coin is inserted, the barrier will become unlocked.
- If a caller picks up the telephone handset, he will hear a dial tone.
- If I push the elevator button, the elevator will eventually arrive at my floor and open its doors.

Explicit vs. Implicit Time

Sometimes we care about the timing of those events.

- If a train comes within 200 meters of a railroad crossing, the gate will be lowered within 10 seconds.

But for the most part, we are concerned only with <u>the order</u> in which events occur.

LTL: Linear Temporal Logic

- Linear Temporal Logic was designed to express the temporal ordering of events and variable values while leaving time implicit.
- In temporal logic, time progresses, but the exact time is abstracted away. Instead, we keep track of changes to variable values and the order in which they occur.

LTL: System State

- The system state is an assignment of values to the model’s variables.
- Intuitively, the system state is a snapshot of the system’s execution. In this snapshot, every variable has some value.
- If we are working with an OO or UML system, then looking at a snapshot of the system, there is an explicit number of instantiated objects executing, each object is in exactly one state of its state diagram, and each of its attributes has some value.
- This is one system state. If the system then executes an assignment statement, the value of one of its variables changes. The system enters a new system state.

LTL: System State

- There is some initial state of the system, defined by the initial values of all the variables.
- As the system executes, the values of the variables change. Each state represents a change from the previous state in the value of some variable. More than one variable can change the value between two consecutive states.

LTL: Executions

- A sequence of system states represents a particular execution of the system. Time progresses during the execution, but there is no keeping track of how long the system is in any specific state.
- An execution or a computation is a sequence of system states $\sigma = S_0, S_1, S_2, \ldots$

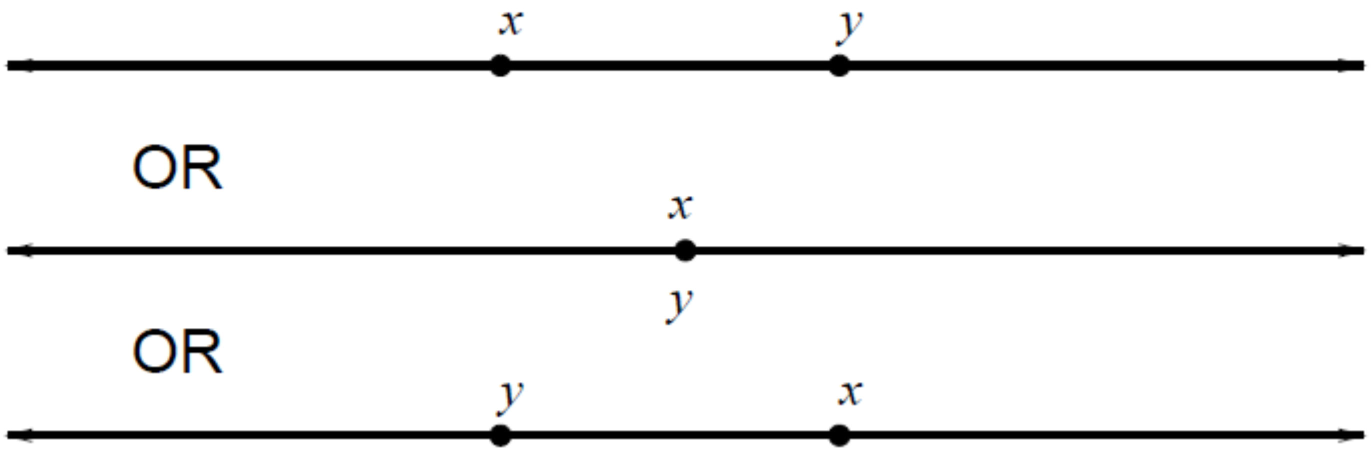
LTL: Semantics

- In linear temporal logic (LTL), formulas are evaluated concerning a particular execution and a particular state in that execution.
- Formulas evaluated concerning a state in an execution.

LTL: Ordered Time

Time is totally ordered.

$$\forall x, y \in Time : ((x < y) \vee (x = y) \vee (y < x))$$

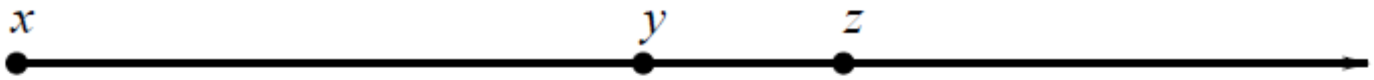


LTL: Boundedness

Time is usually bounded in the past and unbounded in the future.

$$\exists x \in Time : (\neg \exists z \in Time : (z < x))$$

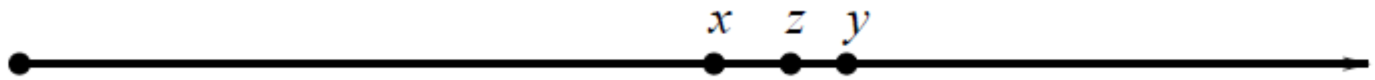
$$\forall y \in Time : (\exists z \in Time : (y < z))$$



LTL: Density

Time is continuous.

$$\forall x, y \in Time : (x < y \rightarrow \exists z \in Time : (x < z < y))$$



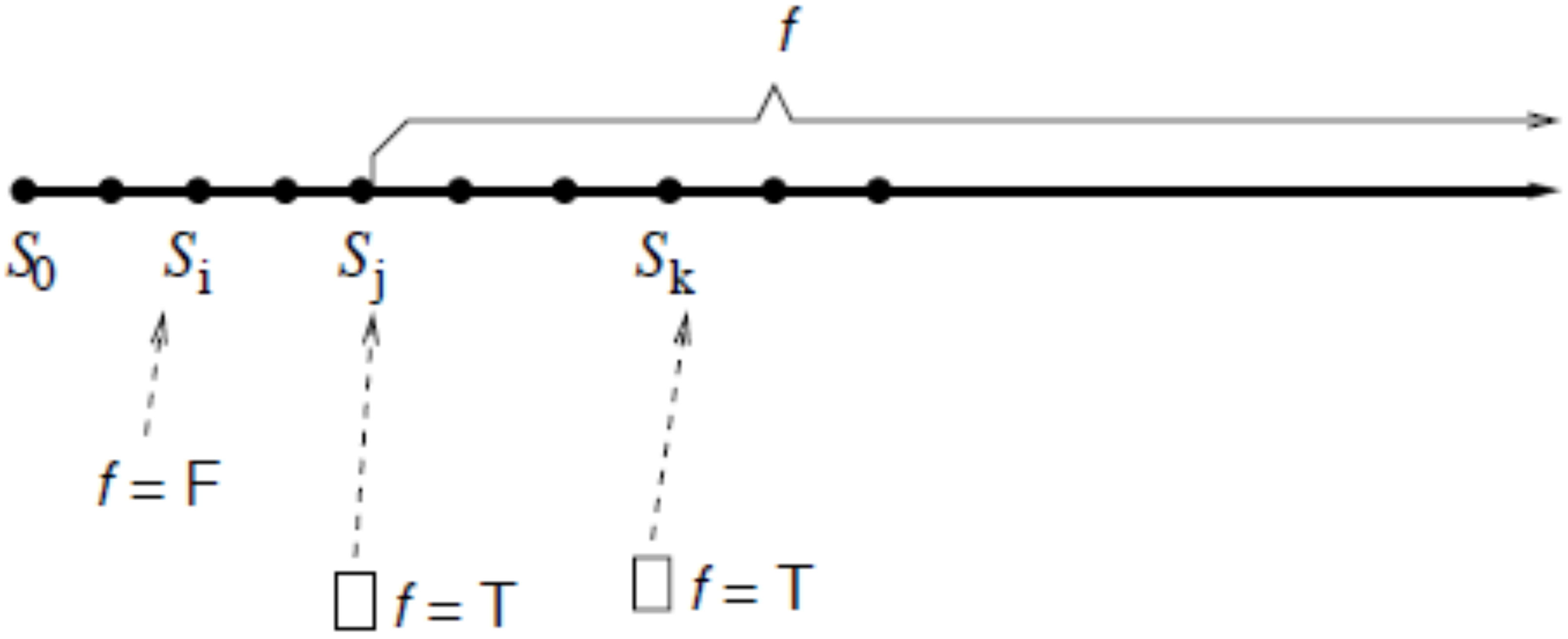
Temporal Connectives

- Connectives are shorthand notations that quantify over future system states.

- henceforth: $\square$
- eventually: $\Diamond$
- next: $\bigcirc$
- until: $\mathcal{U}$
- unless: $\mathcal{W}$

Henceforth

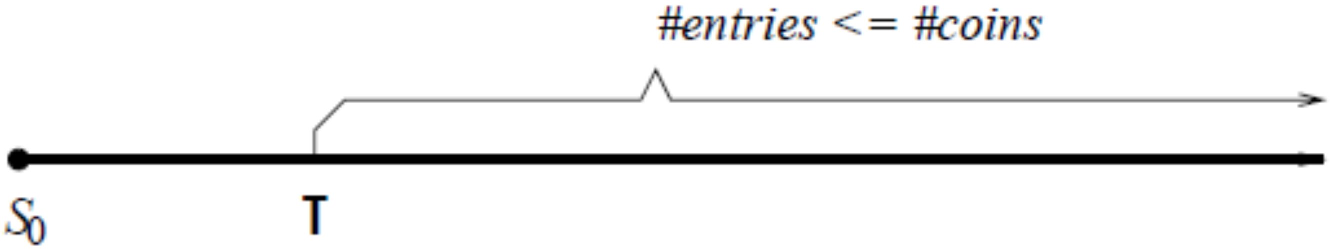
$$\Box f = \begin{cases} T & \text{if } f \text{ is true in the current and } \underline{\text{all}} \text{ future states} \\ F & \text{otherwise} \end{cases}$$



Example

It is always the case that the number of entries into the park is less than or equal to the number of coins received.

$$\models \Box (\#entries \leq \#coins)$$



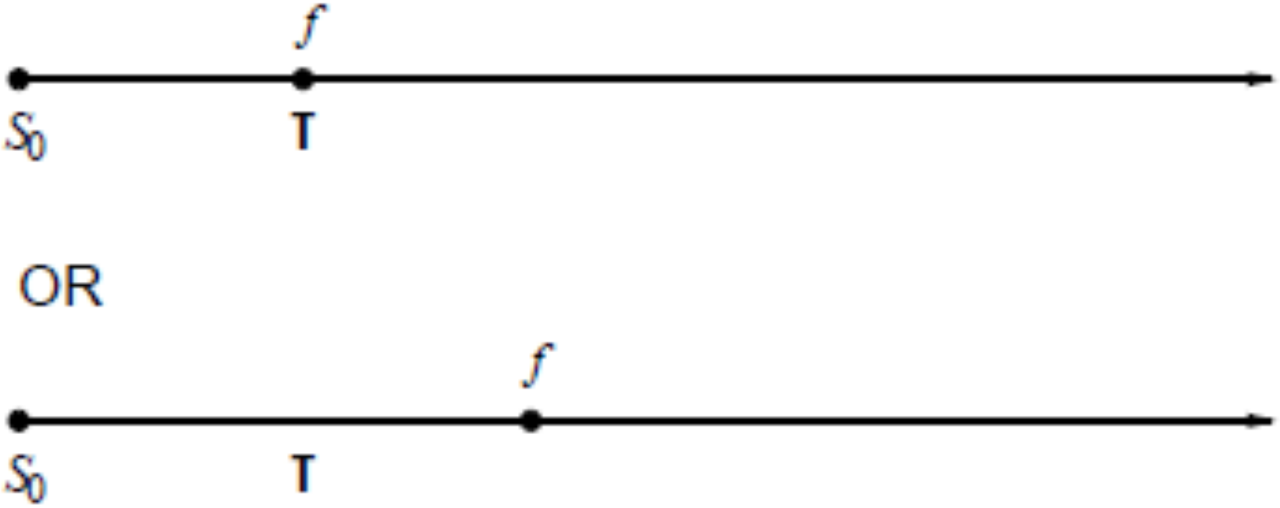
Shorthand for

$\forall t \in Time : (\#entries(t) \leq \#coins(t))$, which is a shorthand for

$\forall t \in Time : (t \geq t_0 \rightarrow (\#entries(t) \leq \#coins(t)))$

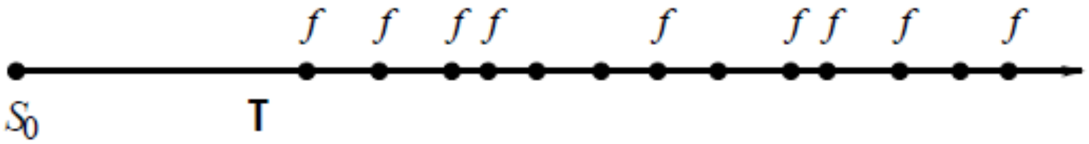
Eventually

$$\Diamond f = \begin{cases} T & \text{if } f \text{ is true in the current or } \underline{\text{some}} \text{ future state} \\ F & \text{otherwise} \end{cases}$$

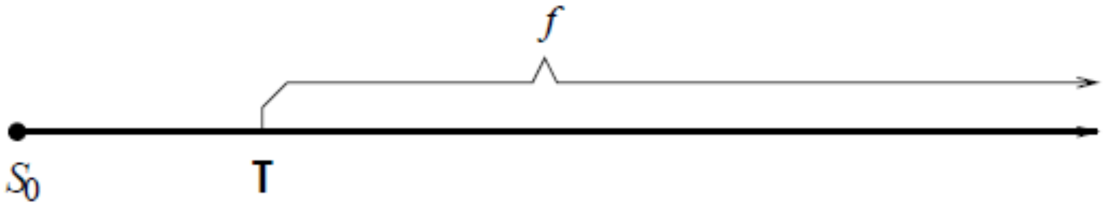


Note

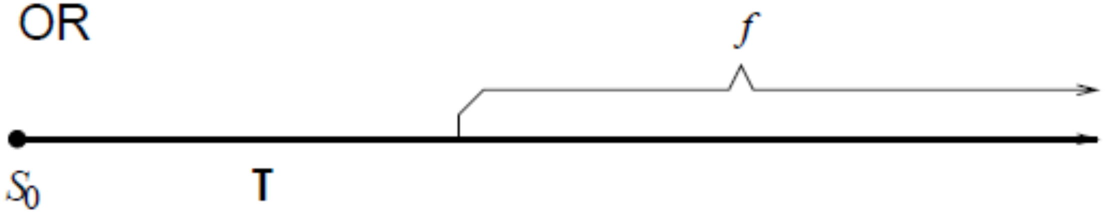
$\models \Box(\Diamond f)$ means that f happens infinitely often.



$\models \Diamond(\Box f)$ means that, eventually, f is true forever.



OR



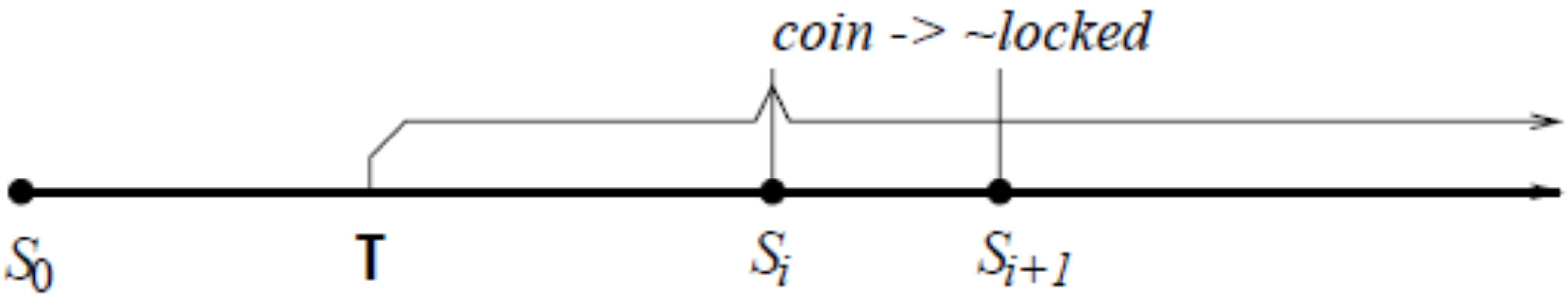
Next

$$\bigcirc f = \begin{cases} T & \text{if } f \text{ is true in the } \underline{\text{next}} \text{ future state} \\ F & \text{otherwise} \end{cases}$$

Example

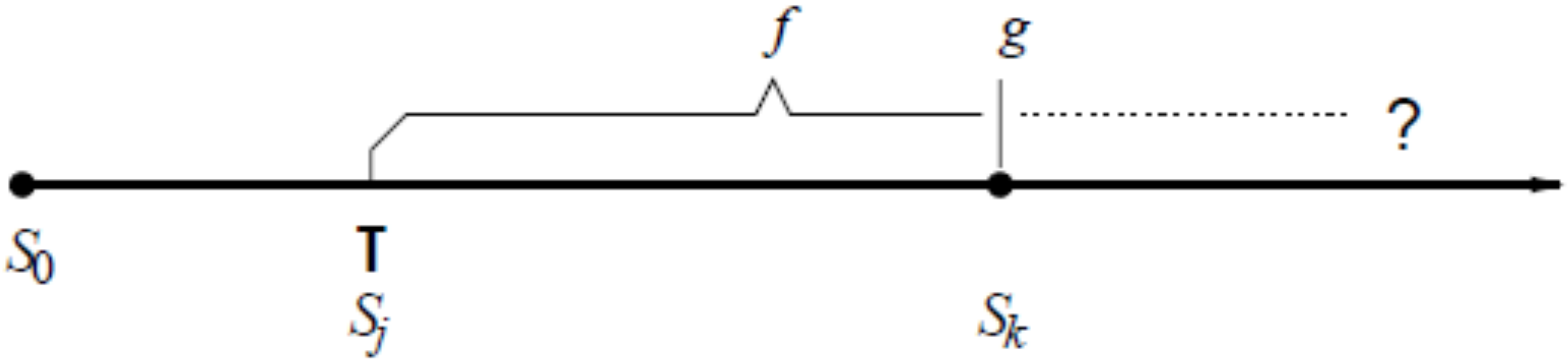
The turnstile is unlocked whenever a coin is inserted in the next state.

$$\models \square(coin \rightarrow \bigcirc \neg locked)$$



Until

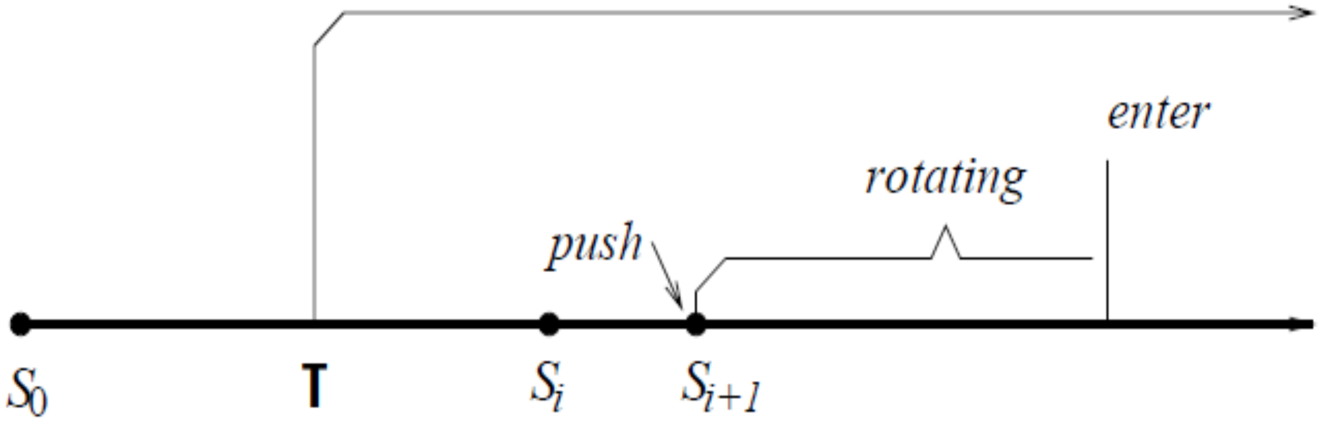
$$f\,\mathcal{U}\,g = \begin{cases} T & \text{if } g \text{ is eventually true and } f \text{ is true until } g \text{ is true} \\ F & \text{otherwise} \end{cases}$$



Example

Henceforth, if the barrier is pushed, then in the next state, the barrier will be rotating until the visitor has entered.

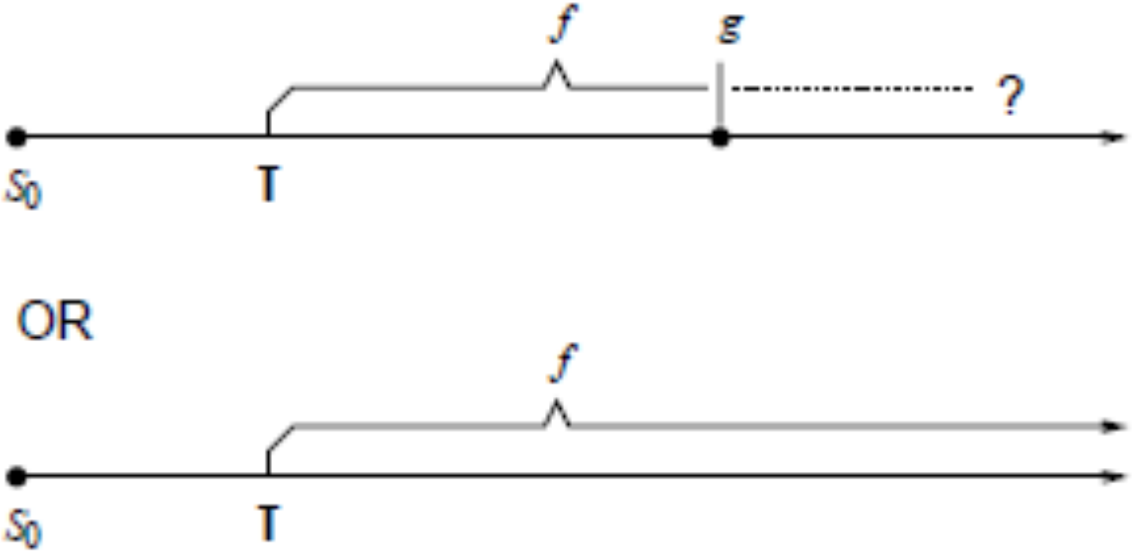
$$\models \Box(push \rightarrow \bigcirc(rotating\ \mathcal{U}\ enter))$$



Unless

Unless is similar to Until, but without the guarantee that g happens.
Unless is also called “weak until”.

$$f\,\mathscr{W}\,g = \begin{cases} T & \text{if } f \text{ is indefinitely true or } f \text{ holds until } g \text{ is true} \\ F & \text{otherwise} \end{cases}$$



$$f\,\mathscr{W}\,g \quad \textit{iff} \quad \Box f \text{ or } f\,\mathscr{U}\,g$$

Note

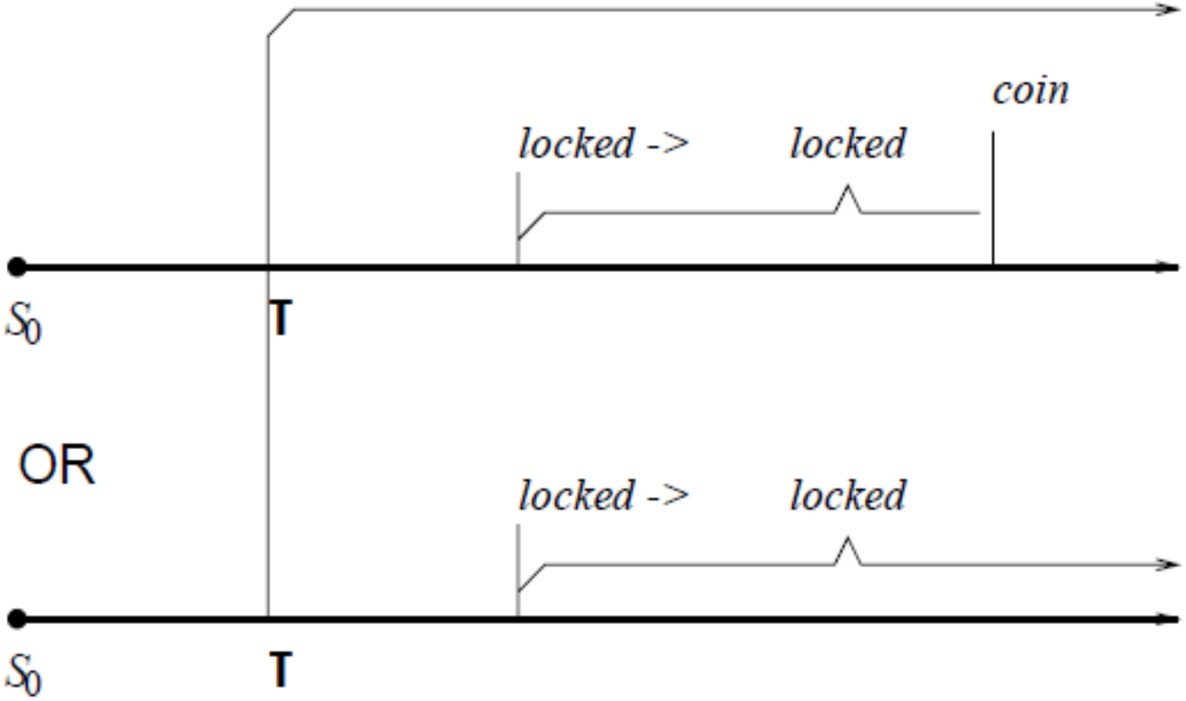
- Until is often used to describe some (temporarily) constant system property.

- Unless is used to describe some (temporarily) constant environmental property.

Example

Henceforth, if the turnstile is locked, it will stay locked unless a coin is entered.

$$\models \Box(locked \rightarrow (locked\, \mathcal{W}\, coin))$$



LTL and Finite State Machines

- LTL can be used to describe properties of a finite state machine.

LTL and Finite State Machines

$\models \Box(locked \rightarrow (locked\ \mathcal{W}\ coin))$

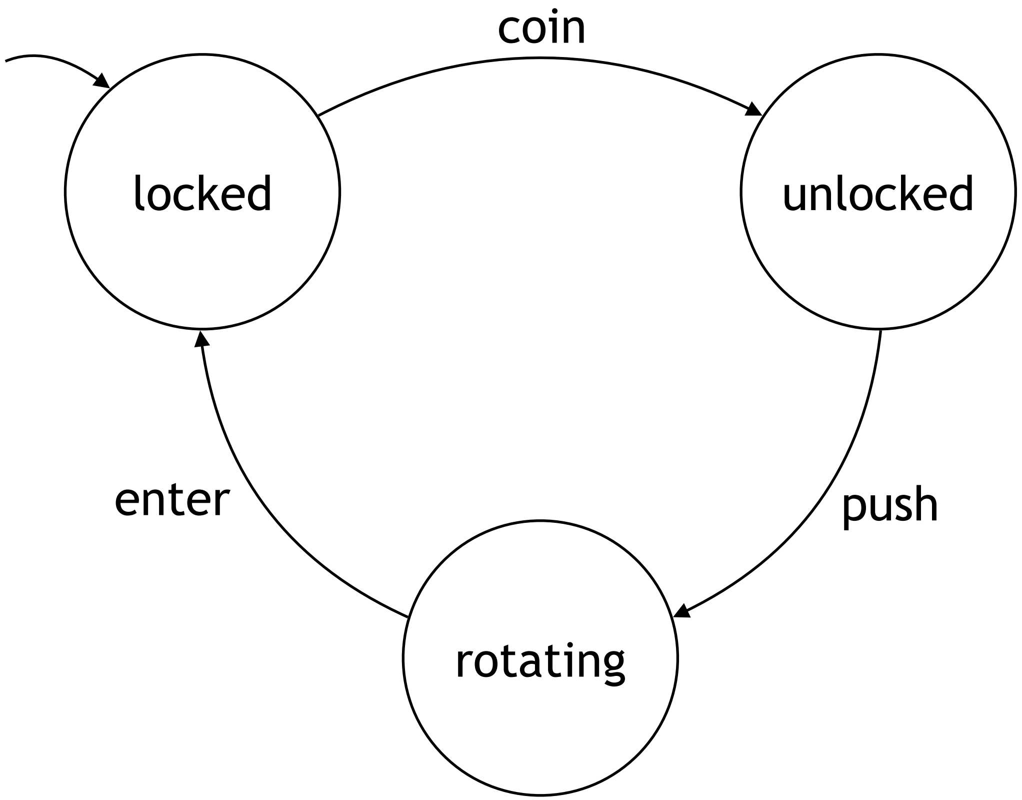
$\models \Box((locked \wedge coin) \rightarrow \bigcirc(unlocked))$

$\models \Box(unlocked \rightarrow (unlocked\ \mathcal{W}\ push))$

$\models \Box((unlocked \wedge push) \rightarrow \bigcirc(rotating))$

$\models \Box(rotating \rightarrow (rotating\ \mathcal{U}\ enter))$

$\models \Box((rotating \wedge enter) \rightarrow \bigcirc(locked))$



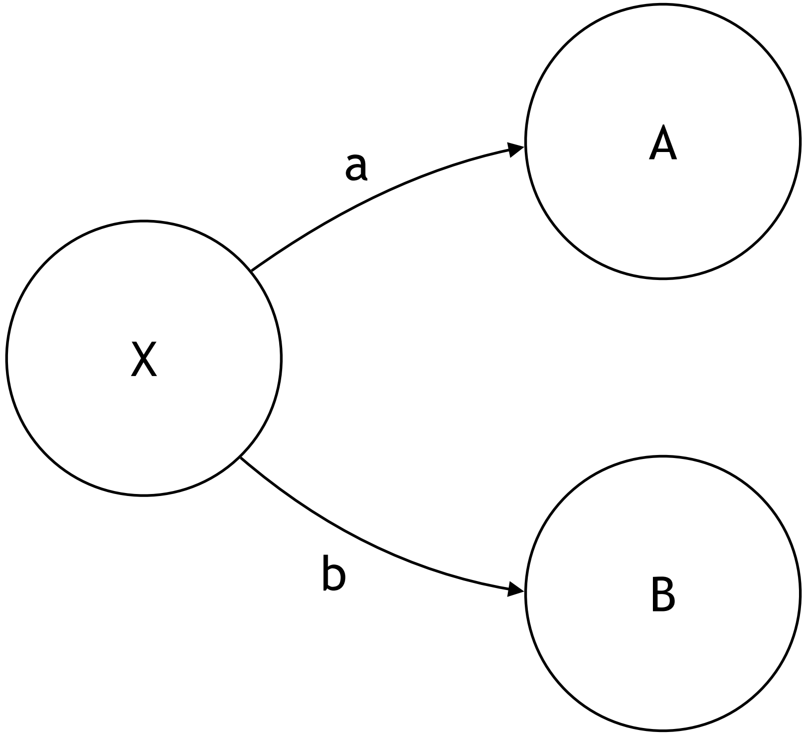
Note: $unlocked \not\equiv \neg locked$ and $\neg locked \equiv rotating \vee unlocked$

LTL and Finite State Machines

$$\models \Box(X \rightarrow (X \mathscr{W} (a \vee b)))$$

$$\models \Box((X \wedge a) \rightarrow \bigcirc(A))$$

$$\models \Box((X \wedge b) \rightarrow \bigcirc(B))$$



Exercise 1: Telephone System

- Variables:
    - Users u, u_1 and u_2.
- Predicates:
    - onhook(user) user's phone is on hook
    - offhook(user) user's phone is off hook
    - dialing(user) user is dialing a number
    - dial(user1, user2) user1 has dialed user2 (user1 ≠ user2)
    - busytone(user) user hears a busy tone
    - idletone(user) user hears an idle tone
    - ringtone(user) user hears a ringtone
    - dialtone(user) user hears a dial tone
    - connection(user1, user2) there is a connection between the phones of users 1 and 2 (user1 ≠ user2)

Exercise 1: Telephone System

1) It is always the case that the user's phone is either on hook or off hook.

For any given user u:

$$\models \Box\,(\text{onhook}(u) \vee \text{offhook}(u))$$

Exercise 1: Telephone System

2) A user always needs to pick up the phone before dialing.

For any given user u:

$$\models \Box(\neg \text{dialing}(u)\ \mathscr{W}\ \text{offhook}(u))$$

Exercise 1: Telephone System

3) After picking up the phone, the user eventually either puts the telephone back on the hook or dials.

For any given user u:

$$\models \Box(\text{offhook}(u) \rightarrow \Diamond(\text{dialing}(u) \vee \text{onhook}(u)))$$

Exercise 1: Telephone System

4) Whenever a user dials a number and hears the ring tone, a connection will only result after the other user picks up the phone.

For any given users u_1 and u_2:

$$\models \Box((\text{dial}(u_1,u_2) \wedge \text{ringtone}(u_1)) \rightarrow (\neg\text{connection}(u_1,u_2)\ \mathscr{W}\ \text{offhook}(u_2)))$$

Exercise 1: Telephone System

5) Immediately after the callee hangs up on a connection, the caller will hear an idle tone. After a time-out, the caller will hear a dial tone.

For any given users u_1 and u_2:

$$\models \Box((\text{connection}(u_1,u_2) \wedge \text{onhook}(u_2)) \rightarrow (\bigcirc\, \text{idletone}(u_1) \wedge \Diamond\, \text{dialtone}(u_1)))$$

Exercise 1: Telephone System

6) Without exception, users will stay connected as long as nobody hangs up.

For any given users u_1 and u_2:

$$\models \Box(\text{connection}(u_1,u_2) \rightarrow (\text{connection}(u_1,u_2)\, \mathcal{U} (\text{onhook}(u_1) \vee \text{onhook}(u_2))))$$

Exercise 2: Trains Crossing

Events:

a = “A train is approaching"

c = “A train is crossing"

l = “The lights are blinking"

b = “The barrier is down"

Exercise 2: Trains Crossing

1) When a train is crossing, the barrier must be down.

$$\models \square (c \rightarrow b)$$

This is a safety property.

Safety properties are usually of the form $\square\neg$bad

Another solution is

$$\models \square \neg (c \wedge \neg b)$$

Exercise 2: Trains Crossing

2) If a train is approaching or crossing, the lights must be blinking.

$$\models \square (a \vee c \rightarrow l)$$

This is a safety property.

Safety properties are usually of the form $\square \neg$bad

Another solution is

$$\models \square \neg((a \vee c) \wedge \neg l)$$

Exercise 2: Trains Crossing

3) If the barrier is up and the lights are off, then no train is coming or crossing.

$$\models \Box(\neg b \wedge \neg l \rightarrow \neg a \wedge \neg c)$$

This is a safety property.

Safety properties are usually of the form $\Box\neg\text{bad}$

Another solution is

$$\models \Box\neg(\neg b \wedge \neg l \wedge (a \vee c))$$

Exercise 2: Trains Crossing

4) When a train is approaching, the train will eventually cross.

$$\models \Box(a \rightarrow \Diamond c)$$

This is a liveness property.

Liveness properties are usually of the form
$\Box$(initiated → $\Diamond$terminates)

Exercise 2: Trains Crossing

5) When a train is approaching, the barrier will eventually be down before it crosses.

$$\models \Box(a \land \neg c \rightarrow \Diamond \neg c \mathscr{W} b)$$

This is a liveness property.

Liveness properties are usually of the form

$\Box$(initiated → $\Diamond$terminates)

Exercise 2: Trains Crossing

6) If a train finishes crossing, the barrier will be eventually risen.

$$\models \Box(c \land \bigcirc \neg c \rightarrow \bigcirc \Diamond \neg b)$$

This is a safety property.

Safety properties are usually of the form $\Box \neg$bad

Another solution is

$$\models \Box \neg(c \land \neg b)$$

Note

- Something happens infinitely often = $\Box \Diamond \alpha$
- Example: The barrier is risen infinitely often = $\Box \Diamond \neg$barrier

- The dual is a latching condition = $\Diamond \Box \alpha$
- Example: At a given point, no more trains are approaching = $\Diamond \Box \neg$approach

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Temporal Logic

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Risk Analysis

Risk

A risk is an uncertain factor whose occurrence may result in some loss of satisfaction with some corresponding objective. The risk is said to negatively impact this objective.

[van Lamsveerde, section 3.4]

- Has a likelihood to occur
- Has consequences
- Product-related risks: may result in the product's inability to deliver the required services or the required quality of services, including safety hazards and security threats.
- Process-related risks: may result in delayed product delivery, cost overruns, deterioration of project team morale, etc.

Risk

- If risks go unrecognized or underestimated, the requirements will be incomplete or inadequate as they will not consider such risks.

Goal: Early risk management at requirements time.

Risk Classification

- Software Requirement Risks
- Software Cost Risks
- Software Scheduling Risks
- Software Quality Risks

Software Requirement Risks

1. Lack of analysis for change of requirements.
2. Change the extension of requirements
3. Lack of report for requirements
4. Poor definition of requirements
5. Ambiguity of requirements
6. Change of requirements
7. Inadequate requirements
8. Impossible requirements
9. Invalid requirements

Software Cost Risks

1. Lack of good estimation in projects
2. Unrealistic schedule
3. The hardware does not work well
4. Human errors
5. Lack of testing
6. Lack of monitoring
7. Complexity of architecture
8. Large size of architecture
9. Extension of requirements change
10. The tools do not work well
11. Personnel change, Management change, technology change, and environment change
12. Lack of reassessment of the management cycle

Software Scheduling Risks

1. Inadequate budget
2. Change of requirements and extension of requirements
3. Human errors
4. Inadequate knowledge of tools and techniques
5. Long-term training for personnel
6. Lack of employment of manager experience
7. Lack of enough skill
8. Lack of good estimation in projects

Software Quality Risks

1. Inadequate documentation
2. Lack of project standard
3. Lack of design documentation
4. Inadequate budget
5. Human errors
6. Unrealistic schedule
7. Extension of requirements change
8. Poor definition of requirements
9. Lack of enough skill
10. Lack of testing and good estimation in projects
11. Inadequate knowledge of techniques, programming language, tools, and so on

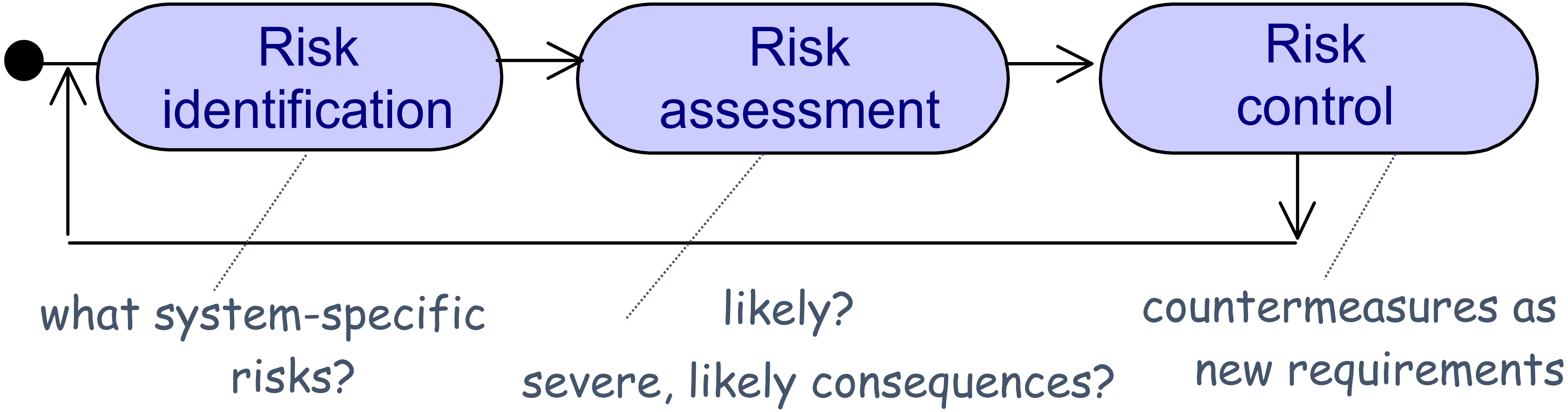
Risk Management

Risk Management attempts to manage the degree to which a project is exposed to risks of quality, delay, or failure.

Some tasks are to

- identify risks
- estimate the likelihood of occurrence of risks
- predict the impact of risks on the project

Risk Management


Risk identification
Risk assessment
Risk control
what system-specific risks?
likely?
severe, likely consequences?
countermeasures as new requirements

- Risk management is iterative
  - countermeasures may introduce new risks
- Poor risk management is a major cause of software failure
  - natural inclination to conceive over-ideal systems (nothing can go wrong)
  - unrecognized, underestimated risks lead to incomplete, inadequate requirements

Risk Identification: Risk Checklists

- Instantiation of risk categories to project specifics
  - associated with corresponding requirements categories (cf. Chap. 1)
- Product-related risks: requirement unsatisfaction in functional or quality requirement categories
  - information inaccuracy, unavailability, unusability, poor response time, poor peak throughput, etc.

  e.g. inaccurate estimates of train speed and positions?
- Process-related risks: top 10 risks [Boehm, 1989]
  - requirement volatility, personnel shortfalls, dependencies on external sources, unrealistic schedules/budgets, etc.
  - poor risk management

    e.g. Unexperienced developer team for train system?

Risk Identification: Component Inspection

- For product-related risks
- Review each component of the system-to-be: human, device, software components
  - can it fail?
  - how?
  - why?
  - what are the possible consequences?

  e.g. onboard train controller, station computer, tracking system, and communication infrastructure, …
- Finer-grained components lead to more accurate analysis

  e.g. acceleration controller, doors controller, track sensors, ...

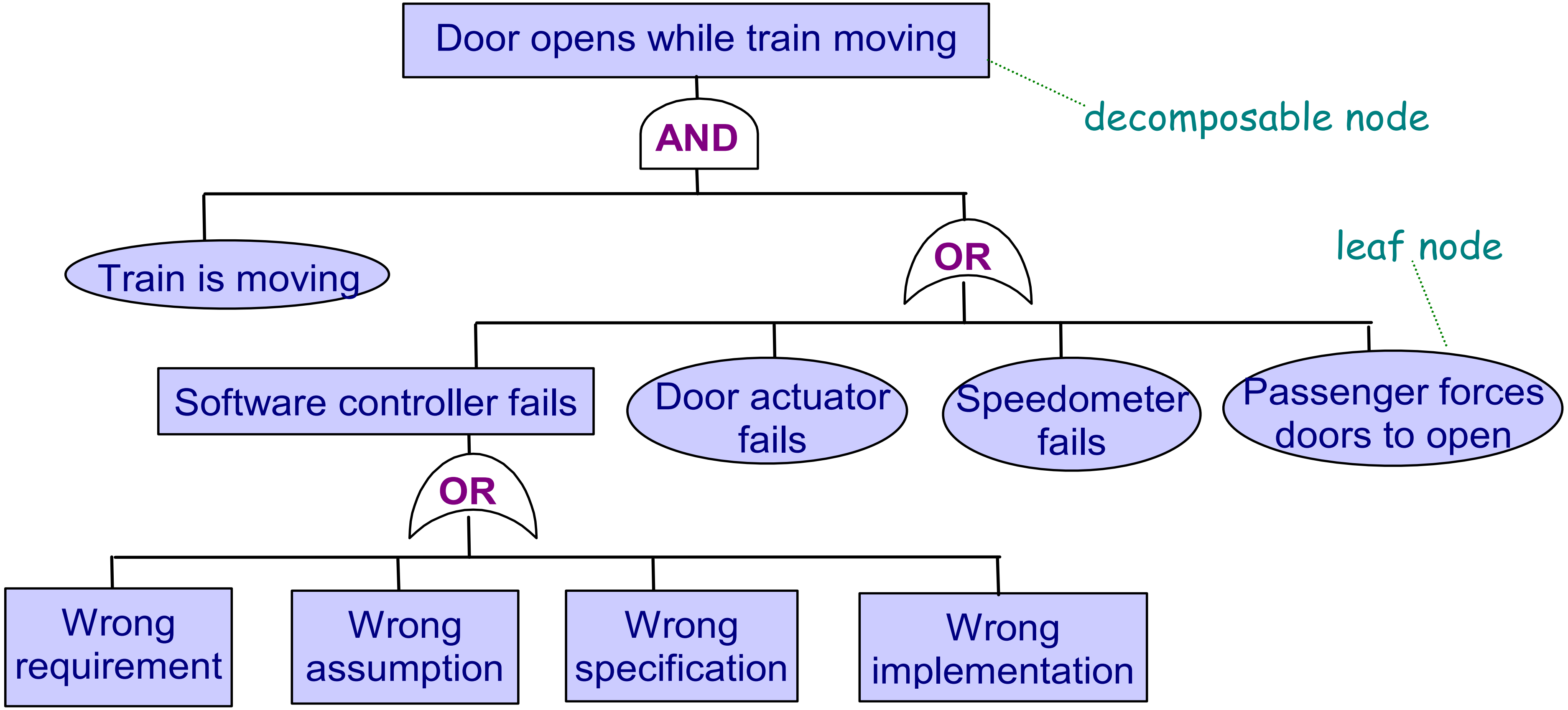
Risk Identification: Risk Trees

- Tree organization for causal linking of failures, causes, consequences

- Failure nodes: independent failure event or condition
  - decomposable into finer-grained nodes

- Logical nodes: AND/OR, causal links through logical nodes
  - AND-node: child nodes must all occur for the parent node to occur as a consequence
  - OR-node: only one child node needs to occur

Risk Tree: Example



decomposable node

leaf node

Building Risk Trees: Heuristic Identification of Failure Nodes

- Checklists, component inspection identify failure nodes
- Guidewords: keyword-based patterns of failure
  - NO: “something is missing”
  - MORE: “there are more things than expected”
  - LESS: “there are fewer things than expected”
  - BEFORE: “something occurs earlier than expected”
  - AFTER: “something occurs later than expected”
- But problems frequently happen due to *combinations* of basic failure events / conditions.

Analyzing Failure Combinations: Cut Set of a Risk Tree

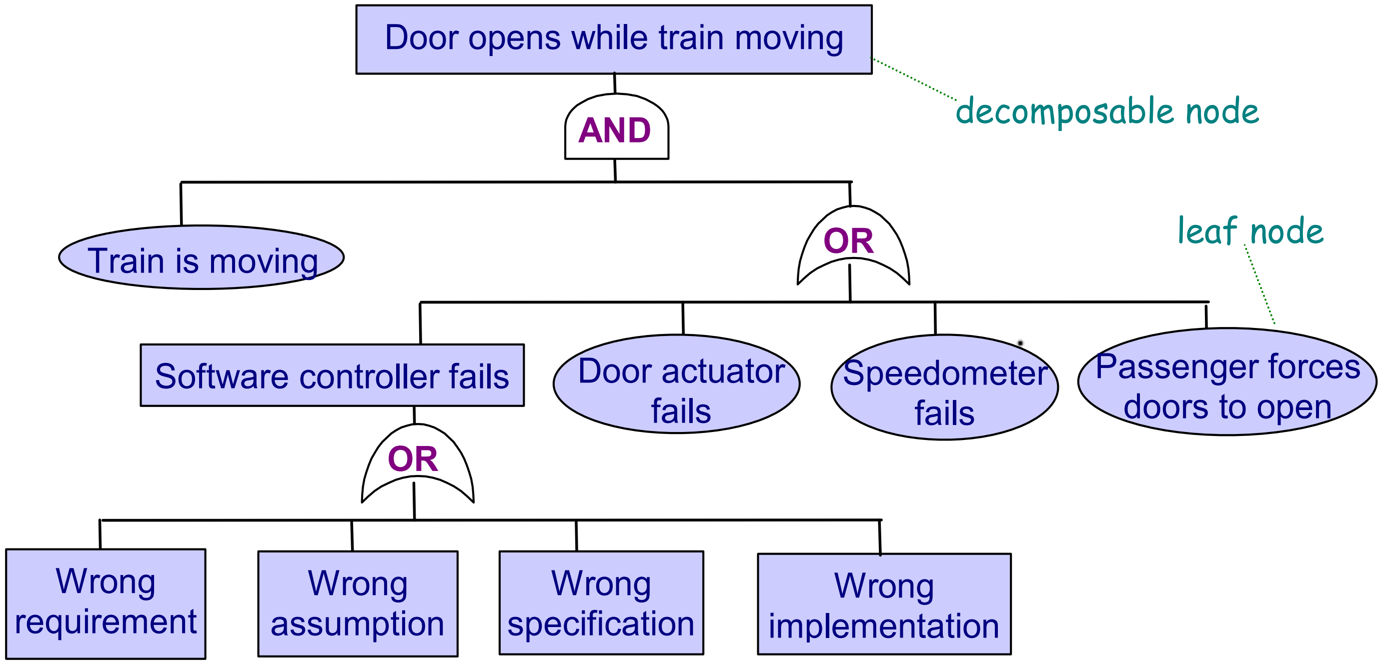
- Cut set of risk tree RT: set of minimal AND-combinations of RT's leaf nodes sufficient for causing RT's root node
  - Cut-set tree of RT: set of its leaf nodes = RT's cut set

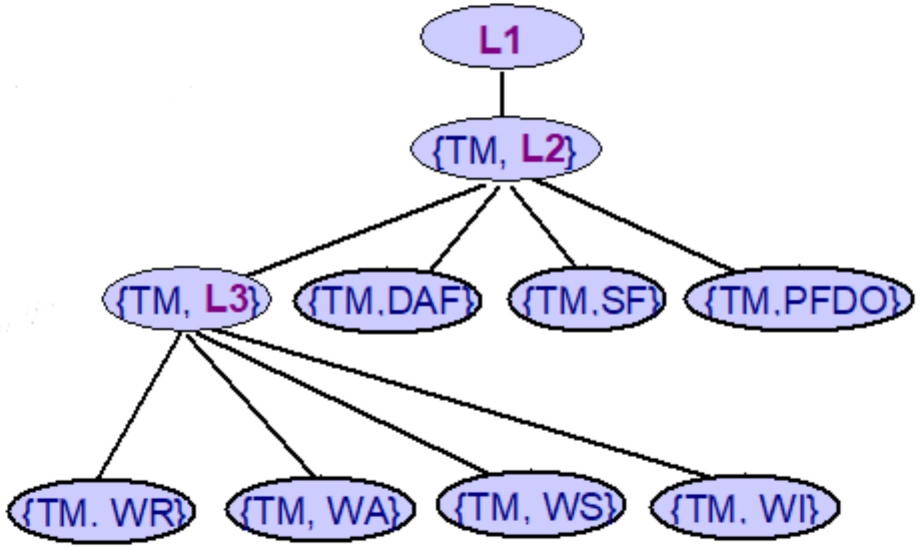
- Derivation of cut-set tree CST of RT:
  - CST's top node := RT's top logical node
  - If current CST node is OR-node:

    expand it with RT's corresponding alternative child nodes
  - If current CST node is AND-node:

    expand it in single aggregation of RT's conjoined child nodes
  - Termination when CST's child nodes are all aggregations of leaf nodes from RT

Cut Set of a Risk Tree: Derivation

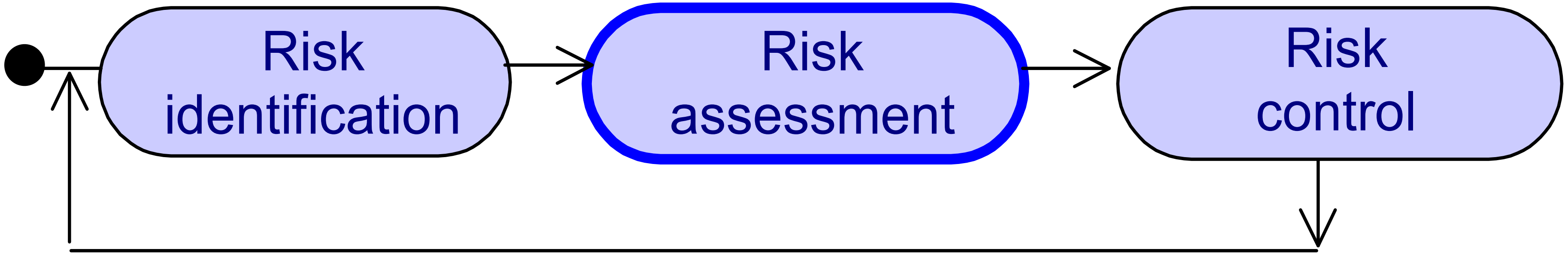




Risk Identification: Using Elicitation Techniques

- Scenarios to point out failures from WHAT IF questions
  - interactions not occurring
  - interactions occurring too late
  - unexpected interactions (e.g. under wrong conditions)
- Knowledge reuse: typical risks from similar systems
- Group sessions: focused on the identification of project-specific risks

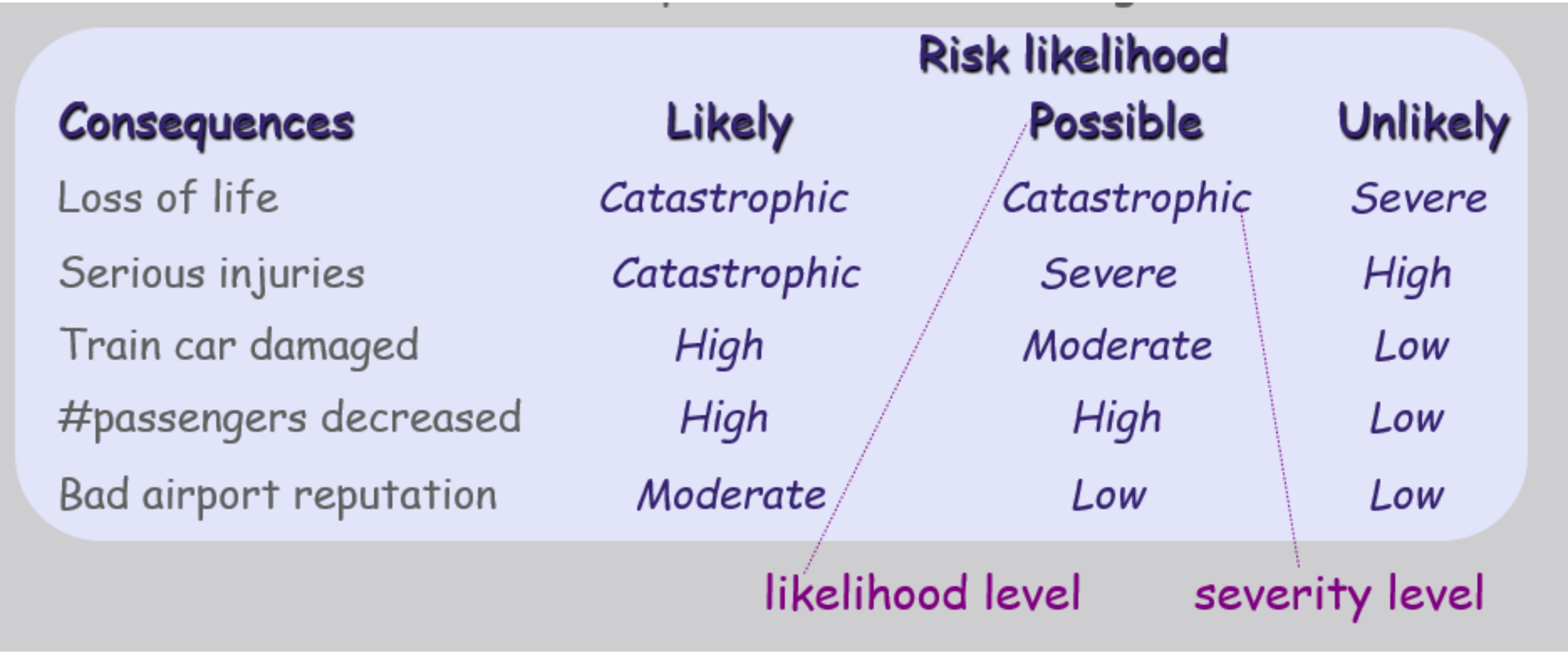
Risk Assessment



- Goal: assess likelihood of risks + severity, likelihood of consequences, to control high-priority risks
- Qualitative assessment: use qualitative estimates (levels)
  - for likelihood: {very likely, likely, possible, unlikely, ...}
  - for severity: {catastrophic, severe, high, moderate, ...}
- Risk likelihood-consequence table for each risk
- Risk comparison/prioritization on severity levels

Qualitative Risk Assessment Table: Example

Risk: “Doors open while train moving:

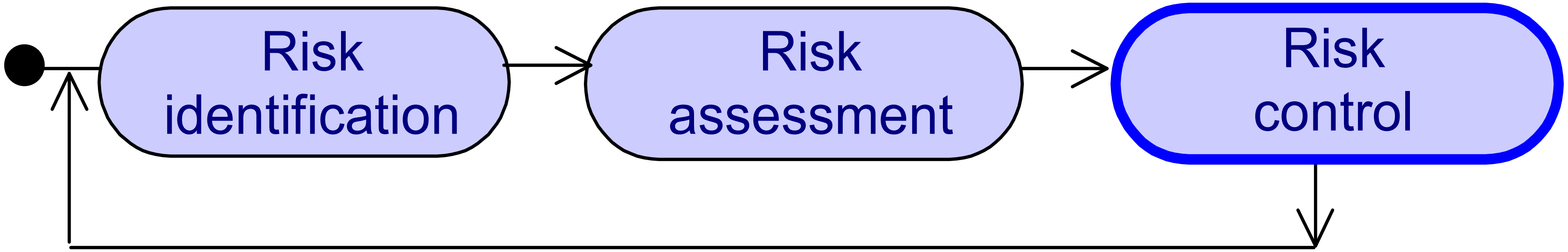


		Risk likelihood	
Consequences	**Likely**	**Possible**	**Unlikely**
Loss of life	*Catastrophic*	*Catastrophic*	*Severe*
Serious injuries	*Catastrophic*	*Severe*	*High*
Train car damaged	*High*	*Moderate*	*Low*
#passengers decreased	*High*	*High*	*Low*
Bad airport reputation	*Moderate*	*Low*	*Low*

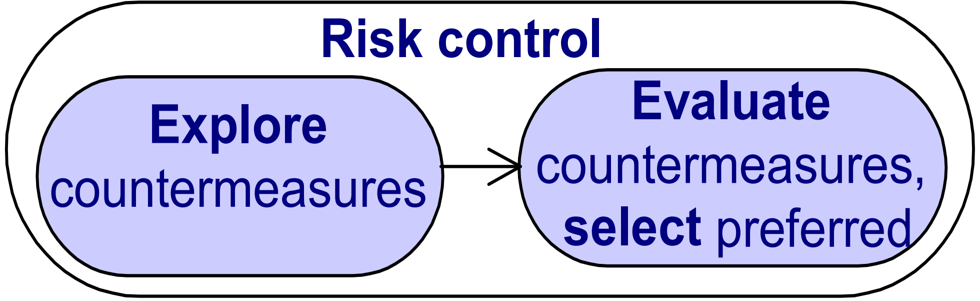
likelihood level

severity level

Risk Control



- Goal: Reduce high-exposure risks through countermeasures
  - yields new or adapted requirements
  - should be cost-effective


Risk control
Explore countermeasures
Evaluate countermeasures, **select** preferred

Exploring Countermeasures

- Using elicitation techniques
  - interviews, group sessions
- Reusing known countermeasures

  e.g. generic countermeasures to top 10 risks [Boehm, 1989]
  - simulation ✂ poor performance
  - prototyping, task analysis ✂ poor usability
  - use of cost models ✂ unrealistic budgets/schedules
- Using risk reduction tactics

Risk Reduction Tactics

- Reduce risk likelihood: new requirements to ensure significant decrease

  e.g. "Prompts for driver reaction regularly generated by software"

- Avoid risk: new requirements to ensure risk may never occur

  e.g. "Doors may be opened by software-controlled actuators only"

- Reduce consequence likelihood: new requirements to ensure significant decrease of consequence likelihood

  e.g. "Alarm generated in case of door opening while train moving"

- Avoid risk consequence: new requirements to ensure consequence may never occur

  e.g. "No collision in case of inaccurate speed/position estimates"

- Mitigate risk consequence: new requirements to reduce severity of consequence(s)

  e.g. "Waiting passengers informed of train delays"

Selecting Preferred Countermeasures

- Evaluation criteria for preferred countermeasure:
  - contribution to critical non-functional requirements
  - contribution to the resolution of *other* risks
  - cost-effectiveness
- Cost-effectiveness is measured by risk-reduction leverage (RRL):

$$\mathrm{RRL}(r, cm) = (\mathrm{Exp}(r) - \mathrm{Exp}(r|cm)) / \mathrm{Cost}(cm)$$

Exp (r): exposure of risk r

Exp ($r|cm$): new exposure of r if countermeasure cm is selected

- Select countermeasures with the highest RRLs
  - Refinable through cumulative countermeasures and RRLs.

Risks Should Be Documented

- To record/explain why these countermeasure requirements, to support system evolution
- For each identified risk:
    - conditions/events for the occurrence
    - estimated likelihood
    - possible causes and consequences
    - estimated likelihood and severity of each consequence
    - identified countermeasures + risk-reduction leverages
    - selected countermeasures

Defect Detection and Prevention (DDP)

DDP is a software tool developed by NASA

DDP Steps:

1. Identify the most critical requirements
2. Identify potential risks
3. Estimate the impact of each risk on each requirement
4. Identify possible countermeasures
5. Identify the most effective countermeasures

Result of DDP: Optimized collection of mitigating actions that may be applied to project

Example: Meeting Scheduler

- A meeting initiator informs potential participants about the need for a meeting and specifies a date range within which the meeting should take place, asking them to return their scheduling constraints

- Constraints are expressed as two sets:
  - one exclusion set (dates when a participant cannot attend)
  - one preference set (dates when a participant prefers to attend)

- Initiator also asks for specific requirements of meeting room

Example: Meeting Scheduler

- All correspondence with participants is via email

- The meeting should be scheduled within the stated date range and not be in any exclusion sets. The date should also belong to as many preference sets as possible, especially of the “important” participants.

- A new schedule cycle is required in case of a date or room conflict.

- Conflicts can be resolved in several ways: the initiator may extend the date range, some participants may remove dates from their exclusion set, or some may decline the invitation to attend the meeting.

Defect Detection and Prevention (DDP)

DDP Steps:

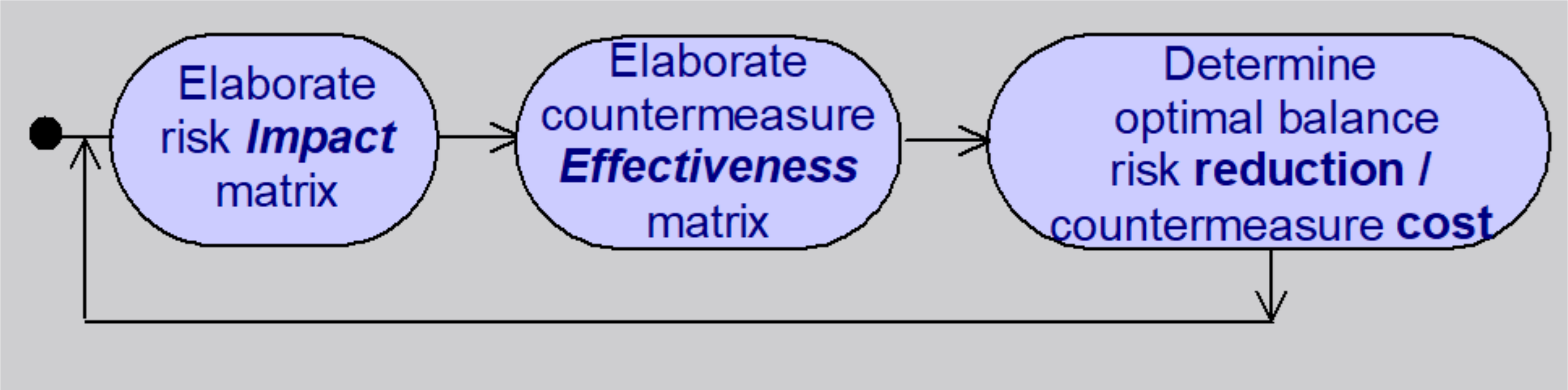
1. Identify the most critical requirements and their relative importance
2. Identify potential risks, and their likelihood
3. Estimate the impact of each risk on each requirement
4. Identify possible countermeasures
5. Identify the most effective countermeasures

Goal:

- To develop a set of prioritized risks to be addressed
- Perhaps to identify which requirements are the most “risk-driving”

DDP Process

- Risk Consequence Table
- Risk Countermeasure Table


Elaborate risk ***Impact*** matrix
Elaborate countermeasure ***Effectiveness*** matrix
Determine optimal balance risk **reduction /** countermeasure **cost**

Risk Consequence Table

		Risks				
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change
Likelihood		0.4	0.3	0.1	0.3	0.5
Reduce time taken to schedule meetings	0.5					
Notify participants when time and place are found	0.4					
Increase participant average attendance	0.3					
Reduce schedule conflicts	0.6					

Risk Impact Matrix

		Risks				
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change
Likelihood		0.4	0.3	0.1	0.3	0.5
Reduce time taken to schedule meetings	0.5	0.6	0.8	0.2	0.7	0.2
Notify participants when time and place are found	0.1	0	0.8	0	1	0.2
Increase participant average attendance	0.3	0.8	0.8	0	0.8	0.5
Reduce schedule conflicts	0.6	0.2	1	0	0	0.7

Impact(risk, req) = estimate of loss of requirement
0 = no loss
1 = total loss

Loss of Objective

		Risks					
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Loss of objective**
Likelihood		0.4	0.3	0.1	0.3	0.5	
Reduce time taken to schedule meetings	0.5	0.6	0.8	0.2	0.7	0.2	**0.405**
Notify participants when time and place are found	0.4	0	0.8	0	1	0.2	**0.256**
Increase participant average attendance	0.3	0.8	0.8	0	0.8	0.5	**0.315**
Reduce schedule conflicts	0.6	0.2	1	0	0	0.7	**0.438**

$$\text{LossOfObjective(req)} = \text{weight(req)} \times \sum_{risk} \text{impact(risk,req)} \times \text{likelihood(risk)}$$

Risk Driving Requirements

		Risks					
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Loss of objective**
Likelihood		0.4	0.3	0.1	0.3	0.5	
Reduce time taken to schedule meetings	0.5	0.6	0.8	0.2	0.7	0.2	**0.405**
Notify participants when time and place are found	0.4	0	0.8	0	1	0.2	**0.256**
Increase participant average attendance	0.3	0.8	0.8	0	0.8	0.5	**0.315**
Reduce schedule conflicts	0.6	0.2	1	0	0	0.7	**0.438**
Risk criticality		**0.264**	**0.468**	**0.01**	**0.297**	**0.375**	

Risk-driving requirements are the requirements that are most at risk of not being achieved.

Risk Criticality

		Risks					
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Loss of objective**
Likelihood		0.4	0.3	0.1	0.3	0.5	
Reduce time taken to schedule meetings	0.5	0.6	0.8	0.2	0.7	0.2	**0.405**
Notify participants when time and place are found	0.4	0	0.8	0	1	0.2	**0.256**
Increase participant average attendance	0.3	0.8	0.8	0	0.8	0.5	**0.315**
Reduce schedule conflicts	0.6	0.2	1	0	0	0.7	**0.438**
Risk criticality		**0.264**	**0.468**	**0.01**	**0.297**	**0.375**	

$$\text{RiskCriticality(risk)} = \text{likelihood(risk)} \times \sum_{req} \text{impact(risk,req)} \times \text{weight(req)}$$

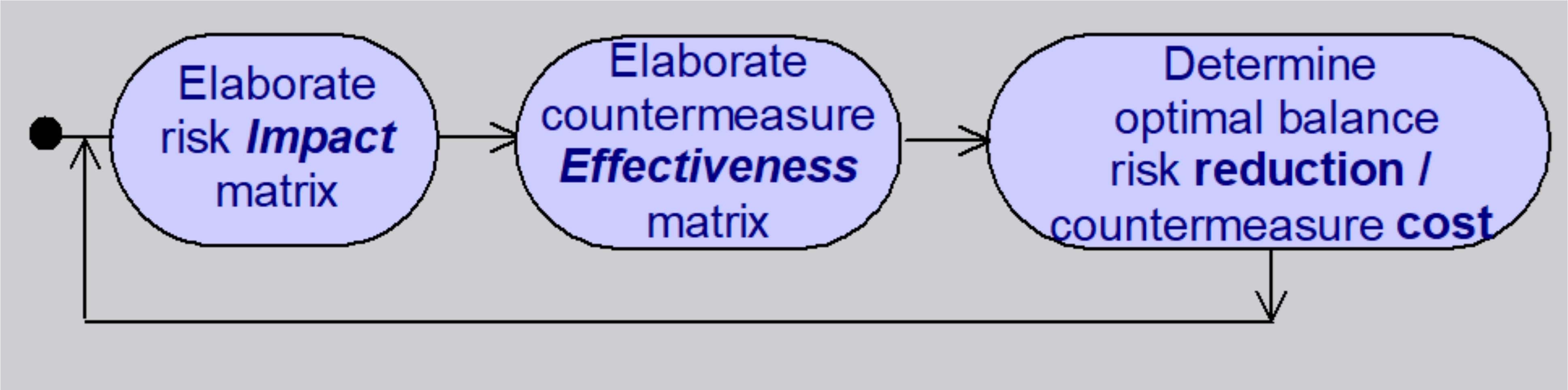
Tall Poles

		Risks					
Requirements	Weight	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Loss of objective**
Likelihood		0.4	0.3	0.1	0.3	0.5	
Reduce time taken to schedule meetings	0.5	0.6	0.8	0.2	0.7	0.2	**0.405**
Notify participants when time and place are found	0.4	0	0.8	0	1	0.2	**0.256**
Increase participant average attendance	0.3	0.8	0.8	0	0.8	0.5	**0.315**
Reduce schedule conflicts	0.6	0.2	1	0	0	0.7	**0.438**
Risk criticality		**0.264**	**0.468**	**0.01**	**0.297**	**0.375**	

Tall Poles are the most critical risks, having the most severe consequences.

DDP Process

- Risk Consequence Table
- Risk Countermeasure Table


Elaborate risk ***Impact*** matrix
Elaborate countermeasure ***Effectiveness*** matrix
Determine optimal balance risk **reduction /** countermeasure **cost**

Defect Detection and Prevention (DDP)

DDP Steps:

1. Identify the most critical requirements
2. Identify potential risks, and their likelihood
3. Estimate the impact of each risk on each requirement
4. Identify possible countermeasures, and their effectiveness in reducing risk
5. Identify the most effective countermeasures

Goal:

- Identify options for preventing or detecting risks
  - Preventative measures, Analyses, Process controls, Tests, Mitigations
- Perhaps to identify the most effective countermeasures

Identify Possible Countermeasures

1. Using elicitation techniques: such as interviews, group sessions, etc.
2. Reusing available countermeasures: Boehm(1989) listed the top ten risks with alternative countermeasures for each.

Boehm’s Top 10 Risks

Risk Item	Risk Management Technique
Personnel shortfall	Staffing with top talent, job matching, team building, key personnel agreements, cross training
Unrealistic schedules and budgets	Detailed milestone cost and schedule estimation, design to cost, incremental development, software reuse, requirements scrubbing
Developing the wrong functions and properties	Organizational analysis, mission analysis, operations-concept formulation, user surveys and user participation, prototyping, early user's manuals
Developing the wrong user interface	Prototyping, scenarios, task analysis, user participation
Gold-plating (e.g. implementing "neat features" not asked for by consumer)	Requirements scrubbing, prototyping, cost-benefit analysis, designing to cost
Continuing stream of requirements changes	High change threshold information hiding, incremental development (deferring changes to later increments)
Shortfalls in externally-furnished components (e.g. component reuse)	Benchmarking, inspections, reference checking, compatibility analysis
Shortfalls in externally performed tasks (e.g. worked performed by a contractor)	Reference checking, pre-award audits, award-fee contracts, competitive design or prototyping, team building
Real-time performance shortfalls	Simulation, benchmarking, modeling, prototyping, instrumentation, tuning
Straining computer science capabilities	Technical analysis, cost-benefit analysis, prototyping, reference checking

Identify Possible Countermeasures

3. Using risk-reduction tactics
    - Reduce risk likelihood
        - to reduce the risk of the train's driver falling asleep or being distracted from controlling the acceleration process requires a prompt reaction to being generated by the software.
    - Avoid risk
        - to avoid the risk of passengers forcing doors to open:
            - require that the door actuator reacts to the software controller exclusively, and
            - the software checks the train's speed before responding to any opening request from passengers.
    - Reduce consequence likelihood
        - the likelihood of severe injuries or loss of life in case of unexpected door opening might be reduced by requiring the software to generate an alarm in case doors open while the train is moving.

Identify Possible Countermeasures

3. Using risk-reduction tactics
   - Avoid risk consequences
     - introduce requirements that ensure that train collisions cannot occur in case the risk of inaccurate train position or speed information occurs.
   - Mitigate risk consequences
     - introduce requirements that reduce the severity of consequences of this tolerated risk. For example, request videoconferencing etc., in case of a last-minute absence of a participant in a meeting.

Risk Countermeasure Table

	Risks				
Countermeasures	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change
Criticality	0.264	0.468	0.01	0.297	0.375
Send e-mail reminder					
Change the meeting, increase time range					
Allow system to have access to personal e-agendas					
Change the meeting, fewer constraints (equipment)					
Cancel a meeting and send e-mail confirmation					

Countermeasure Effectiveness Matrix

	Risks				
Countermeasures	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change
Criticality	0.264	0.468	0[illegible]1	0.297	0.375
Send e-mail reminder	0.7	0.7	0	0.1	0
Change the meeting, increase time range	0.2	0.2	0	0.1	0
Allow system to have access to personal e-agendas	0.3	0.2	0.1	0.2	0.3
Change the meeting, fewer constraints (equipment)	0	0	0.9	0	0
Cancel a meeting and send e-mail confirmation	0.8	0.8	1	0.7	0.9

Effect(cm, risk) = estimate of reduction of risk
0 = no reduction
1 = risk eliminated

Combined Risk Reduction

	Risks				
Countermeasures	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change
Criticality	0.264	0.468	0.01	0.297	0.375
Send e-mail reminder	0.7	0.7	0	0.1	0
Change the meeting, increase time range	0.2	0.2	0	0.1	0
Allow system to have access to personal e-agendas	0.3	0.2	0.1	0.2	0.3
Change the meeting, fewer constraints (equipment)	0	0	0.9	0	0
Cancel a meeting and send e-mail confirmation	0.8	0.8	1	0.7	0.9
Combined Risk Reduction	**0.966**	**0.962**	**1**	**0.806**	**0.93**

$$\text{CombinedRiskReduction(risk)} = 1 - \prod (1 - \text{reduction(cm,risk)})$$

Effect of Countermeasure

	Risks					
Countermeasures	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Effect of Counter measure**
Criticality	0.264	0.468	0.01	0.297	0.375	
Send e-mail reminder	0.7	0.7	0	0.1	0	**0.542**
Change the meeting, increase time range	0.2	0.2	0	0.1	0	**0.176**
Allow system to have access to personal e-agendas	0.3	0.2	0.1	0.2	0.3	**0.346**
Change the meeting, fewer constraints (equipment)	0	0	0.9	0	0	**0.009**
Cancel a meeting and send e-mail confirmation	0.8	0.8	1	0.7	0.9	**1.141**
Combined Risk Reduction	**0.966**	**0.962**	**1**	**0.806**	**0.93**	

$$\text{EffectOfCountermeasure(cm)} = \sum_{risk} (\text{reduction(cm,risk)} \times \text{criticality(risk)})$$

Most Effective Countermeasure

	Risks					
Countermeasures	Participant does not read e-mails	Participant does not reply to requests	Room with equipment is not available	System response is too close to meeting	Important participant has last minute change	**Effect of Counter measure**
Criticality	0.264	0.468	0.01	0.297	0.375	
Send e-mail reminder	0.7	0.7	0	0.1	0	**0.542**
Change the meeting, increase time range	0.2	0.2	0	0.1	0	**0.176**
Allow system to have access to personal e-agendas	0.3	0.2	0.1	0.2	0.3	**0.346**
Change the meeting, fewer constraints (equipment)	0	0	0.9	0	0	**0.009**
Cancel a meeting and send e-mail confirmation	0.8	0.8	1	0.7	0.9	**1.141**
Combined Risk Reduction	**0.966**	**0.962**	**1**	**0.806**	**0.93**	

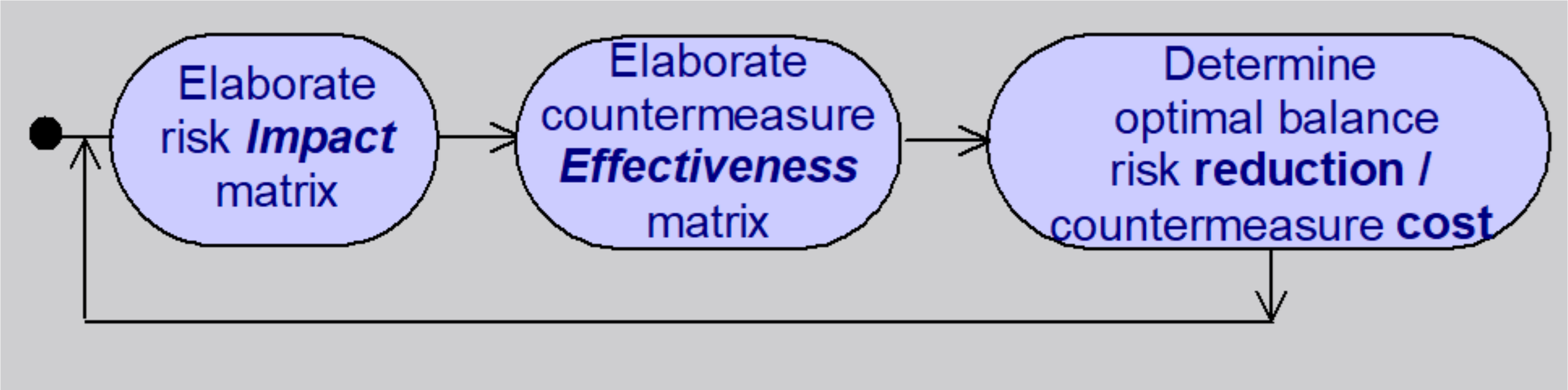
Most reduced risk

Most effective countermeasure

Least effective countermeasure

DDP Process

- Risk Consequence Table
- Risk Countermeasure Table



Elaborate risk ***Impact*** matrix

Elaborate countermeasure ***Effectiveness*** matrix

Determine optimal balance risk **reduction /** countermeasure **cost**

Determine Optimal Balance Risk Reduction *vs.* Countermeasure Cost

- Cost of each countermeasure *cm* to be estimated with domain experts.
- DDP can then visualize
  - risk balance charts: residual impact of each risk on all objectives if *cm* is selected
  - optimal combinations of countermeasures for risk balance under cost constraints
    - simulated annealing search for near-optimal solutions
    - user can set optimality criterion

e.g. "maximize the satisfaction of objectives under this cost threshold."

"minimize cost above this satisfaction threshold."

Risk Documentation

- Recall: risk management is an iterative process (identify, assess, control)
- The process should be documented:
  - to provide the rationale for countermeasure requirements
  - to support requirements evolution
  - needed for risk monitoring at system runtime
  - needed for the dynamic selection of more appropriate countermeasures

Risk Documentation

- Risk document should include for each identified risk:
  - The conditions or events characterizing its occurrence.
  - Its estimated likelihood of occurrence.
  - Its possible cause and consequences.
  - The estimated likelihood and severity of each possible consequence.
  - The countermeasures that were identified together with their respective risk-reduction leverage
  - The selected subset of countermeasures.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Risk Analysis

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Cost Estimation

Fundamental Estimation Questions

- How much effort is required to complete an activity?
- How much calendar time is needed to complete an activity?
- What is the total cost of an activity?
- Project estimation and scheduling and interleaved management activities

Estimation

Our job is to estimate:

1. Time to develop
2. Cost
3. Number of developers / month

Why is it hard to estimate well?

It is not easy to estimate the cost and effort to build a project when you do not know very much about that project.

- Yes, software engineering is still a relatively new field.
- We are not estimating repeatable, objective phenomena.
- The earlier the estimate (e.g., requirements phase), the less is known about the project.
- Unlike building bridges, most of the time and effort in software development is in creating a new design.
- A goal to estimate within 10% of the actual cost is unrealistic. Experience has shown that the product is almost complete when we know enough about a project to estimate its cost to be within 10% of its actual cost.

Why estimate software cost and effort?

- To provide a basis for agreeing to a job.
- To make commitments that you can meet.
- To help you track progress.

Estimation Techniques

- Delphi Method
- Function Point Analysis
- CoCoMo

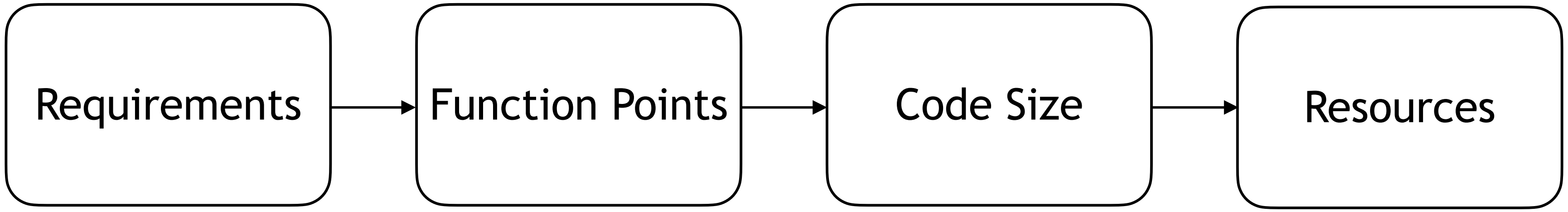
Delphi Method

Delphi methods are based on expert judgment:

1. Each expert submits a secret prediction, using whatever process each one chooses.
2. The average estimate is sent to the whole group.
3. Each expert revises their prediction privately.
4. Repeat until no expert wants to revise their estimate, i.e., until a fixed point is reached.

Function Point Analysis

- Estimating Cost based on what we know at requirements time
  1. Estimate the number of function points from the requirements,
  2. Estimate code size from function points, and
  3. Estimate resources required (time, personnel, money) from a code size



1. Estimate Function Points

Idea: Predict the complexity of the system in terms of the number of functions to write

The Basic Model is:

$$FPs = a_1 EI + a_2 EO + a_3 EQ + a_4 EIF + a_5 ILF$$

FPs = number of function points

EI (External Inputs) = number of user inputs (data entry, input event).

EO (External Outputs) = number of user outputs (screen error messages, report).

EQ (External Inquiries) = number of user queries (request or response function that doesn't require a change to system state).

EIF (External Interface File) = number of external interfaces (other systems,…).

ILF (Internal Logical Files) = number of internal files.

a_1, a_2, …, a_5 - empirically observed weights per function type

Weights

	Complexity		
	Low	Average	High
External Input (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiry (EQ)	3	4	6
External Interface File (EIF)	5	7	10
Logical Internal File (LIF)	7	10	15

2. Estimating Code Size From FPs

- There are tables that list, for each programming language, the number of statements in it that are required to implement one function point.
- These tables must be calibrated for each shop, each domain, etc.

Language	SLOC / UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI / Quick / Turbo Basic	64
Basic - Compiled	91
Basic - Interpreted	128
C	128
C++	29
ANSI Cobol 85	91
Fortran 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

Problems with KLOC

- How do you measure them?
  - How do you count one line that has several statements?
  - How do you count a statement that is over several lines?
  - How do you count constructs, e.g., conditionals?
- One person's line may be another's several lines

But they are used as the unit of code size with care and with *standards* that answer these questions.

3. Estimate Cost

COnstructive COst MOdel (COCOMO) - used to predict the cost of a project from an estimate of its size (LOC or KLOC):

- is one of the earliest cost models widely used in cost estimation.
- was initially published in Software Engineering Economics by Dr. Barry Boehm in 1981.
- is a regression-based model considering various historical programs' software sizes and multipliers.
- its most fundamental calculation is using the Effort Equation to estimate the number of developers in a month required to develop a project.

3. Estimate Cost

COnstructive COst MOdel (COCOMO) - used to predict the cost of a project from an estimate of its size in lines of code (LOC).

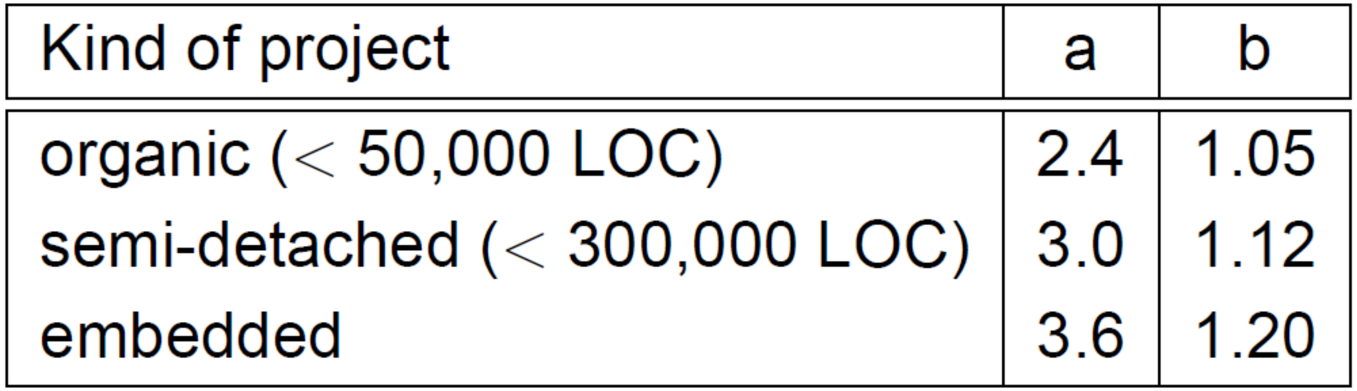
$$E = a \times KLOC^{b} \times X$$

E is for Effort - estimated in man-months or person-months (the amount of work performed by the average worker in a month)

KLOC - estimated project size (thousands of lines of code)

a, b - empirically observed weightings; depend on the type of system being developed

X - project attribute multipliers



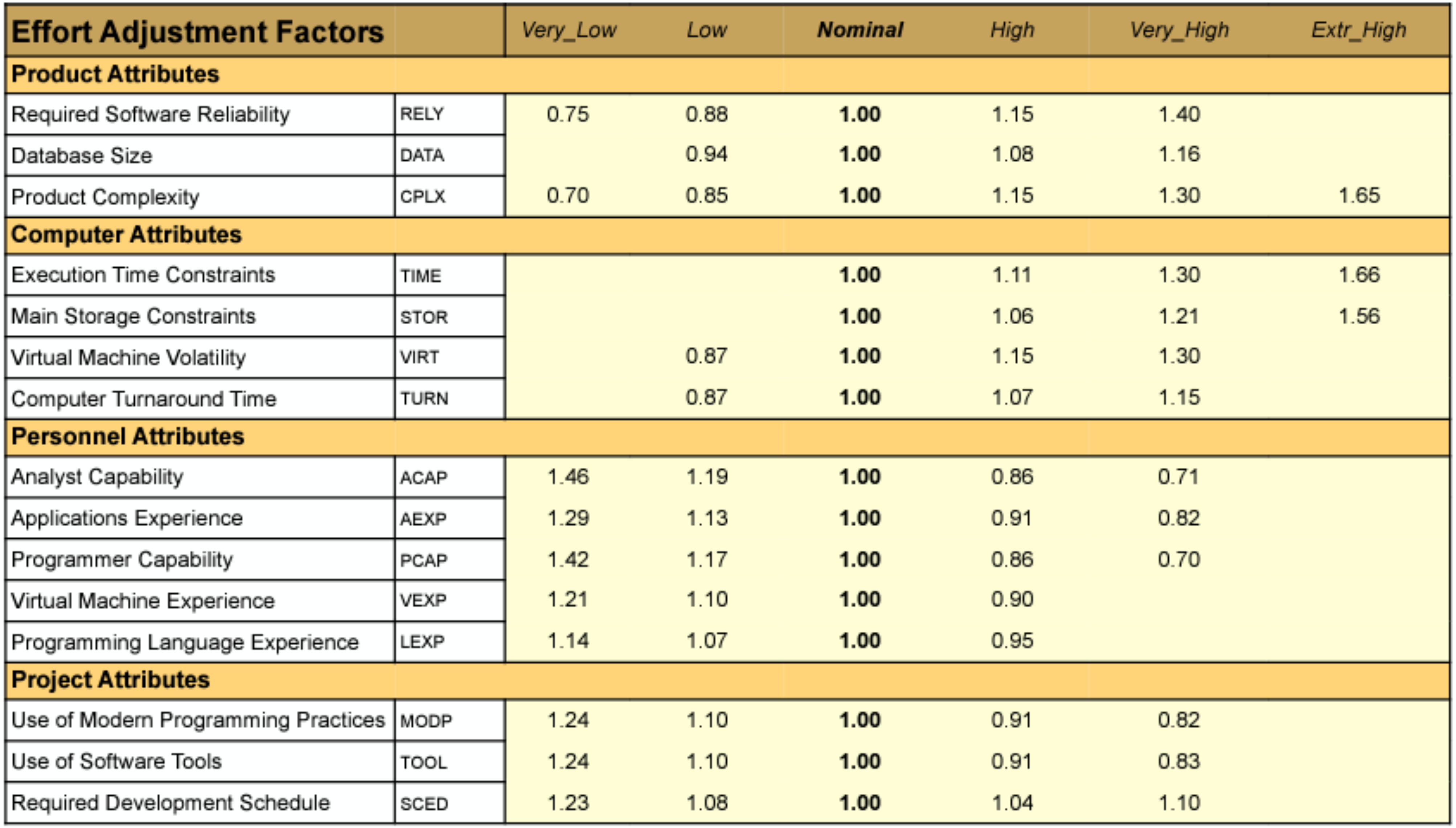
Kind of project	a	b
organic ($<$ 50,000 LOC)	2.4	1.05
semi-detached ($<$ 300,000 LOC)	3.0	1.12
embedded	3.6	1.20

Project Attributes

Adjust Effort estimation according to attributes of the project:

- Product attributes (reliability, complexity): required reliability↑, complexity↑, database size ↑
- Resource constraints (execution time, memory constraints): execution time↑, memory↑, hardware volatility↑, tight response time↑
- Personnel attributes (experience of developers): quality of analysts↓, quality of programmers↓, experience with the product↓, hardware experience↓, programming language (PL) experience↓
- Project attributes (techniques, programming languages): use of software tools (e.g., debugger)↓, use of modern PL↓, schedule constraints↑

Project Attributes



Effort Adjustment Factors		*Very_Low*	*Low*	***Nominal***	*High*	*Very_High*	*Extr_High*
Product Attributes							
Required Software Reliability	RELY	0.75	0.88	**1.00**	1.15	1.40	
Database Size	DATA		0.94	**1.00**	1.08	1.16	
Product Complexity	CPLX	0.70	0.85	**1.00**	1.15	1.30	1.65
Computer Attributes							
Execution Time Constraints	TIME			**1.00**	1.11	1.30	1.66
Main Storage Constraints	STOR			**1.00**	1.06	1.21	1.56
Virtual Machine Volatility	VIRT		0.87	**1.00**	1.15	1.30	
Computer Turnaround Time	TURN		0.87	**1.00**	1.07	1.15	
Personnel Attributes							
Analyst Capability	ACAP	1.46	1.19	**1.00**	0.86	0.71	
Applications Experience	AEXP	1.29	1.13	**1.00**	0.91	0.82	
Programmer Capability	PCAP	1.42	1.17	**1.00**	0.86	0.70	
Virtual Machine Experience	VEXP	1.21	1.10	**1.00**	0.90		
Programming Language Experience	LEXP	1.14	1.07	**1.00**	0.95		
Project Attributes							
Use of Modern Programming Practices	MODP	1.24	1.10	**1.00**	0.91	0.82	
Use of Software Tools	TOOL	1.24	1.10	**1.00**	0.91	0.83	
Required Development Schedule	SCED	1.23	1.08	**1.00**	1.04	1.10	

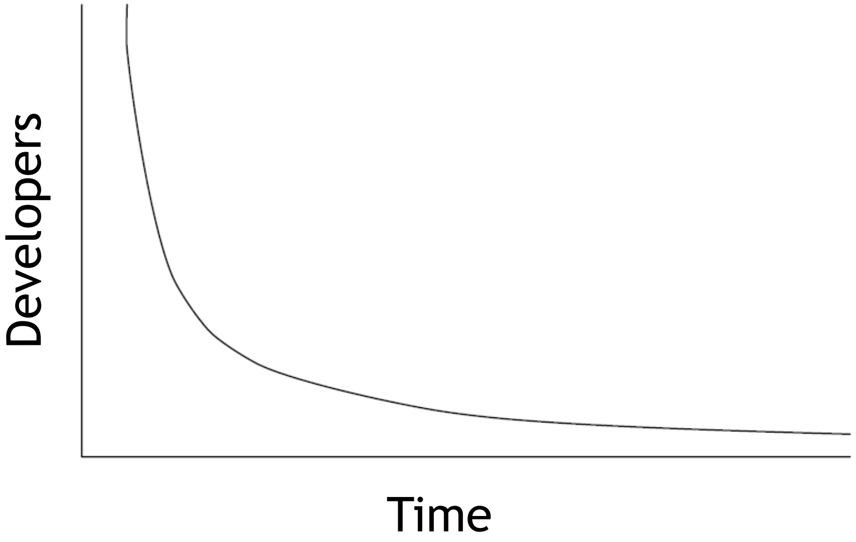
Other Equations

- Development Time (D): cE^{d} months.
- People Required (P): E/D people.

Software Project	c	d
Organic	2.5	0.38
Semi-detached	2.5	0.35
Embedded	2.5	0.32

Notes

- The FPs are calculated from the requirements and translated into estimated LOCs, then used in the COCOMO estimation method.
- Technically, the more developers the less time it takes to finish the project.
- Why is the formula not linear?

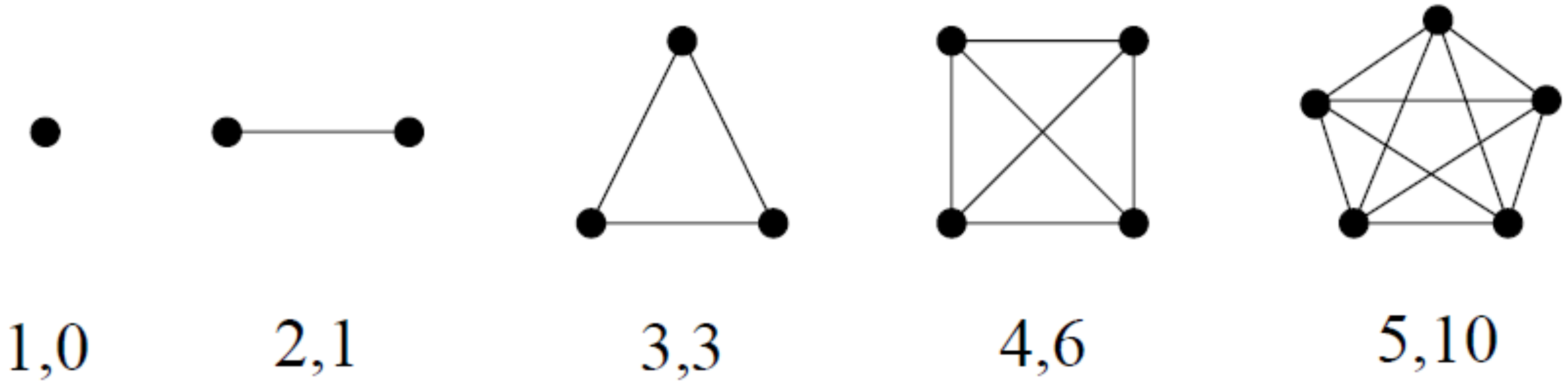


Notes

- But it does not work.
- Main counter-example:
  - It does work for painting a fence. Why?
  - It does not work for software development teams. Why?

Communication in a Group Project

- At some point, a new person costs in communication more than they add to the work that can be done.
- This is not even counting the fact that a new person wastes their own and others’ time getting up to speed.



Team size, Lines of communication

Communication in a Group Project

The other side of the coin is that any given project needs at least some minimum number *X* of people, and if you do not have that many people, you need to add more, even though it will cost delays. It is a choice between delay and never finishing.

Experience, Experience and Experience

- Models have to be calibrated to an organization
  Local factors include expertise, process, product type, and definition of LOC perturb accuracy.
- 100%+ errors are normal
  A software cost estimate model is doing well if it can estimate within 20% of the actual costs and within 70% of the actual time, assuming that the model has been calibrated to this type and size of the project!
- Model parameters based on old projects/technology
  Weights and coefficients are based on empirical studies of past projects using old technology and may be entirely unlike new projects.

So why to bother?

Poor estimates may be better than no estimates:

- We need this information to negotiate the cost of the product.
- We need to plan for the project.
  - to determine how many developers to hire or to assign to this project,
  - to know how long they'll be dedicated to this project and not to others
- We cannot control what we cannot measure.

Our estimation ability improves with practice and experience.

Do not get too caught up in an estimate. It is wrong. You will get better, but you will never master the problem.

So why to bother?

- Some people will be better at estimating than others.

Cost estimation is not a science.

It’s an art based on intuition and experience.

Be wary of any method or tool vendor that claims to predict cost or effort to unrealistic precision, i.e., more than one significant digit!

Cross-checks and Validation

- After an estimate has been created, the next step involves validating the estimate by cross-checking.
  - Cross-checking means using a different approach to create the estimate.
  - If both estimates are close, the target estimate has some validity.
  - If both estimates are very different.
    - This increases the uncertainty level, which must be reflected in a risk analysis.
    - This may lead to another estimating method to increase cost estimate confidence.
- It is a good practice to cross-check significant cost drivers.
  - If time is available, cross-checking other cost elements can further validate the estimate.

Cross-checks and Validation

- Validation also includes a demonstration that:
  - The data relationships are logical,
  - The data used are credible/convincing,
  - Model users have sufficient experience and training,
  - Calibration processes are thoroughly documented,
  - Formal estimating policies and procedures are established, and
  - When applicable, information system controls are maintained to ensure the integrity of the used models.

Cost Estimating Challenges

- Access to historical data
  - Need to invest in database capture of historical costs and technical data for proper CER development
    - Costly, time-consuming, and usually not funded
    - Development costs for IT systems can quickly become outdated by new programming languages
    - Maintenance costs are even more challenging to capture because they are seen as ongoing support or overhead and not as metrics
    - System architecture change effects on cost estimates can be hard to determine

- Validity and uncertainty of data
  - Garbage in = Garbage out

Cost Estimating Challenges

- Limited time to develop estimates
  - Can result in rough-order magnitude costs being used as budget quality estimates
  - Cause necessary steps like validation and Monte Carlo simulation to be omitted

- Resources
  - Lack of trained people is a problem

Why you underestimate by an order of magnitude

Fred Brook observes:

- Everybody thinks program when they should think of software system product.
- Program - what you write for yourself (and thus what you know)
- System - a program that interfaces with other programs, directly or indirectly, costs three times as much as a central program (more stuff to write)
- Product - a program written for others that must therefore be robust, costs three times as much as a central program
- Software system product - a program that is system and product costs nine times as much as a central program

COCOMO - Constructive Cost Model

- COCOMO II - Constructive Cost Model
- http://softwarecost.org/tools/COCOMO/

- COCOMO III - Constructive Cost Model
- https://boehmcsse.org/tools/cocomo-iii/
- https://www.youtube.com/watch?v=5sxKi-QsIOU

“The models are just there to help, not to make the management decisions for you.”

-- Barry Boehm

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Cost Estimation

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Prioritizing Requirements

Klaus Pohl . Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2010.

Why prioritize requirements?

- When customer expectations are high, and timelines are short.
- When you need to make sure the product delivers the most critical or valuable functionality as early as possible.
- It is a way to deal with competing demands for limited resources.
- It is a critical strategy for agile or other projects that develop products through a series of fixed-schedule timeboxes.
- On every project, a project manager must balance the desired project scope against the constraints of schedule, budget, staff, and quality goals.
  - to drop, or to defer to a later release, low-priority requirements when new, more essential requirements are accepted or when other project conditions change.

Some Prioritization Pragmatics

Successful prioritization requires an understanding of six issues:

1. The needs of the customers
2. The relative importance of requirements to the customers
3. The timing at which capabilities need to be delivered
4. Requirements that serve as predecessors for other requirements and other relationships among requirements
5. Which requirements must be implemented as a group
6. The cost to satisfy each requirement

Stakeholders May Resist

To encourage stakeholders to acknowledge that some requirements have lower priority, the analyst can ask questions such as the following:

- Is there some other way to satisfy the need that this requirement addresses?
- What would the consequences be of omitting or deferring this requirement?
- What effect would it have on the project’s business objectives if this requirement was not implemented for several months?
- Why might customers be unhappy if this requirement was deferred to a later release?
- Is having this feature worth delaying the release of all of the other features with this same priority?

Some Prioritization Techniques

- In or Out
- Three-Level Scale
- MoSCoW
- Cost-Value Approach

In or Out

- Simple
- Group of stakeholders
- Binary decision
- Keep referring to the project’s business objectives

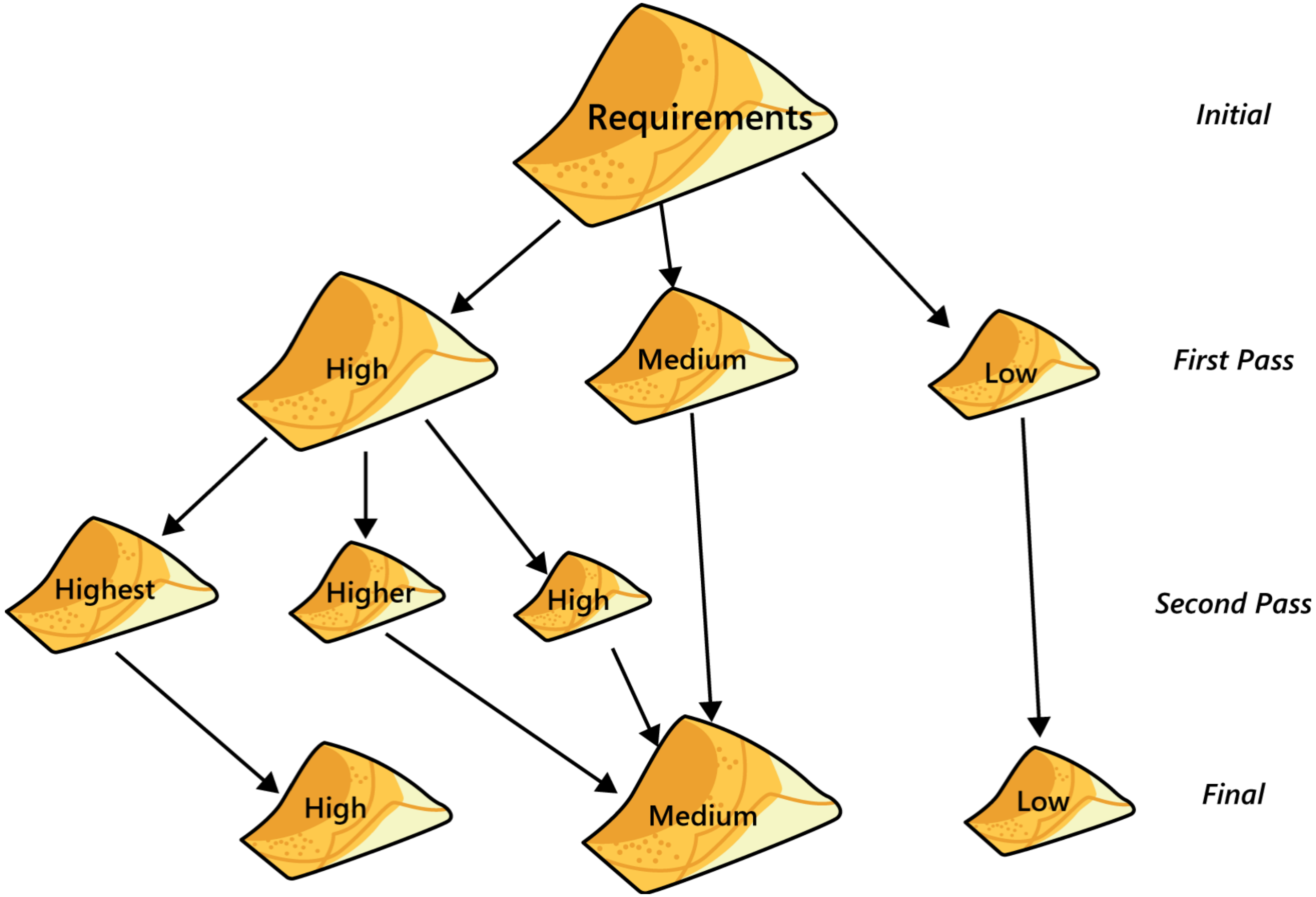
Three-Level Scale

- Consider the two dimensions of *importance* and *urgency*

	Important	*Not So Important*
Urgent	High Priority	Don’t Do These!
Not So Urgent	Medium Priority	Low Priority

Prioritize Iteratively

- Sometimes, particularly on a large project, you might want to perform prioritization iteratively.



MoSCoW

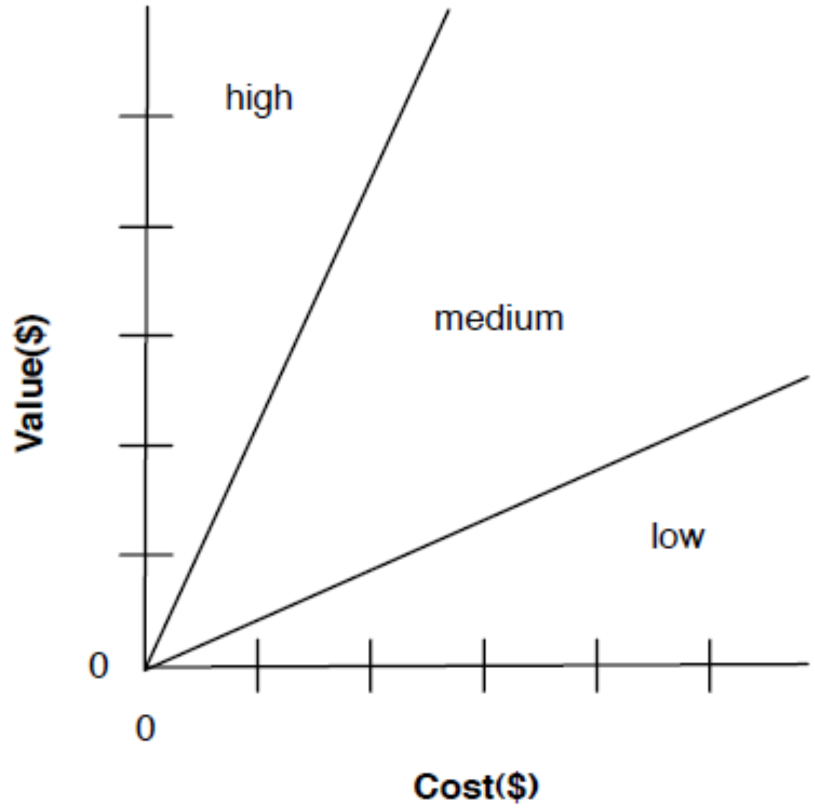
The four capitalized letters in the MoSCoW prioritization scheme stand for four possible priority classifications for the requirements in a set.

- **M**ust: The requirement must be satisfied for the solution to be considered a success.
- **S**hould: The requirement is essential and should be included in the solution if possible, but it’s not mandatory for success.
- **C**ould: It’s a desirable capability that could be deferred or eliminated. Implement it only if time and resources permit.
- **W**on’t: This indicates a requirement that will not be implemented at this time but could be included in a future release.

Cost-Value Approach

Want to sort requirements by their potential value and cost?

- Value is a requirement’s potential contribution to customer satisfaction
- Cost is the cost of implementing the requirement
- Can prioritize requirements according to their cost-value ratios
- absolute values and costs are complex to estimate
- relative comparisons are easier



- Based on the Analytic Hierarchy Process (AHP); an approach for supporting decision-making.
- It includes five steps.

Step I:

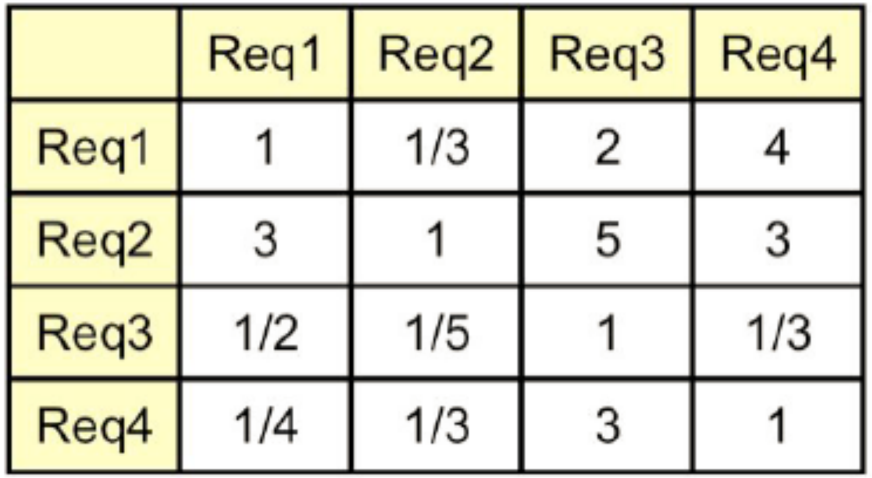
The requirements engineers review the candidate requirements to ensure that the requirements are complete and clearly defined.

Step II:

Customers and users determine the relative value of each requirement using the pairwise comparison method of the AHP, which includes five steps.

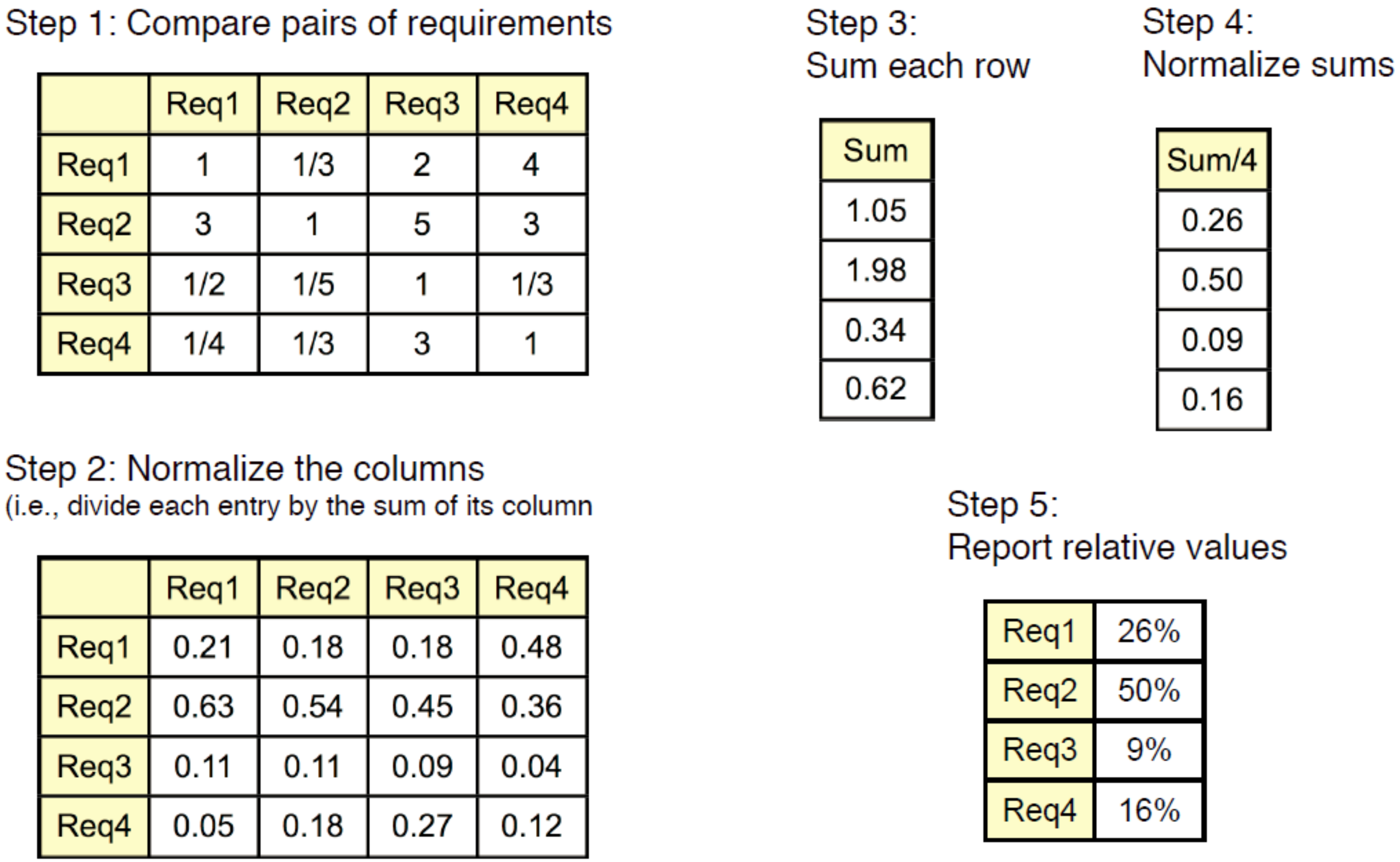
step 1: compare pairs of requirements

- 1 - requirements are of equal value
- 3 - one is slightly preferred over the other
- 5 - one is strongly preferred over the other
- 7 - one is very strongly preferred over the other
- 9 - one is highly preferred over the other
- Intermediate values 2, 4, 6, and 8 used when compromise is needed
- if pair (x,y) has relative value n, complementary pair (y,x) has reciprocal value 1/n



	Req1	Req2	Req3	Req4
Req1	1	1/3	2	4
Req2	3	1	5	3
Req3	1/2	1/5	1	1/3
Req4	1/4	1/3	3	1

Averaging Over Normalized Columns



Step 1: Compare pairs of requirements

	Req1	Req2	Req3	Req4
Req1	1	1/3	2	4
Req2	3	1	5	3
Req3	1/2	1/5	1	1/3
Req4	1/4	1/3	3	1

Step 2: Normalize the columns
(i.e., divide each entry by the sum of its column

	Req1	Req2	Req3	Req4
Req1	0.21	0.18	0.18	0.48
Req2	0.63	0.54	0.45	0.36
Req3	0.11	0.11	0.09	0.04
Req4	0.05	0.18	0.27	0.12

Step 3:
Sum each row

Sum
1.05
1.98
0.34
0.62

Step 4:
Normalize sums

Sum/4
0.26
0.50
0.09
0.16

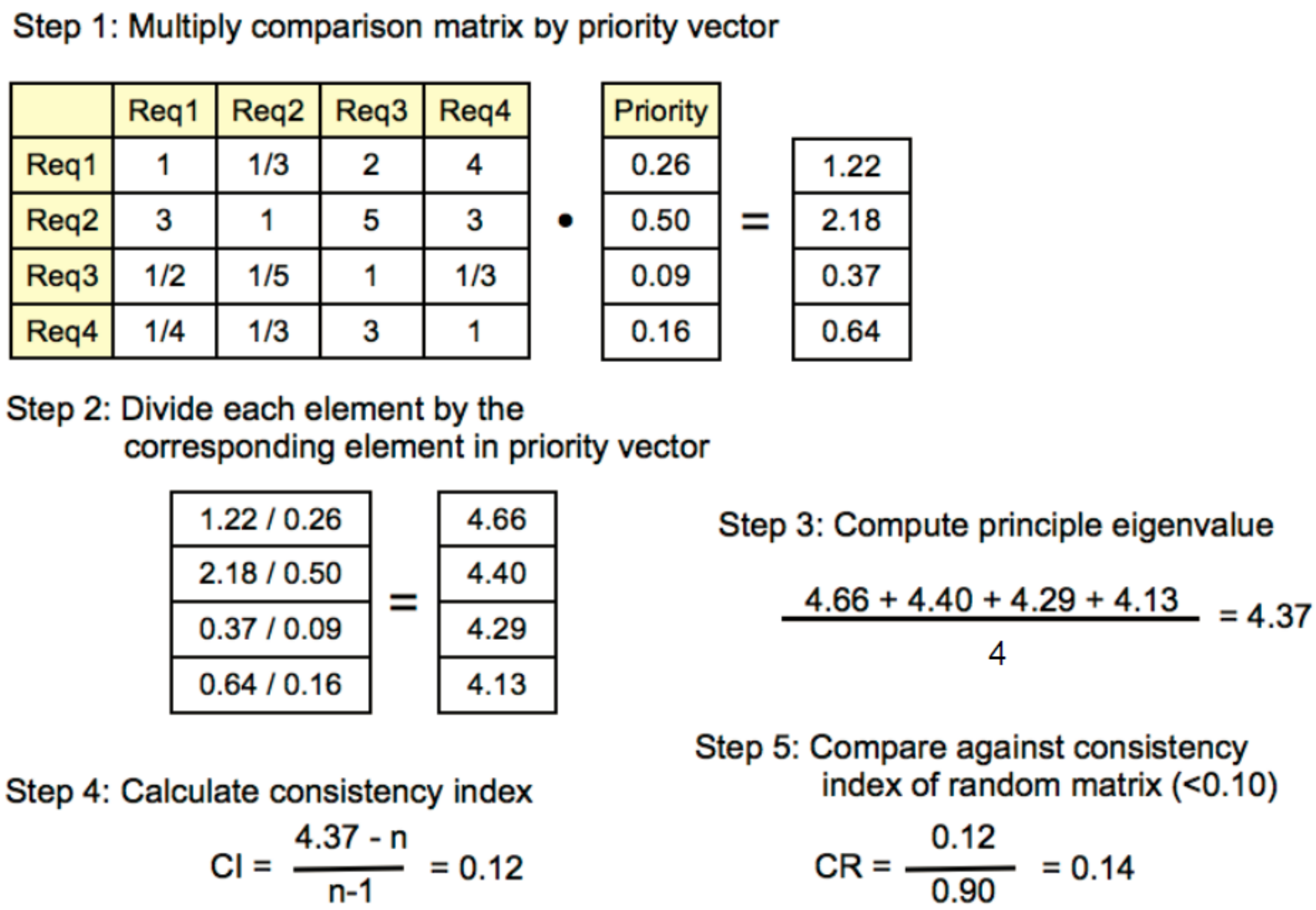
Step 5:
Report relative values

Req1	26%
Req2	50%
Req3	9%
Req4	16%

Checking Consistency

The consistency Index is the first indicator of the result accuracy of the pairwise comparison.

Checking Consistency



Step 1: Multiply comparison matrix by priority vector

$$\begin{array}{|c|c|c|c|c|}\hline & \text{Req1} & \text{Req2} & \text{Req3} & \text{Req4} \\ \hline \text{Req1} & 1 & 1/3 & 2 & 4 \\ \hline \text{Req2} & 3 & 1 & 5 & 3 \\ \hline \text{Req3} & 1/2 & 1/5 & 1 & 1/3 \\ \hline \text{Req4} & 1/4 & 1/3 & 3 & 1 \\ \hline \end{array} \bullet \begin{array}{|c|}\hline \text{Priority} \\ \hline 0.26 \\ \hline 0.50 \\ \hline 0.09 \\ \hline 0.16 \\ \hline \end{array} = \begin{array}{|c|}\hline 1.22 \\ \hline 2.18 \\ \hline 0.37 \\ \hline 0.64 \\ \hline \end{array}$$

Step 2: Divide each element by the corresponding element in priority vector

$$\begin{array}{|c|}\hline 1.22 / 0.26 \\ \hline 2.18 / 0.50 \\ \hline 0.37 / 0.09 \\ \hline 0.64 / 0.16 \\ \hline \end{array} = \begin{array}{|c|}\hline 4.66 \\ \hline 4.40 \\ \hline 4.29 \\ \hline 4.13 \\ \hline \end{array}$$

Step 3: Compute principle eigenvalue

$$\frac{4.66 + 4.40 + 4.29 + 4.13}{4} = 4.37$$

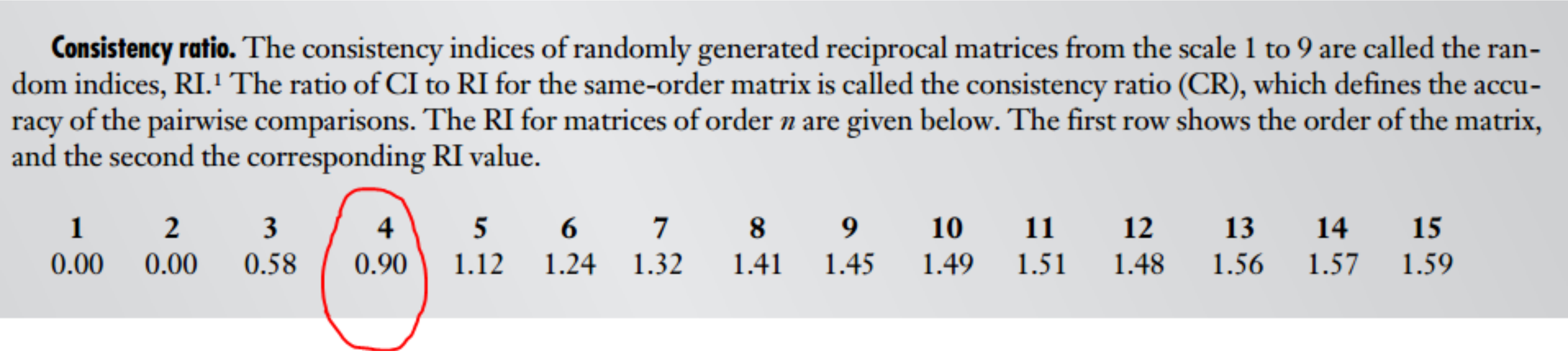
Step 4: Calculate consistency index

$$CI = \frac{4.37 - n}{n-1} = 0.12$$

Step 5: Compare against consistency index of random matrix (<0.10)

$$CR = \frac{0.12}{0.90} = 0.14$$

Checking Consistency



Consistency ratio. The consistency indices of randomly generated reciprocal matrices from the scale 1 to 9 are called the random indices, RI.[1] The ratio of CI to RI for the same-order matrix is called the consistency ratio (CR), which defines the accuracy of the pairwise comparisons. The RI for matrices of order *n* are given below. The first row shows the order of the matrix, and the second the corresponding RI value.

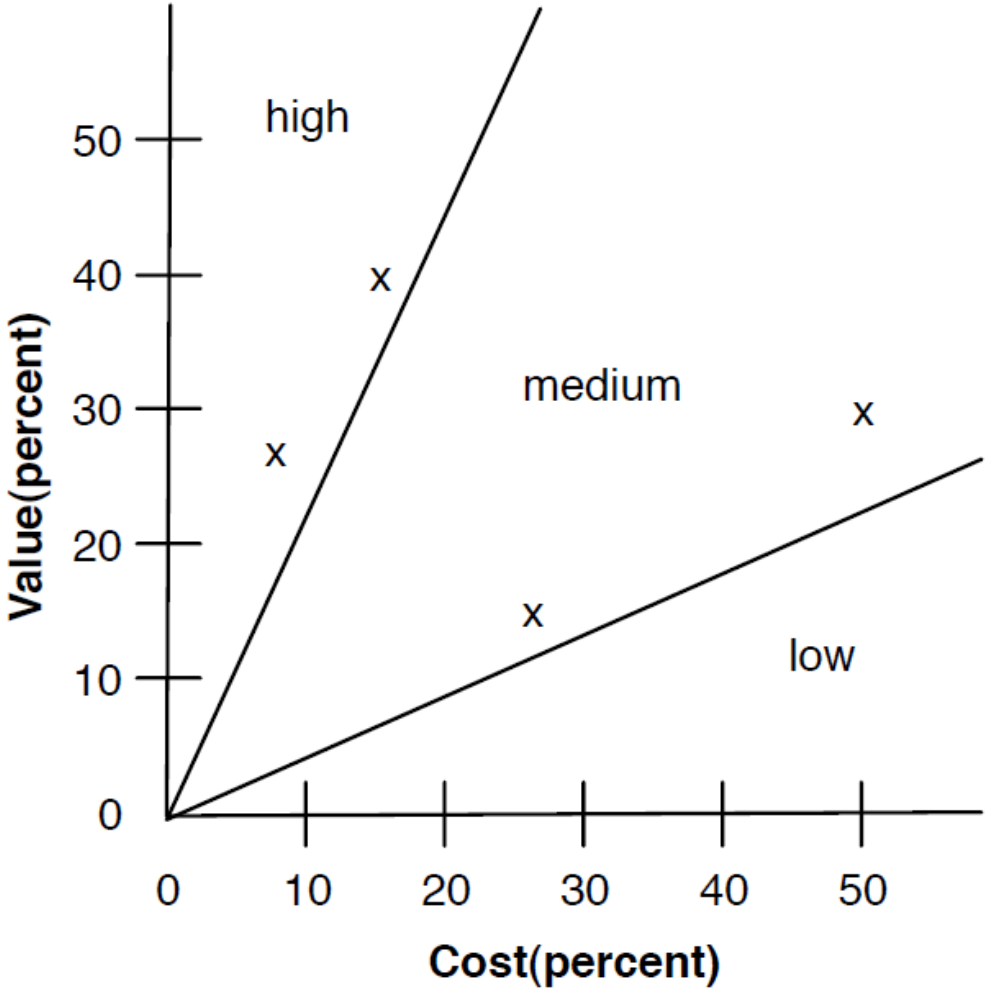
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Step III:

Perform Step II of AHP to estimate relative cost

Step IV:

Create a cost-value diagram where the value is depicted on the y-axis, and the cost is depicted on the x-axis.



Step V:

Stakeholders use the cost-value diagram as a conceptual map for analyzing and discussing the requirements.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Prioritizing Requirements

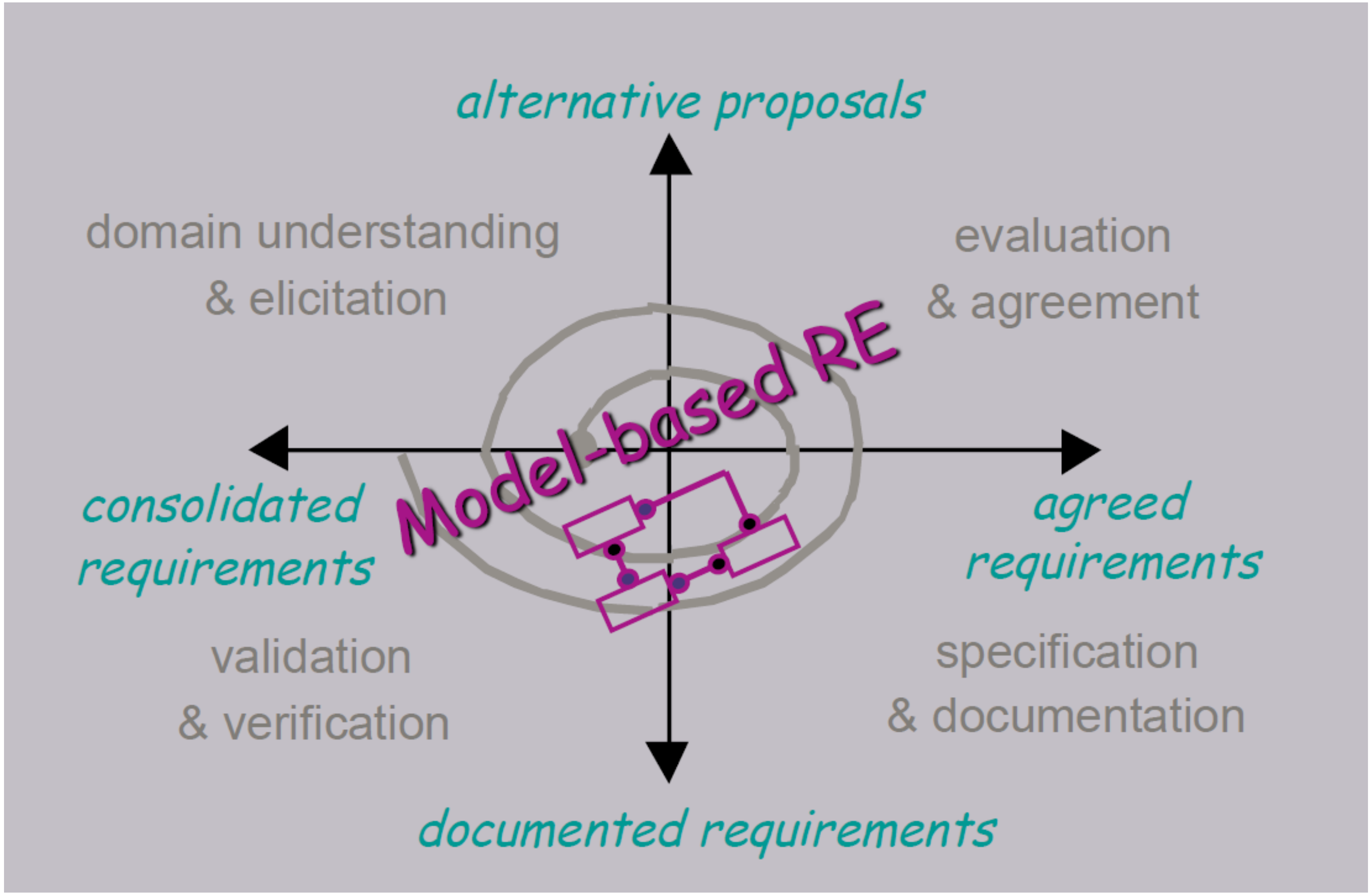
Klaus Pohl . Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2010.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Requirements Negotiation and Conflict Management

Klaus Pohl . Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2010.


alternative proposals
domain understanding
& elicitation
evaluation
& agreement
Model-based RE
consolidated
requirements
agreed
requirements
validation
& verification
specification
& documentation
documented requirements

Overview

One goal of the requirement engineering process is to establish sufficient agreement among the stakeholders regarding the already known requirements for the system.

You achieve this goal by:

- Identifying conflicts
- Analyzing conflicts
- Resolving conflicts
- Documenting conflict resolutions

Those activities are mainly supported by the following:

- Win-Win approach
- Interaction matrix

Conflict in Requirements Engineering

- Exists if the needs and wishes of different stakeholders regarding the system contradict each other or if some needs and desires cannot be considered.
- Examples:
  - Maintenance staff of an email system demand that the incoming and outgoing emails are recorded in the log file to support the system, while users demand high confidentiality of the exchanged emails.
  - Some stakeholders demand radar sensors for distance measurement, while others demand ultrasound sensors.
  - some stakeholders demand that safety information for drivers be displayed on a head-up display, while others think it could distract drivers and reject this requirement.

Conflict in Requirements Engineering

- Risks: unresolved conflicts may cause
  - stakeholders to no longer support the development of the system, or
  - a failure of the development of the system
- Conflicts should be treated as a source of
  - new ideas
  - innovative requirements
- To resolve conflict:
  - involve the relevant stakeholders to resolve.
  - involve software architects, developers, and testers to be trained to report (not to resolve) detected conflicts.
  - Both jointly should resolve the conflicts.

Use of Goals and Scenarios

- First, identify conflicts at the goal level as far as possible, then document, analyze, and resolve them at that level as far as possible.
- Conflict analysis benefits from using scenarios: a scenario can clarify conflict by describing the sequence of interactions in which the conflict occurs.
- Scenarios can be used to discuss how to reduce the conflict or avoid it altogether.
- Stakeholders can evaluate different scenarios and choose the ones that offer the best solution.

Activities of Conflict Management

1. Identifying conflicts
2. Analyzing conflicts
3. Resolving conflicts
4. Documenting conflict resolutions

1. Identifying Conflicts

- Conflicts about requirements may surface during all requirement engineering activities, such as:
  - during elicitation in workshops
  - during documenting the requirements that have been elicited during different interviews
  - during prioritization of requirements (different opinions)
  - during requirements validation (some consider requirement correct, and others object to it).
  - during conflict resolution, a new conflict identified

2. Analyzing Conflicts

- Goal: to determine the conflict types
- Resolving the conflict depends on the type
- One suggestion for classifying the conflicts is
    - Data conflict
    - Interest conflict
    - Value conflict

2.1 Data Conflict

Caused by:

- a lack of information
- misinformation
- different interpretations of an issue

Example: The following requirement is defined for a car entertainment system:

R4: The DVD player shall be able to handle re-writeable CDs (CD-RW) and DVDs (DVD-RW).

A stakeholder disagrees with the requirement. In his opinion, it does not make sense for a DVD player in the car to be able to write data onto CDs or DVDs

2.2 Interest Conflict

It exists if:

- stakeholders’ interests or goals about the system contradict each other.

Example:

Stakeholder#1: wants the car entertainment system to include Mp3 functionality, an optional hard disk, and a USB interface to attract technology-oriented customers.

Stakeholder#2: wants the system to be equipped with a CD player and radio. His goal is to reduce costs to attract price-conscious customers.

2.3 Value Conflict

It exists if:

- different stakeholders evaluate a requirement differently
- each stakeholder considers the importance of the requirement differently

The evaluation of facts is affected by the following:

- experience in life
- profession
- education
- training
- personal ideals
- culture
- religion
- and other characteristics.

2. Analyzing Conflicts

The type of conflict can be determined by the following three steps (in this order):

1. Checking for a data conflict: based on misinterpretations or incorrect information. Ask stakeholders to write their interpretation of the requirements to detect a potential conflict.
2. Checking for an interest conflict: based on different goals. Ask stakeholders to name their goals associated with the conflicting requirements. Document the goals of each stakeholder separately in a goal model. Compare the models to detect the conflict.

2. Analyzing Conflicts

3. Checking for a value conflict: based on different values. Check the stakeholders’ evaluation backgrounds (find out why they evaluate the requirements the way they do).

3. Resolving Conflicts

One of the following three basic strategies can be applied:

1. Negotiation
2. Creative Solution
3. Decision

3.1 Negotiation

- Exchange of arguments
- Agreement upon a solution
- Advantage: the viewpoints of all parties are considered, and a win-win situation is created. (I will talk about it later)
- Disadvantage: it can be time-consuming, and the compromise may not be the best solution from an objective viewpoint.

3.2 Creative Solution

- Discard the old solutions
- Develop creative, novel solutions
- Advantage: all parties come off as winners, as solutions are acceptable to all parties
- Disadvantage: The process can be time-consuming (to develop creative, novel solutions) and might impact other requirements influenced by the solutions.

3.3 Decision

- Complete agreement is rarely achievable. Conflict must be resolved in due time by a decision-maker.
- Decision-maker: higher authority, project leader, client representative, etc.
- Advantage: can be quick, without consuming too many resources.
- Disadvantage: there are cases where there is no higher authority. Usually, the decision is made favouring one viewpoint while ignoring the others.
- (Alt) Voting: on the viewpoints of all involved stakeholders

Negotiation Techniques

1. Win-Win approach: All stakeholders become winners
2. Interaction matrix: Visualizing overlapping and conflict about requirements.

Win-Win approach

1. Understand how stakeholders want to win: what is considered a benefit?
2. Raise adequate/realistic expectations by:
    1. Joint discussion about stakeholders’ expectations to identify wrong or unrealistic expectations.
    2. putting oneself in the other stakeholders’ place to improve understanding of their viewpoints.
    3. Expectations shall be defined based on objective criteria.
    4. Expectations shall be oriented towards experience (e.g. benchmarks, expert knowledge).
3. Win-win approach is beneficial for negotiation and creative solution strategies.
4. Resolving a conflict through a decision generally leads to a win-lose situation since it is typically made in favour of a single viewpoint.

Interaction matrix

- Each cell represents a pair of requirements and describes their interaction.
- Values of cells:
  - 1: if the conflict exists
  - 1,000: if the requirements overlap
  - 0: if the requirements are independent of each other
- Analysis:
  - calculate the sum of each column
  - if the sum=0 for a column: the req. represented by this column doesn’t overlap or conflict with any other requirements
  - #overlaps= sum div 1000
  - #conflicts= sum module 1000 (remainder of sum/1000)

Example

	R1	R2	R3	R4
R1	0	0	1	1
R2	0	0	0	0
R3	1	0	0	1,000
R4	1	0	1,000	0
Sum	2	0	1,001	1,001

Summary

One goal of the requirement engineering process is to establish sufficient agreement among the stakeholders regarding the already known requirements for the system.

You achieve this goal by:

- Identifying conflicts
- Analyzing conflicts
- Resolving conflicts
- Documenting conflict resolutions

Those activities are mainly supported by the following:

- Win-Win approach
- Interaction matrix

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Requirements Negotiation and Conflict Management

Klaus Pohl . Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2010.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Quality Requirements

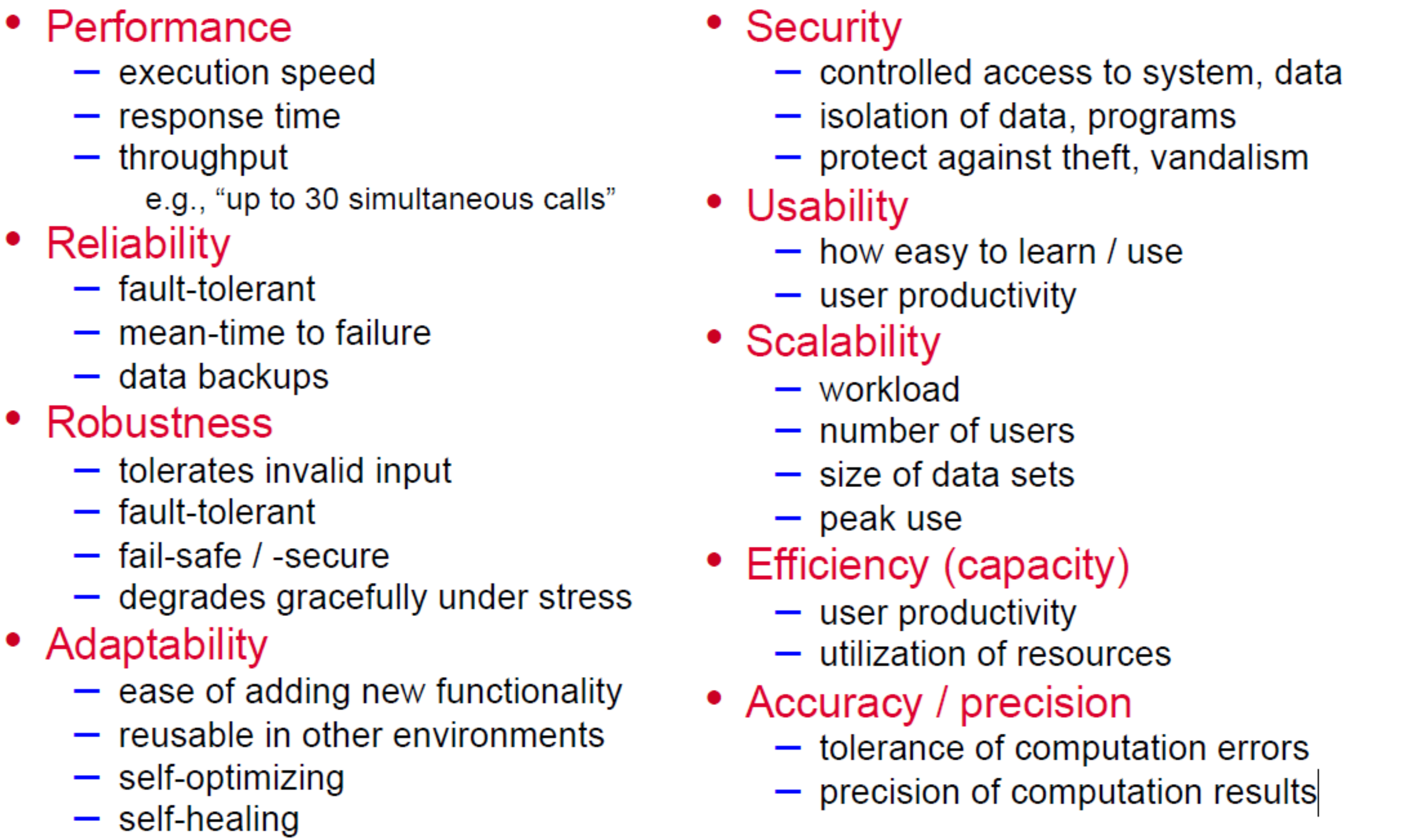
Overview

- There is more to software success than just delivering the proper functionality.
- Users also have expectations, often unstated, about how well the product will work.
  - how easy it is to use,
  - how quickly it executes,
  - how rarely it fails,
  - how it handles unexpected conditions,
  - perhaps, how loud it is.
- Such characteristics, collectively known as quality attributes, quality factors, quality requirements, quality of service requirements.

Overview

- Quality attributes can distinguish a product that merely does what it is supposed to do from one that delights its users.
- Excellent products reflect an optimum balance of competing quality characteristics.
- If you do not explore the customers’ quality expectations during elicitation, you are lucky if the product satisfies them.
  - Disappointed users and frustrated developers are the more typical outcome.
- Quality attributes serve as the origin of many functional requirements.
- They also drive significant architectural and design decisions.
  - It’s far more costly to re-architect a completed system to achieve essential quality goals than to design for them at the outset.

Software Quality Attributes



- Performance
  - execution speed
  - response time
  - throughput

    e.g., “up to 30 simultaneous calls”
- Reliability
  - fault-tolerant
  - mean-time to failure
  - data backups
- Robustness
  - tolerates invalid input
  - fault-tolerant
  - fail-safe / -secure
  - degrades gracefully under stress
- Adaptability
  - ease of adding new functionality
  - reusable in other environments
  - self-optimizing
  - self-healing
- Security
  - controlled access to system, data
  - isolation of data, programs
  - protect against theft, vandalism
- Usability
  - how easy to learn / use
  - user productivity
- Scalability
  - workload
  - number of users
  - size of data sets
  - peak use
- Efficiency (capacity)
  - user productivity
  - utilization of resources
- Accuracy / precision
  - tolerance of computation errors
  - precision of computation results

Software Quality Attributes

Design constraints

- interfaces to other systems
- COTS components
- programming language

Operating Constraints

- location
- size, power consumption
- temperature, humidity
- operating costs
- accessibility (for maintenance)

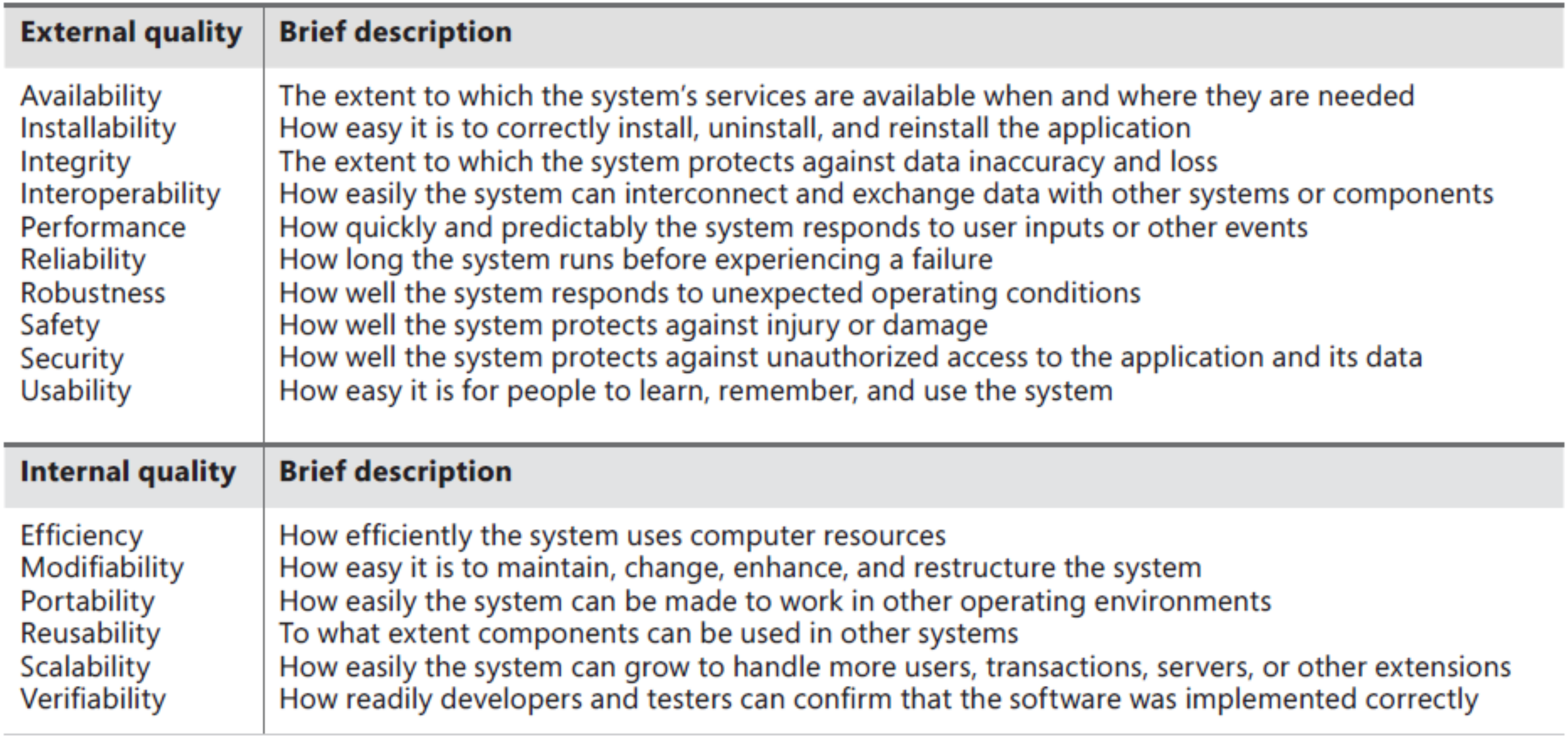
Product-family requirements

- modifiability
- portability
- reusability
- UI

Process Requirements

- Resources
  - personnel development
  - costs
  - development schedule
- Documentation
  - audience
  - conventions
  - readability
- Complexity (of code)
  - comments / KLOC
  - coupling / cohesion
  - cyclomatic complexity
  - use of multiple inherit. overloading, templates
- Standards compliance

Software Quality Requirements



External quality	Brief description
Availability	The extent to which the system's services are available when and where they are needed
Installability	How easy it is to correctly install, uninstall, and reinstall the application
Integrity	The extent to which the system protects against data inaccuracy and loss
Interoperability	How easily the system can interconnect and exchange data with other systems or components
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system runs before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application and its data
Usability	How easy it is for people to learn, remember, and use the system

Internal quality	Brief description
Efficiency	How efficiently the system uses computer resources
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions
Verifiability	How readily developers and testers can confirm that the software was implemented correctly

Trade-offs

- **In an ideal universe**, every system would exhibit the maximum possible value for all its attributes.
- The system would be available at all times, would never fail, would supply instantaneous results that are always correct, would block all attempts at unauthorized access, and would never confuse a user.
- **In reality**, trade-offs and conflicts between specific attributes make it impossible to maximize all of them simultaneously.
- Because perfection is unattainable, you must determine which attributes are most important to your project's success. Then you can craft specific quality objectives for these essential attributes so designers can make appropriate choices.

Examples of Quality Requirements

- The interface shall be user friendly
- The system should be available the vast majority of the time
- The user shall be able to learn to use the system very quickly

Fit criteria

Fit criteria express quality requirements in a way that makes it possible, objectively, to divide solutions into those that are acceptable and those that are not.

A fit criterion quantifies the extent to which a quality requirement must be met.

Example: Measuring Reliability

Reliability can be defined as a percentage likelihood of success, downtime, the absolute number of failures, …

Example: Telephone network

The entire network can fail no more than, on average, 5 minutes per Year, but failures of individual switches can fail up to 2 hours per Year.

Example: Patient monitoring system

The system may fail for up to 1 hour per year, but doctors or nurses should be alerted of the failure in those cases. More frequent failure of individual components is unacceptable.

Richer Fit Criteria

Requirement	Outstanding	Target	Minimum
Response Time	0.1s	0.5s	1s
CPU Utilization	20%	25%	30%
Usability	40 tasks/hour	30 tasks/hour	20 tasks/hour

Fit Criteria - Measurement

- The hardest part of testing a requirement against an agreed-upon measurement is defining the appropriate measure for the requirement.
- Example:
  - Stakeholder asks for a “nice” product. How to measure “nice”?
  - Need measurement of niceness!
  - Must be agreeable to stakeholder.
- Once you define how to measure “niceness”, you can define a requirement to build the product agreeably.

So how to measure nice ?

- Interrogate users and find:
  - nice => “liked by staff members”
  - “liked by staff members” => “take to product instinctively” and “don’t hesitate to use”
- We can measure the duration of hesitation!
- If our stakeholder agrees, we now have a good measurement for “nice.”

If you can’t quantify something, it cannot be a requirement.

External Quality Attributes

- Availability
- Installability
- Integrity
- Interoperability
- Performance
- Reliability
- Robustness
- Safety
- Security
- Usability

Availability

- Availability measures the planned-up time during which the system’s services are available for use and fully operational.
- Formally, availability equals the uptime ratio to the sum of uptime and downtime.

AVL-1. The system shall be at least 95 percent available on weekdays between 6:00 A.M. and midnight Eastern Time and at least 99 percent available on weekdays between 3:00 P.M. and 5:00 P.M. Eastern Time.

AVL-2. Downtime excluded from the calculation of availability consists of maintenance scheduled from 6:00 P.M. Sunday Pacific Time through 3:00 A.M. Monday Pacific Time.

Installability

- Installability describes how easy it is to perform these operations correctly.

- Increasing a system’s installability reduces the time, cost, user disruption, error frequency, and skill level needed for an installation operation.

Examples:

INS-1. An untrained user shall be able to successfully perform an initial installation of the application in an average of 10 minutes.

INS-2. When installing an upgraded application version, all customizations in the user's profile shall be retained and converted to the new version's data format if needed.

INS-3. The installation program shall verify the correctness of the download before beginning the installation process.

INS-4. Installing this software on a server requires administrator privileges.

INS-5. Following successful installation, the installation program shall delete all temporary, backup, obsolete, and unneeded files associated with the application.

Integrity

- Integrity deals with preventing information loss and preserving the correctness of data entered into the system.

- Integrity requirements have no tolerance for error: the data is either in good shape and protected or not.

- Data integrity also addresses the accuracy and proper formatting of the data

Examples:

INT-1. After a file backup, the system shall verify the backup copy against the original and report any discrepancies.

INT-2. The system shall protect against the unauthorized addition, deletion, or modification of data.

INT-3. The Chemical Tracking System shall confirm that an encoded chemical structure imported from third-party structure-drawing tools represents a valid chemical structure.

INT-4. The system shall confirm daily that the application executables have not been modified by adding unauthorized code.

Interoperability

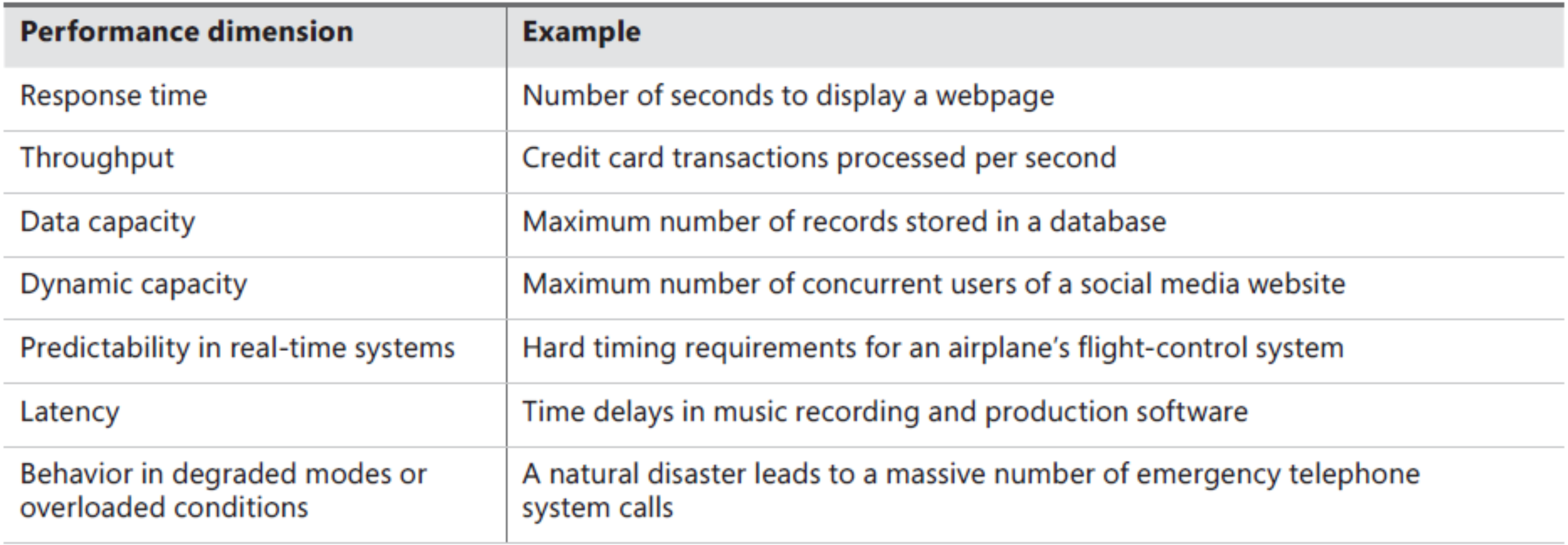
- Interoperability indicates how readily the system can exchange data and services with other software systems and how easily it can integrate with external hardware devices.

IOP-1. The Chemical Tracking System shall be able to import any valid chemical structure from the ChemDraw (version 13.0 or earlier) and MarvinSketch (version 5.0 or earlier) tools.

IOP-2. The Chemical Tracking System shall be able to import any chemical structure encoded using the SMILES (simplified molecular-input line-entry system) notation.

Performance

- Some aspects of performance



Performance dimension	Example
Response time	Number of seconds to display a webpage
Throughput	Credit card transactions processed per second
Data capacity	Maximum number of records stored in a database
Dynamic capacity	Maximum number of concurrent users of a social media website
Predictability in real-time systems	Hard timing requirements for an airplane's flight-control system
Latency	Time delays in music recording and production software
Behavior in degraded modes or overloaded conditions	A natural disaster leads to a massive number of emergency telephone system calls

Examples:

PER-1. Authorization of an ATM withdrawal request shall take no more than 2.0 seconds.

PER-2. The anti-lock braking system speed sensors shall report wheel speeds every two milliseconds with a variation not to exceed 0.1 milliseconds.

PER-3. Webpages shall fully download in an average of 3 seconds or less over 30 megabits/second Internet connection.

PER-4. At least 98 percent of the time, the trading system shall update the transaction status display within 1 second after the completion of each trade.

Reliability

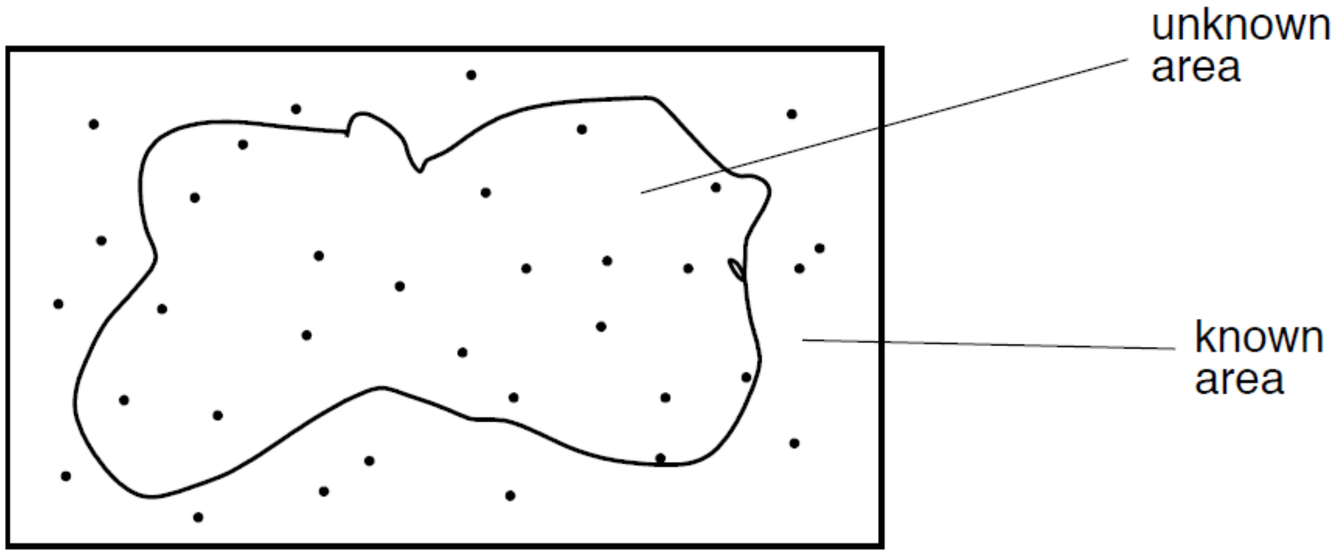
- The probability of the software executing without failure for a specific period.
- Ways to specify and measure software reliability include:
  - the percentage of operations that are completed correctly,
  - the average length of time the system runs before failing (mean time between failures, or MTBF), and
  - the maximum acceptable probability of a failure during a given period.

REL-1. At most, five experimental runs out of 1,000 can be lost because of software failures.

REL-2. The mean time between failures of the card reader component shall be at least 90 days.

Monte Carlo Techniques

Monte Carlo techniques: estimate an unknown quantity using a known amount.

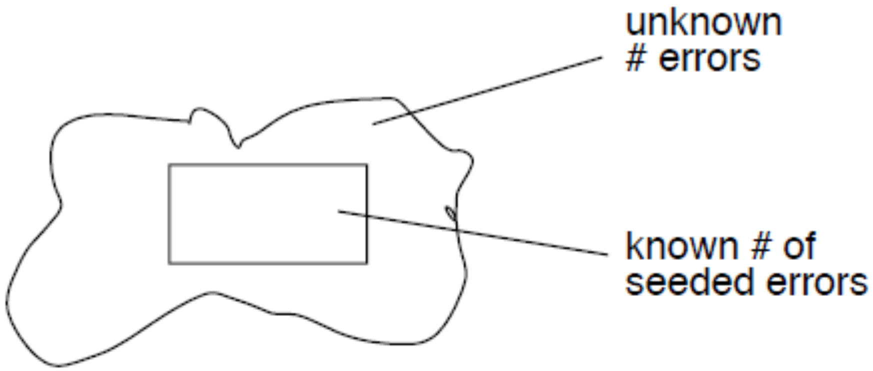


$$\frac{\text{Number of points in shape}}{\text{Total number of points}} \approx \frac{\text{Area of Shape}}{\text{Known Area of Rectangle}}$$

Monte Carlo Techniques

We can use Monte Carlo techniques to estimate the number of bugs remaining in a program (reliability).

- Plant a known number of errors into the program which the testing team does not know about.
- Then compare the number of seeded errors the team detects with the total number of errors it detects to estimate the total number of bugs in the program.



$$\frac{\#\text{ detected seeded errors}}{\#\text{ seeded errors}} \approx \frac{\#\text{ detected errors}}{\#\text{ errors in the program}}$$

Problems with this approach:

- Not all bugs are equal
  - some are more difficult to find/detect than others
  - some are more difficult to seed than others
  - some have a more significant negative impact than others
- Fixing bugs will create more bugs

Robustness

- The degree to which a system functions correctly when confronted with invalid inputs, defects in connected software or hardware components, external attacks, or unexpected operating conditions.
- Other attribute terms associated with robustness are:
  - fault tolerance
  - survivability
  - recoverability

ROB-1. If the text editor fails before the user saves the file, it shall recover the contents of the file being edited as of, at most, one minute before the failure the next time the same user launches the application.

ROB-2. All plot description parameters shall have default values specified, which the Graphics Engine shall use if a parameter's input data is missing or invalid.

Safety

- The need to prevent a system from doing any injury to people or damage to property.
- Might be dictated by government regulations or other business rules, and legal or certification issues could be associated with satisfying such requirements.
- Safety requirements frequently are written in the form of conditions or actions the system must not allow to occur.

SAF-1. The user shall be able to see a list of all ingredients in any menu items, with highlighted ingredients known to cause allergic reactions in more than 0.5 percent of the North American population.

SAF-2. If the reactor vessel's temperature rises faster than 5°C per minute, the Chemical Reactor Control System shall turn off the heat source and signal a warning to the operator.

Security

- Security deals with blocking unauthorized access to system functions or data, ensuring that the software is protected from malware attacks, and so on.
- Some considerations to examine when eliciting security requirements:
  - User authorization or privilege levels and user access controls
  - User identification and authentication
  - Data privacy
  - Deliberate data destruction, corruption, or theft
  - Protection against viruses, worms, Trojan horses, spyware, rootkits, and other malware
  - Firewall and other network security issues
  - Encryption of secure data
  - Building audit trails of operations performed and access attempts

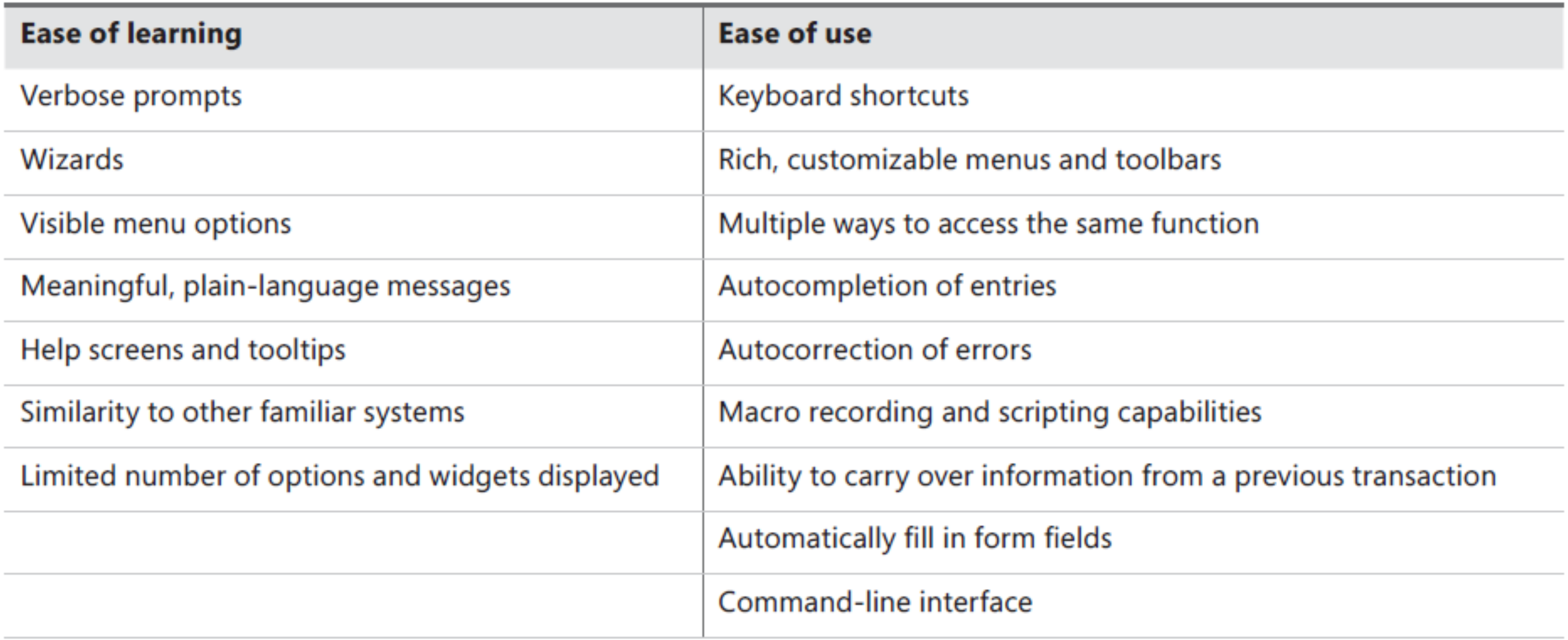
Examples:

- *SEC-1. The system shall lock a user’s account after four unsuccessful login attempts within five minutes.*
- *SEC-2. The system shall log all attempts to access secure data by users having insufficient privilege levels.*
- *SEC-3. A user shall have to change the temporary password assigned by the security officer to a previously unused password immediately following the first successful login with the temporary password.*
- *SEC-4. A door unlocks that results from a successful security badge read shall keep the door unlocked for 8.0 seconds, with a tolerance of 0.5 seconds.*
- *SEC-5. The resident antimalware software shall quarantine any incoming Internet traffic that exhibits characteristics of known or suspected virus signatures.*
- *SEC-6. The magnetometer shall detect at least 99.9 percent of prohibited objects, with a false positive rate not to exceed 1 percent.*
- *SEC-7. Only users who have Auditor access privileges shall be able to view customer transaction histories.*

Usability

- Usability addresses the countless factors that constitute what people describe colloquially as user-friendliness, ease of use, and human engineering.

Possible Design Approaches for Ease of Learning and Ease of Use



Ease of learning	Ease of use
Verbose prompts	Keyboard shortcuts
Wizards	Rich, customizable menus and toolbars
Visible menu options	Multiple ways to access the same function
Meaningful, plain-language messages	Autocompletion of entries
Help screens and tooltips	Autocorrection of errors
Similarity to other familiar systems	Macro recording and scripting capabilities
Limited number of options and widgets displayed	Ability to carry over information from a previous transaction
	Automatically fill in form fields
	Command-line interface

Usability

- Usability indicators include:
  - The average time needed for a specific type of user to complete a particular task correctly.
  - How many transactions can the user complete correctly in a given period?
  - What percentage of tasks can the user complete correctly without needing help?
  - How many errors the user makes when completing a task?
  - How many tries it takes the user to accomplish a particular task, like finding a specific function buried somewhere in the menus?
  - The delay or wait time when performing a task.
  - The number of interactions (mouse clicks, keystrokes, touch-screen gestures) required to get to a piece of information or to accomplish a task.

Examples

USE-1. A trained user shall be able to submit a request for a chemical from a vendor catalogue in an average of three minutes and a maximum of five minutes, 95 percent of the time.

USE-2. All functions on the File menu shall have defined shortcut keys that use the Control key pressed simultaneously with one other. Menu commands appearing in Microsoft Word shall use the same default shortcut keys Word uses.

USE-3. 95 percent of chemists who have never used the Chemical Tracking System before shall be able to request a chemical correctly within 15 minutes of orientation.

Internal quality attributes

- Efficiency
- Modifiability
- Portability
- Reusability
- Scalability
- Verifiability

Efficiency

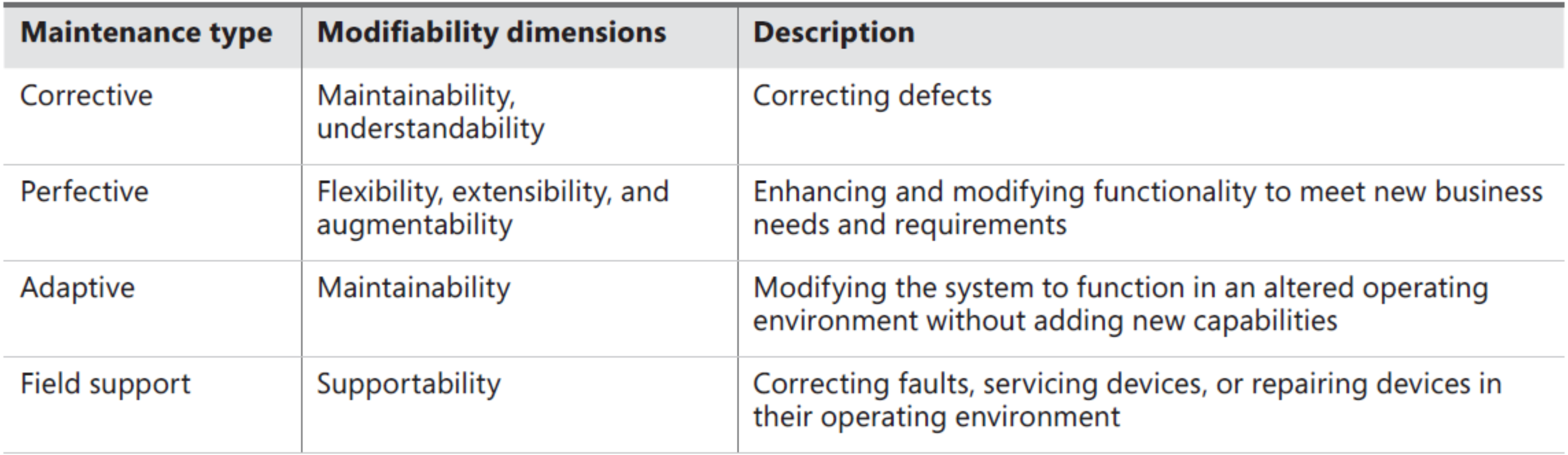
- Efficiency is closely related to the external quality attribute of performance.
- Efficiency measures how well the system utilizes processor capacity, disk space, memory, or communication bandwidth.
- If a system consumes too much of the available resources, users will encounter degraded performance.

EFF-1. At least 30 percent of the processor capacity and memory available to the application shall be unused at the planned peak load conditions.

EFF-2. The system shall warn the operator when the user load exceeds 80 percent of the maximum planned capacity.

Modifiability

- Modifiability addresses how easily the software designs and code can be understood, changed, and extended.
- Some aspects of modifiability



Maintenance type	Modifiability dimensions	Description
Corrective	Maintainability, understandability	Correcting defects
Perfective	Flexibility, extensibility, and augmentability	Enhancing and modifying functionality to meet new business needs and requirements
Adaptive	Maintainability	Modifying the system to function in an altered operating environment without adding new capabilities
Field support	Supportability	Correcting faults, servicing devices, or repairing devices in their operating environment

Examples

MOD-1. A maintenance programmer experienced with the system shall be able to modify existing reports to conform to revised chemical-reporting regulations from the federal government with 10 hours or less of development effort.

MOD-2. Function calls shall not be nested more than two levels deep.

SUP-1. A certified repair technician shall be able to replace the scanner module in no more than 10 minutes.

SUP-2. The printer shall display an error message if replacement ink cartridges were not inserted in the proper slots.

Portability

- The effort needed to migrate software from one operating environment to another.
- The ability to internationalize and localize a product.
- Portability has become increasingly important as applications must run in multiple environments, such as Windows, Mac, and Linux; iOS and Android; and PCs, tablets, and phones. Data portability requirements are also necessary.

POR-1. Modifying the iOS version of the application to run on Android devices shall require changing at most 10 percent of the source code.

POR-2. The user shall be able to port browser bookmarks to and from Firefox, Internet Explorer, Opera, Chrome, and Safari.

POR-3. The platform migration tool shall transfer customized user profiles to the new installation without user action.

Reusability

- The relative effort required to convert a software component for other applications.
- Reusable software must be modular, well documented, independent of a specific application and operating environment, and somewhat generic in capability.
- Reusability goals are challenging to quantify. Specify which elements of the new system need to be constructed in a manner that facilitates their reuse.

REU-1. The chemical structure input functions shall be reusable in other object code-level applications.

REU-2. At least 30 percent of the application architecture shall be reused from the approved reference architectures.

REU-3. The pricing algorithms shall be reusable by future store-management applications.

Scalability

- Scalability requirements address the applicant's ability to grow to accommodate more users, data, servers, geographic locations, transactions, network traffic, searches, and other services without compromising performance or correctness.

SCA-1. The capacity of the emergency telephone system must be able to be increased from 500 calls per day to 2,500 calls per day within 12 hours.

SCA-2. The website shall be able to handle a page-view growth rate of 30 percent per quarter for at least two years without user-perceptible performance degradation.

SCA-3. The distribution system shall accommodate up to 20 new warehouse centers.

Verifiability

- More narrowly referred to as *testability*, verifiability refers to how well software components or the integrated product can be evaluated to demonstrate whether the system functions as expected.
- Designing for verifiability is critical if the product has complex algorithms and logic or contains subtle functionality interrelationships.
- Verifiability is also essential if the product is often modified because it will undergo frequent regression testing to determine whether the changes damaged any existing functionality.
- Designing software for verifiability means making it easy to place the software into the desired pretest state, provide the necessary test data, and observe the test result.

Examples

VER-1. The development environment configuration shall be identical to the test configuration environment to avoid irreproducible testing failures.

VER-2. A tester shall be able to configure which execution results are logged during testing.

VER-3. The developer shall be able to set the computational module to show the interim results of any specified algorithm group for debugging purposes.

VER-4. The maximum cyclomatic complexity of a module shall not exceed 20.

Cyclomatic complexity measures the number of logic branches in a source code module. Adding more branches and loops to a module makes it harder to understand, test, and maintain.

Quality Attributes Trade-offs

	Availability	Efficiency	Installability	Integrity	Interoperability	Modifiability	Performance	Portability	Reliability	Reusability	Robustness	Safety	Scalability	Security	Usability	Verifiability
Availability									+		+					
Efficiency	+				−	−	+	−			−		+		−	
Installability	+								+					+		
Integrity			−		−		−			−		+		+	−	−
Interoperability	+		−	−			−	+	+		+	−		−		
Modifiability	+		−				−		+	+			+			+
Performance		+			−	−		−			−		−			
Portability		−			+	−	−			+				−	−	+
Reliability	+	−		+		+	−				+	+		+	+	+
Reusability		−		−	+	+	−	+						−		+
Robustness	+	−	+	+	+		−		+			+	+	+	+	
Safety		−		+	+		−				+			+	−	−
Scalability	+	+		+			+	+	+		+					
Security	+			+	+		−									
Usability		−	+				−	−	+		+	+				−
Verifiability	+		+	+		+			+	+	+	+		+	+	

Constraints

- A constraint places restrictions on the design or implementation choices available to the developer.

- External stakeholders can impose constraints. These other systems interact with the one you are building or maintaining or other life cycle activities for your systems, such as transition and maintenance.

- Other constraints result from existing agreements, management, and technical decisions (ISO/IEC/IEEE 2011).

Constraints

- Sources of constraints include:
  - Specific technologies, tools, languages, and databases that must be used or avoided.
  - Restrictions because of the product’s operating environment or platform, such as the types and versions of web browsers or operating systems used.
  - Required development conventions or standards. (For instance, if the customer’s organization maintains the software, the organization might specify design notations and coding standards that a subcontractor must follow.)
  - Backward compatibility with earlier products and potential forward compatibility, such as knowing which software version was used to create a specific data file.
  - Limitations or compliance requirements imposed by regulations or other business rules.

Constraints

- Sources of constraints include:
  - Hardware limitations include timing requirements, memory or processor restrictions, size, weight, materials, or cost.
  - Physical restrictions because of the operating environment, user characteristics, or limitations.
  - Existing interface conventions to be followed when enhancing an existing product.
  - Interfaces with other systems, such as data formats and communication protocols.
  - Restrictions because of the display size, as when running on a tablet or phone.
  - Standard data interchange formats used, such as XML or RosettaNet for e-business.

Examples

CON-1. The user clicks at the top of the project list to change the sort sequence. [specific user interface control imposed as a design constraint on a functional requirement]

CON-2. Only open-source software available under the GNU General Public License may be used to implement the product. [implementation constraint]

CON-3. The application must use Microsoft .NET framework 4.5. [architecture constraint]

CON-4. ATMs contain only $20 bills. [physical constraint]

CON-5. Online payments may be made only through PayPal. [design constraint]

CON-6. All textual data the application uses shall be stored in XML files. [data constraint]

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Quality Requirements

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

Behavioural Modelling

Glossary

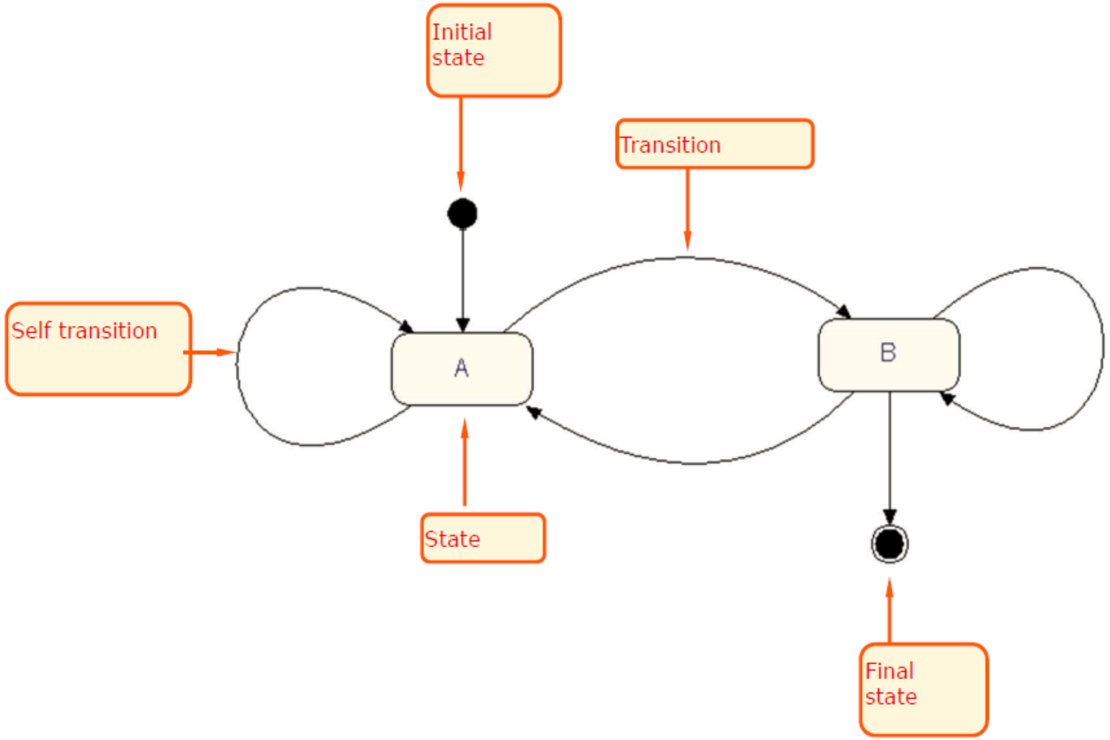
- System behaviour: how a system acts and reacts.
- Behavior model: a view of a system that emphasizes the system’s behaviour as a whole (as it appears to outside users).
- State-driven behaviour: means that the object’s behaviour can be divided into disjoint sets.

UML State Diagrams

- **State diagram**: Shows data and behaviour of a single object throughout its lifetime.
  - set of states (including an initial start state)
  - transitions between states
  - entire diagram is drawn from that object's perspective
- What objects are best used with state diagrams?
  - large, complex objects with a long lifespan
  - domain (“model”) objects
  - not valuable for doing state diagrams for every class in the system!
- Commonly used in design to describe an object’s behaviour as a guide to implementation
- Used in RE to model interface specifications (e.g. UI)
- Specify each object’s contribution to all scenarios of all use cases.

UML State Diagrams

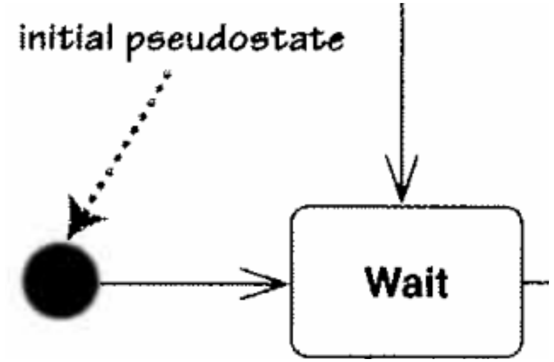
- Represented by Finite State Machine (FSM)
  - Finite State Automaton (FSA) is another term for FSM.



States

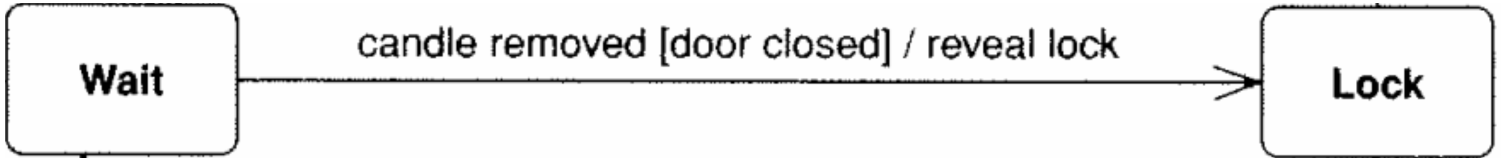
- **State**: conceptual description of the data in the object
  - represented by the object's field values

- Entire diagram is drawn from the central object's perspective
  - only include states/concepts that this object can see and influence
  - do not include every possible value for the fields; only ones that are conceptually different

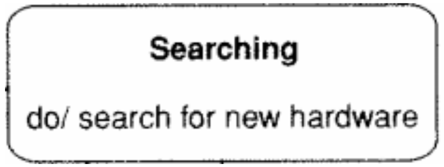


Transitions

- **Transition**: movement from one state to another



- *Event* [*condition*] / *action*
  - event: triggers (potential) state change
  - condition: a boolean condition that must be true
  - action: any behaviour executed during the transition *(optional)*



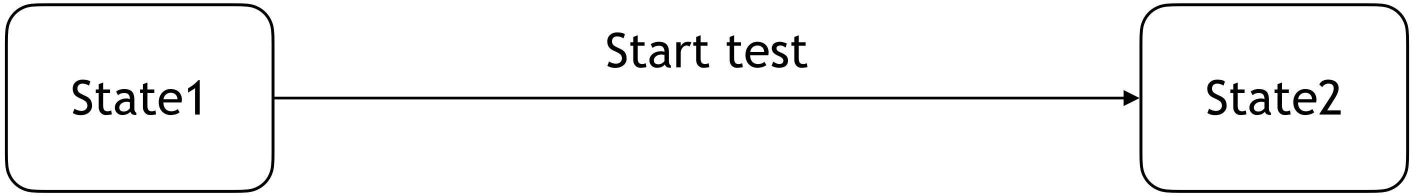
- Transitions must be mutually exclusive (deterministic)
  - it must be clear on what transition to take for an event
  - most transitions are instantaneous (existing or measured at a particular instant), except “do” activities

Note:

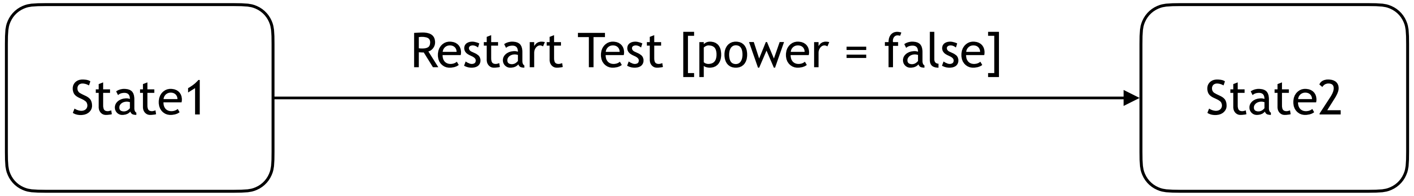
- Event is a noteworthy or significant occurrence in the environment.
  - input message from the env. (login request)
  - change in the env. (coin inserted, elevator button pressed)
  - passage of time
  - multiple events on a transition label are alternative triggers
- Condition is a Boolean expression:
  - over domain model phenomena
  - over state-machine variables
- Action is the system’s response to an event; it is non-interruptible.
  - output message
  - change to env phen. (Turnstile.locked := true.   AddLoan(m:LibraryMember, p:Publication, today:Date)
  - multiple actions are separated by “;” and execute sequentially

Example

- Event Start Test changes the state from State 1 to State 2.

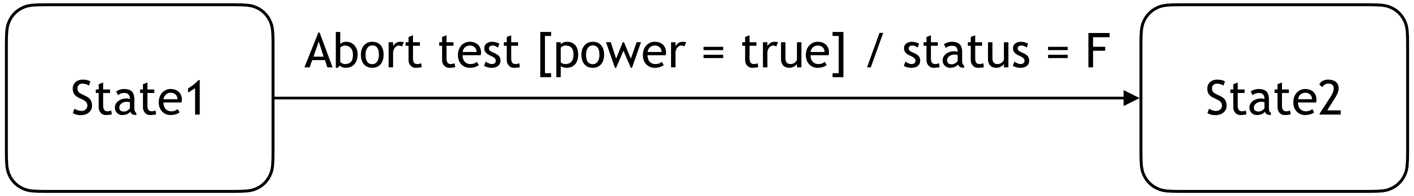


- The transition takes place when the event Restart test occurs and the power is false.

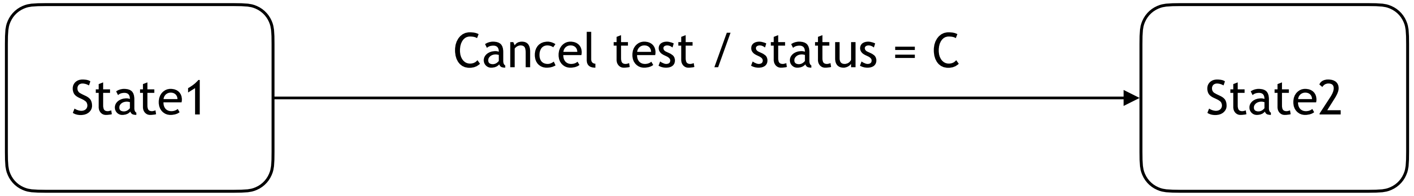


Example

- The variable status is set to F the event Abort occurs, provided that power is true.

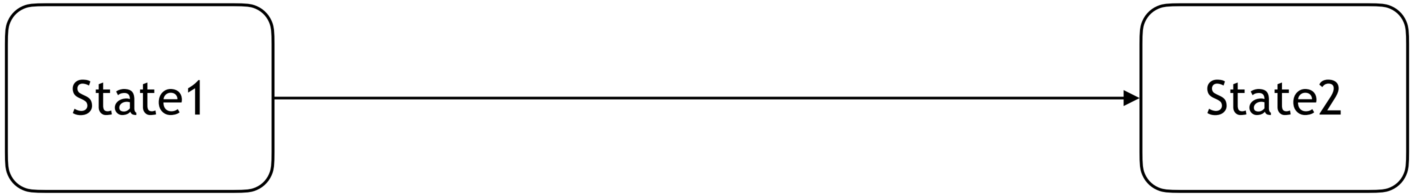


- No conditions on cancelling test. The variable status is set to C.

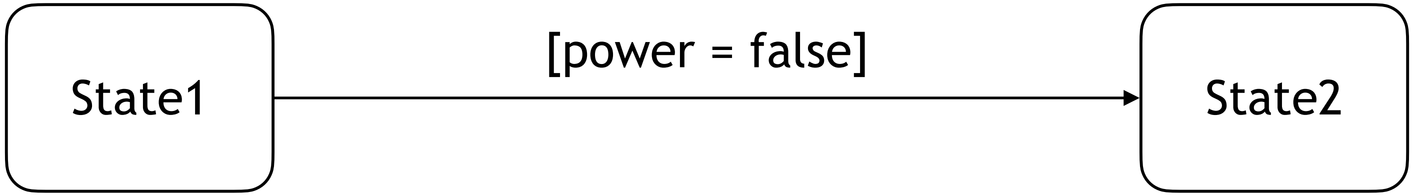


Example

- No event on the transition. The transition happens automatically.

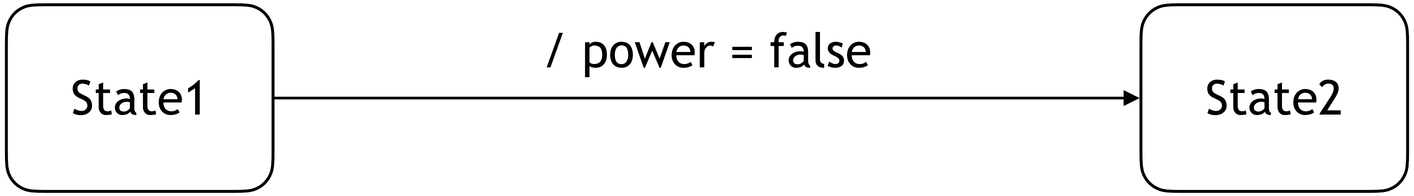


- No event on the transition. The transition happens automatically, provided that the condition evaluates to true.

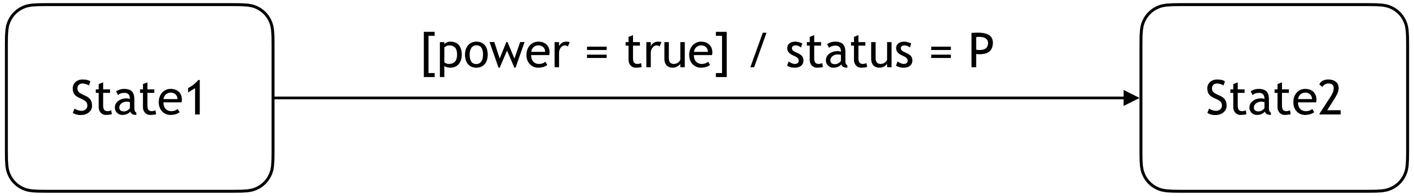


Example

- No event on the transition. The transition happens automatically, and power is set to false.



- No event on the transition. The transition happens automatically, provided that the condition evaluates to true. Status is set to P.

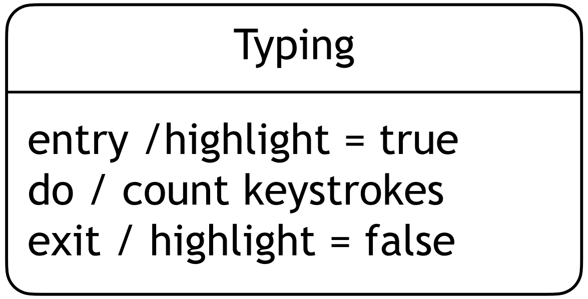


How are transactions handled?

- If an object is in a state S that responds to an event E, it acts upon that event.
  - It transitions to the specified state if the event triggers a transition, and the condition (if any) on that transition evaluates to true.
  - It executes any actions associated with that transition.
- Events are quietly discarded if:
  - A transition is triggered, but the transition’s condition evaluates to false.
  - The event does not explicitly trigger a transition or reaction.

Internal Activities

- **Internal activity**: actions that the central object takes on itself
  - sometimes drawn as self-transitions (events that stay in the same state)



Typing

entry /highlight = true
do / count keystrokes
exit / highlight = false

- entry/exit activities
  - reasons to start/stop being in that state

- Take time; interruptible; may require computation.

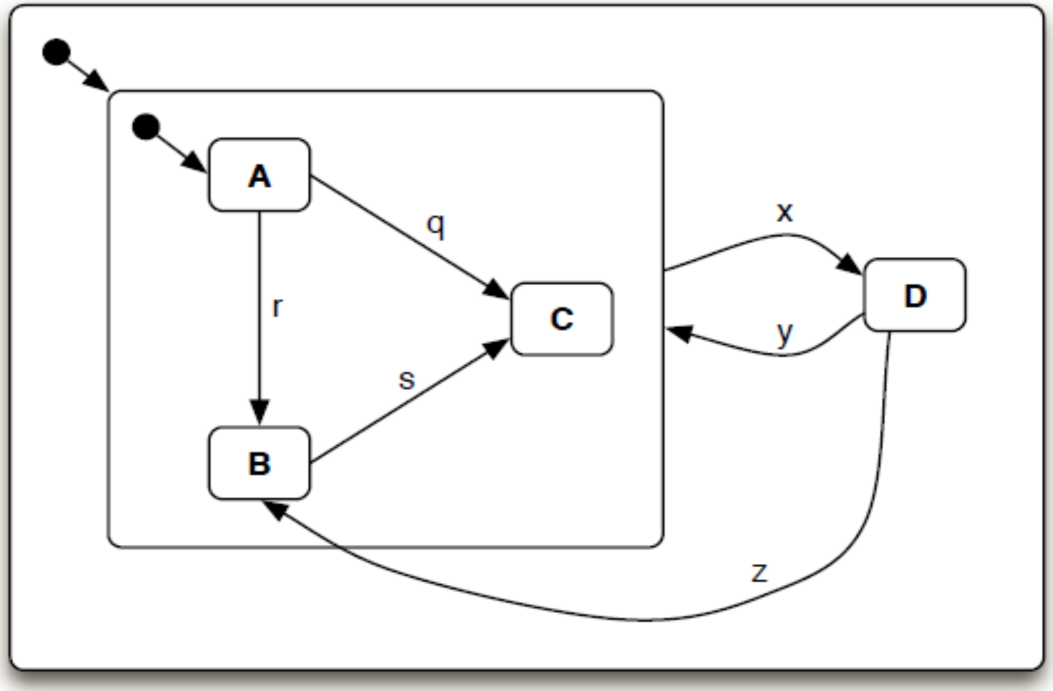
Composite State

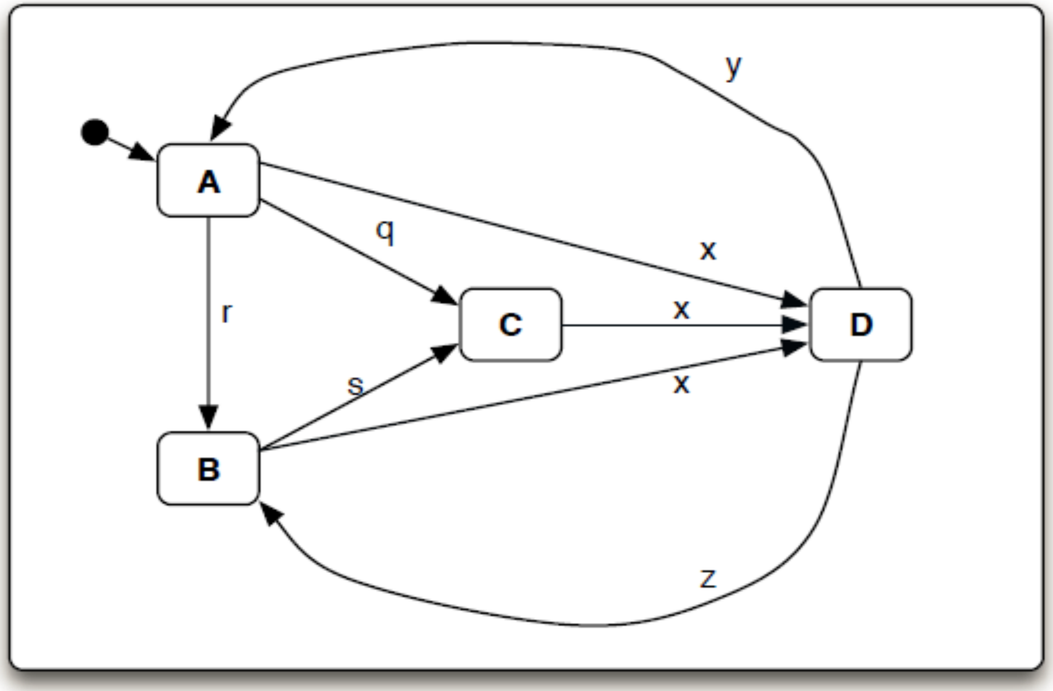
- Combines states and transitions that work together towards a common goal. There are two kinds:
  1. Hierarchical (simple / or-states)
  2. Concurrent (orthogonal / and-states)

Hierarchical State

Hierarchy is used to cluster states with similar behaviours.

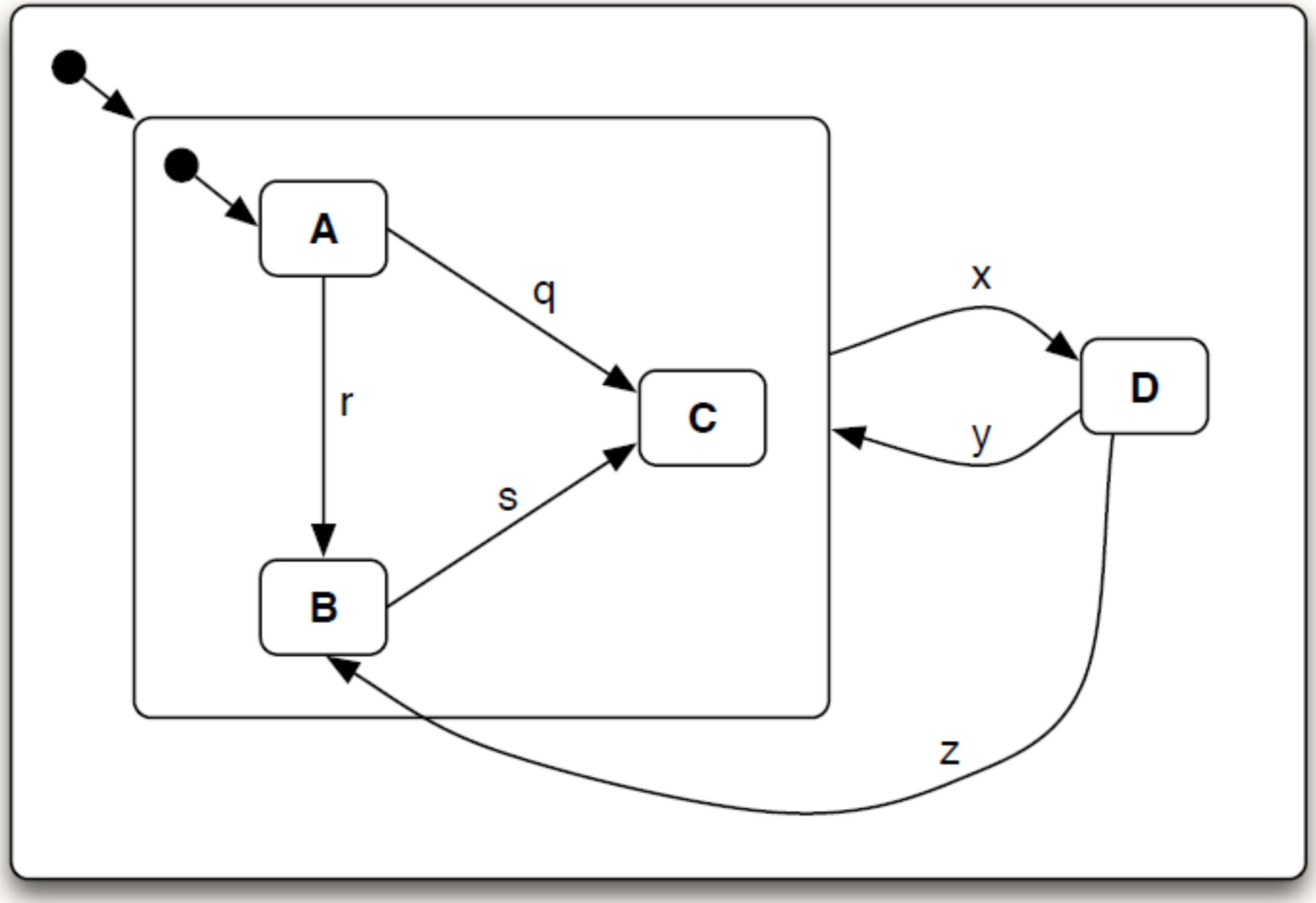
- One transition leaving a superstate represents a transition from each of the superstate’s descendent states.





Exercise

1. What happens if event z occurs when in state D?
2. What happens if event y occurs when in state D?
3. Can the execution ever leave state C?

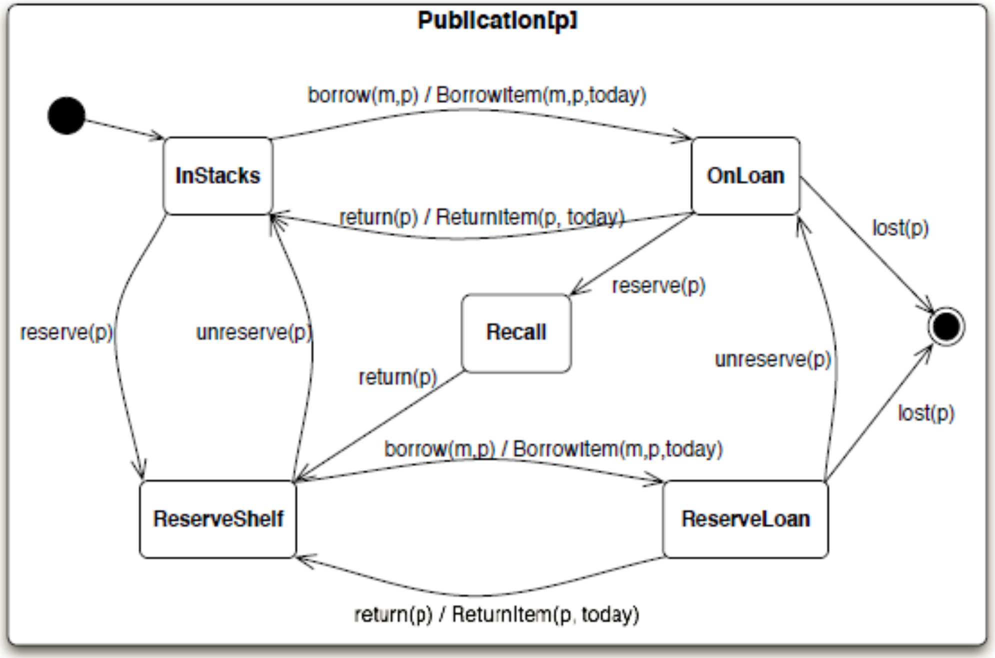


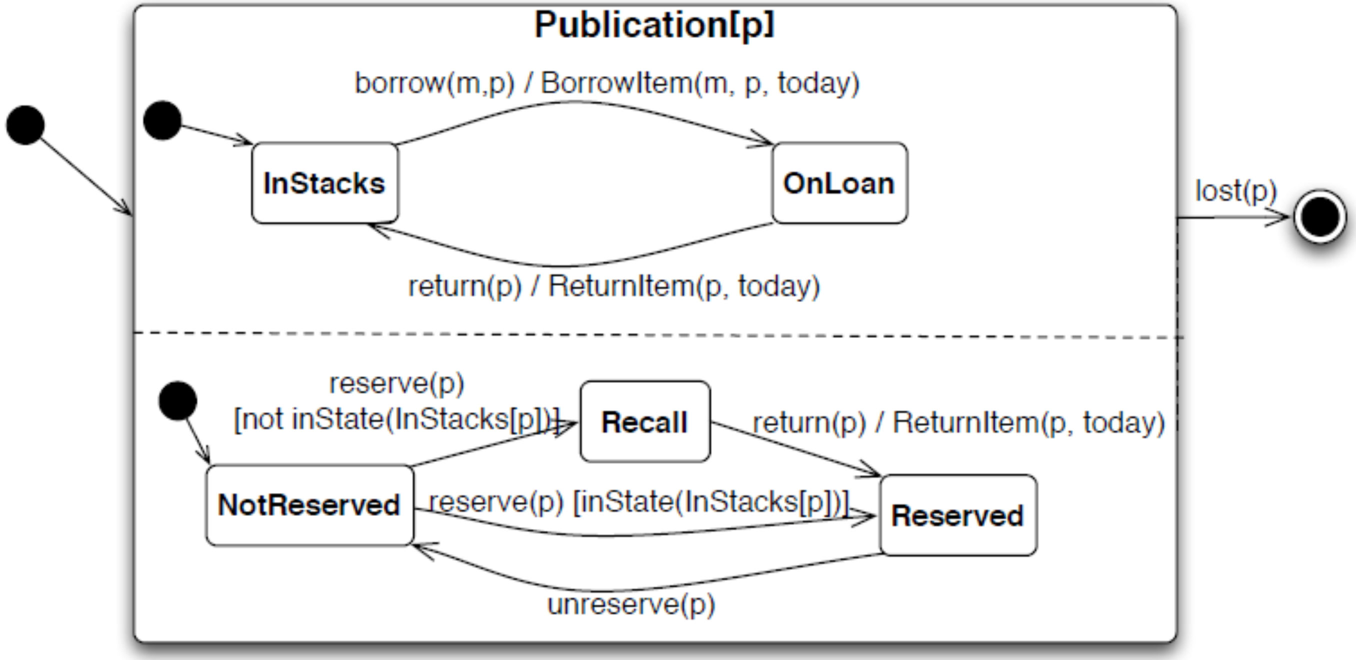
Concurrent State

Some systems have orthogonal behaviors that are best modelled as concurrent state machines

- Regions within a concurrent state execute in parallel.
- Each has its own thread of control.
- Each can “see” and react to events /conditions in the world

Concurrent State

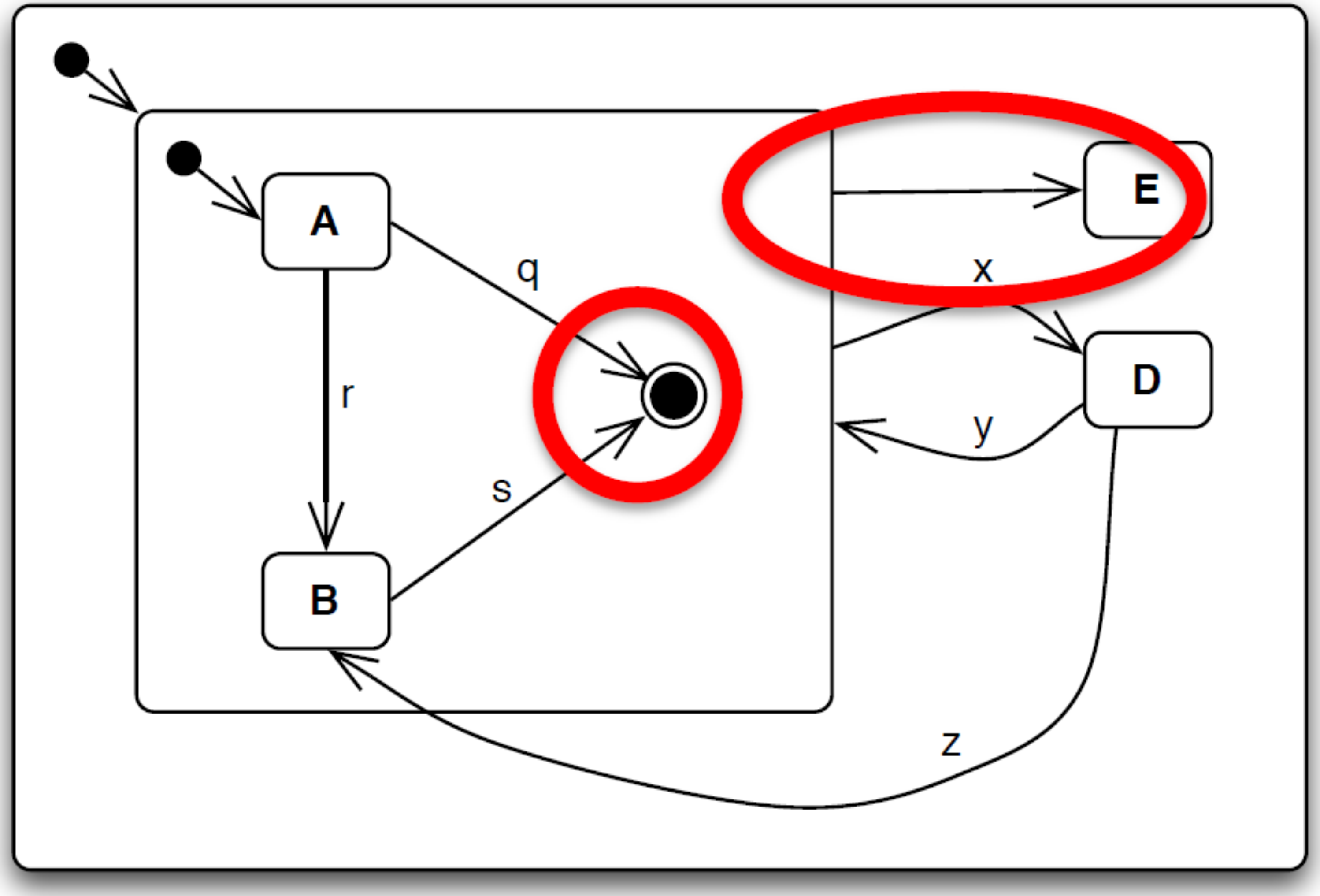

Publication[p]
borrow(m,p) / BorrowItem(m,p,today)
InStacks
OnLoan
return(p) / ReturnItem(p, today)
lost(p)
reserve(p)
Recall
reserve(p)
unreserve(p)
unreserve(p)
return(p)
lost(p)
borrow(m,p) / BorrowItem(m,p,today)
ReserveShelf
ReserveLoan
return(p) / ReturnItem(p, today)


Publication[p]
borrow(m,p) / BorrowItem(m, p, today)
InStacks
OnLoan
return(p) / ReturnItem(p, today)
lost(p)
reserve(p)
[not inState(InStacks[p])]
Recall
return(p) / ReturnItem(p, today)
NotReserved
reserve(p) [inState(InStacks[p])]
Reserved
unreserve(p)

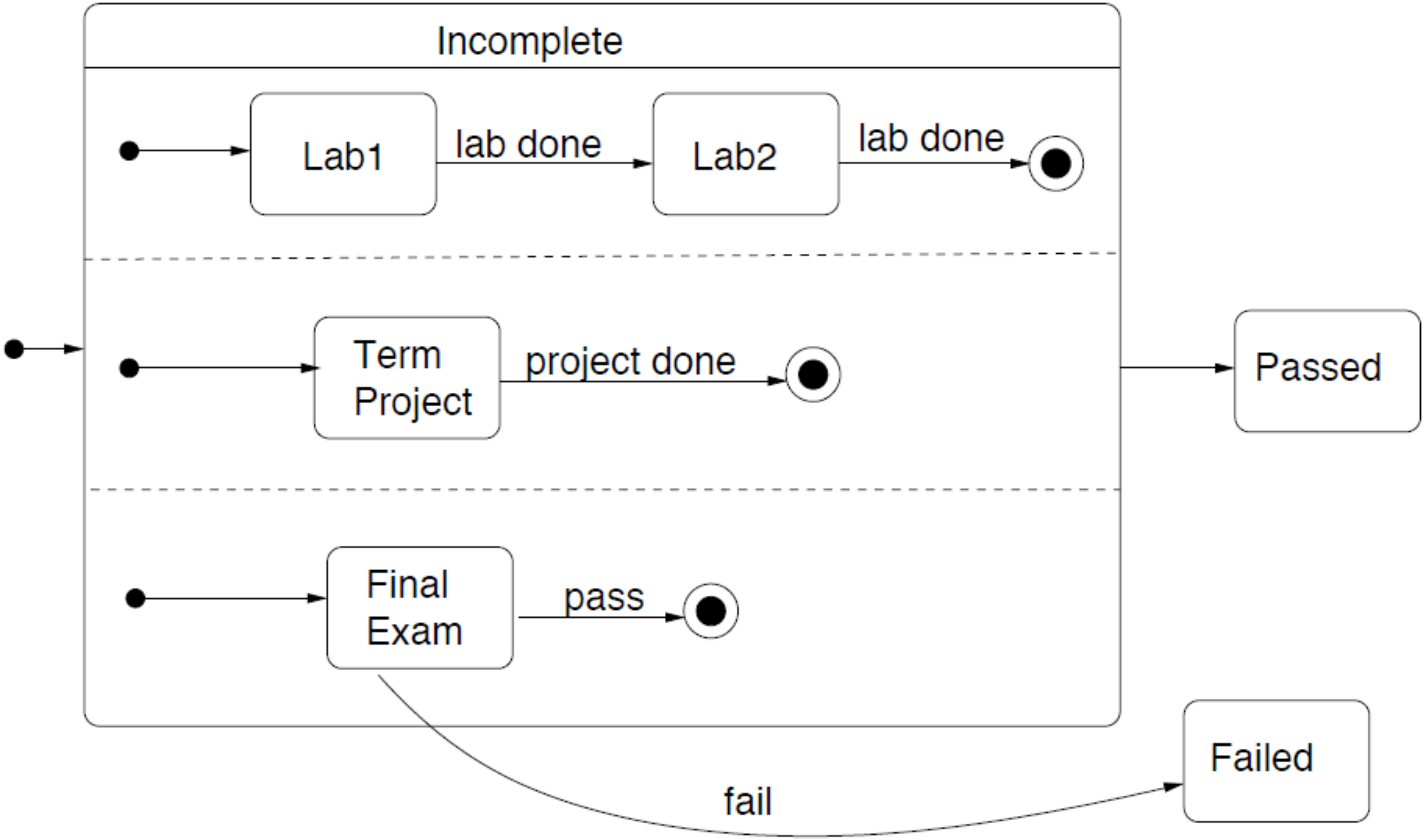
Final State

A transition that has no event or condition in its label is enabled when its

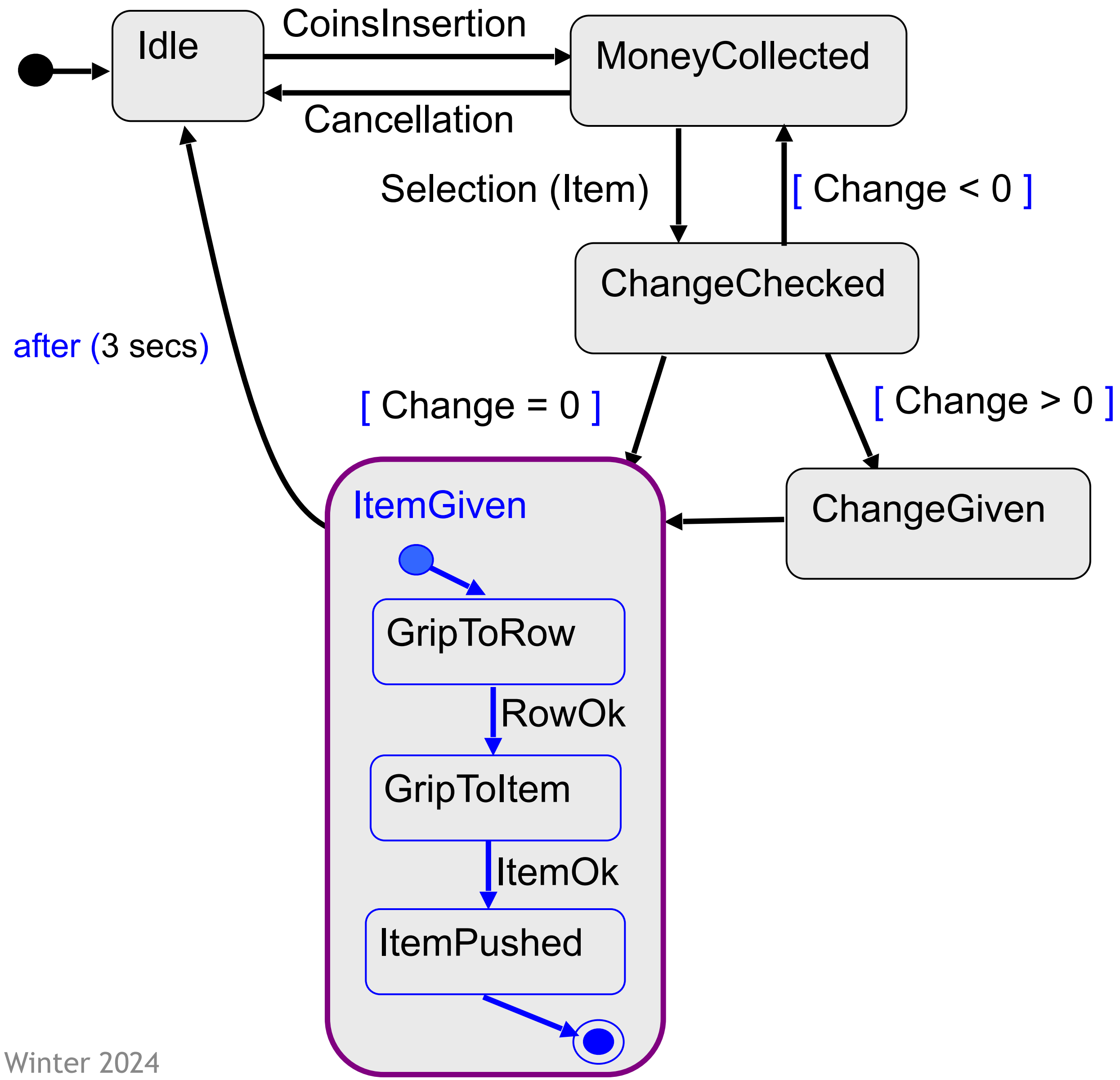
- source state is basic and idle, or
- source superstate entered its final state, or
- source basic state has finished internal activity



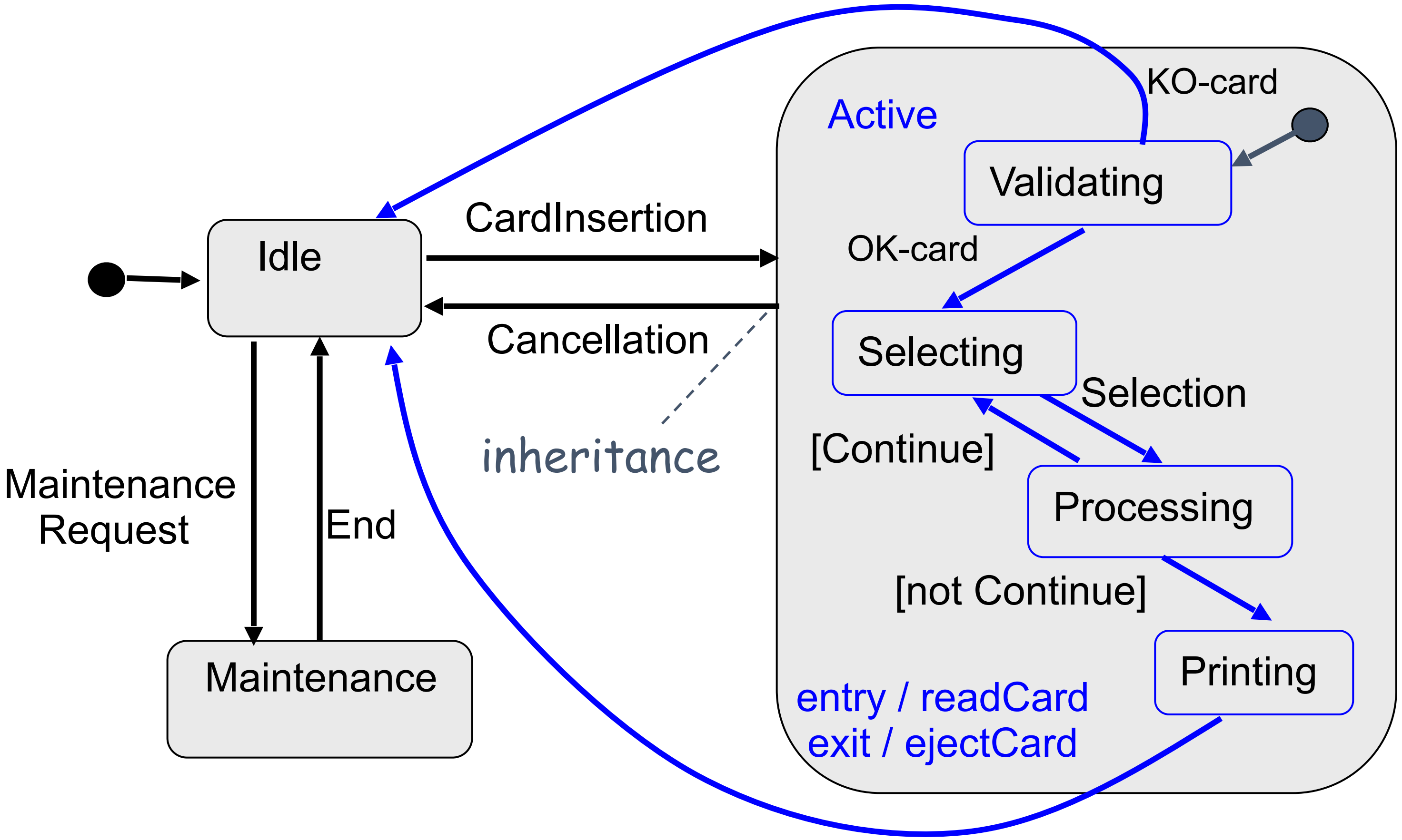
Concurrency and Final States



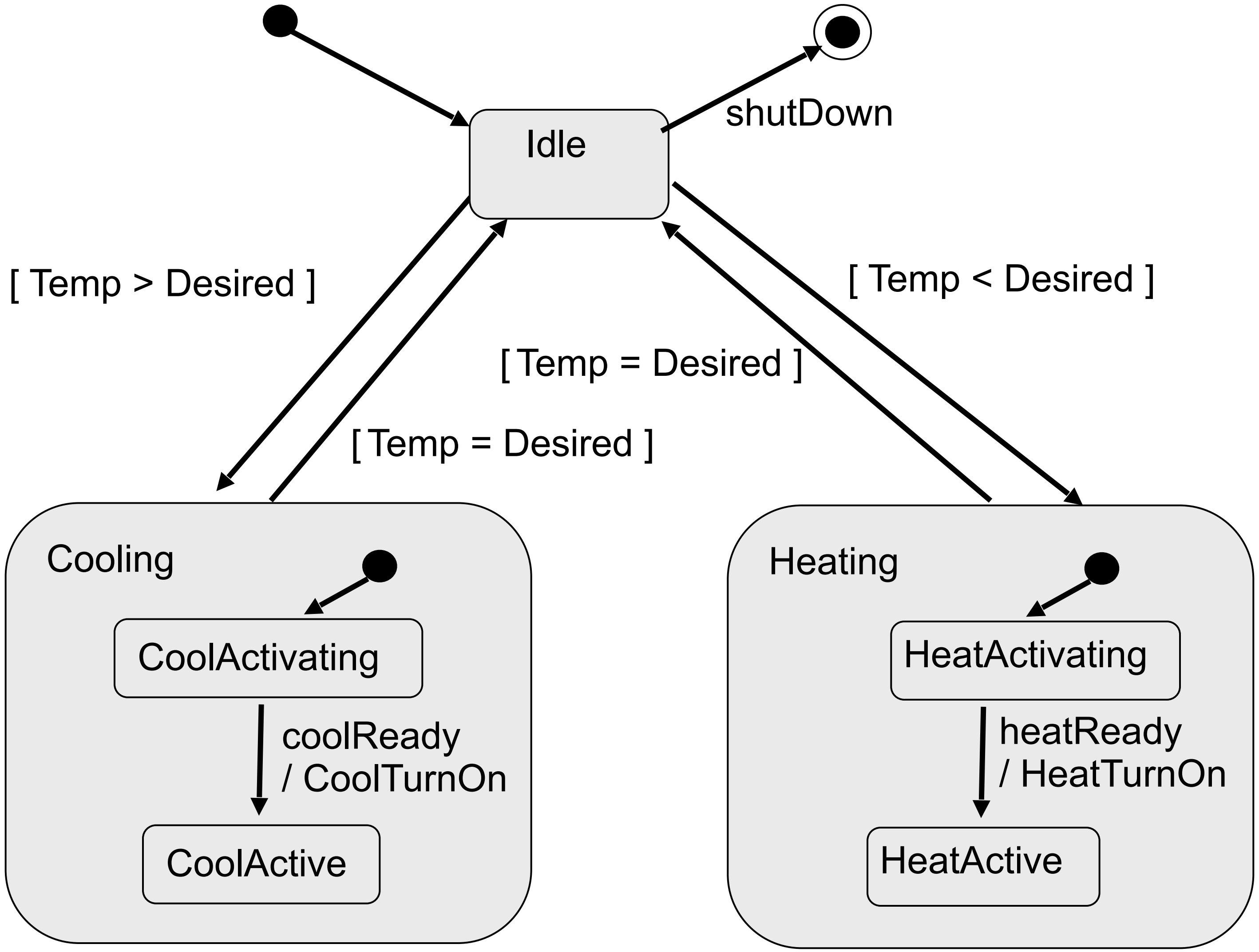
Sequential decomposition: vending machine example



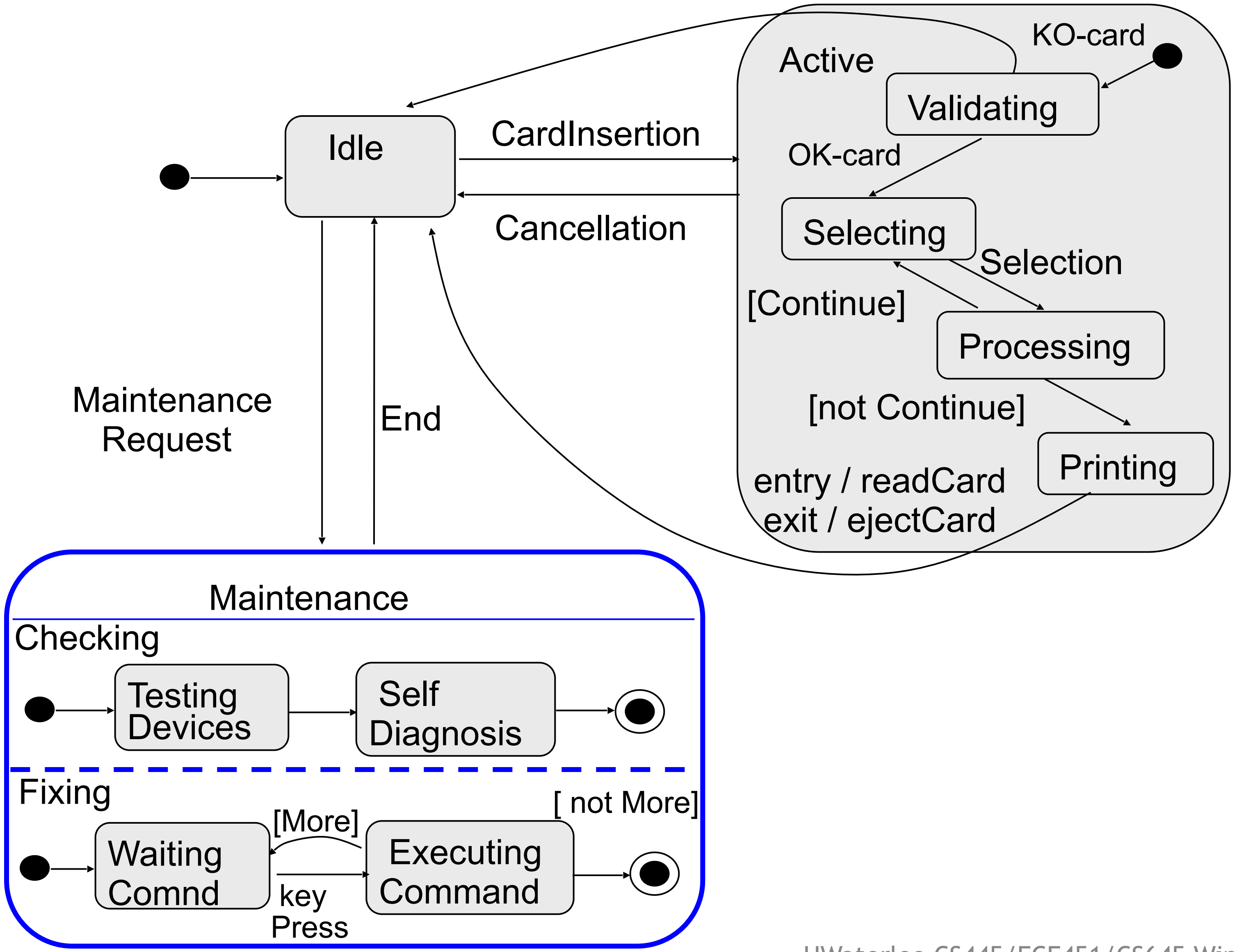
Sequential decomposition: cash machine example

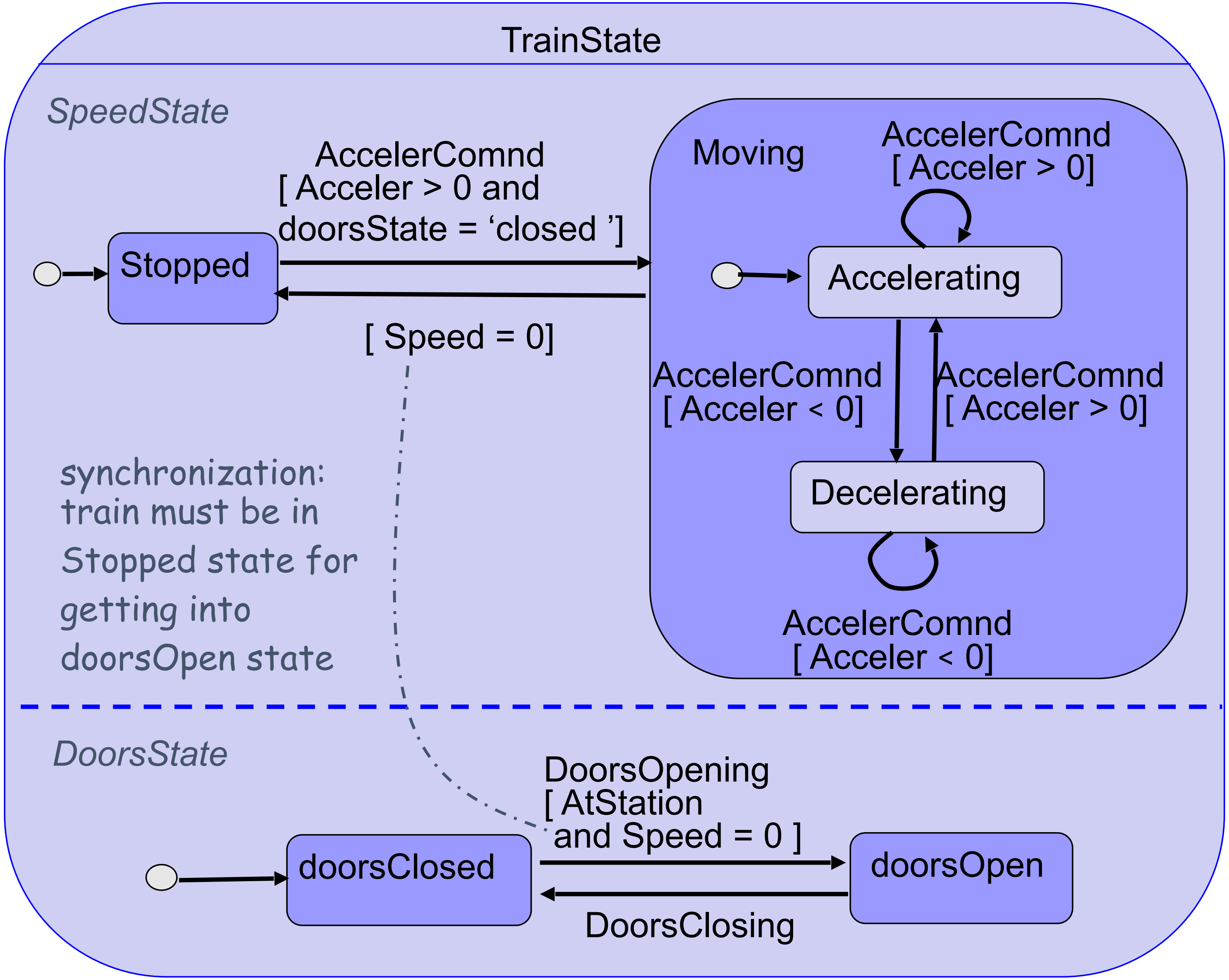


Sequential decomposition: thermostat controller



Parallel and sequential decomposition: example

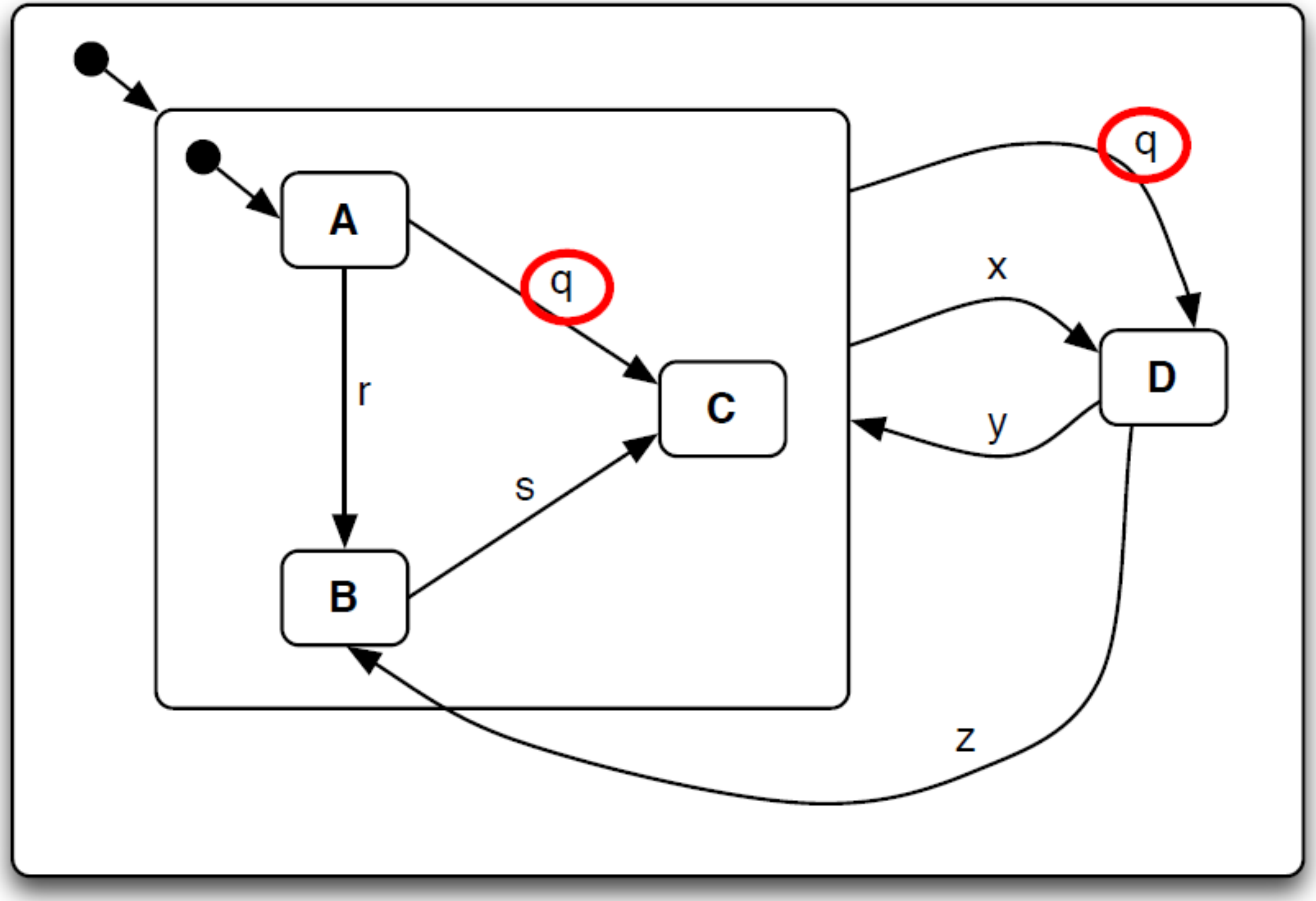



TrainState
SpeedState
AccelerComnd
[Acceler > 0 and
doorsState = ‘closed ’]
Stopped
[Speed = 0]
Moving
AccelerComnd
[Acceler > 0]
Accelerating
AccelerComnd
[Acceler < 0]
AccelerComnd
[Acceler > 0]
Decelerating
AccelerComnd
[Acceler < 0]
synchronization:
train must be in
Stopped state for
getting into
doorsOpen state
DoorsState
DoorsOpening
[AtStation
and Speed = 0]
doorsClosed
doorsOpen
DoorsClosing

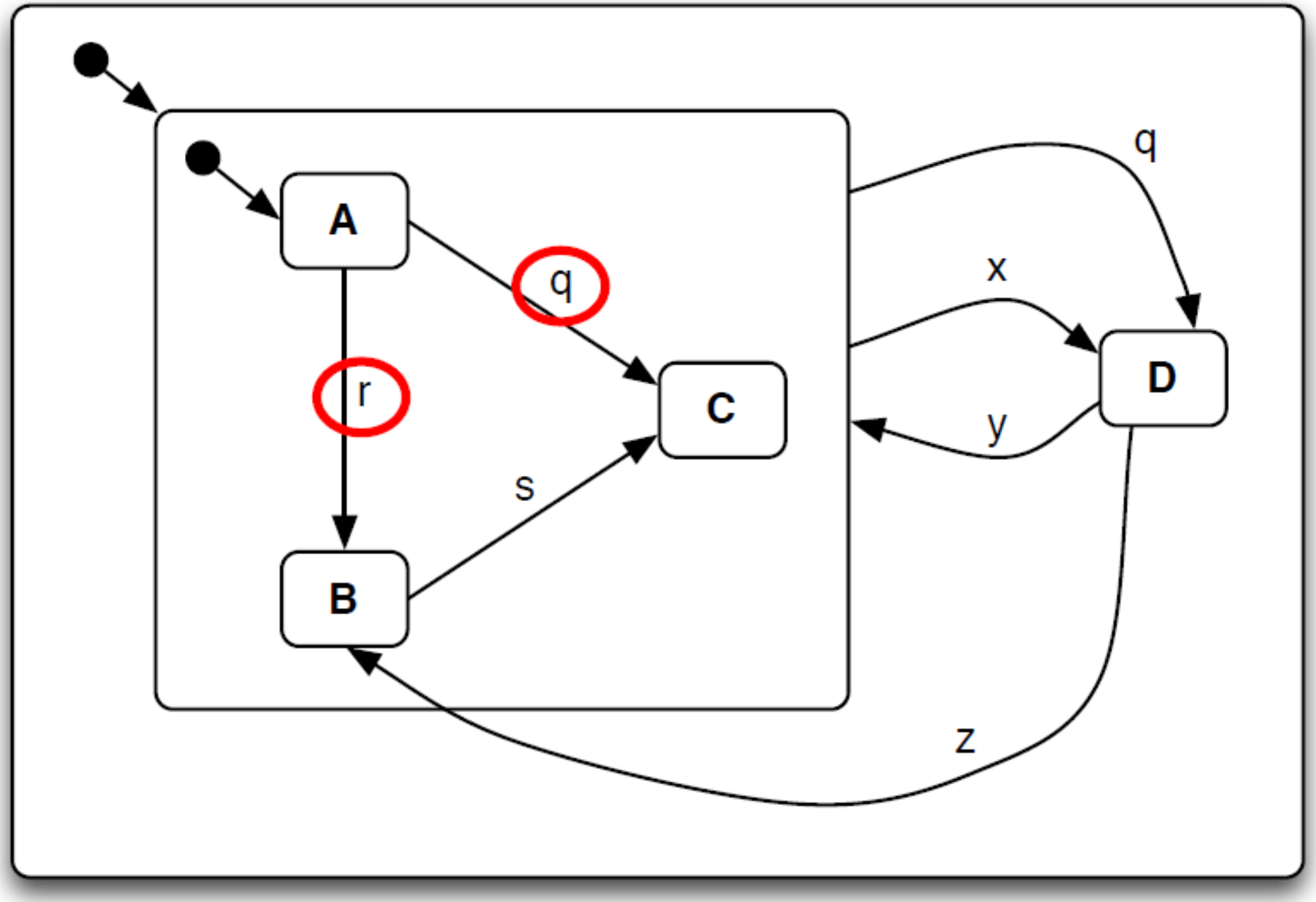
Check Resulting Concurrent SM

- **Within** the concurrent state, for one controlled variable
    - Unreachable states? (from the initial state)
    - Missing states? (incl. final state)
    - Missing or inadequate transitions? (events, guards)
    - Missing actions?
- **Between** concurrent states, for different controlled variables
    - Synchronization needed? (as seen before)
        - Shared events? Synchronizing guards? Event notification?
    - Lexical consistency of event names? (as seen before)

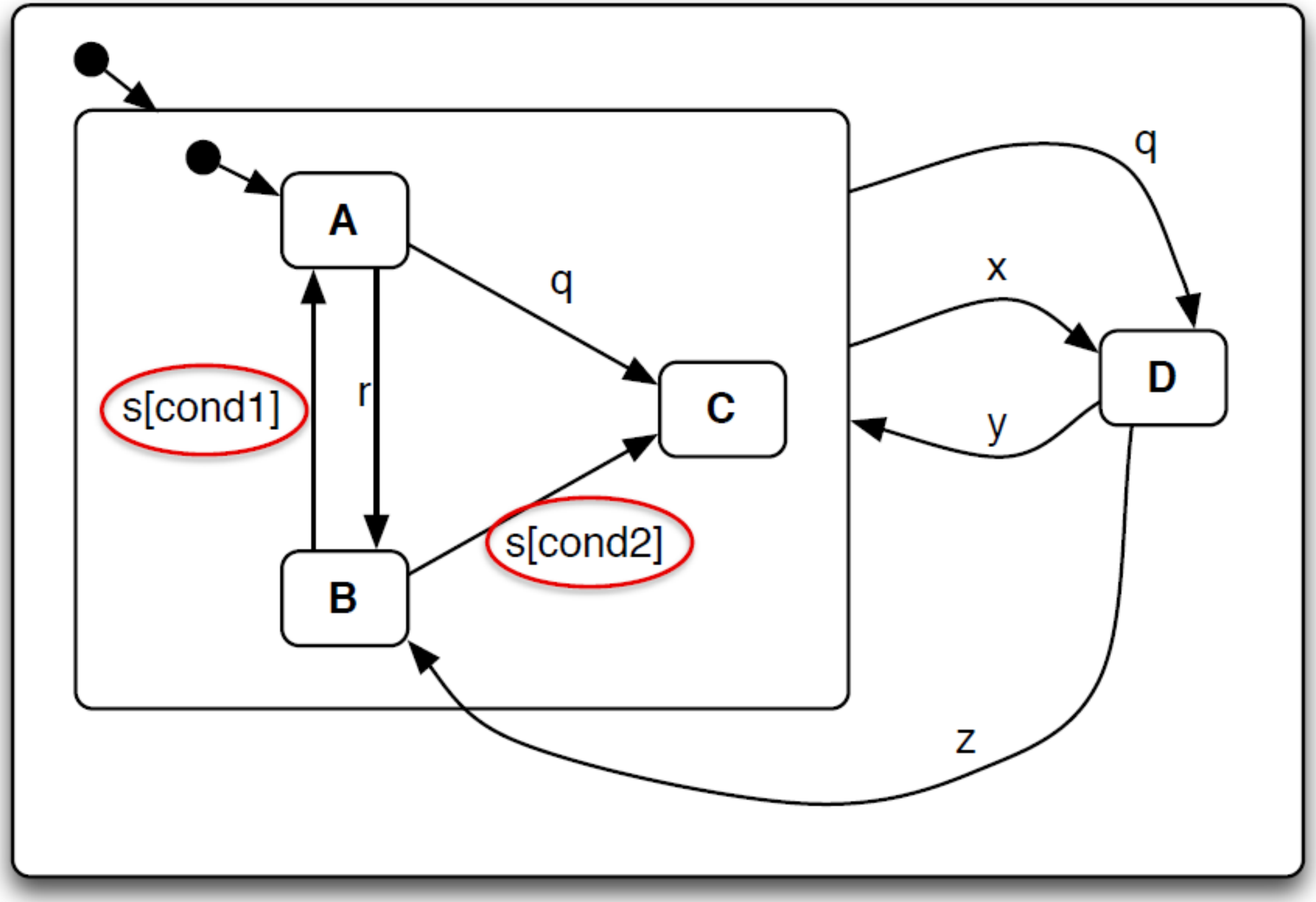
Priority



Priority



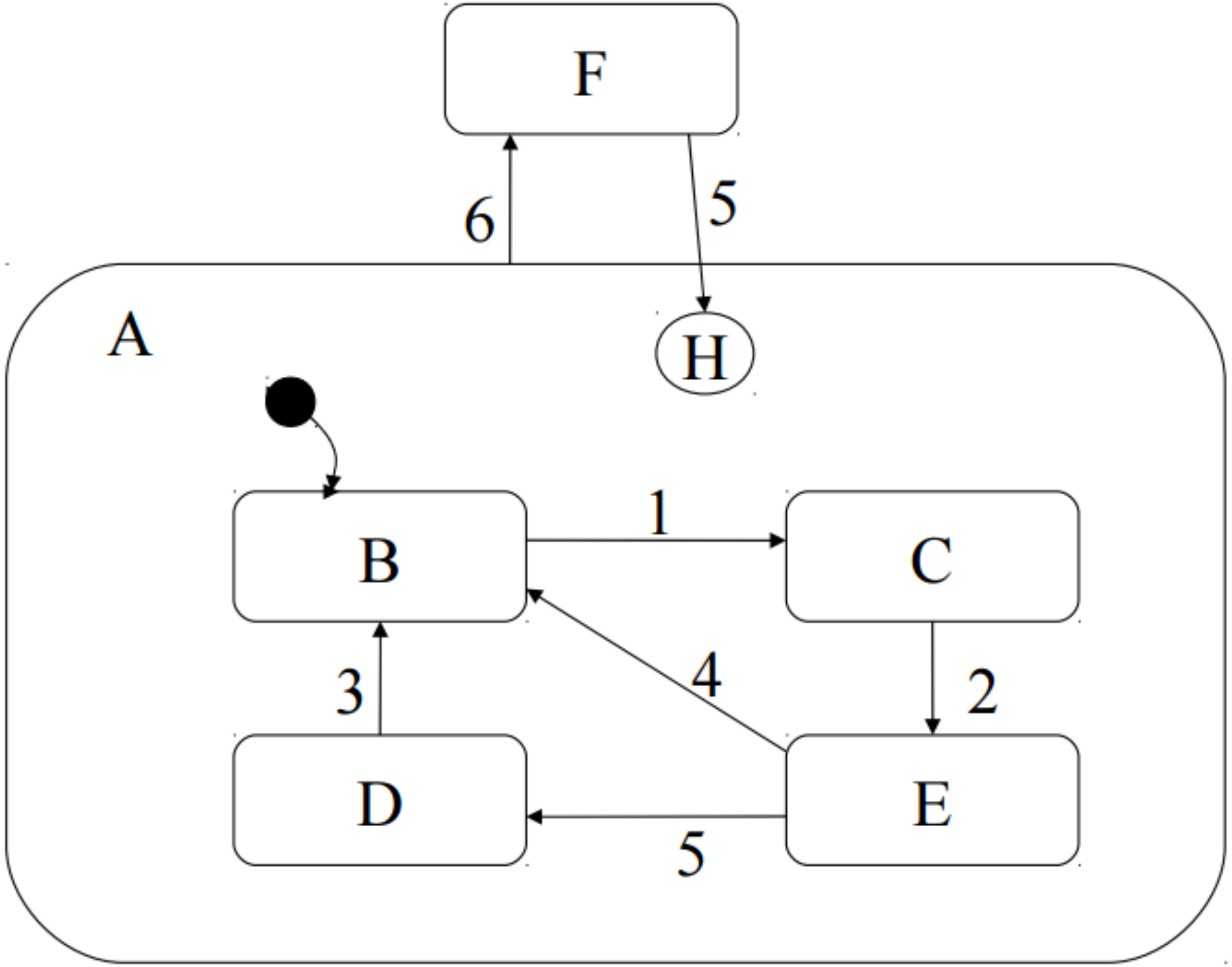
Determinism



History

- Provides a way of entering a group of states based on the system’s history in that group.
  - That is, the state entered is the most recently visited state in that group.
  - In the next slide, when event 5 occurs and state A is entered, the history mechanism is used to determine the next state within A.
    - This is read as “enter the most recently visited state in the group (B, C, D, E) or enter state B if this is the first visit to the state.”

History

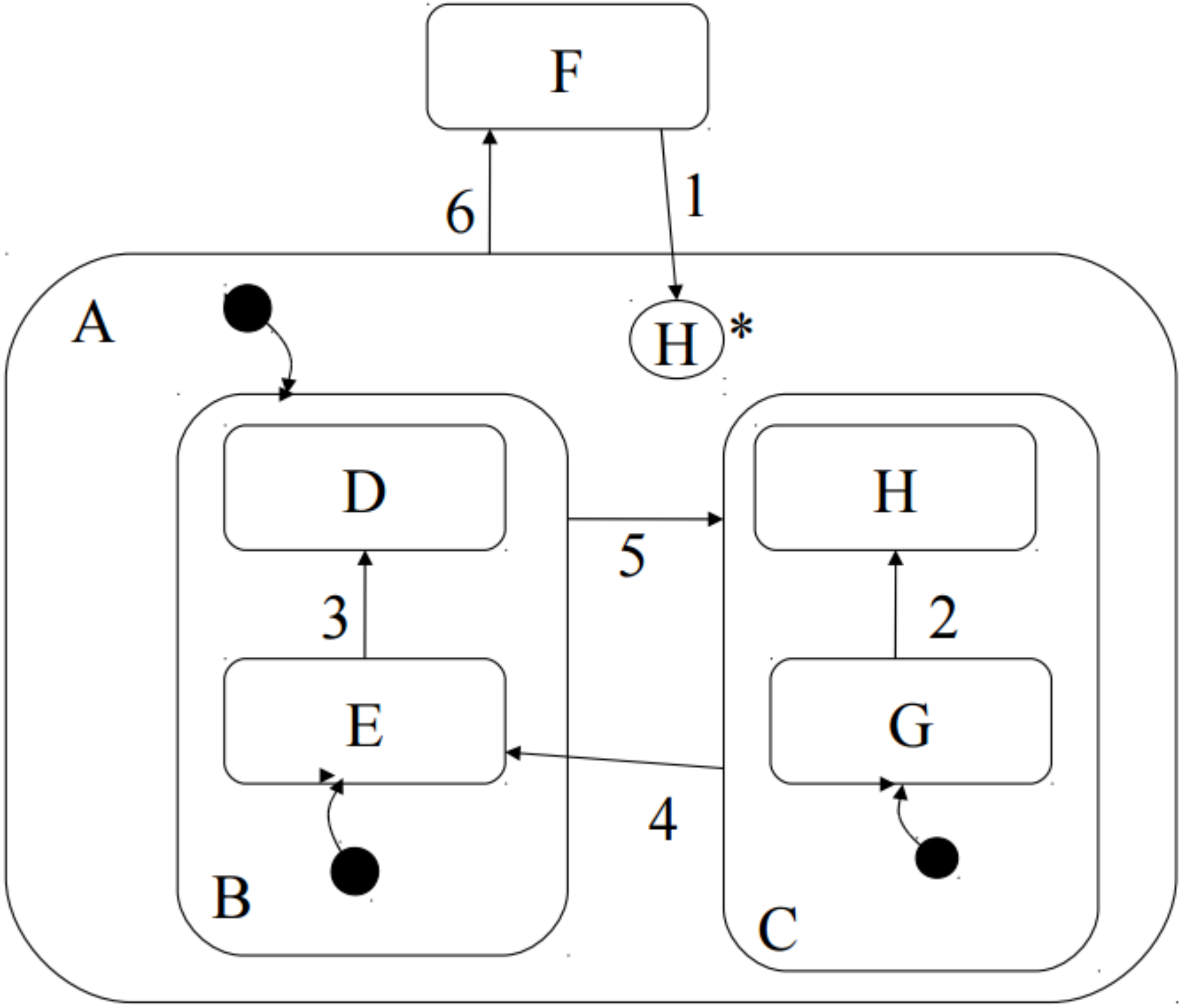


History Usage

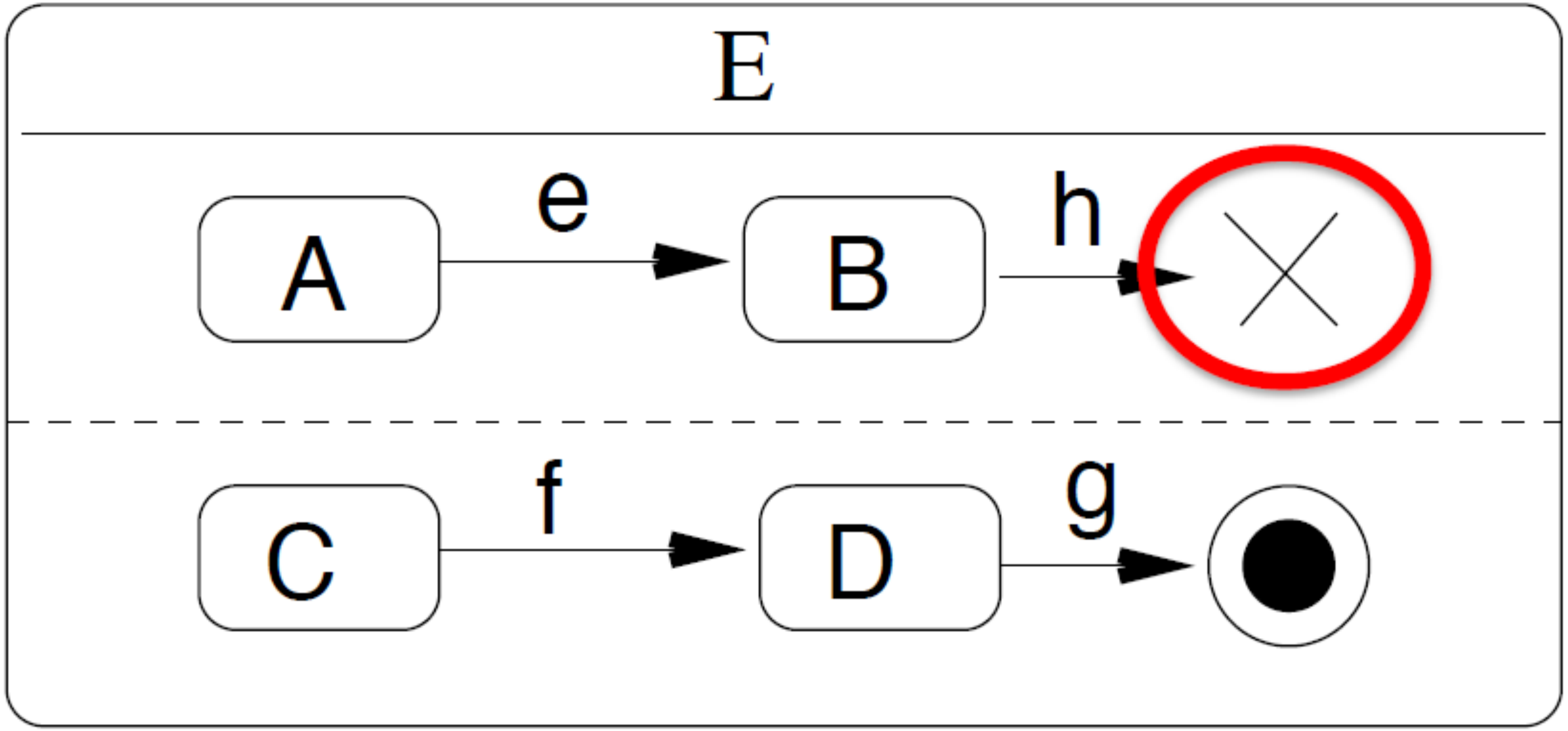
- The history of a system overrides the default start state.
- A default start state must be specified for a group that uses the history mechanism when the group is entered for the first time.
- The history of a system is only applied to the level in the hierarchy in which it appears.
- To apply the history mechanism at a lower level in the state hierarchy, it is necessary to use a history symbol at the lower levels.

Deep History

- An asterisk can be attached to the history symbol to indicate that the history of the system should be applied all the way down to the lowest level in the state hierarchy.



Termination



Time Event

A time event is the occurrence of a specific date/time or the passage of time.

- Absolute time:
  - at (12:12 pm, 12 Dec 2012)
- Relative time:
  - after (10 seconds since exit from state A)
  - after (10 seconds since x)
  - after (20 minutes) // since the transition’s source state was entered

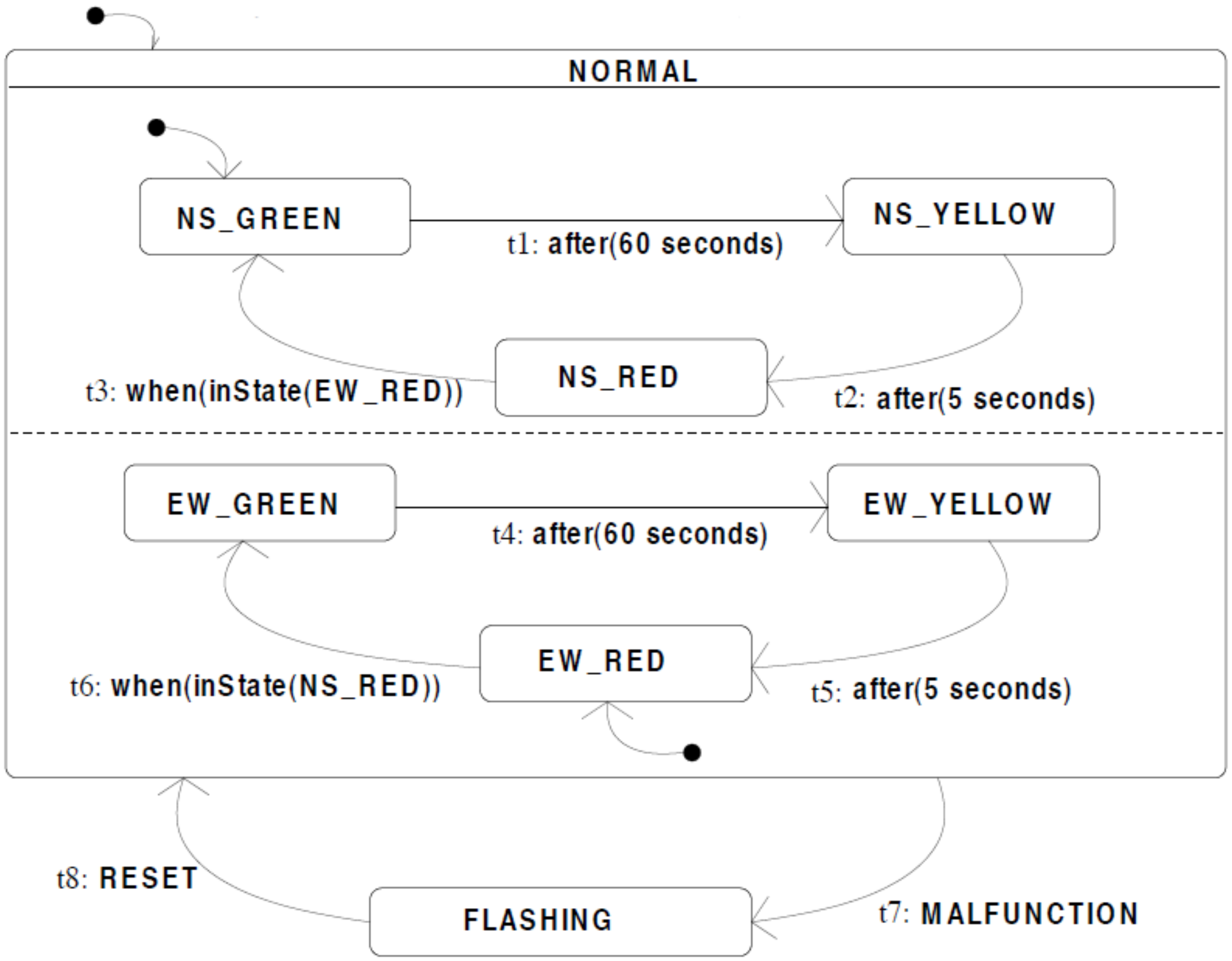
Change Events

A change event is the event of a condition becoming true.

- The event “occurs” when the condition changes value from *false* to *true*.
  - when (temperature > 100 degrees)
  - when (on)
- The event does not reoccur unless the value of the condition becomes *false* and then returns to *true*.

- when(X) vs. [X]

Traffic Light Example



Creating a Behavior Model

1. Identify input and output events
2. Think of a natural partitioning into states
   - *Activity states* – system performs activity or operation
   - *System modes* – use different states to distinguish between different reactions to an event
3. Consider the system’s behaviour for each state input.
4. Revise (using hierarchy, concurrency, and state events)
   - Use concurrency to separate orthogonal behaviour
   - Use hierarchy, and entry/exit actions, to abbreviate a common behavior

Behavioral Models Validation

- Avoid inconsistency: multiple transitions that leave the same state under the same event/conditions.
- Ensure completeness: specify a reaction for every possible input at a state.
  - If transitions are triggered by an event conditioned on some guard, what happens if the guard is false?
- Walkthrough: compare the behaviour of your state diagrams with the use-case scenarios.
  - All paths through the scenarios should be pathed in the state machines.

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

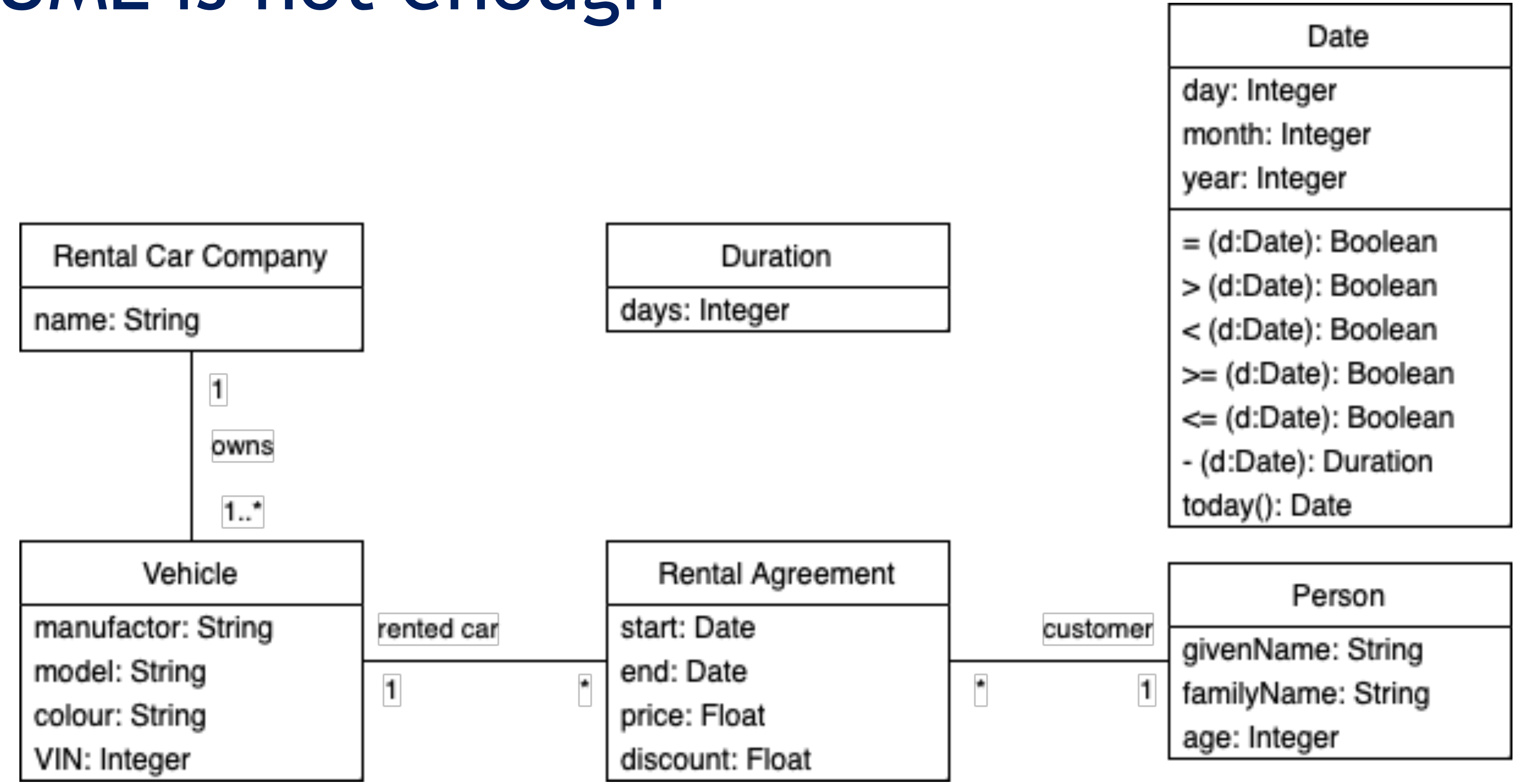
Behavioural Modelling

CS445/ECE 451/CS645

Software Requirements Specifications and Analysis

OCL

UML is not enough



Date

day: Integer
month: Integer
year: Integer

= (d:Date): Boolean
> (d:Date): Boolean
< (d:Date): Boolean
>= (d:Date): Boolean
<= (d:Date): Boolean
- (d:Date): Duration
today(): Date

Rental Car Company

name: String

Duration

days: Integer

1

owns

1..*

Vehicle

manufactor: String
model: String
colour: String
VIN: Integer

rented car

1

*

Rental Agreement

start: Date
end: Date
price: Float
discount: Float

customer

*

1

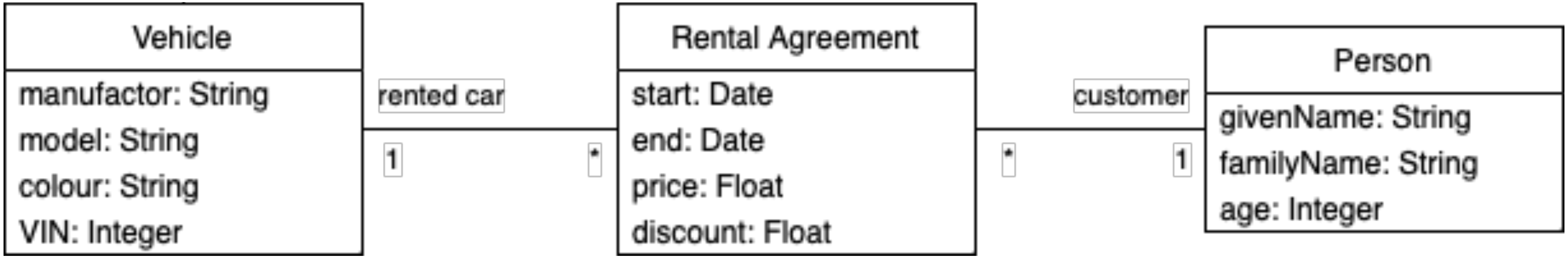
Person

givenName: String
familyName: String
age: Integer

Object Constraint Language

- Standardized by OMG
- Used to express constraints on UML models
- Not one of the UML notations
- Precise, yet easy to read
- It has language constructs for
  - relating classes that have no direct association
  - expressing queries over objects and collections of objects
- OCL constraints :
  - Are declarative; they specify what must be true, not what must be done
  - Have no side effects; do not change the state of the system
  - Have formal syntax and semantics; their interpretation is unambiguous

Navigation Across Associations



Consider the object p:Person

Expression	Value
p	
p.RentalAgreement	
p.RentalAgreement.rented_car	
p.RentalAgreement.rental_car.colour	

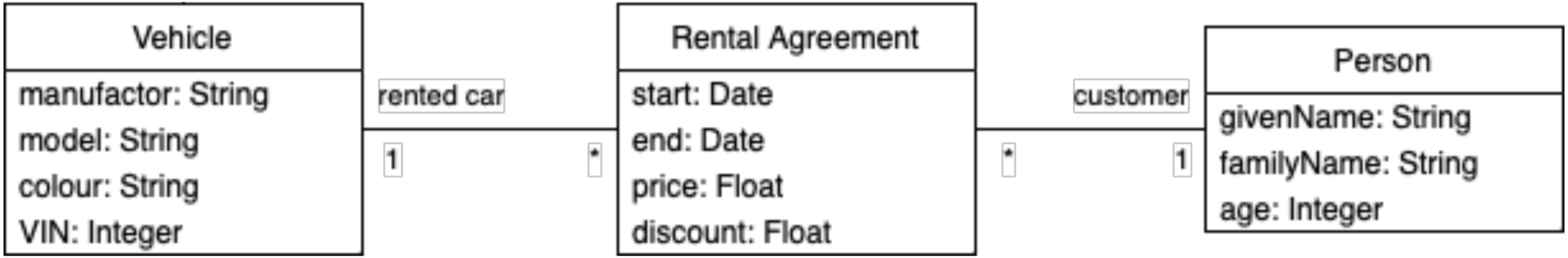
Collections

An OCL expression may be over a collection(set, bag, sequence) of objects:

context Person

self.RentalAgreement

returns set of rental agreements.



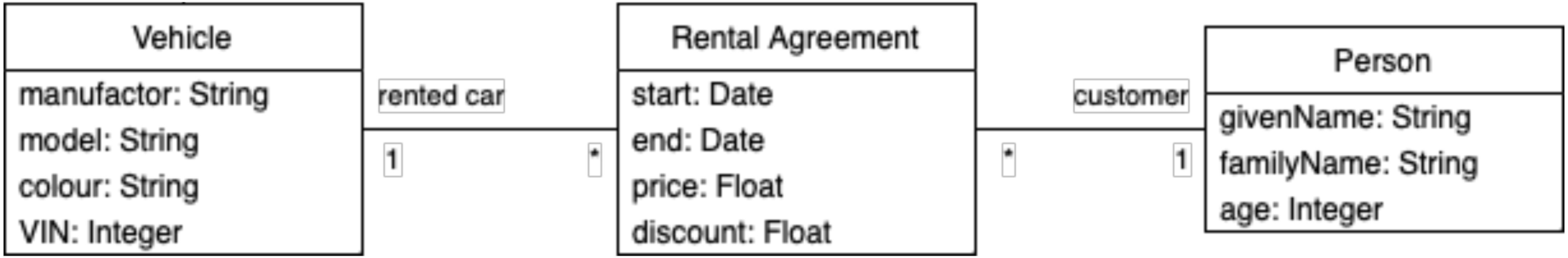
Collections

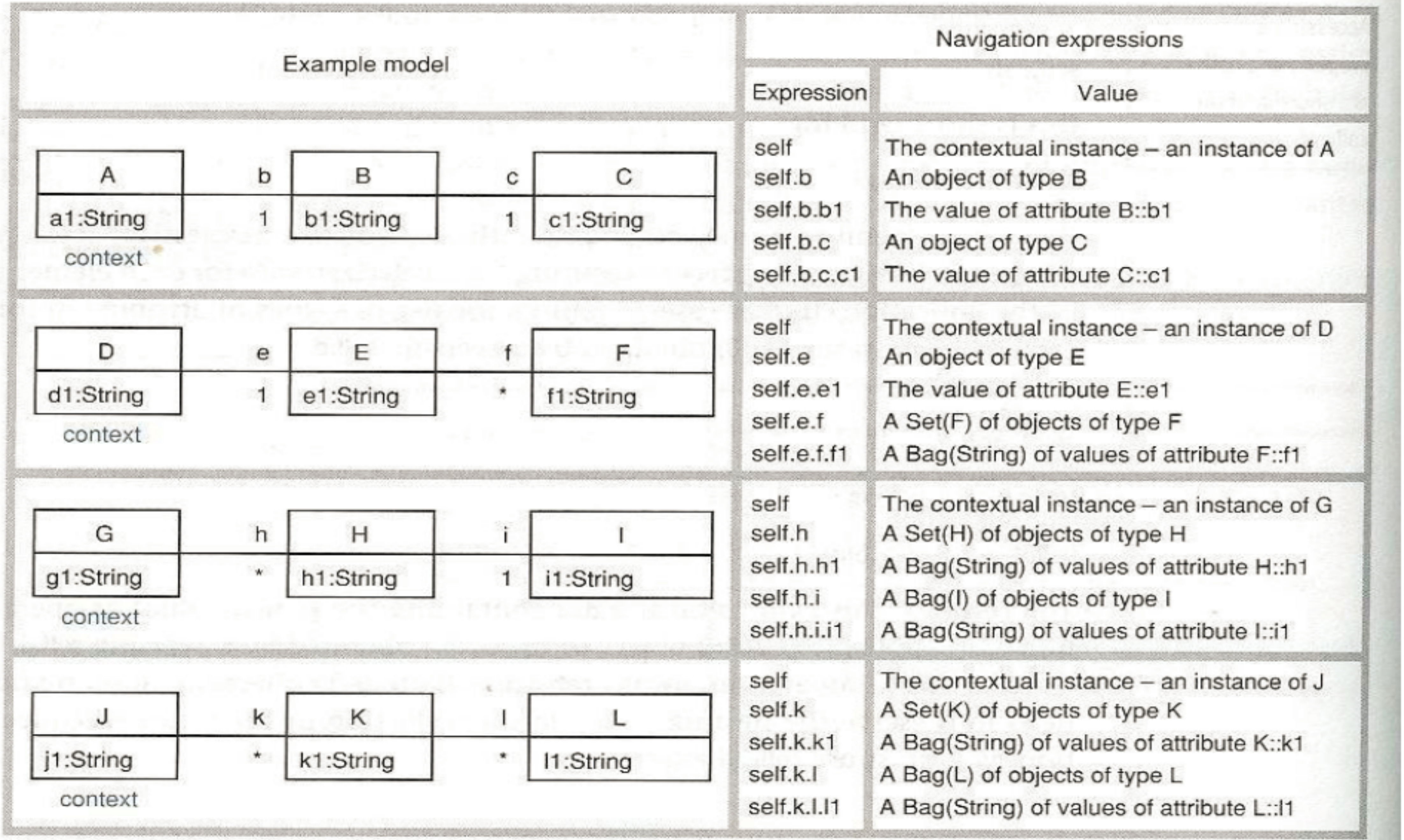
Defined properties of collections are denoted using the arrow notation (->), to distinguish from properties defined on model elements.

Note: size() is OCL operation that counts number of vehicles

context Person **inv:**

self.RentalAgreement.Vehicle->size() <=3.



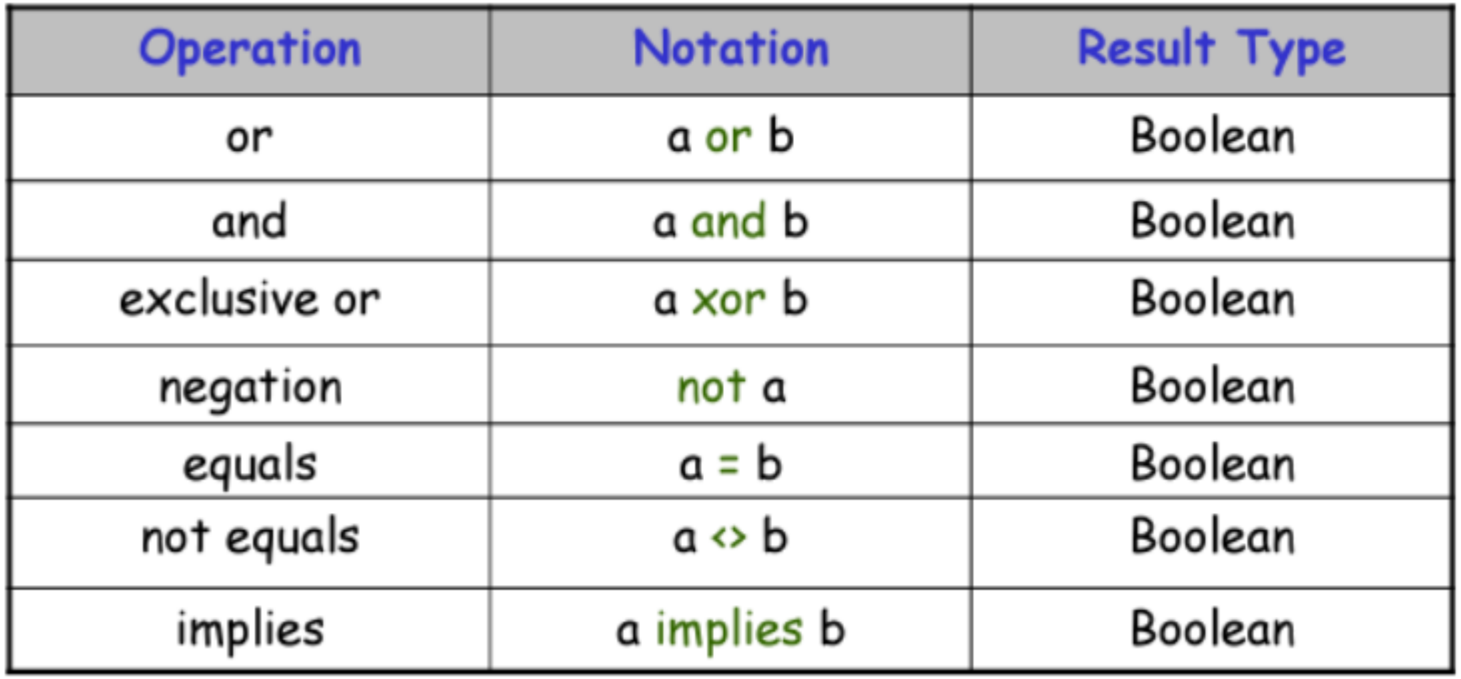


Example model	Navigation expressions	
	Expression	Value
A a1:String context b 1 B b1:String c 1 C c1:String	self	The contextual instance – an instance of A
	self.b	An object of type B
	self.b.b1	The value of attribute B::b1
	self.b.c	An object of type C
	self.b.c.c1	The value of attribute C::c1
D d1:String context e 1 E e1:String f * F f1:String	self	The contextual instance – an instance of D
	self.e	An object of type E
	self.e.e1	The value of attribute E::e1
	self.e.f	A Set(F) of objects of type F
	self.e.f.f1	A Bag(String) of values of attribute F::f1
G g1:String context h * H h1:String i 1 I i1:String	self	The contextual instance – an instance of G
	self.h	A Set(H) of objects of type H
	self.h.h1	A Bag(String) of values of attribute H::h1
	self.h.i	A Bag(I) of objects of type I
	self.h.i.i1	A Bag(String) of values of attribute I::i1
J j1:String context k * K k1:String l * L l1:String	self	The contextual instance – an instance of J
	self.k	A Set(K) of objects of type K
	self.k.k1	A Bag(String) of values of attribute K::k1
	self.k.l	A Bag(L) of objects of type L
	self.k.l.l1	A Bag(String) of values of attribute L::l1

Figure 25.12

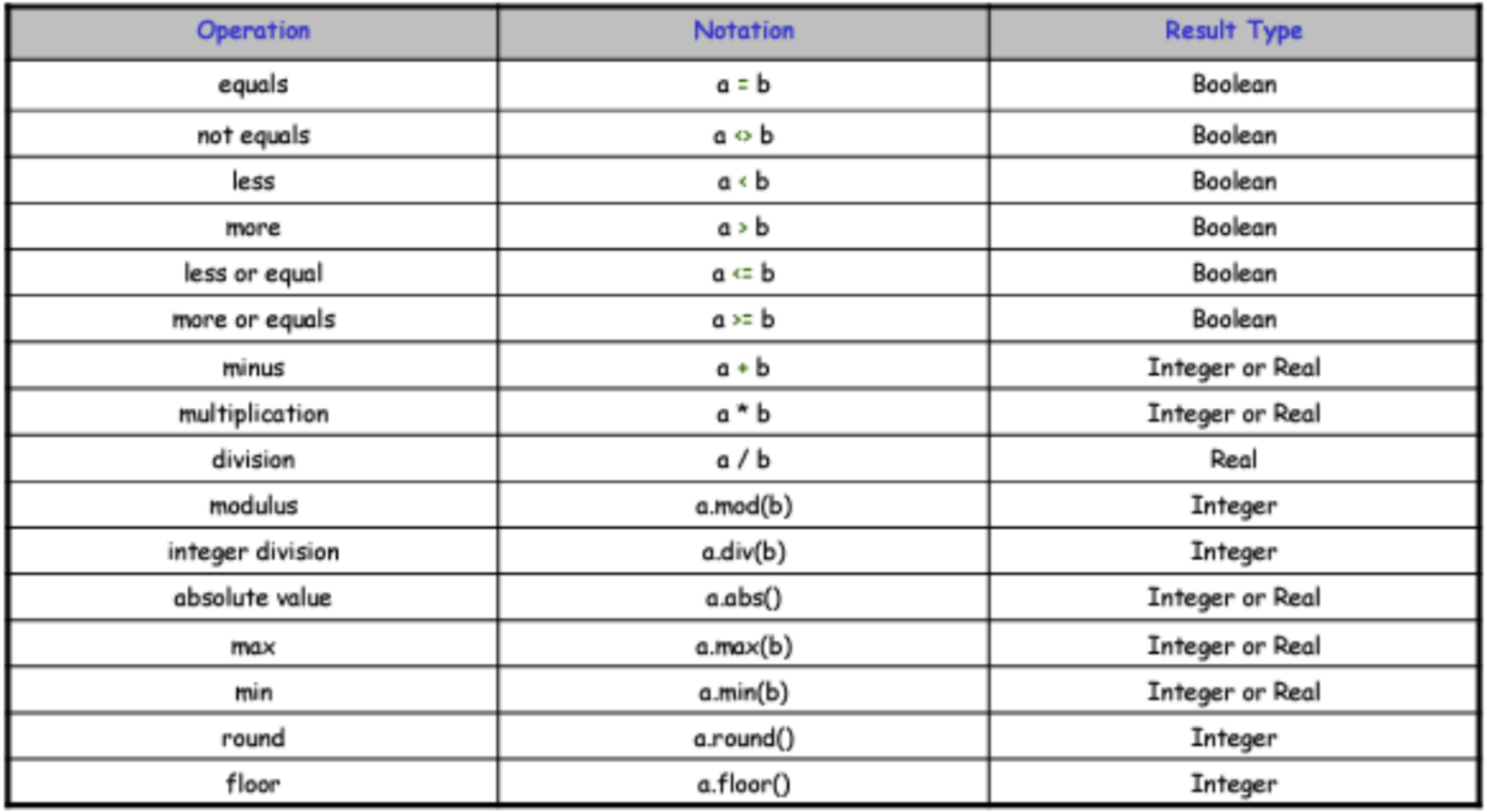
Arlow and Neustadt, *UML 2 and the Unified Process*

Basic Operators



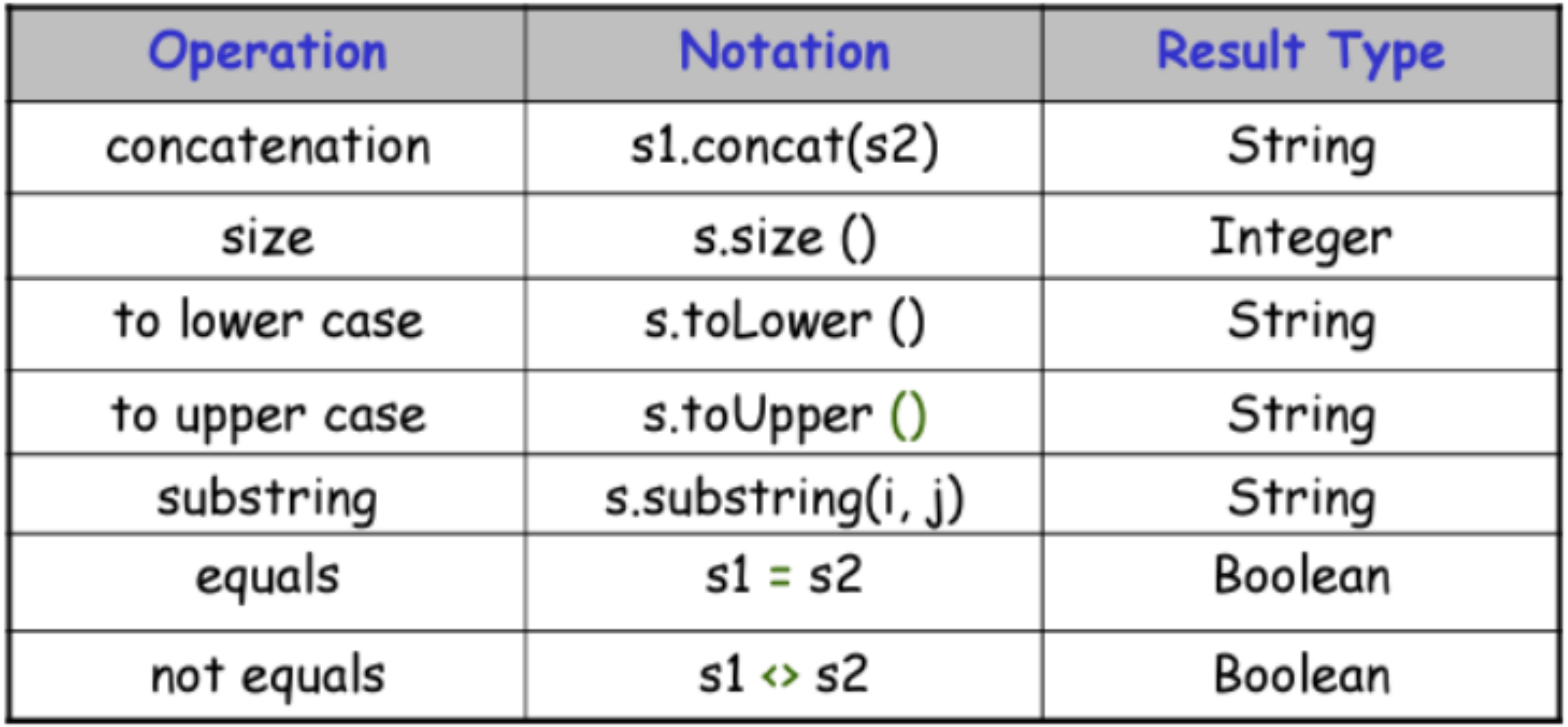
Operation	Notation	Result Type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean

Basic Operators



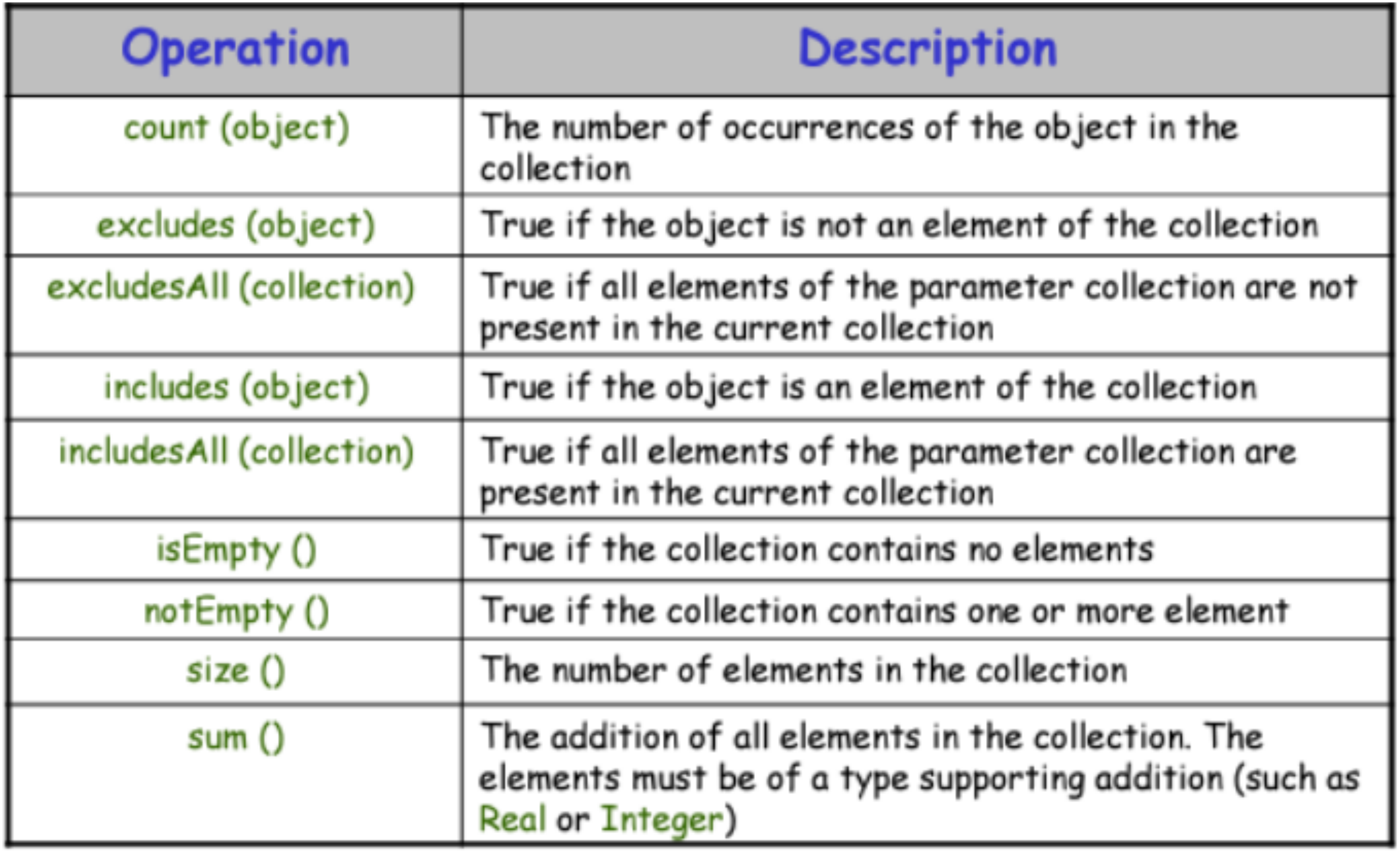
Operation	Notation	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equals	a >= b	Boolean
minus	a + b	Integer or Real
multiplication	a * b	Integer or Real
division	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs()	Integer or Real
max	a.max(b)	Integer or Real
min	a.min(b)	Integer or Real
round	a.round()	Integer
floor	a.floor()	Integer

Basic Operators



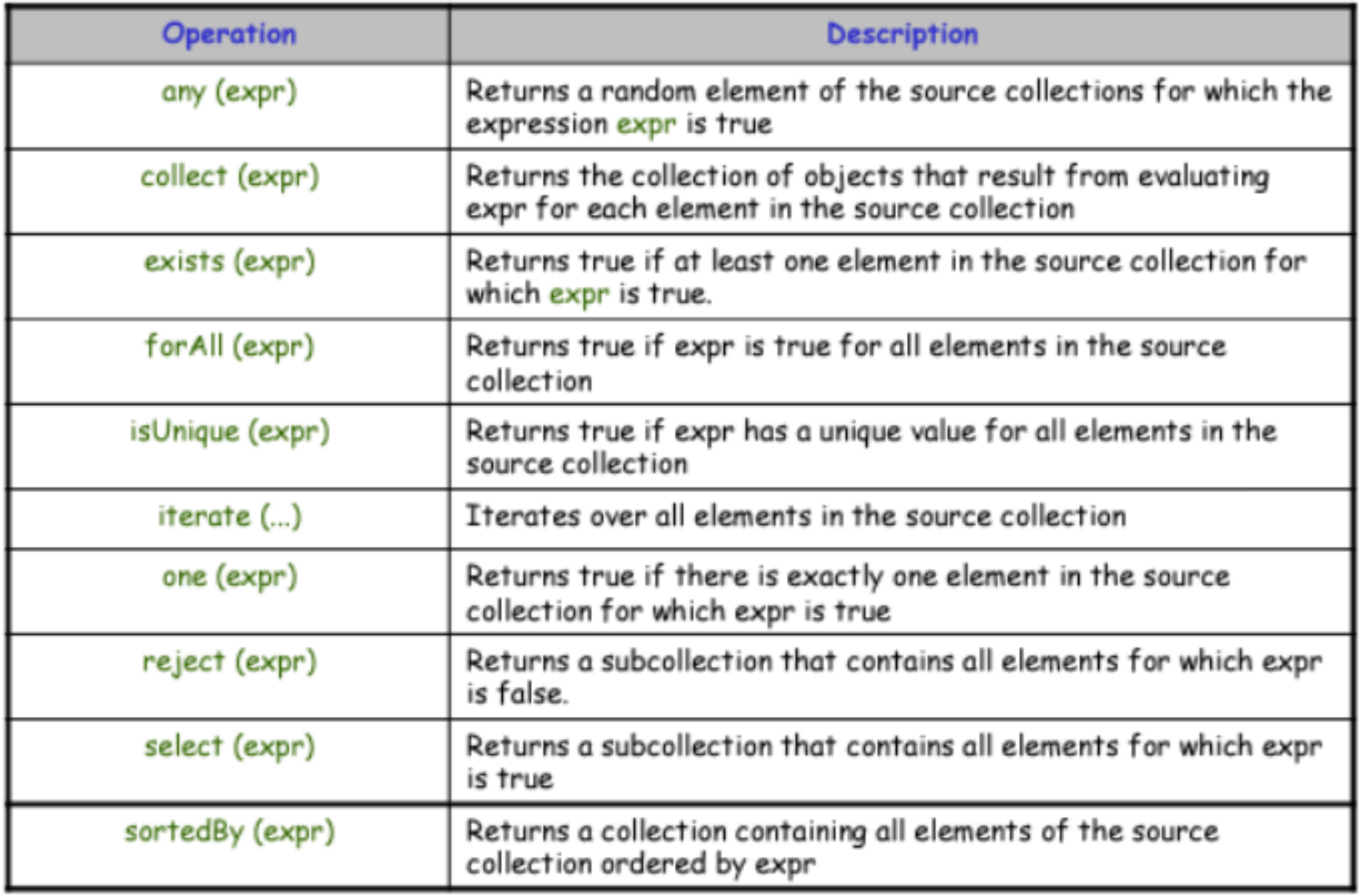
Operation	Notation	Result Type
concatenation	s1.concat(s2)	String
size	s.size ()	Integer
to lower case	s.toLower ()	String
to upper case	s.toUpper ()	String
substring	s.substring(i, j)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

Basic Operators



Operation	Description
count (object)	The number of occurrences of the object in the collection
excludes (object)	True if the object is not an element of the collection
excludesAll (collection)	True if all elements of the parameter collection are not present in the current collection
includes (object)	True if the object is an element of the collection
includesAll (collection)	True if all elements of the parameter collection are present in the current collection
isEmpty ()	True if the collection contains no elements
notEmpty ()	True if the collection contains one or more element
size ()	The number of elements in the collection
sum ()	The addition of all elements in the collection. The elements must be of a type supporting addition (such as Real or Integer)

Basic Operators



Operation	Description
any (expr)	Returns a random element of the source collections for which the expression expr is true
collect (expr)	Returns the collection of objects that result from evaluating expr for each element in the source collection
exists (expr)	Returns true if at least one element in the source collection for which expr is true.
forAll (expr)	Returns true if expr is true for all elements in the source collection
isUnique (expr)	Returns true if expr has a unique value for all elements in the source collection
iterate (...)	Iterates over all elements in the source collection
one (expr)	Returns true if there is exactly one element in the source collection for which expr is true
reject (expr)	Returns a subcollection that contains all elements for which expr is false.
select (expr)	Returns a subcollection that contains all elements for which expr is true
sortedBy (expr)	Returns a collection containing all elements of the source collection ordered by expr

Filtering Operators

To extract specific elements from an existing collection based on the value of an expression.

- select: returns the elements that satisfy the given expression
- reject: returns the elements that falsify the given expression

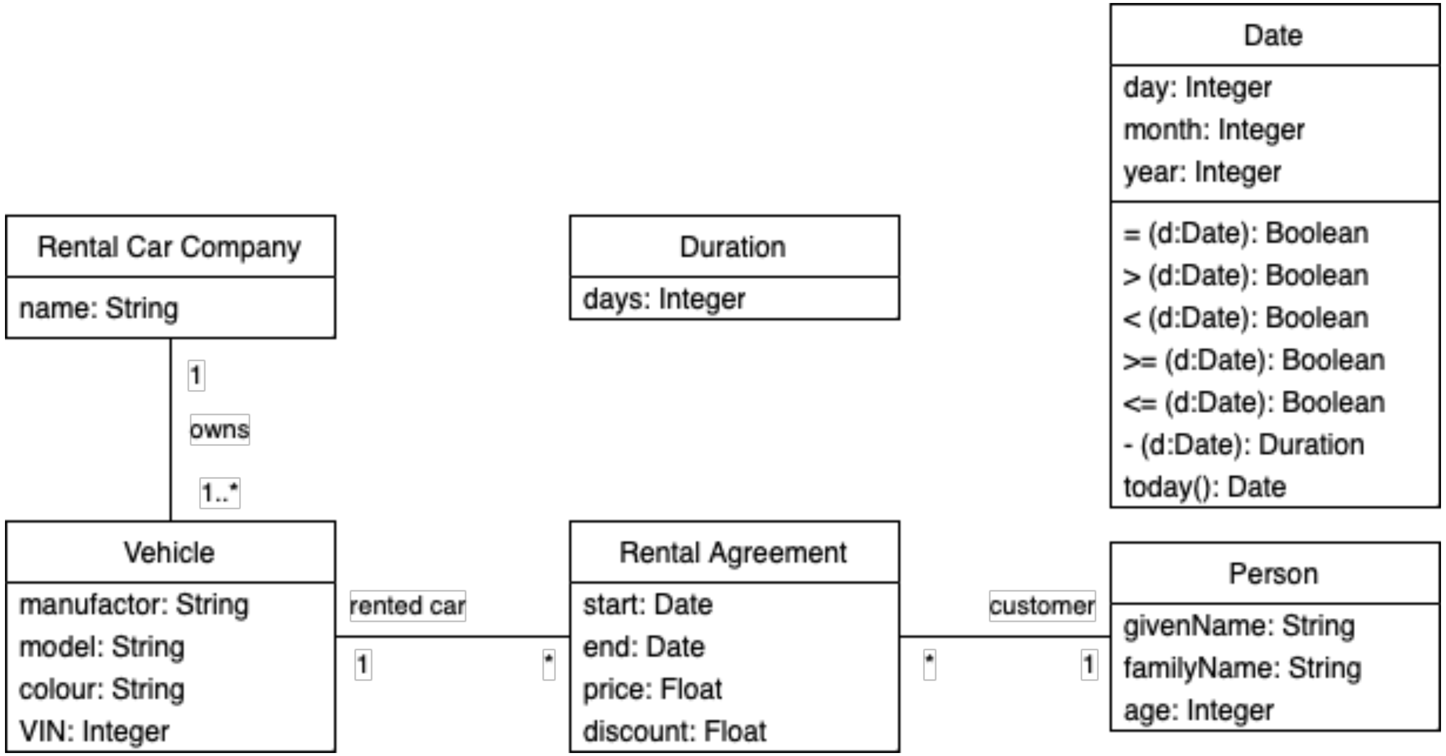
Example:

Rental car companies never own red cars.

Example

Rental car companies never own red cars.

context RentalCarCompany **inv:**
self.owns->select(colour="red")->isEmpty()



Quantification

exists: Boolean operation asserts that at least one element in a collection satisfies some expression.

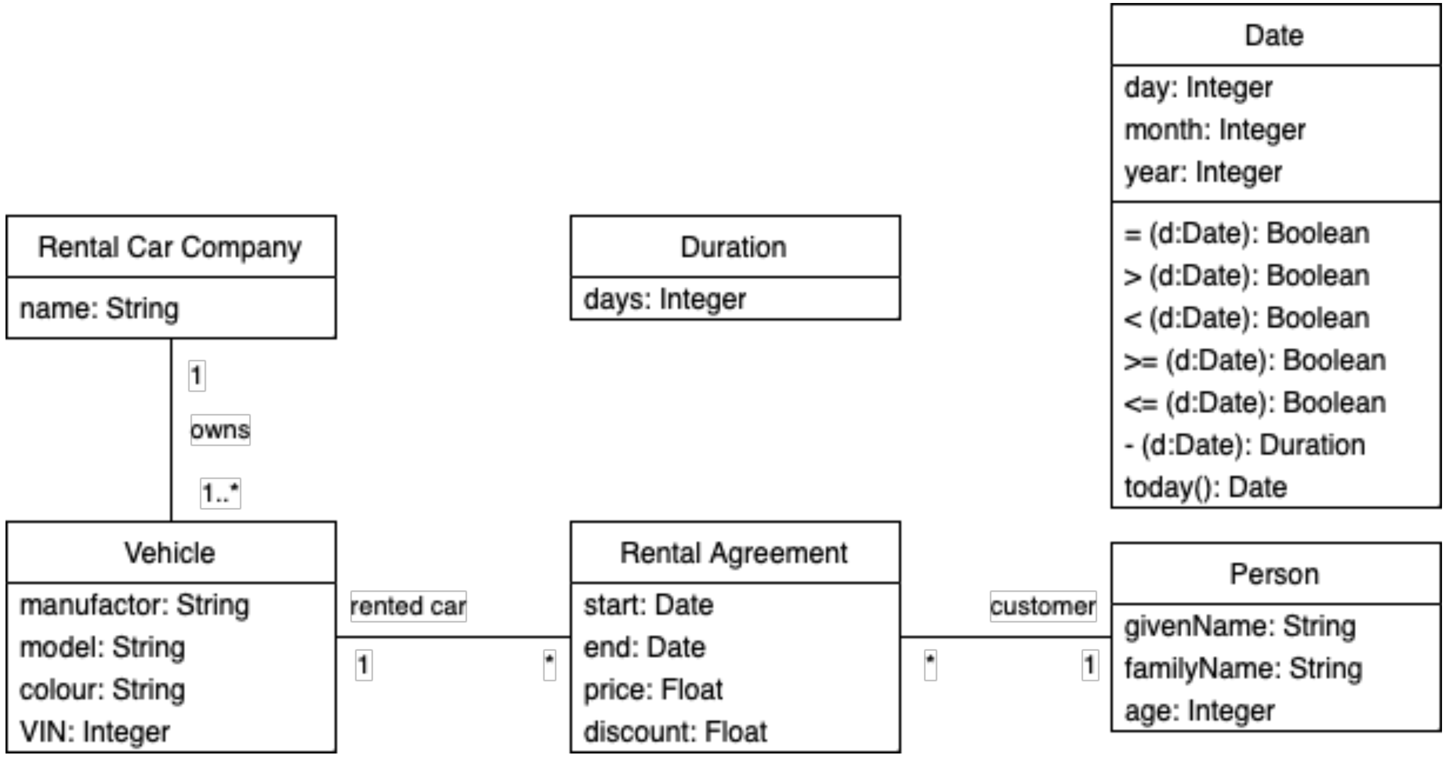
Example:

Every customer rents at least one black car.

Example

Every customer rents at least one black car.

context Person **inv:**
self.RentalAgreement.Vehicle->exists(colour=“black”)



Date
day: Integer
month: Integer
year: Integer
= (d:Date): Boolean
> (d:Date): Boolean
< (d:Date): Boolean
>= (d:Date): Boolean
<= (d:Date): Boolean
- (d:Date): Duration
today(): Date

Rental Car Company
name: String

Duration
days: Integer

1
owns
1..*

Vehicle
manufactor: String
model: String
colour: String
VIN: Integer

rented car
1
*

Rental Agreement
start: Date
end: Date
price: Float
discount: Float

customer
*
1

Person
givenName: String
familyName: String
age: Integer

Quantification

forAll: Boolean operation used to assert that all set members satisfy a given expression.

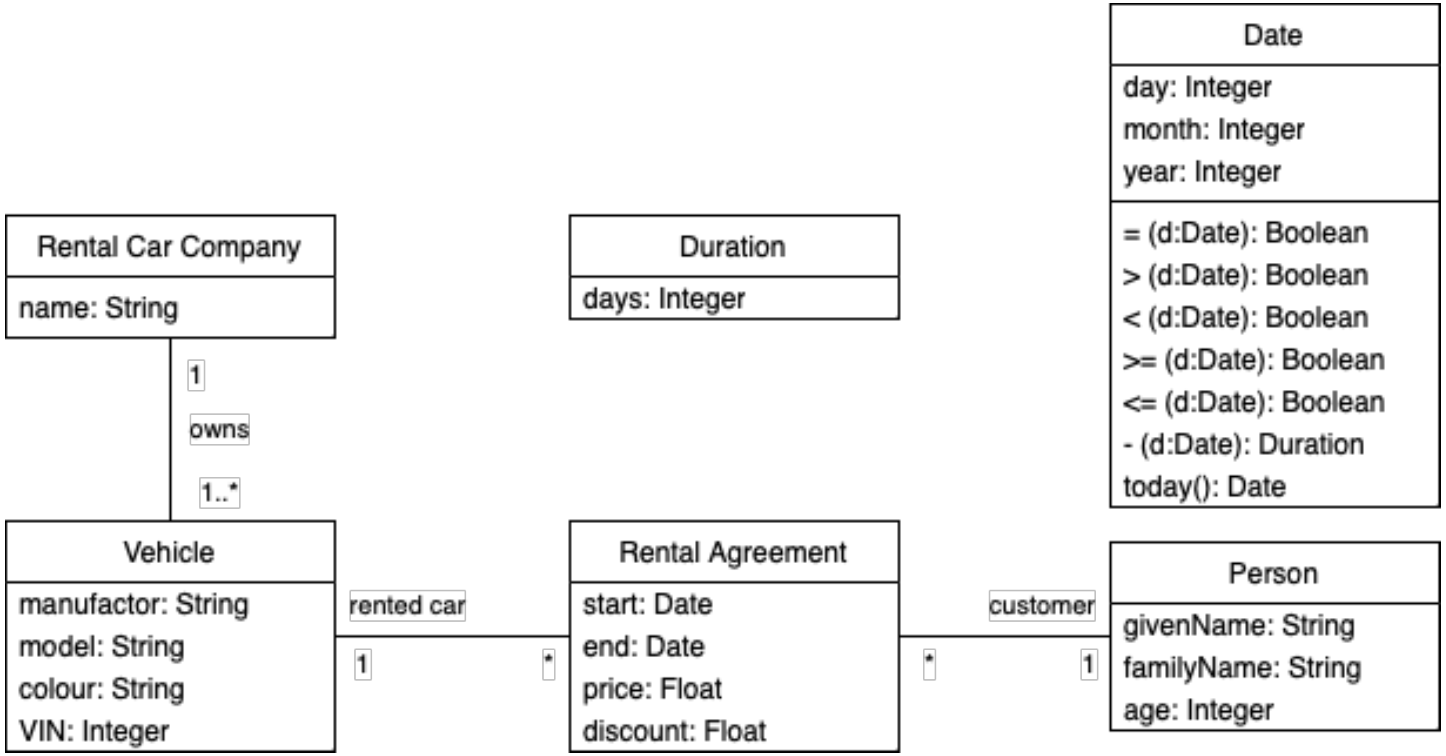
Examples:

- All cars are rented.
- No car is rented more than once each day.
- All rental cars are white.

Example

All cars are rented.

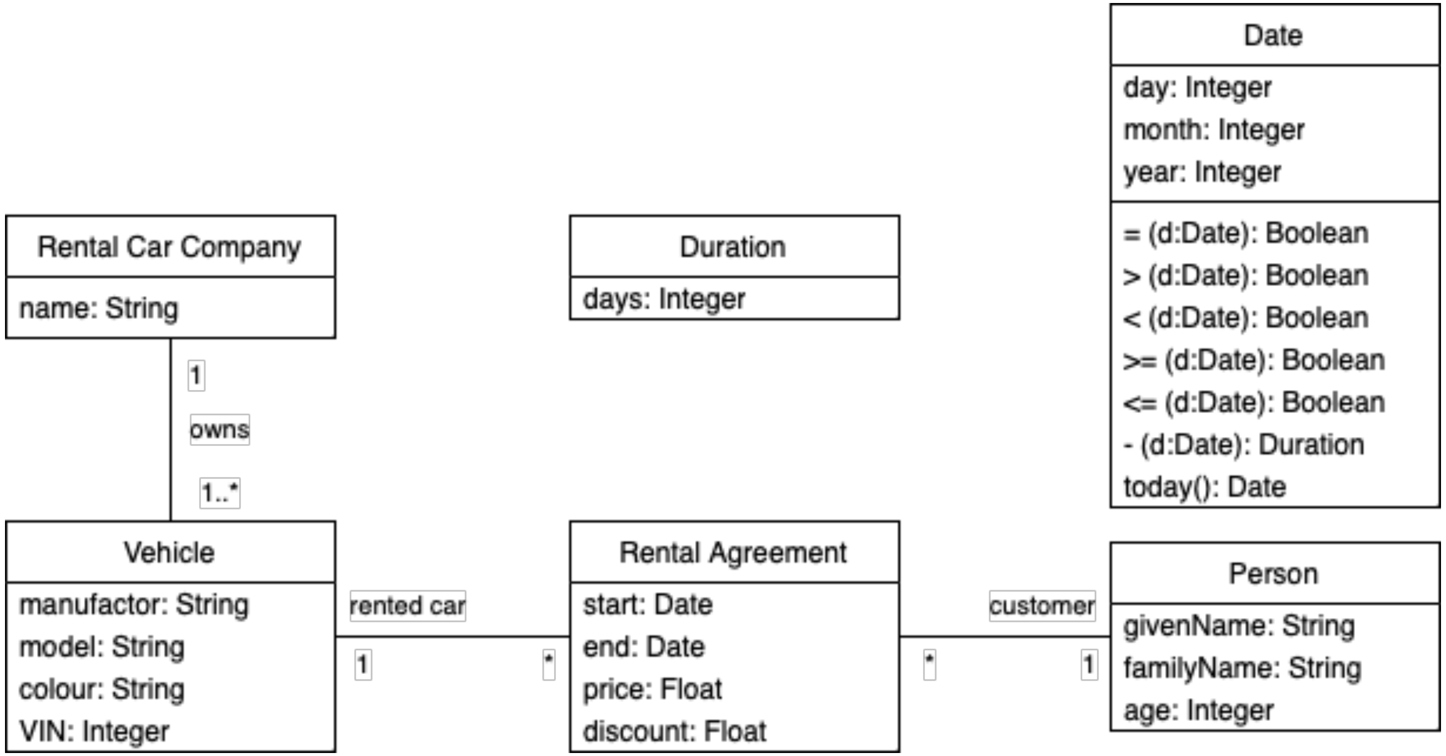
context Vehicle **inv:**
self.RentalAgreement->forAll(r:RentalAgreement |
r.start <= Date.today() and Date.today() <= r.end)



Example

No car is rented more than once each day.

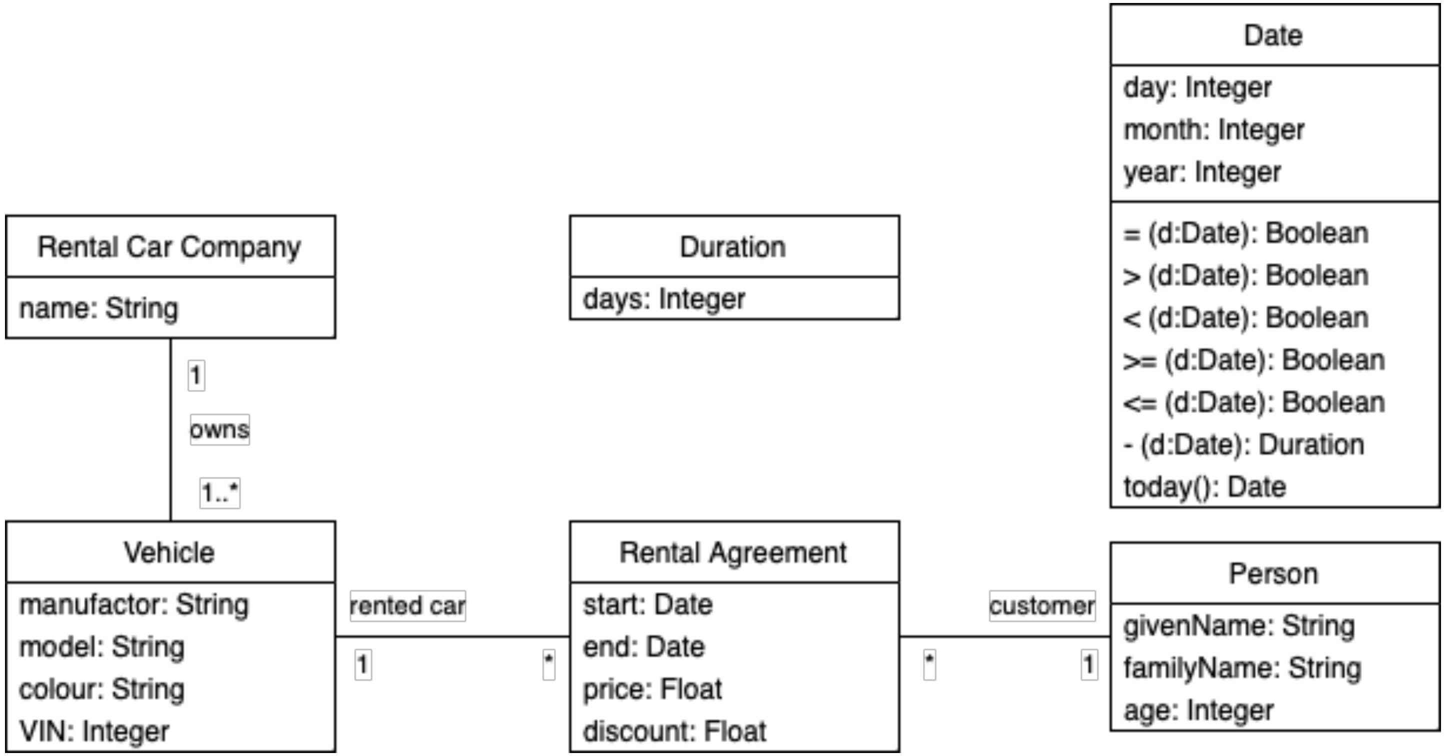
```
context Vehicle inv:
self.RentalAgreement->forAll(r1,r2: RentalAgreement |
   (r1 <> r2) implies (r1.start > r2.end or r2.start > r1.end))
```



Example

All rental cars are white.

context RentalCarCompany **inv:**
self.owns->forAll(colour="white")



Exercise

- Every person is aged 18 or older.

```
context Person inv:
self.age >= 18
```

- No rental agreements are made whose price is less than $100.

```
context RentalAgreement inv:
self.price >= 100
```

Exercise

- All rental agreements started in or after the year 2000.

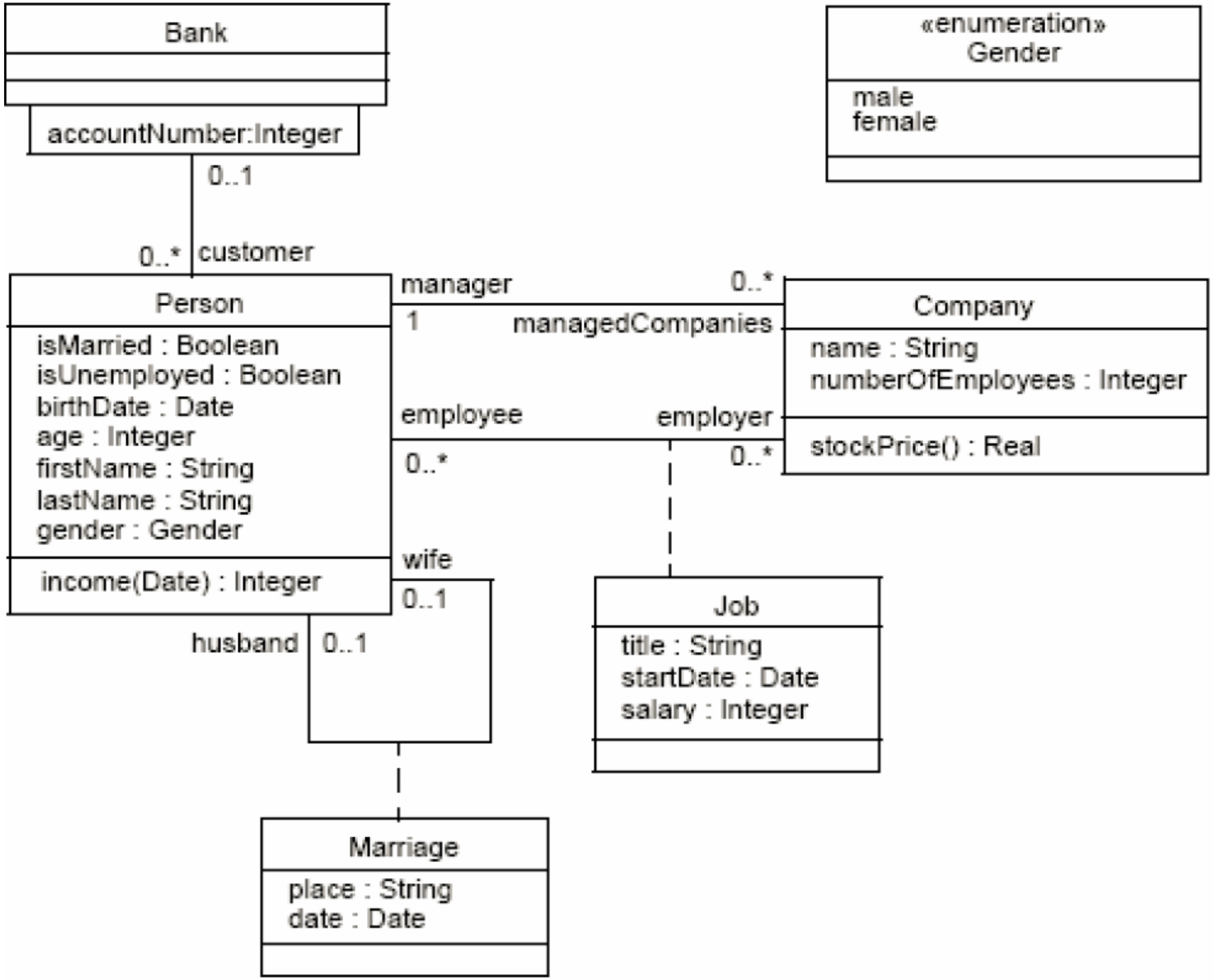
```
context RentalAgreement inv:
self.start.year >= 2000
```

- People aged 65 or older have a 10% discount in rental agreements created in or after 2024.

```
context Person inv:
self.age >= 65 implies
(self.RentalAgreement->select(start.year >= 2024)->forAll(discount = 10))
```

```
context RentalAgreement inv:
self.customer.age >= 65 implies
 ((self.start.year >= 2024) implies (self.discount = 10))
```

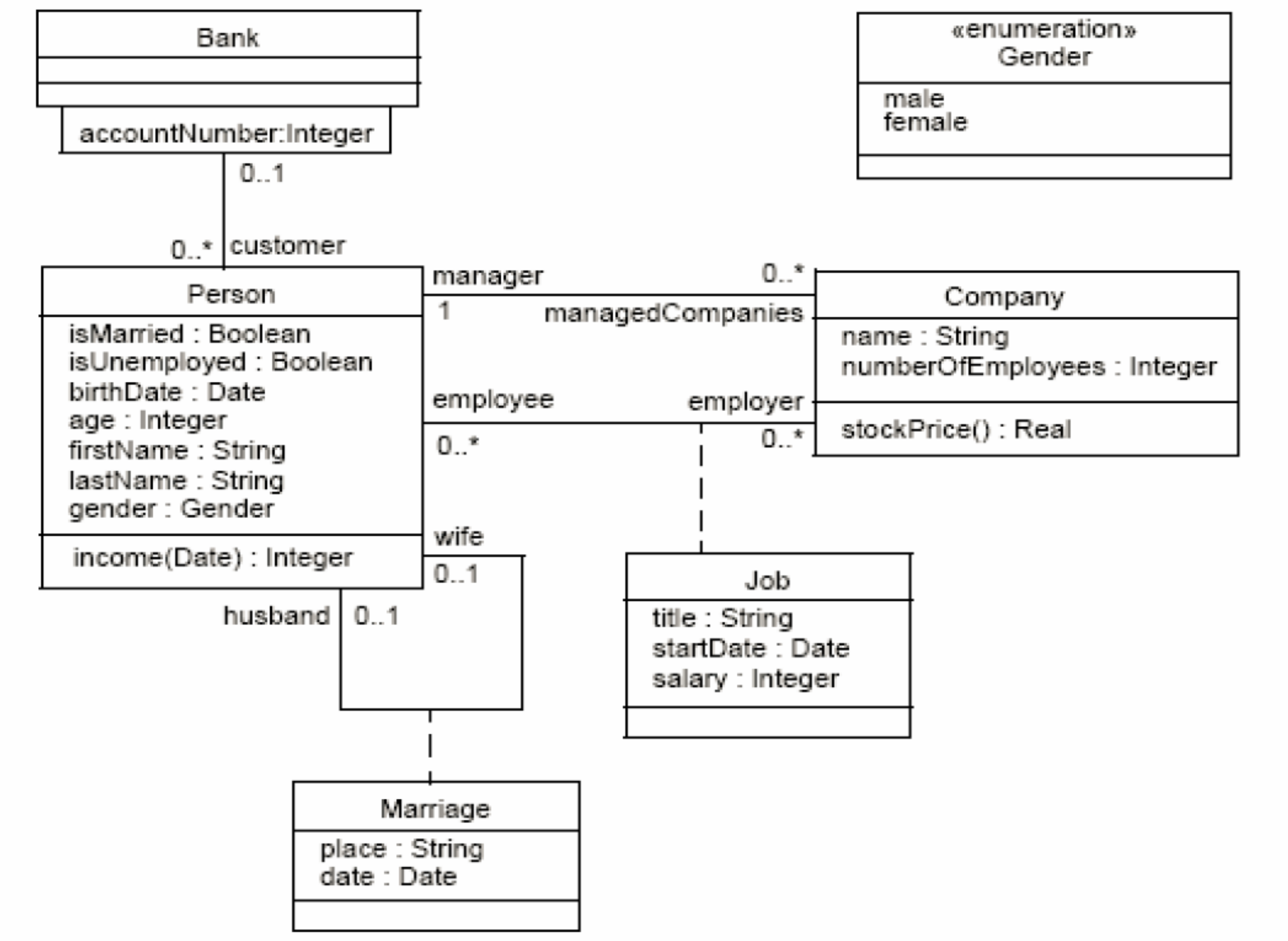
Exercise



Bank
accountNumber:Integer
0..1
0..*
customer
«enumeration»
Gender
male
female
Person
isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
gender : Gender
income(Date) : Integer
manager
1
0..*
managedCompanies
Company
name : String
numberOfEmployees : Integer
stockPrice() : Real
employee
0..*
employer
0..*
wife
0..1
husband
0..1
Job
title : String
startDate : Date
salary : Integer
Marriage
place : String
date : Date

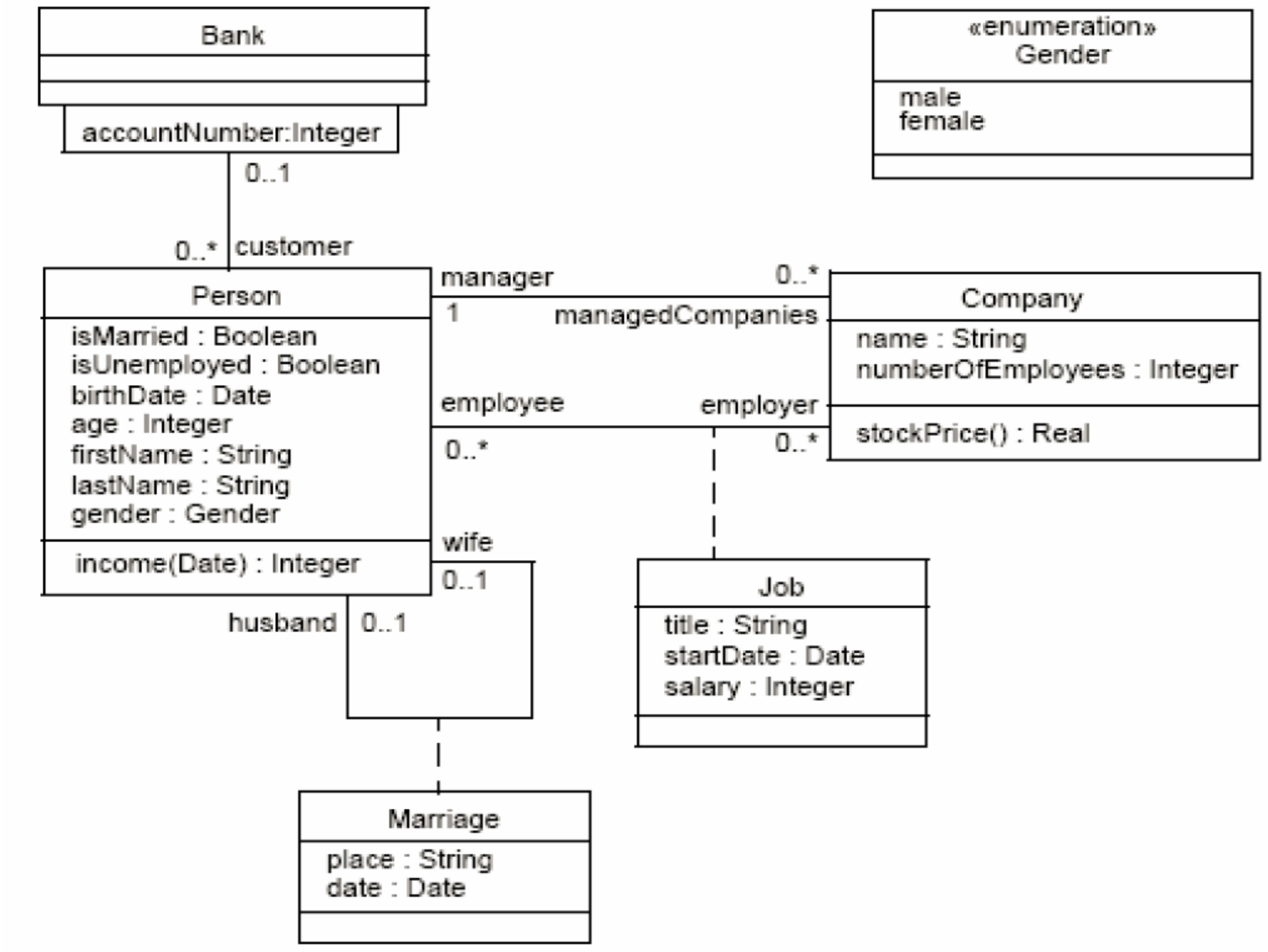
Married people are of age >= 18

context Person **inv:**



A company has at most 50 employees

context Company **inv:**



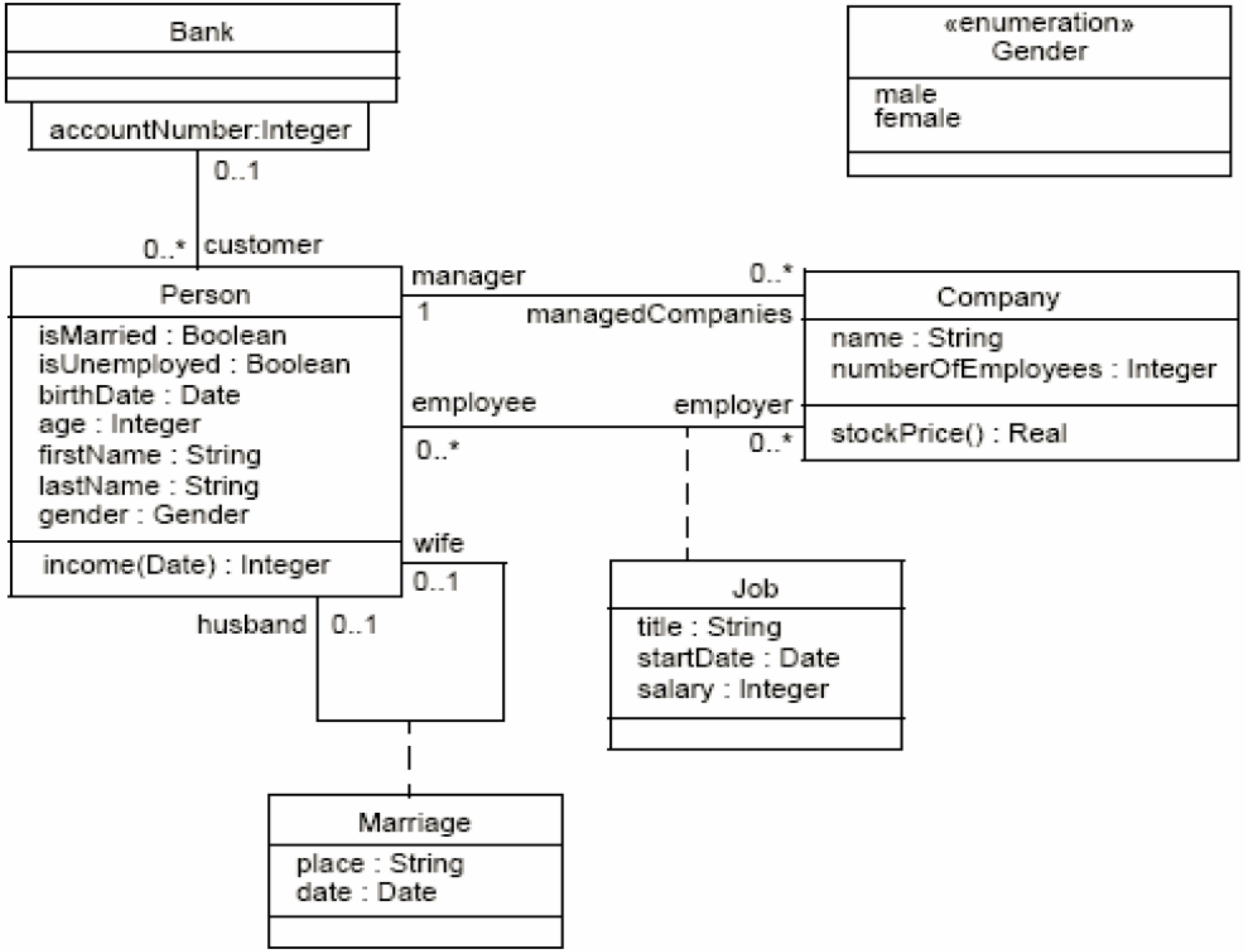
Note

1) context Company **inv:**

self.manager.age > 40

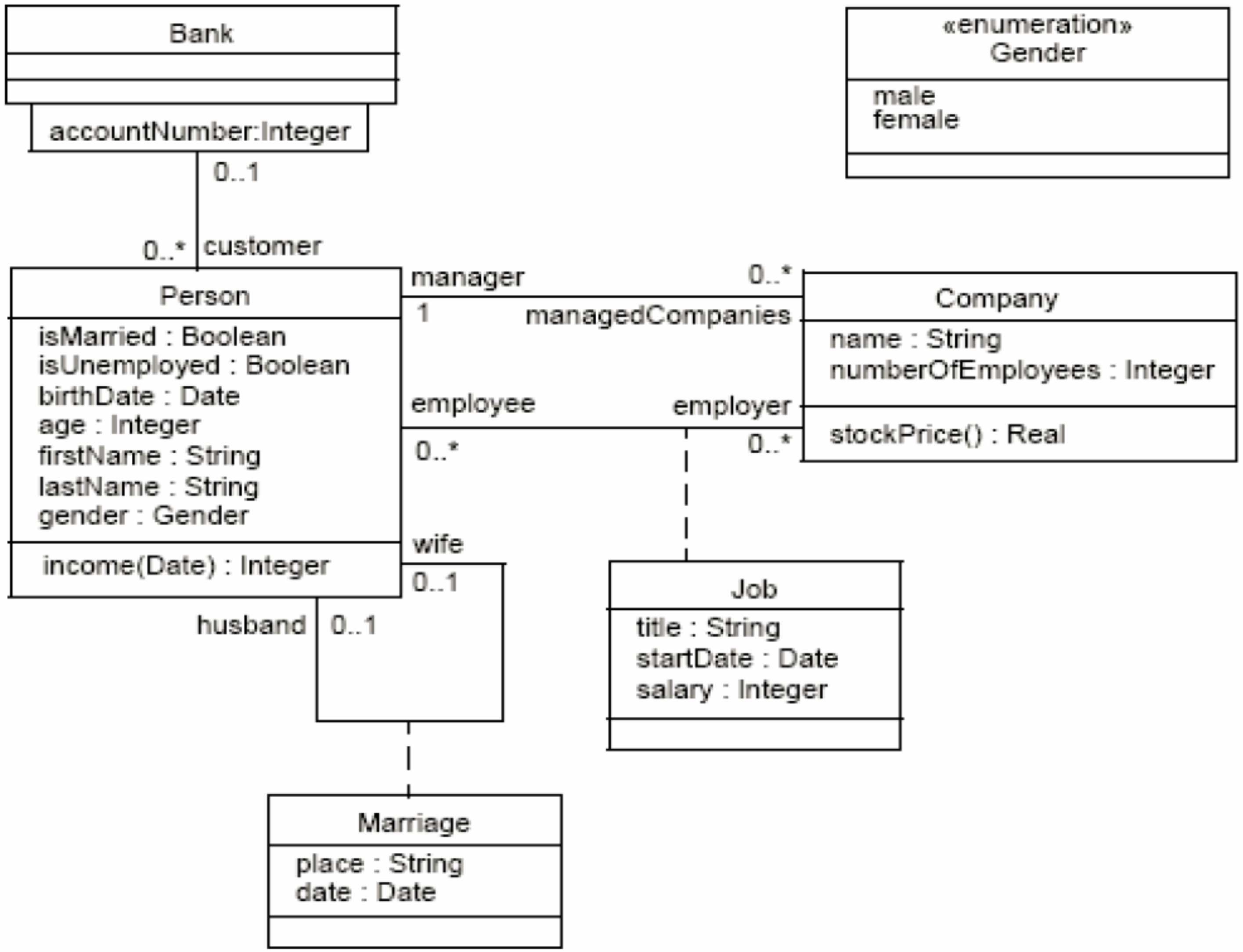
2) context Person **inv:**

self.wife->notEmpty() implies self.wife.age <= 65



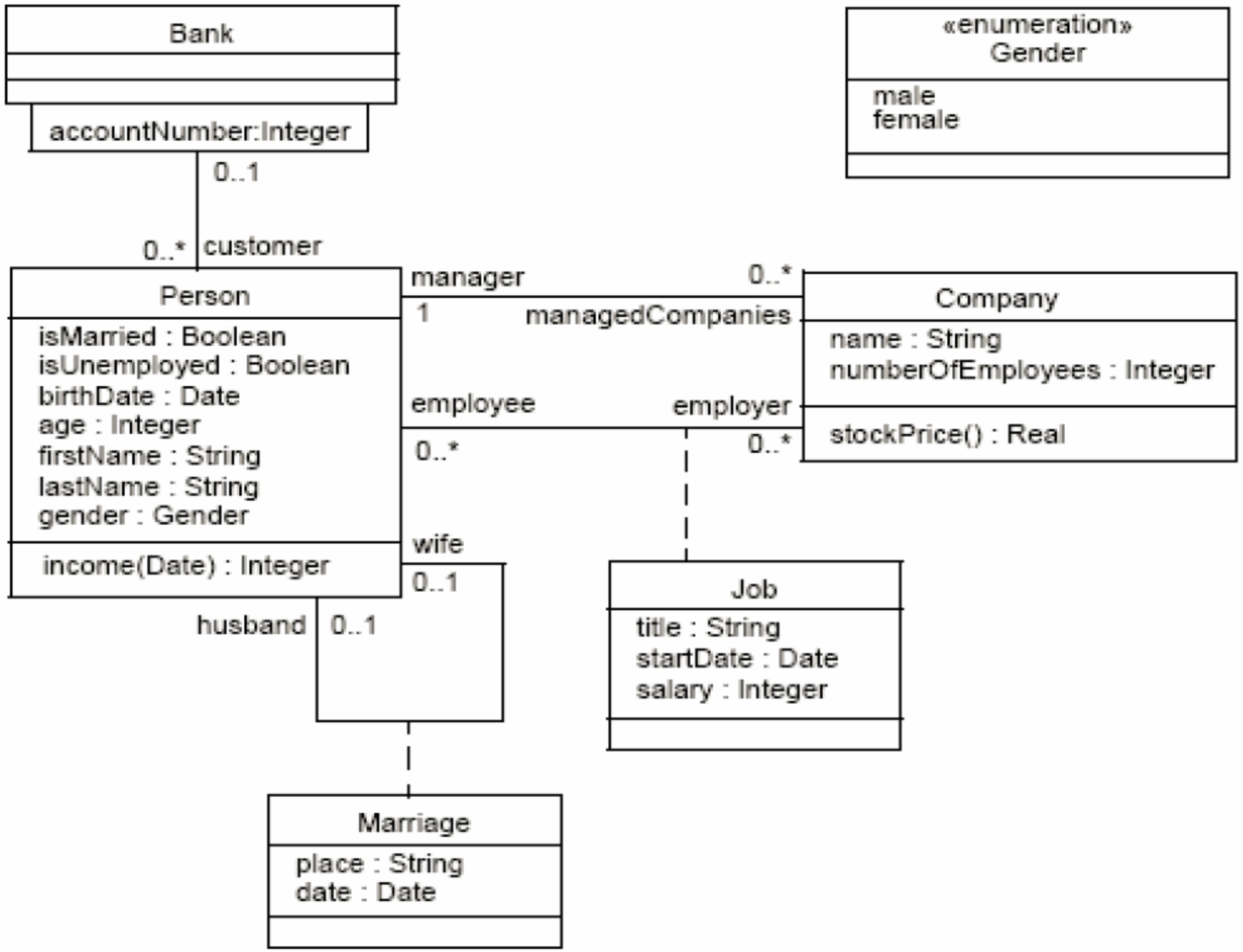
All instances of Person in a Bank have unique first names

context Bank **inv:**



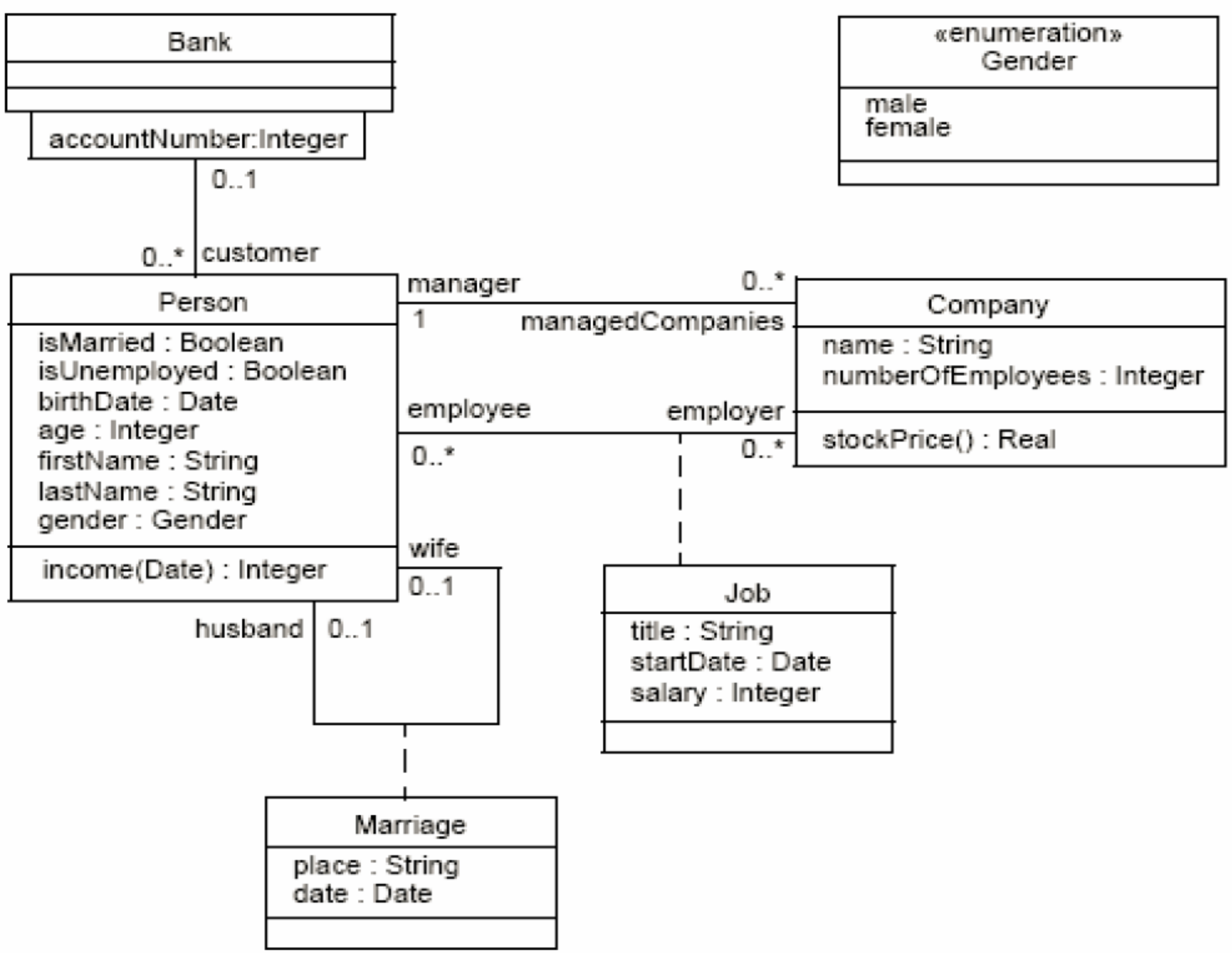
There is at least one employee above 50

context Company **inv:**



All employees are married

context Company **inv:**



Object Constraint Language

The OCL enables one to write formal expressions and constraints on object-oriented models.

Types of OCL expressions/constraints include

- Invariant properties about objects, links, and attribute values
- Initial variable or attribute values
- Pre/Postconditions of functions
- Guard conditions and assignment expressions in State Machine diagrams

Broken Constraints

Note that constraints simply state what ought to be true. If the execution of the system leads to an object model for which a constraint is not true, we say that the constraint is broken or violated.

Nothing in a constraint specification says how to recover from a broken constraint.

OCL Tools

Several tools support OCL from both universities and industry. These tools range from

- Parsers and type checkers
- Evaluators that can check an OCL expression against all instances of a UML class model
- Debuggers that step through an OCL expression and check each subsection (to locate faulty subexpression)
- Code generators that translate OCL expressions into run-time assertions

Summary

Object Constraint Language (OCL) expresses domain assumptions as constraints on the domain model

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

OCL

Exercise

Model the following constraints as OCL expressions over the Flix.net domain model.

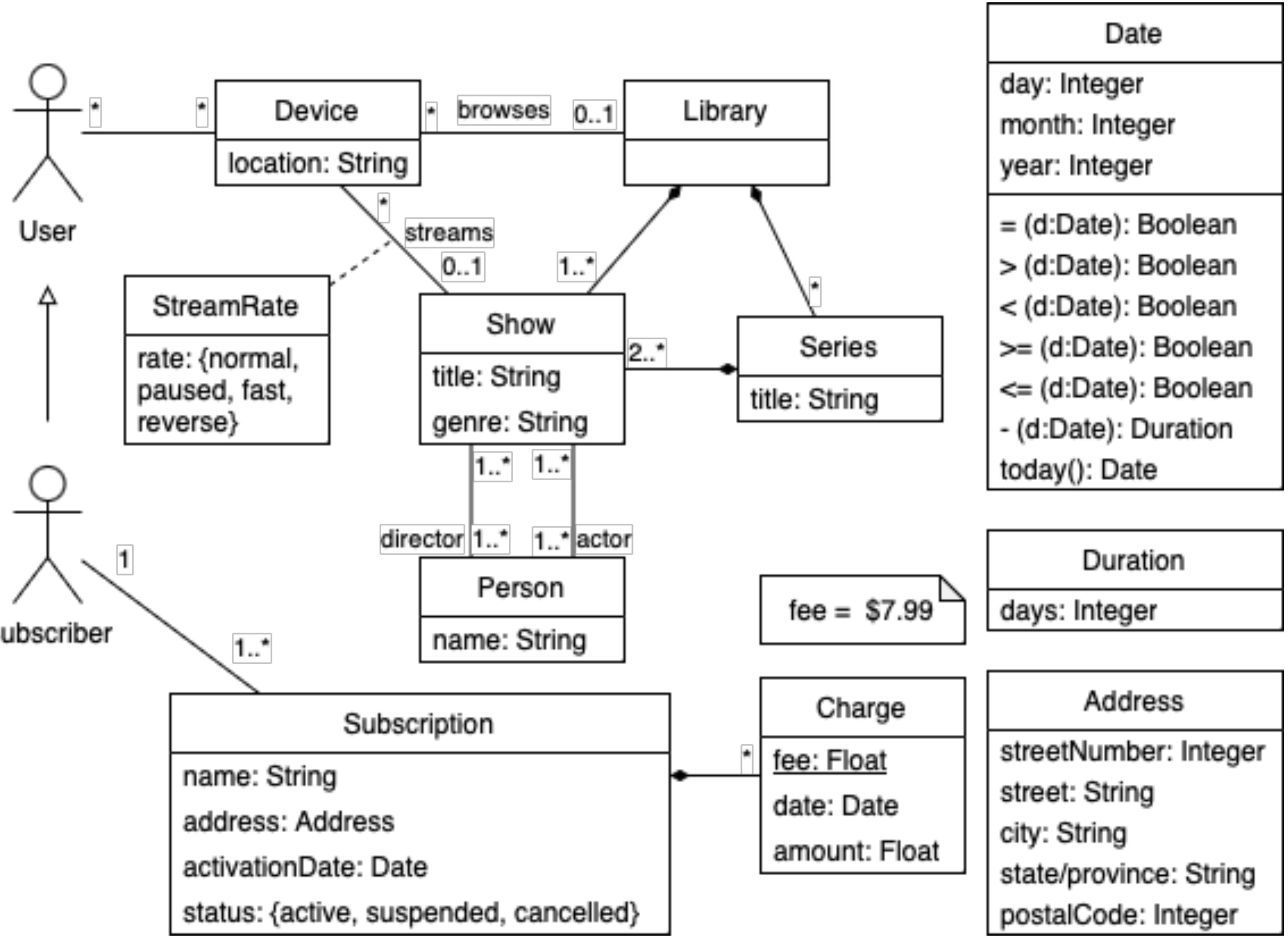
a) Every charge of the subscription fee is $7.99.

b) Every subscription fee that is charged to a subscription is charged on the monthly anniversary of the day on which the subscription was activated.

c) One must have an active subscription to stream videos.

d) Videos can be streamed only to devices (i.e., TVs, computers) within Canada.

Exercise


User
*
*
Device
location: String
*
browses
0..1
Library
*
streams
0..1
1..*
*
StreamRate
rate: {normal, paused, fast, reverse}
Show
title: String
genre: String
2..*
Series
title: String
1..*
1..*
director
1..*
1..*
actor
Person
name: String
Subscriber
1
1..*
fee = $7.99
Subscription
name: String
address: Address
activationDate: Date
status: {active, suspended, cancelled}
*
Charge
fee: Float
date: Date
amount: Float
Date
day: Integer
month: Integer
year: Integer
= (d:Date): Boolean
> (d:Date): Boolean
< (d:Date): Boolean
>= (d:Date): Boolean
<= (d:Date): Boolean
- (d:Date): Duration
today(): Date
Duration
days: Integer
Address
streetNumber: Integer
street: String
city: String
state/province: String
postalCode: Integer

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Functional Modelling

So far we learned how to model the system by:

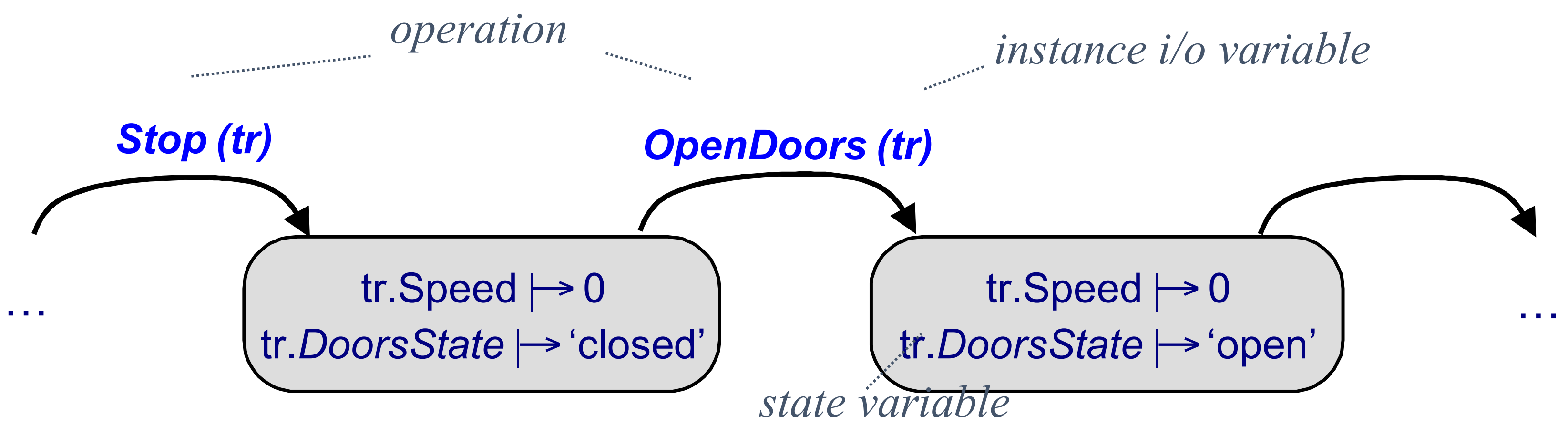
- Use case Diagrams
  - Describe the functional behaviour of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, Associations
- Sequence diagrams
  - Describe the dynamic behaviour between actors and the system and between objects of the system

The operation model

- Functional view of the system being modelled
- Multiple uses:
    - software specifications
        - input for development team
    - description of environment tasks and procedures
    - basis for deriving:
        - black-box test data
        - executable specs for animation, prototyping
    - definition of function points (for size estimation), work units, user manual sections
    - satisfaction arguments, traceability management

What are operations?

- **Operation** *Op* = set of input-output state pairs (binary relation)
  - input variable: object instance that its state affects the application of the operation
  - output variable: object instance its state is changed by the application of the operation
- **Operation application yields state transition from a state in InputStateSet to a state in OutputStateSet**

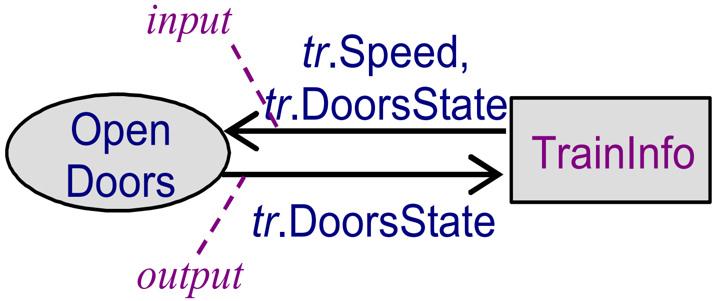


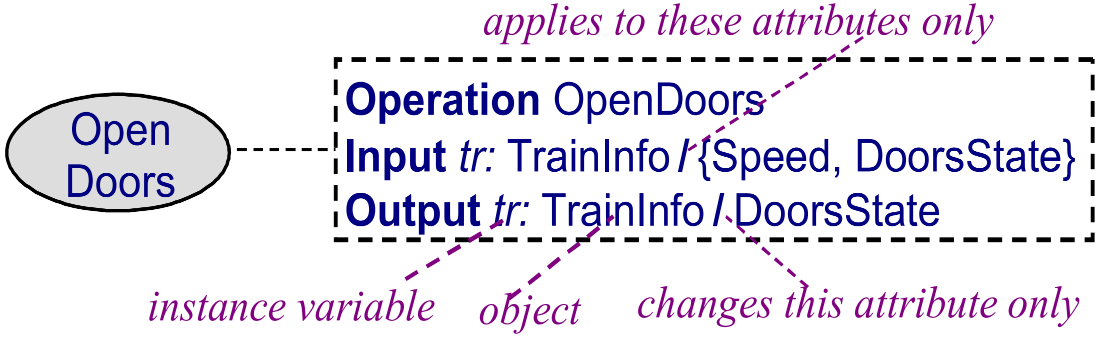
What are operations? (2)

- Op must **operationalize** underlying goals from the goal model
  - To make these satisfied => application under restricted conditions
- Generally deterministic: relation over states is a *function*
  - No multiple alternative outputs from the same input
- **Atomic**: map input state to state at next smallest time unit
  - For operations lasting some duration: use **startOp/endOp** events
- May be applied concurrently with others
  - e.g. OpenDoors || DisplayWhichPlatform
- Software operations, environment operations (tasks)
  - e.g. PlanMeeting, SendConstraints

Characterizing system operations

- Basic features: Name, Definition, Category
- Signature
  - declares the input-output relation over states
    - input/output variables and their type (object from object model)
    - scope may be restricted to specific attributes (nothing else changes)
    - used in pre-, postconditions
  - graphical or textual annotation





applies to these attributes only

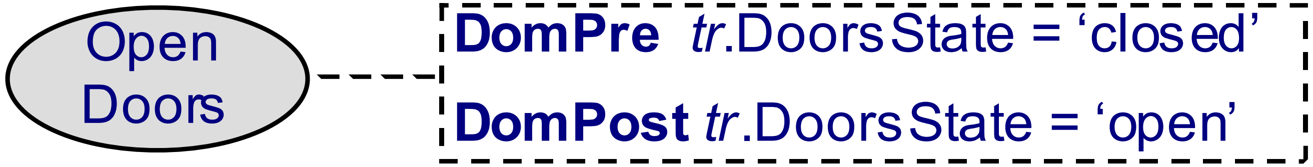
Operation OpenDoors
Input *tr:* TrainInfo / {Speed, DoorsState}
Output *tr:* TrainInfo / DoorsState

instance variable *object* *changes this attribute only*

Characterizing system operations (2)

- Conditions capturing the class of state transitions that define the operation

- **DomPre:** condition characterizing the class of input states in the domain
- **DomPost:** condition characterizing the class of output states in the domain



DomPre *tr*.DoorsState = ‘closed’

DomPost *tr*.DoorsState = ‘open’

Characterizing system operations (3)

- An agent **performs** an operation if the applications of this operation are activated by instances of this agent
- Consistency rules between the *operation* model and *agent* model:
  - Every *input/output* state variable in the signature of operation performed by an agent must be *monitored/controlled* by it in the agent model
  - Unique performer; every operation is performed by precisely one agent

Textual Functional Model

Operation

Def:

Input:

Output:

DomPre:

DomPost:

Example

Operation OpenDoors

Def: Operation controlling the opening of all train doors.

Input: tr:Train / {Speed, Position, DoorsState},

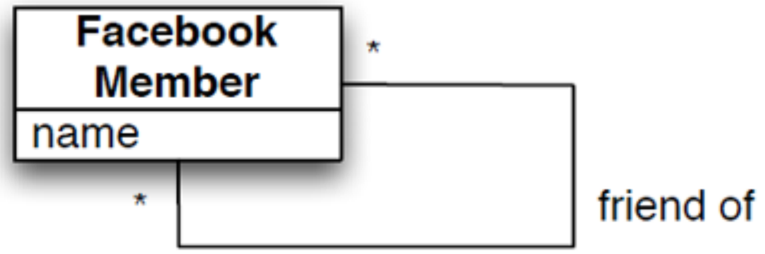
Output: tr:Train / DoorsState

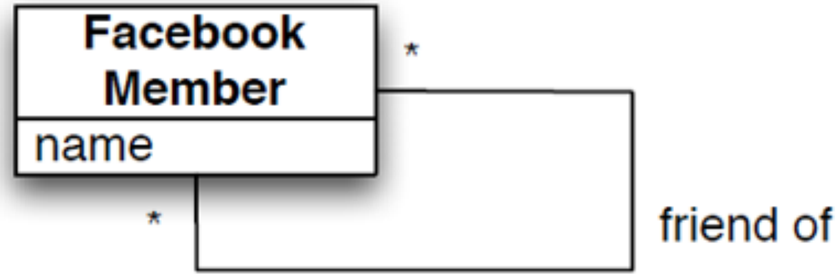
DomPre: The doors of train tr are closed.
The speed of train tr is 0.
Train tr is at a platform.

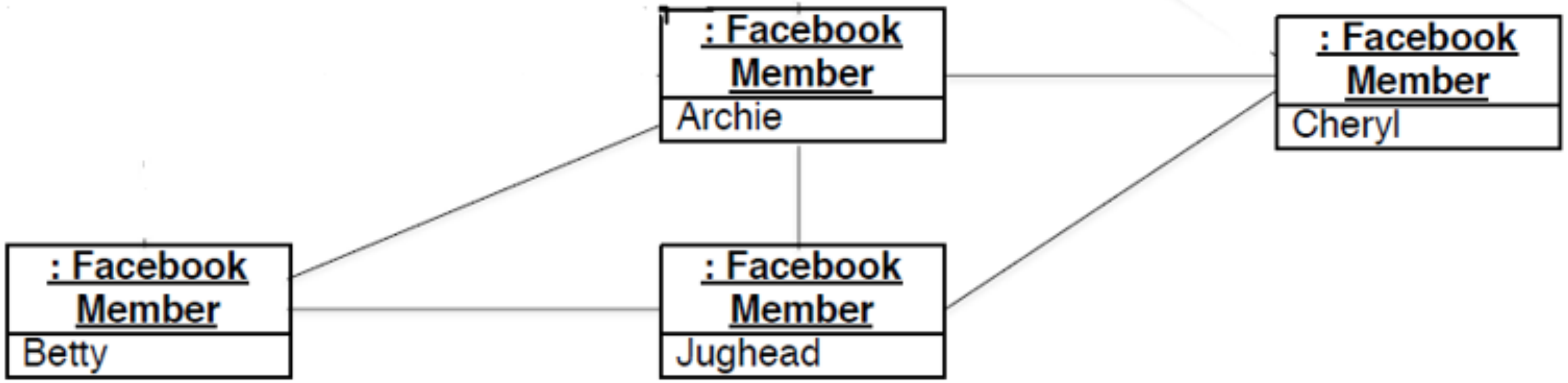
DomPost: The doors of train tr are open.

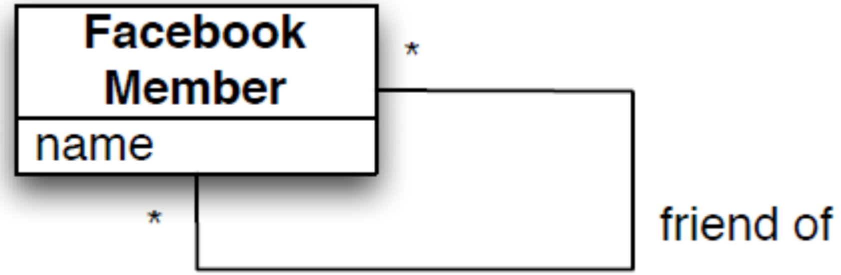
World States

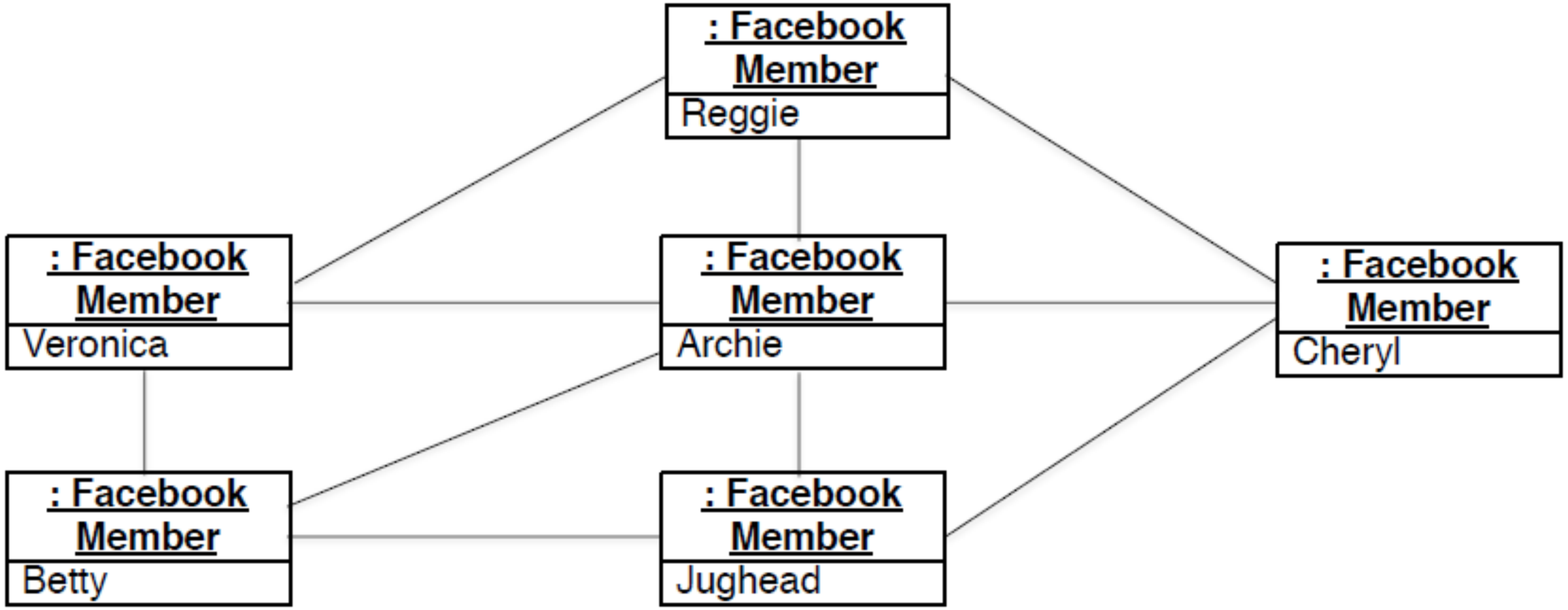
The domain model represents the set of possible states of the world (called world states).



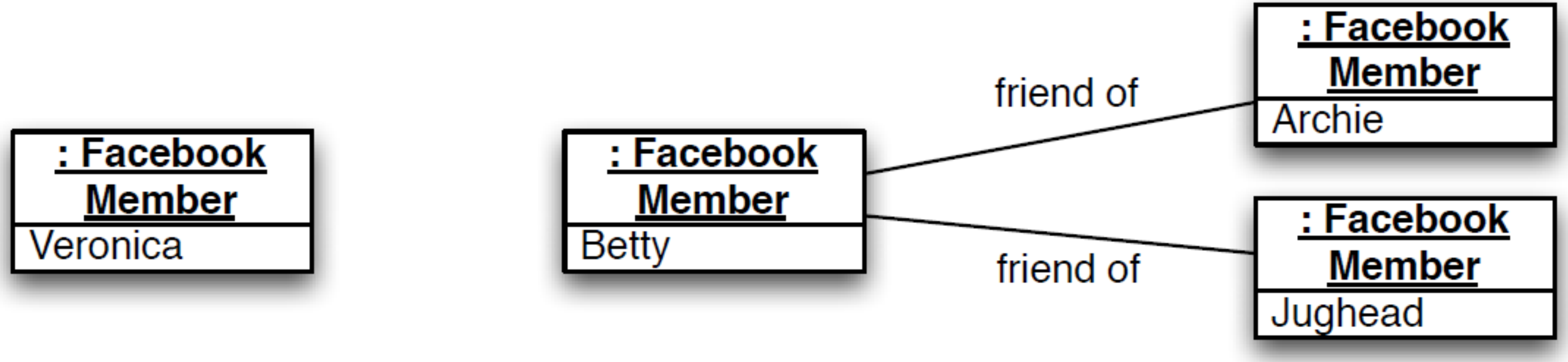

Facebook Member
name
*
*
friend of


: Facebook Member
Archie
: Facebook Member
Cheryl
: Facebook Member
Betty
: Facebook Member
Jughead

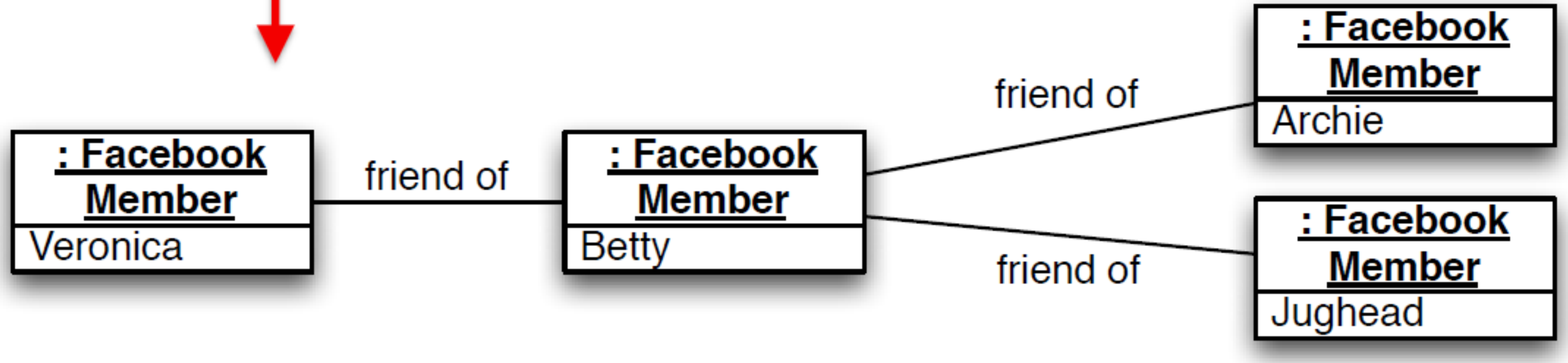

Facebook Member
name
*
*
friend of


: Facebook Member
Reggie
: Facebook Member
Veronica
: Facebook Member
Archie
: Facebook Member
Cheryl
: Facebook Member
Betty
: Facebook Member
Jughead

The functional model expresses the system functionality in terms of system changes to the world state.



FRIEND (Betty : FacebookMember,
Veronica : FacebookMember)



Abstract World State

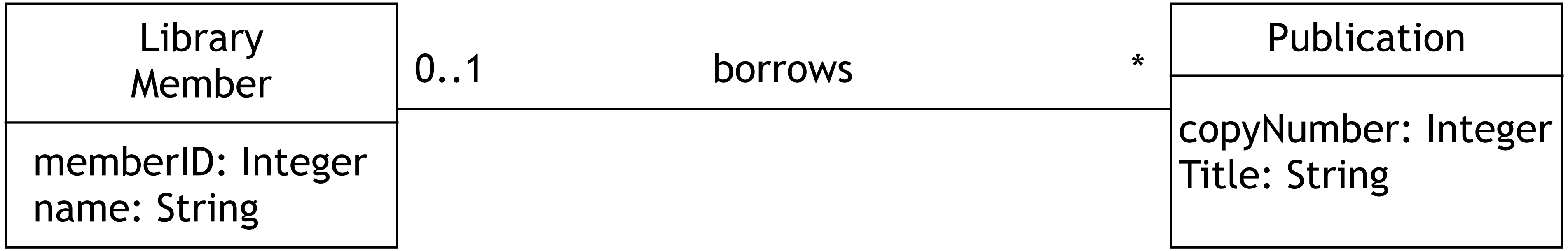
- From the domain model (which defines types), we derive an abstract world state model (which defines sets of instances)
- Example:

Abstract World State

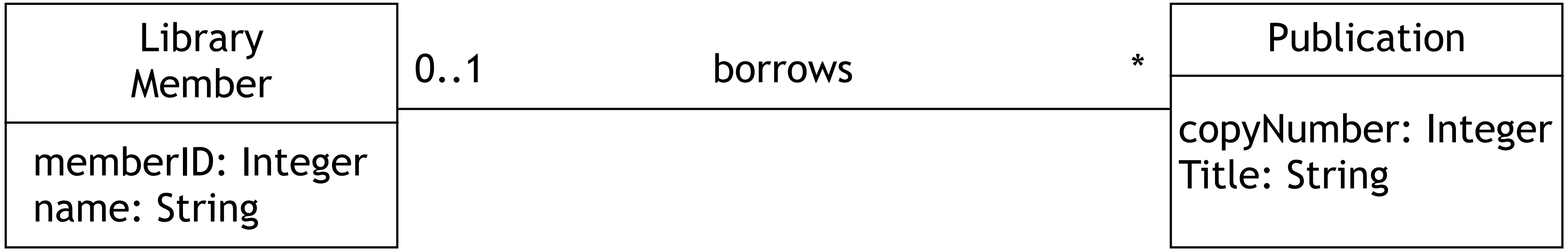
Members: set of Library Member

Pubs: set of Publication

Borrows $\subseteq$ Library Member × Publication



Exceptions



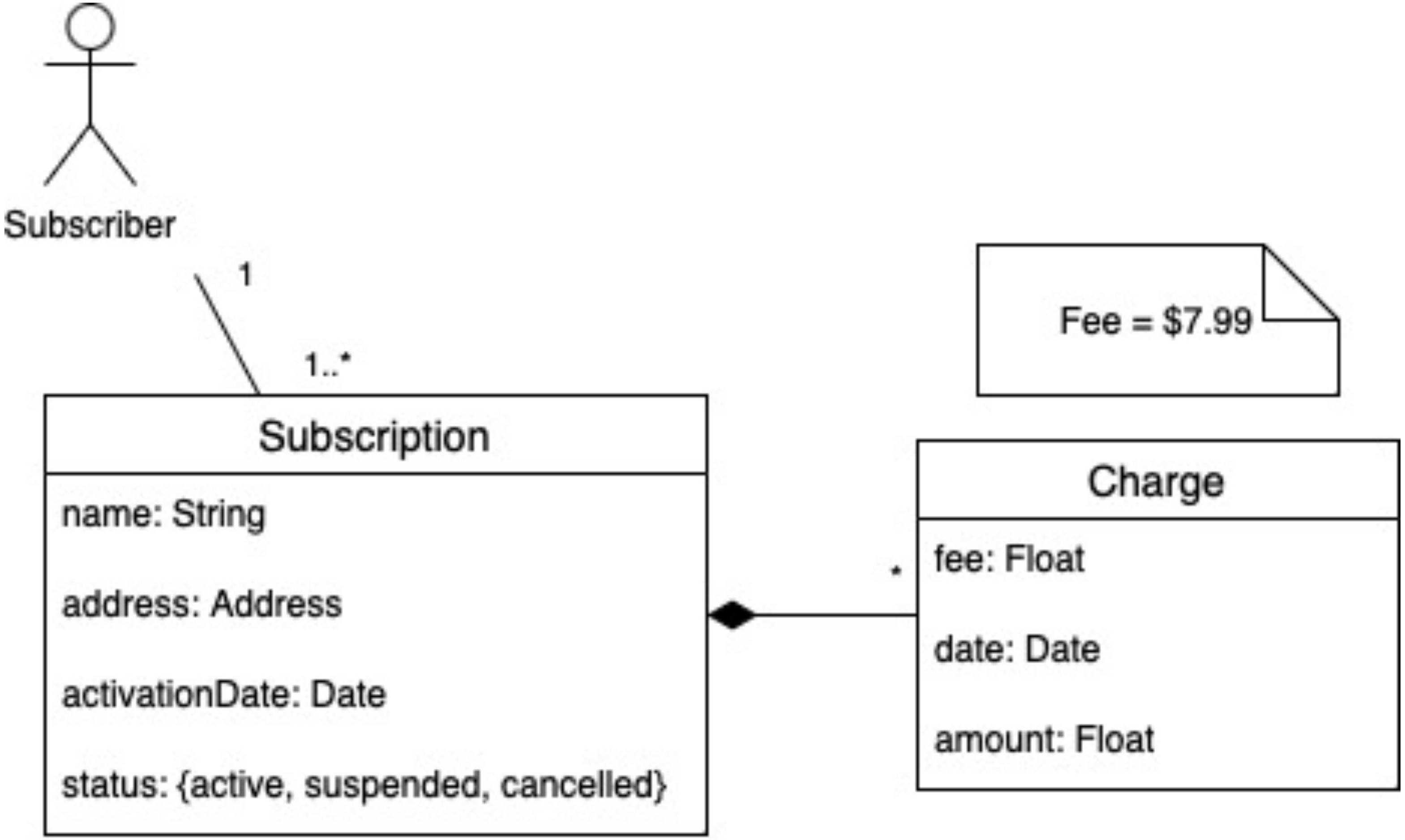
FindBorrowedPubs(memberID): set of Publication
pre: Members[memberID] ≠ ø
modifies: <none>
post: return Borrows[memberID]
exception: if(Members[memberID]) = ø
then **return** error message

Using the domain model diagram for Flix.net

Model the following operations of the system as functions over an abstract world state of your domain model. Your functions should specify all changes to the domain that a function realizes, including new or deleted links -- even those with actors. Specify exceptions if appropriate. You do not need to specify an initial abstract world state. Include also short descriptions of the parameters of your functions.

- Suspending a subscription
- Charging a subscription fee to a subscription
- Initiating a video stream to a device (i.e., TV or computer)

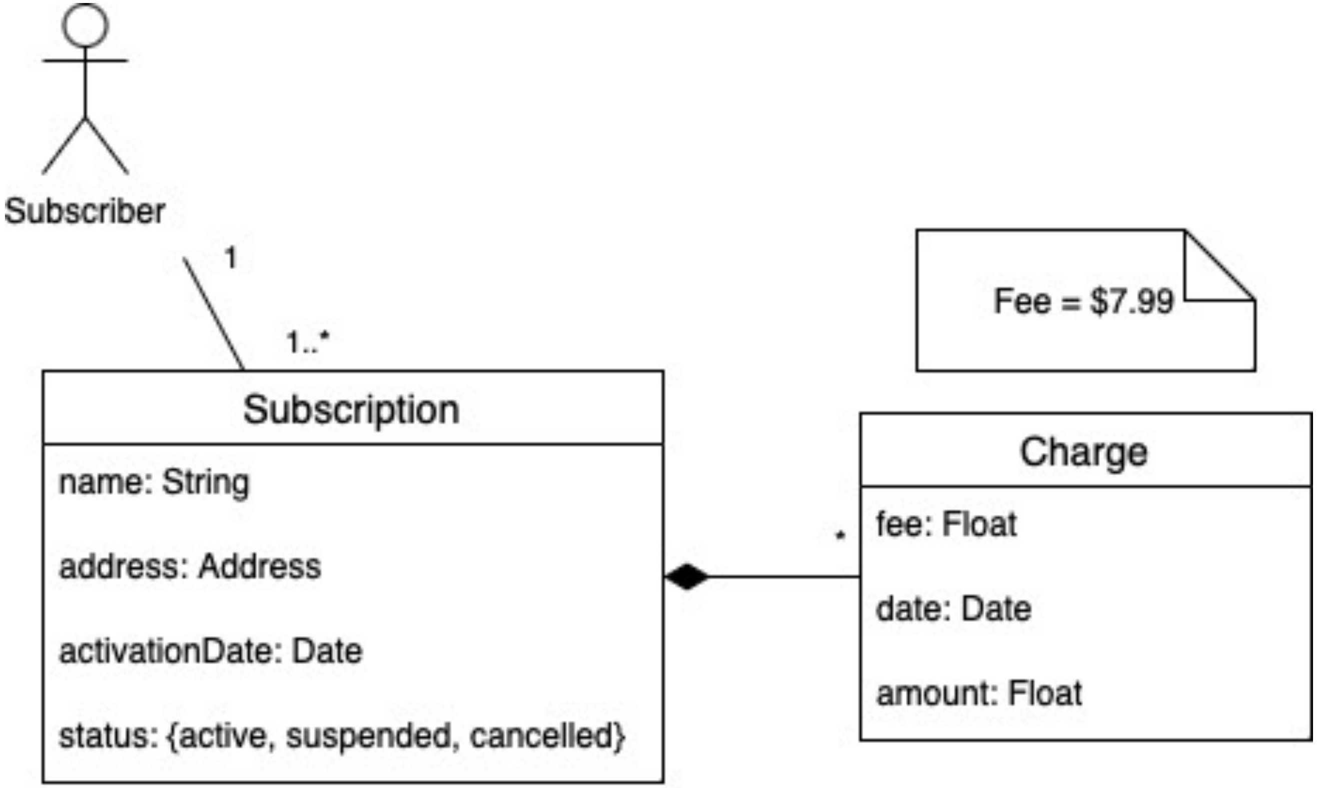
Suspending a subscription



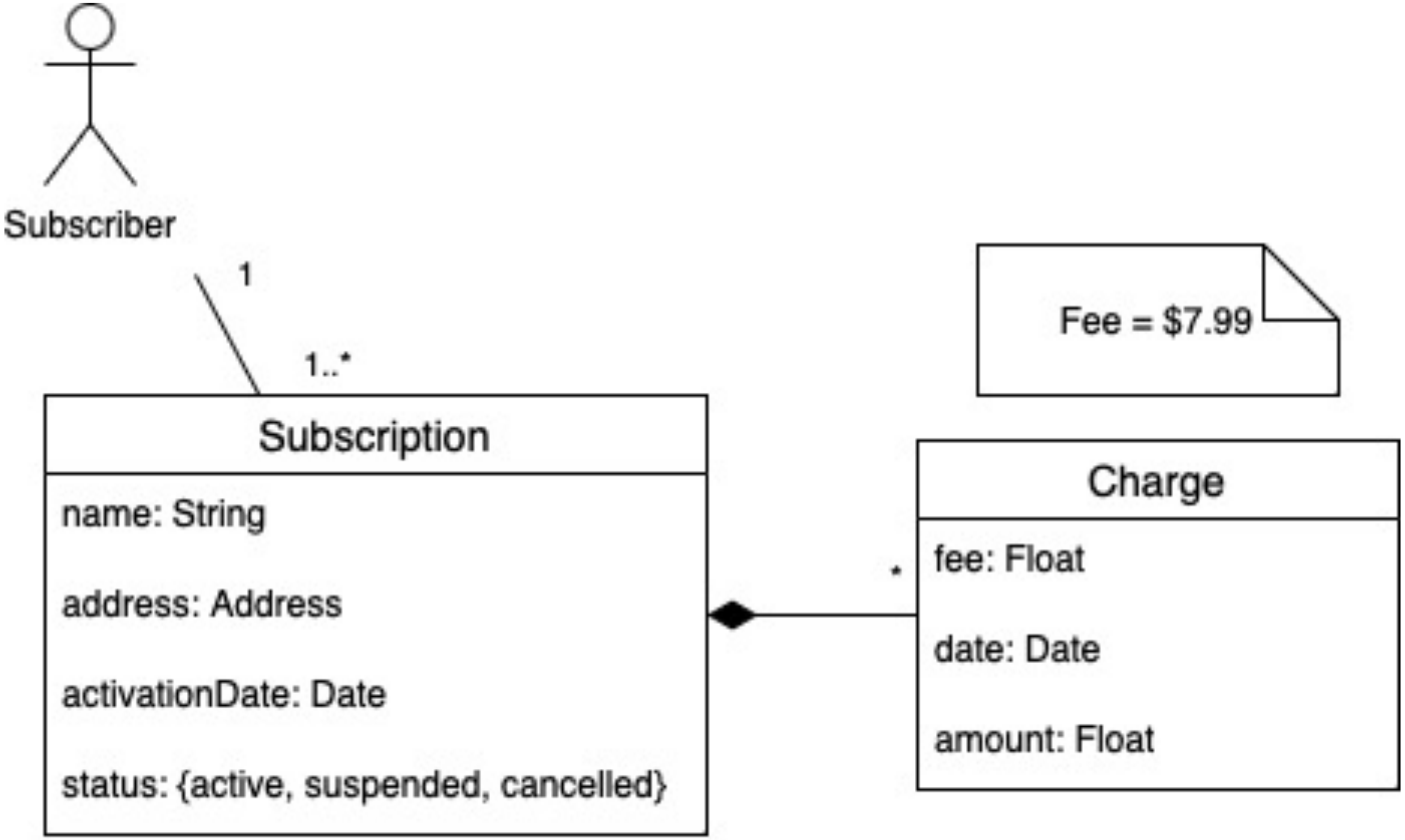
Suspending a subscription

Let s be the subscription to be suspended.

Suspend (s:Subscription)
pre: <none>
modifies: Subscription
post: s.status = “suspended”
exception: <none>



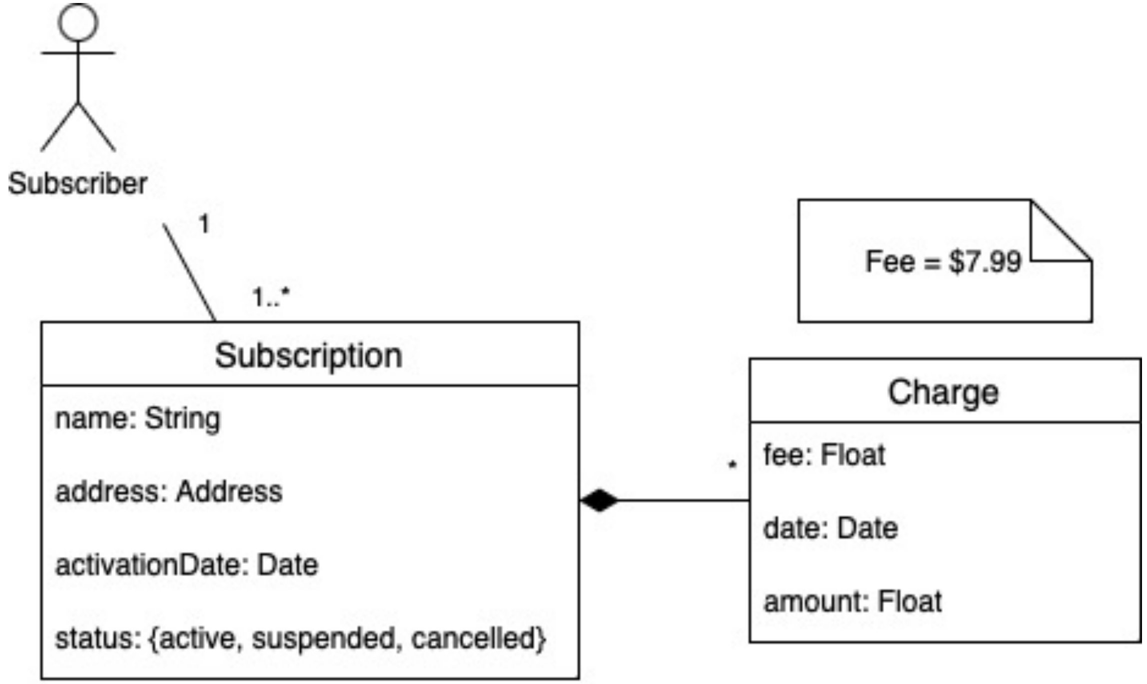
Charging a subscription fee to a subscription



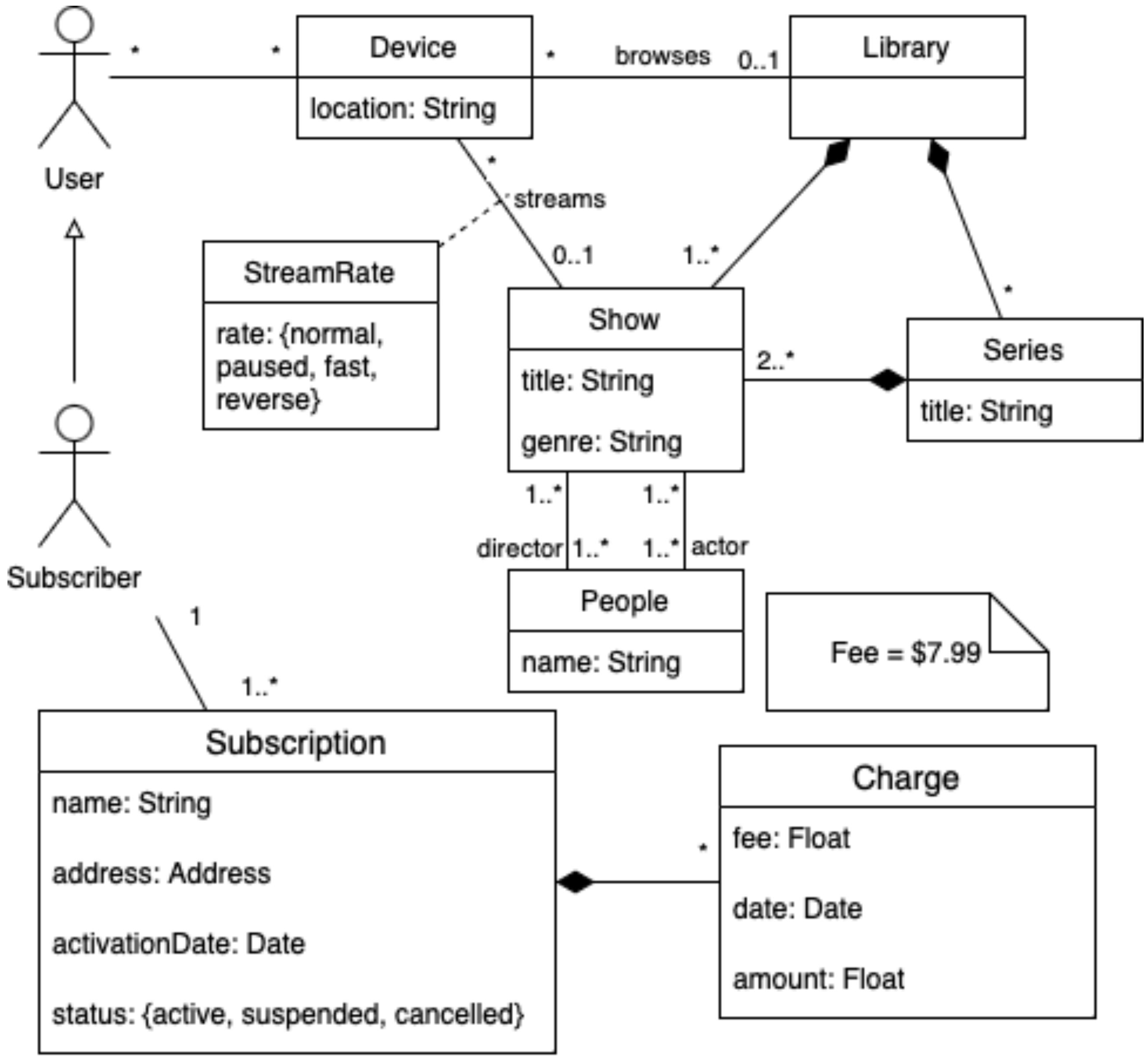
Charging a subscription fee to a subscription

Let s be the subscription to be charged.

ChargeFee(s:Subscription)
pre: s.status="active"
modifies: s.charge, Charges
post: new c':Charge
        c' = (date::today, amount::fee)
        Charges' = Charges ∪ c'
        s'.charge = s.charge ∪ c'
exception: if s.status <> "active"
                then no change to the world state



Initiating a video stream to a device (i.e., TV or computer)



Initiating a video stream to a device (i.e., TV or computer)

Let v be the Show to be streamed, let d be the receiving device and let s be the requesting subscriber.

InitiateStream(s:Subscriber, d:Device, v:Show)
pre: s.status = “active” and d.location = “Canada”
modifies: d, StreamRates
post: new sr’:StreamRate
        sr’ = (d, v, “normal”)
        StreamRates’ = StreamRates ∪ sr’
        d’ = d ⊕ (d’.streams = v) ⊕ (d’.StreamRate = sr’)
exception: if(s.status <> “active” or d.location <> “Canada”)
                then no change to the world state

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Functional Modelling

Axel van Lamsweerde

Requirements Engineering

From System Goals to UML Models to Software Specifications

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Sequence Diagrams

Sequence Diagrams

- UML has a language for describing scenarios, that of the sequence diagram.
- Show step-by-step what’s involved in a use case
  - Which objects are relevant to the use case
  - How those objects participate in the function
- You may need several sequence diagrams to describe a single-use case
  - Each sequence diagram describes one possible scenario for the use case
- Show all events external actors generate, their order, and inter-system events. All systems are treated as a black box; the <u>emphasis</u> of the diagram is <u>events</u> that <u>cross the system boundary</u> from actors to systems.

Sequence Diagrams

- Vertical line is called an object’s **lifeline**
    - Represents an object’s life during interaction
- Object deletion denoted by X, ending a lifeline
    - Horizontal arrow is a message between two objects
- Order of messages sequences top to bottom
- Messages labeled with message name
    - Optionally arguments and control information
- Control information may express conditions:
    - such as [hasStock], or iteration
- Returns (dashed lines) are optional
    - Use them to add clarity

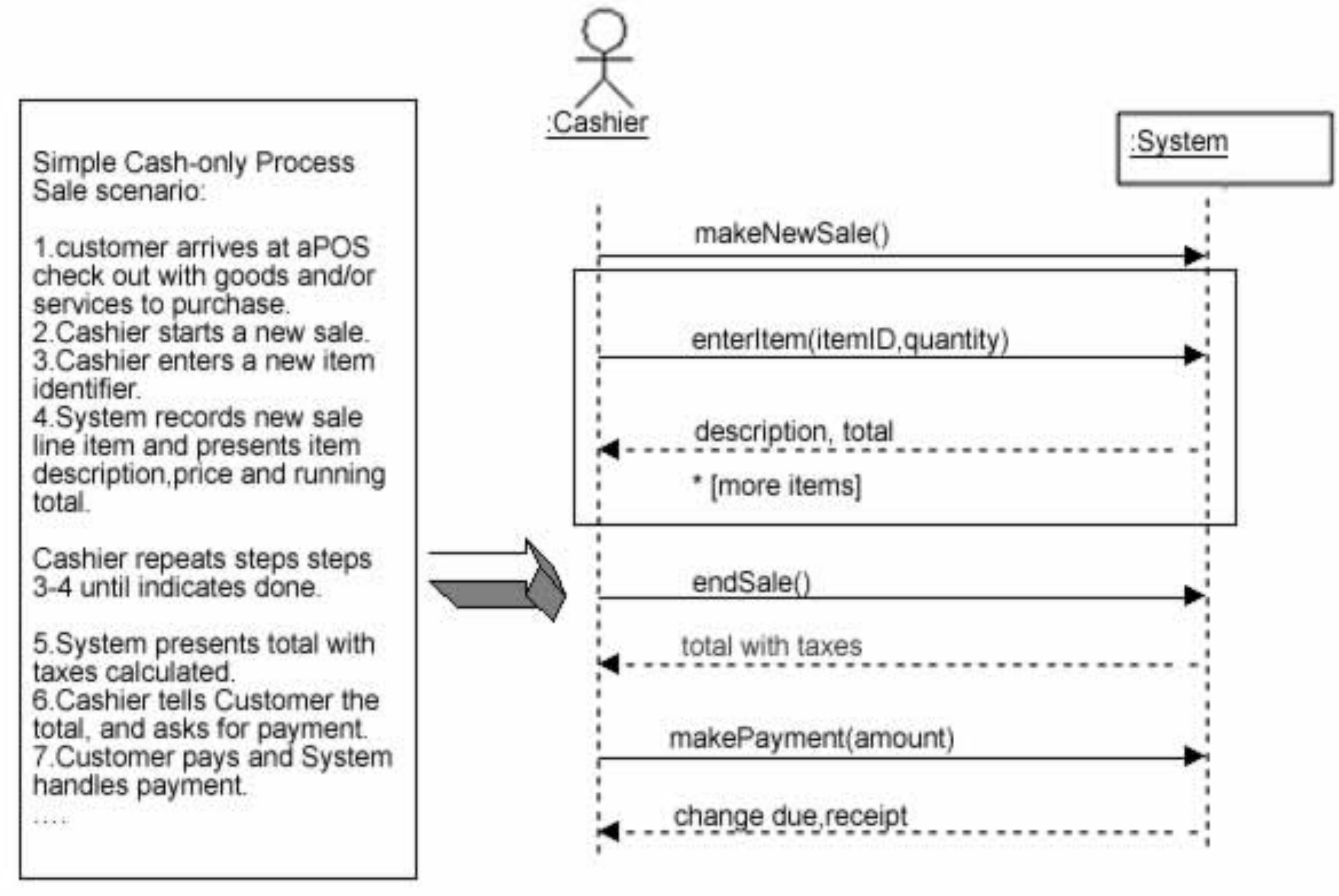
System Sequence Diagram (SSD)

For a use case scenario, a SSD shows:

- The System (as a black box)

- The external actors that interact with System
- The System events that the actors generate
- SSD shows operations of the System in response to events, in temporal order
- Develop SSDs for the main success scenario of a selected use case, then frequent and salient alternative scenarios

Example: Use Case to SSD



Simple Cash-only Process Sale scenario:

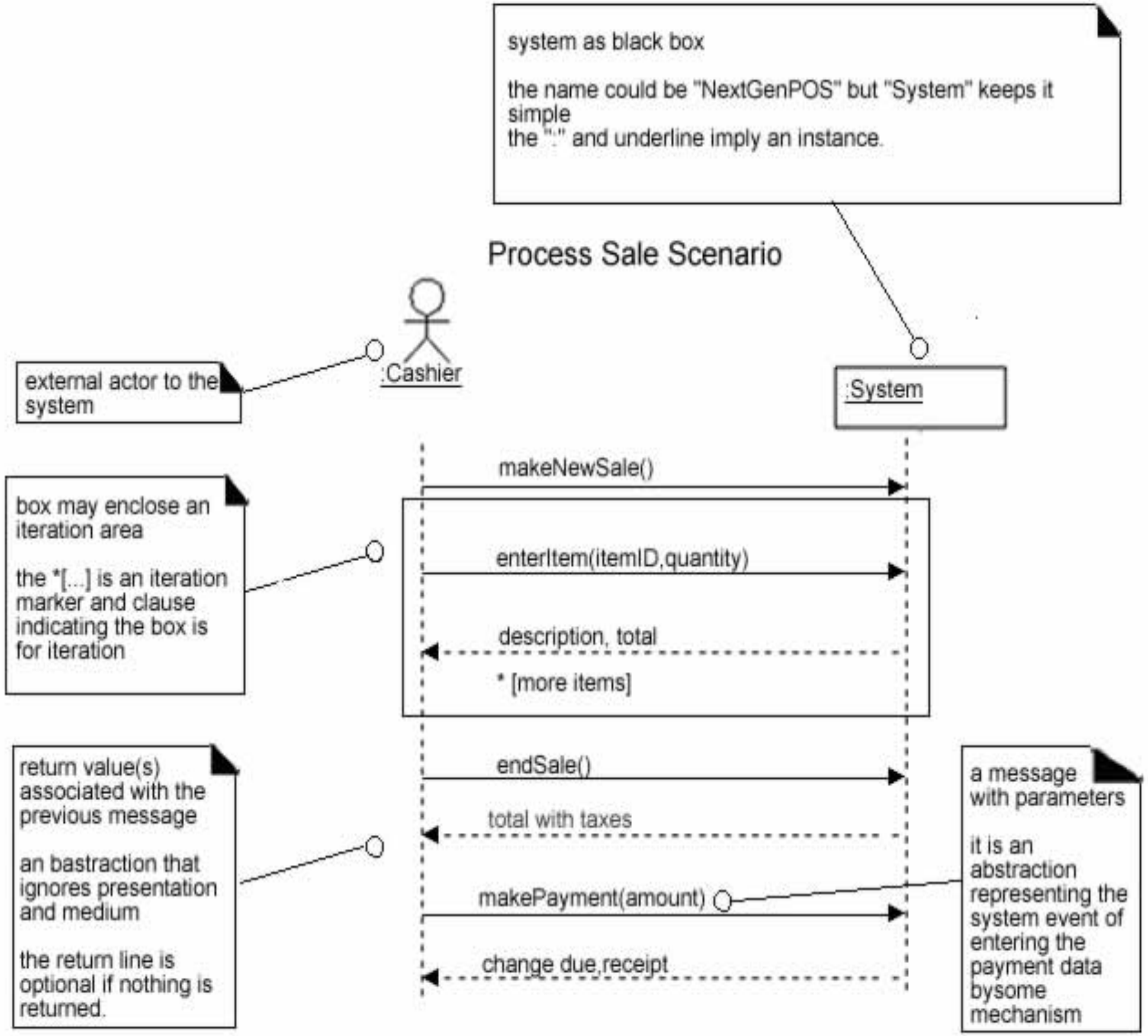
1.customer arrives at aPOS check out with goods and/or services to purchase.
2.Cashier starts a new sale.
3.Cashier enters a new item identifier.
4.System records new sale line item and presents item description,price and running total.

Cashier repeats steps steps 3-4 until indicates done.

5.System presents total with taxes calculated.
6.Cashier tells Customer the total, and asks for payment.
7.Customer pays and System handles payment.
....

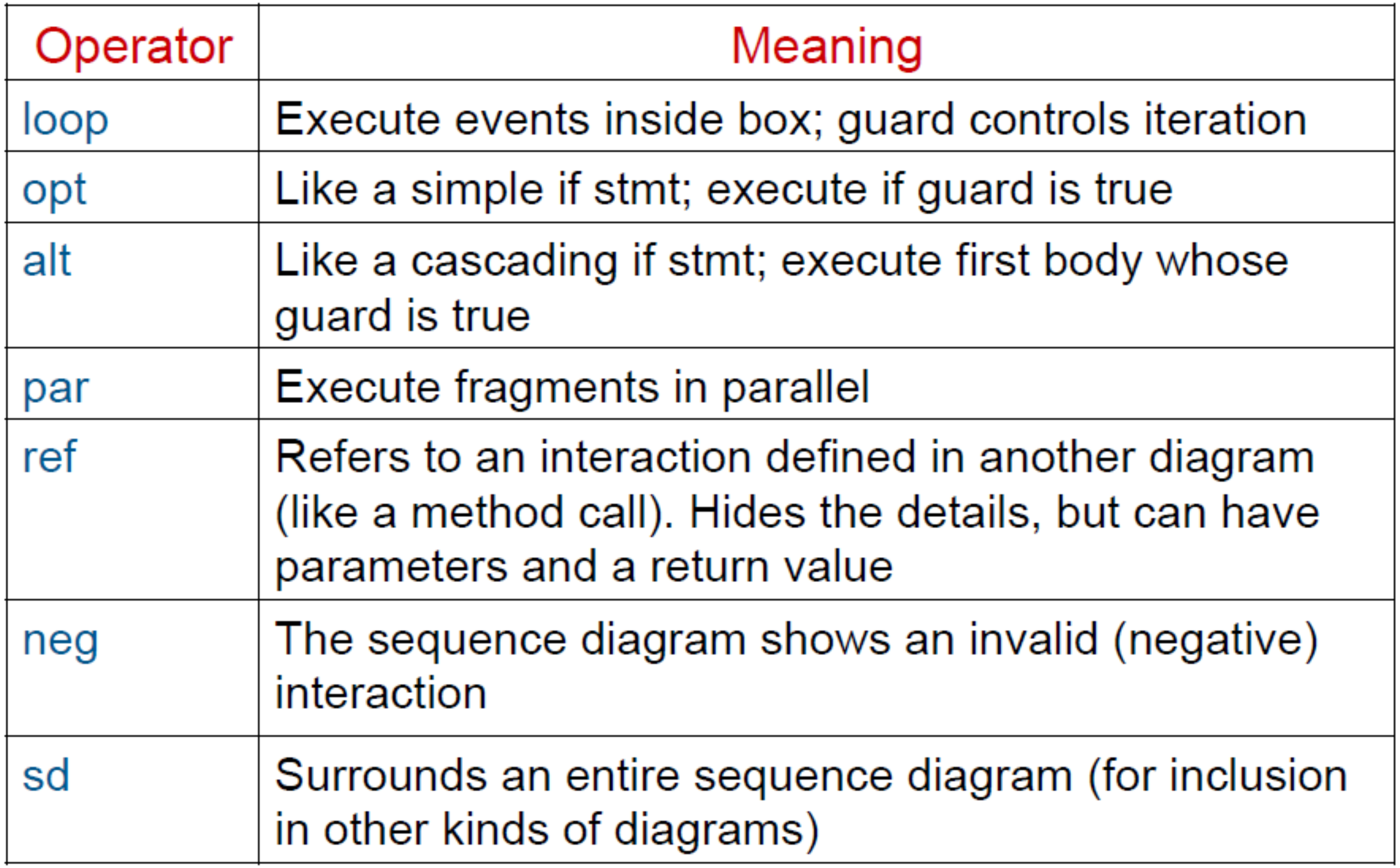
SSD for Process Sale scenario

(Larman, page 175)



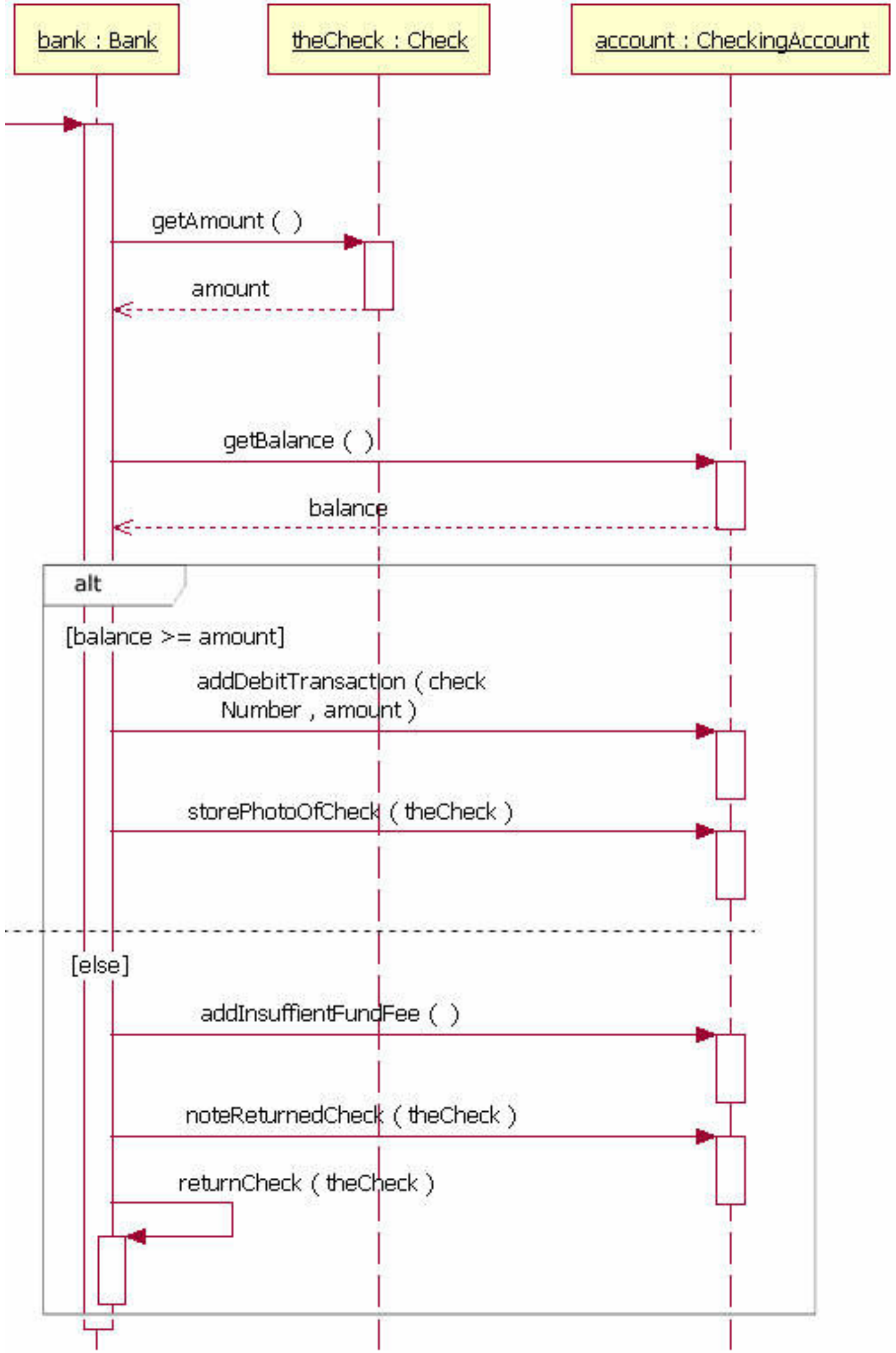
system as black box
the name could be "NextGenPOS" but "System" keeps it simple
the ":" and underline imply an instance.
Process Sale Scenario
external actor to the system
:Cashier
:System
makeNewSale()
box may enclose an iteration area
the *[...] is an iteration marker and clause indicating the box is for iteration
enterItem(itemID,quantity)
description, total
* [more items]
return value(s) associated with the previous message
an bastraction that ignores presentation and medium
the return line is optional if nothing is returned.
endSale()
total with taxes
makePayment(amount)
change due,receipt
a message with parameters
it is an abstraction representing the system event of entering the payment data bysome mechanism

Interaction Frame Operators

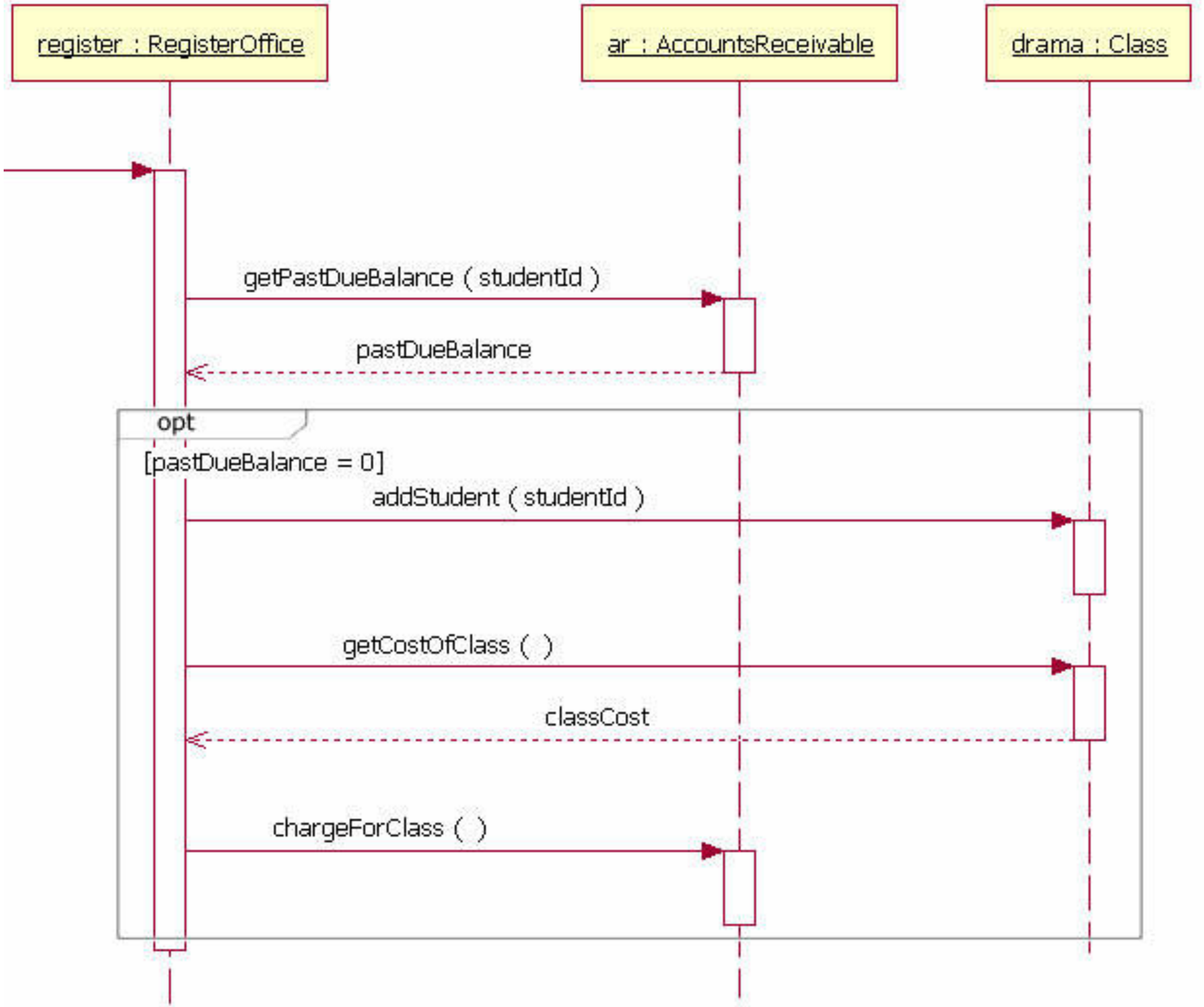


Operator	Meaning
loop	Execute events inside box; guard controls iteration
opt	Like a simple if stmt; execute if guard is true
alt	Like a cascading if stmt; execute first body whose guard is true
par	Execute fragments in parallel
ref	Refers to an interaction defined in another diagram (like a method call). Hides the details, but can have parameters and a return value
neg	The sequence diagram shows an invalid (negative) interaction
sd	Surrounds an entire sequence diagram (for inclusion in other kinds of diagrams)

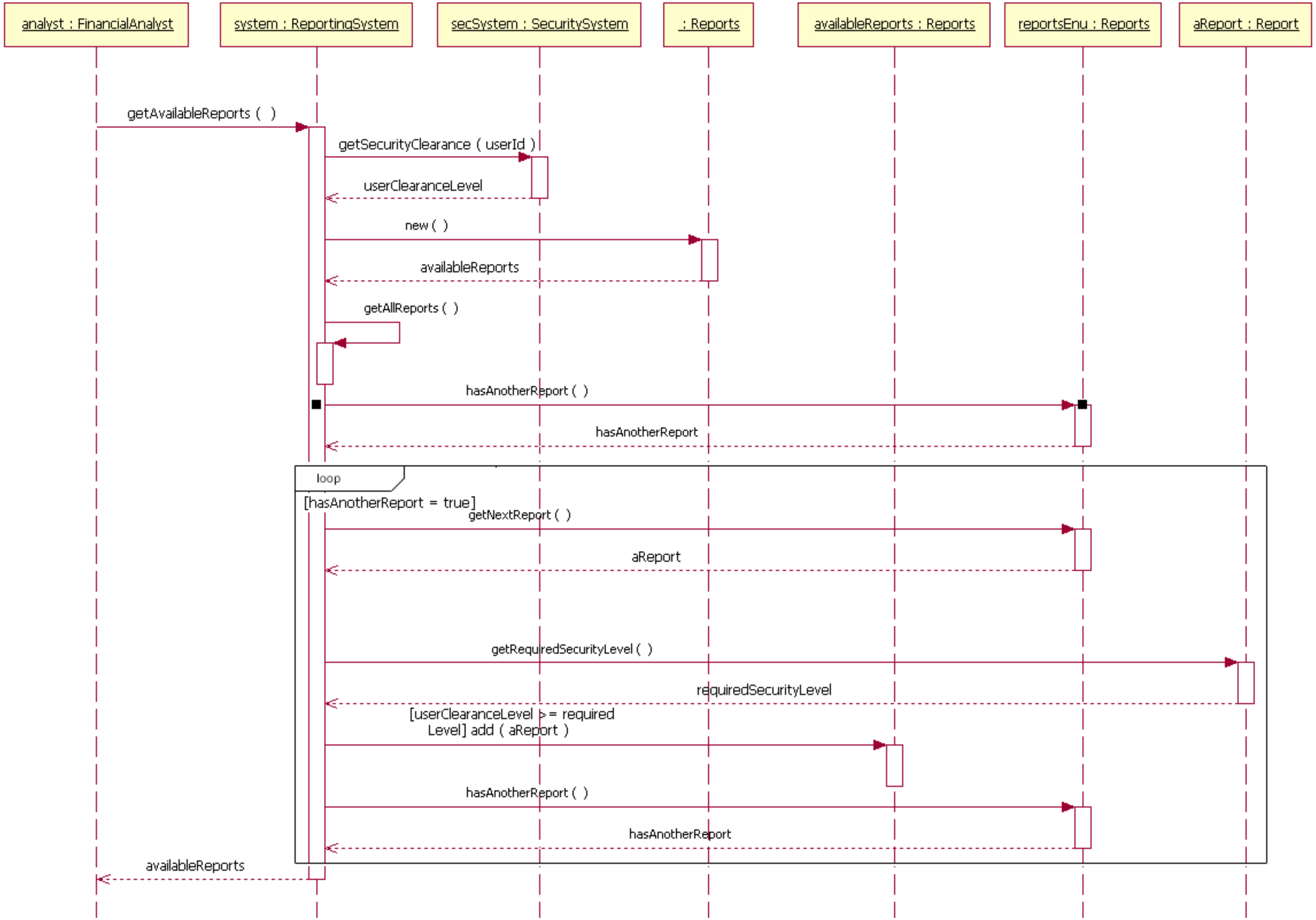
alt


bank : Bank
theCheck : Check
account : CheckingAccount
getAmount ( )
amount
getBalance ( )
balance
alt
[balance >= amount]
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
[else]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)

opt


register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
opt
[pastDueBalance = 0]
addStudent (studentId)
getCostOfClass ( )
classCost
chargeForClass ( )

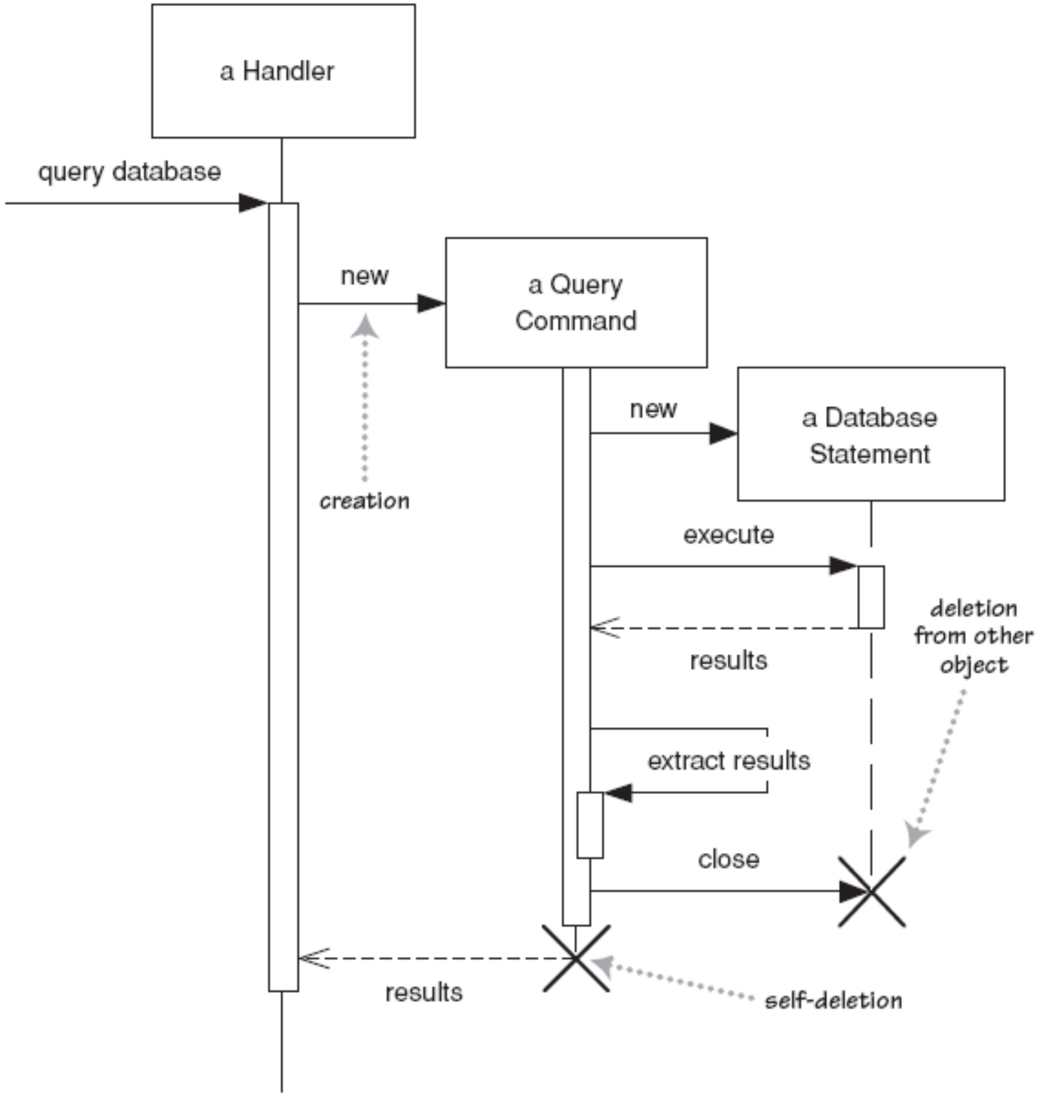
loop


analyst : FinancialAnalyst
system : ReportingSystem
secSystem : SecuritySystem
: Reports
availableReports : Reports
reportsEnu : Reports
aReport : Report
getAvailableReports ( )
getSecurityClearance (userId)
userClearanceLevel
new ( )
availableReports
getAllReports ( )
hasAnotherReport ( )
hasAnotherReport
loop
[hasAnotherReport = true]
getNextReport ( )
aReport
getRequiredSecurityLevel ( )
requiredSecurityLevel
[userClearanceLevel >= required Level] add (aReport)
hasAnotherReport ( )
hasAnotherReport
availableReports

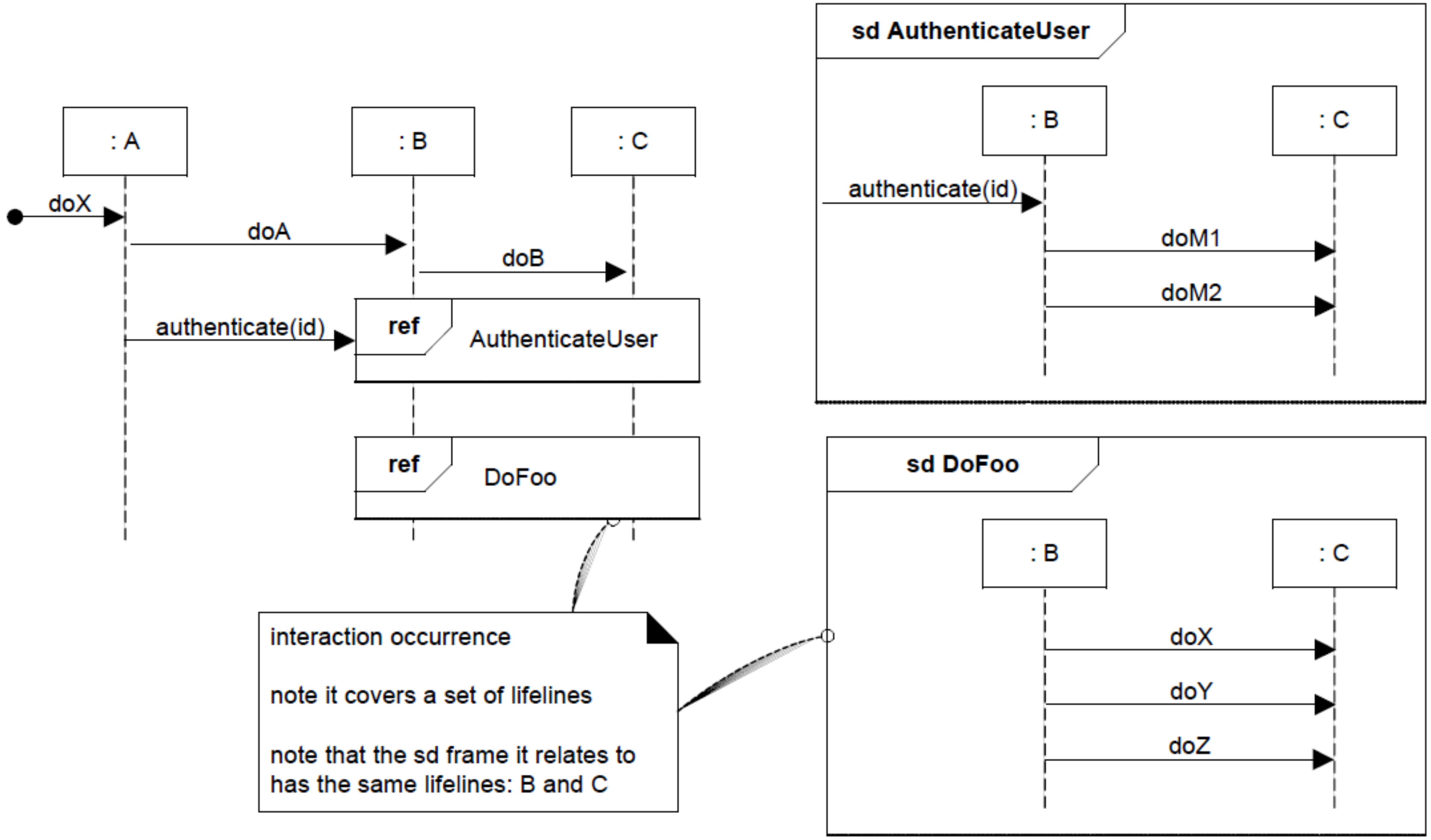
Rules of thumb

- Rarely use option, loop, alt/else
  - These constructs complicate a diagram and make them hard to read/interpret.
  - Frequently it is better to create multiple simple diagrams
- Create sequence diagrams for use cases when it helps clarify and visualize a complex flow
- Remember: the goal of UML is communication and understanding

Lifetime: Creation / Deletion



“Call” to Sub Diagram



interaction occurrence

note it covers a set of lifelines

note that the sd frame it relates to has the same lifelines: B and C

Larman, *Appliying UML and Patterns, 3ed*

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Sequence Diagrams

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Use Cases and Scenarios

Requirements / Specification Models

Model: a simplified version of something complex used in analyzing and solving problems or making predictions.

Modeling consists of building an abstraction of reality.

Uses of Models:

- Can guide elicitation
- Can provide a measure of progress
- Can help to uncover problems
- Can help us check our understanding

UML (Unified Modeling Language)

- Provides a standard way to visualize the design of a system
- Unified: it has become a world standard (OMG Object Management Group, www. omg.org)
- Modeling: it describes a software system at a high level of abstraction
- Language: it expresses an idea, not a methodology
- More…
    - It is an industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
    - The UML uses mostly graphical notations to express software projects' OO analysis and design.
    - Simplifies the complex process of software design

Types of UML Diagrams (first pass):

- Class diagrams
    - Describe the static structure of the system: Objects, Attributes, Associations
- Use case Diagrams
    - Describe the functional behavior of the system as seen by the user.
- Sequence diagrams
    - Describe the dynamic behavior between actors and the system and between objects of the system
- State diagrams
    - Describe the dynamic behavior of an individual object (essentially a finite state automaton)
- Activity Diagrams
    - Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart)

This is only a subset of diagrams, but are most widely used

Use Case (UC)

- Each particular way to use a system is called a use case.
- It is one case of the many ways to use the system.
- Use cases are a summary of the way that all types of users will interact with the (proposed) system
- Use cases can help us discover/document requirements
- Should be easy to read
- Defines the interactions between system and actors
- Focuses on interaction, not internal system activities.
- A use case should not be confused with a scenario. A scenario of system is a particular sequence of interaction steps between a user and the system.

Which of the following is a use case?

- Order cost = order item costs * 1.06 tax.
- Promotions may not run longer than 6 months.
- Customers only become Preferred after 1 year.
- A customer has one and only one sales contact.
- Response time is less than 2 seconds.
- Uptime requirement is 99.8%.
- Number of simultaneous users will be 200 max.

Consider software to run a cell phone:

Use Cases

- call someone
- receive a call
- send a message
- memorize a number

Point of view: user

Internal Functions

- transmit / receive data
- energy (battery)
- user I/O (display, keys, ...)
- phone-book mgmt.

Point of view: developer / designer

Actors and stakeholders

- Actor: anything with behavior that acts on the system. An actor might be a person, a company or organization, a computer program, or a computer system-hardware, software, or both.

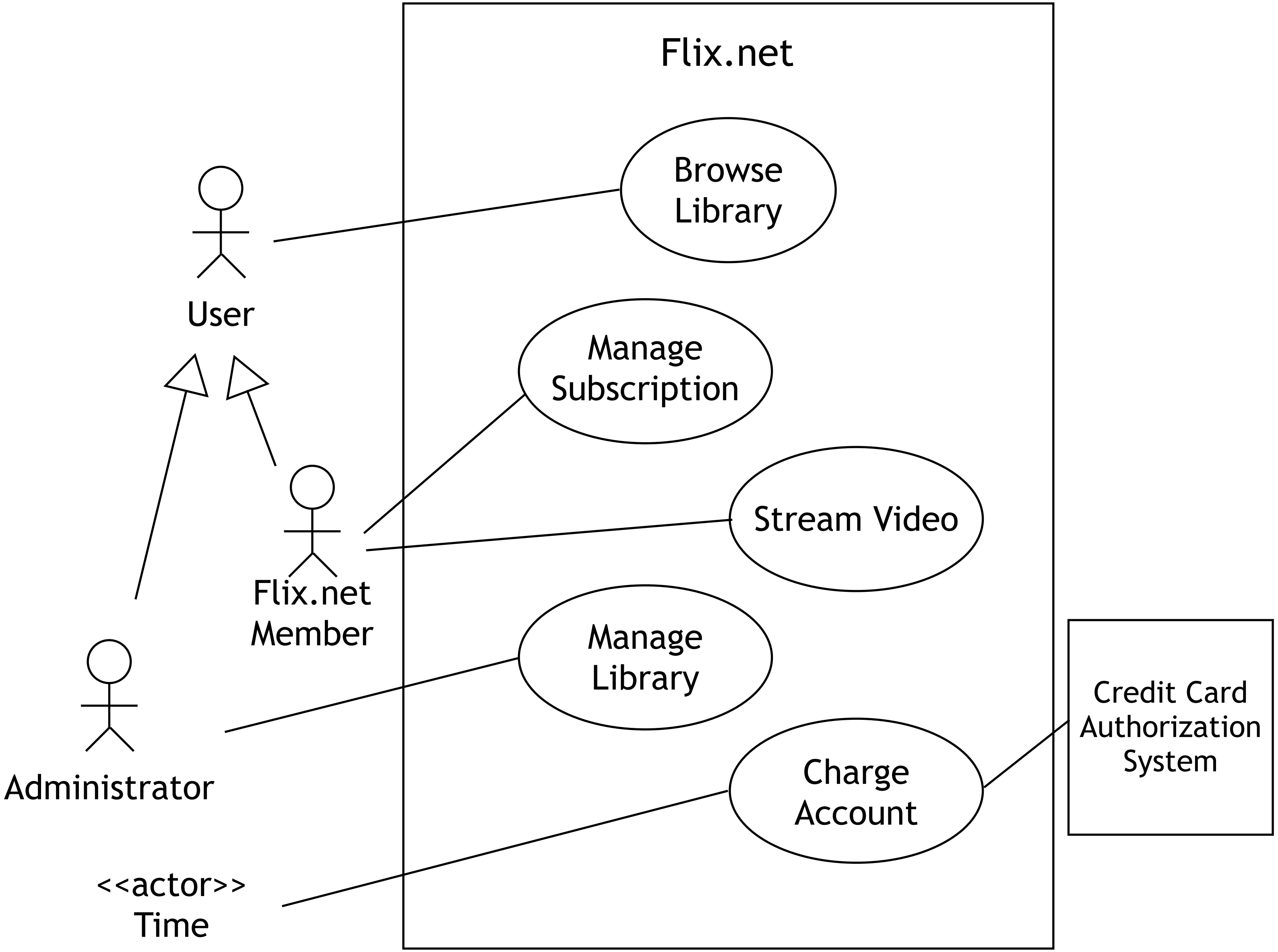
- Stakeholder: anyone interested in the system. Stakeholder might not “act” in any case/scenario.

- Primary vs secondary actors.

Exercise: Use case Diagram

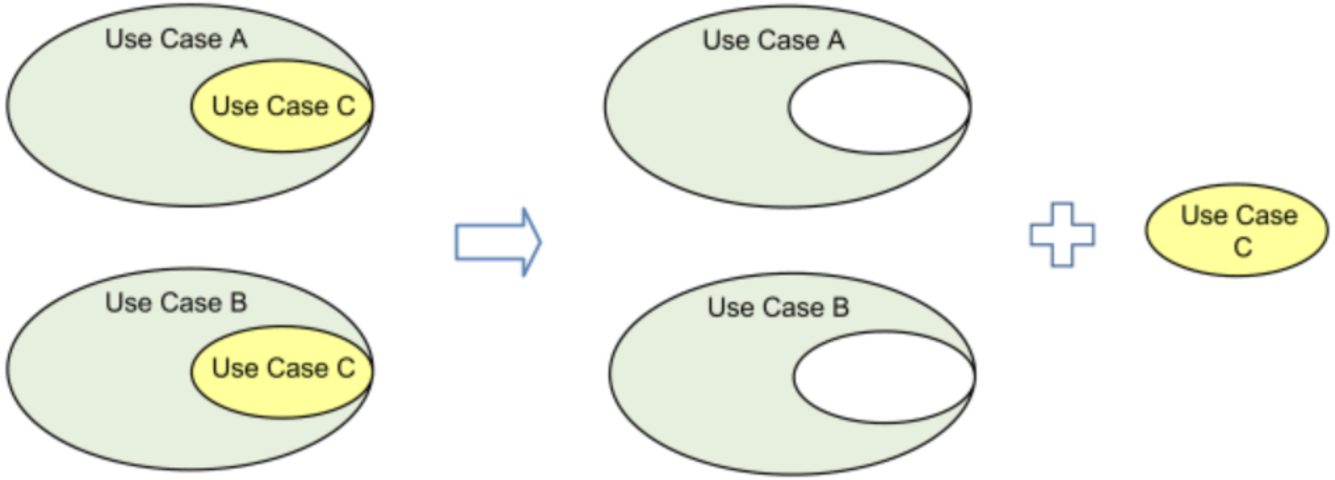
- Flix.net is an internet service for streaming movies and TV shows to personal computers and TVs.
- Anyone can browse the Flix.net library (by title, actor, director, and genre), but one must have a subscription to stream videos.
- A user can activate (i.e., create), suspend, or cancel membership.
- An account is active if it has not been suspended (and not re-activated) or cancelled.
- The subscription fee is $7.99 per month, charged on the monthly anniversary of the subscription's activation.
- If a user has an active subscription and accesses the website from within Canada, the user can stream as many videos (from the Flix.net library) as desired at any time of the day.
- A user can pause, rewind, fast-forward or stop a stream as often as they like.

- Flix.net is an internet service for streaming movies and TV shows to personal computers and TVs.
- Anyone can browse the Flix.net library (by title, actor, director, and genre), but one must have a subscription to stream videos.
- A user can activate (i.e., create), suspend, or cancel membership.
- An account is active if it has not been suspended (and not re-activated) or cancelled.
- The subscription fee is $7.99 per month, charged on the monthly anniversary of the subscription's activation.
- If a user has an active subscription and accesses the website from within Canada, the user can stream as many videos (from the Flix.net library) as desired at any time of the day.
- A user can pause, rewind, fast-forward or stop a stream as often as they like.

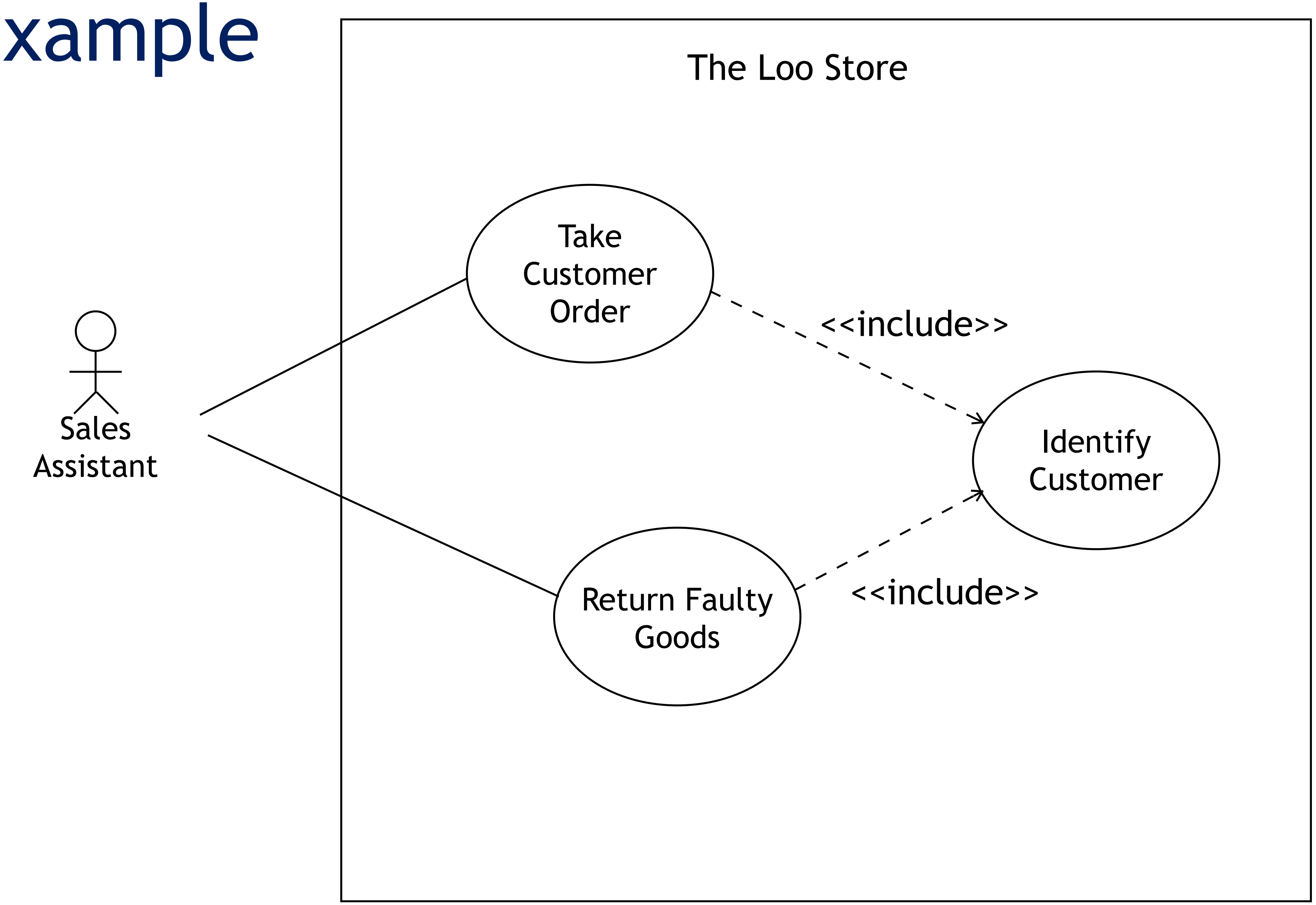


<<include>>

- A sub-use case that is used within multiple other use cases.
- You have a piece of behaviour that is similar across many use cases
- Break this out as a separate use case and let the other ones "include" it
- Examples include
  - Valuation
  - Validate user interaction
  - Sanity check on sensor inputs
  - Check for proper authorization



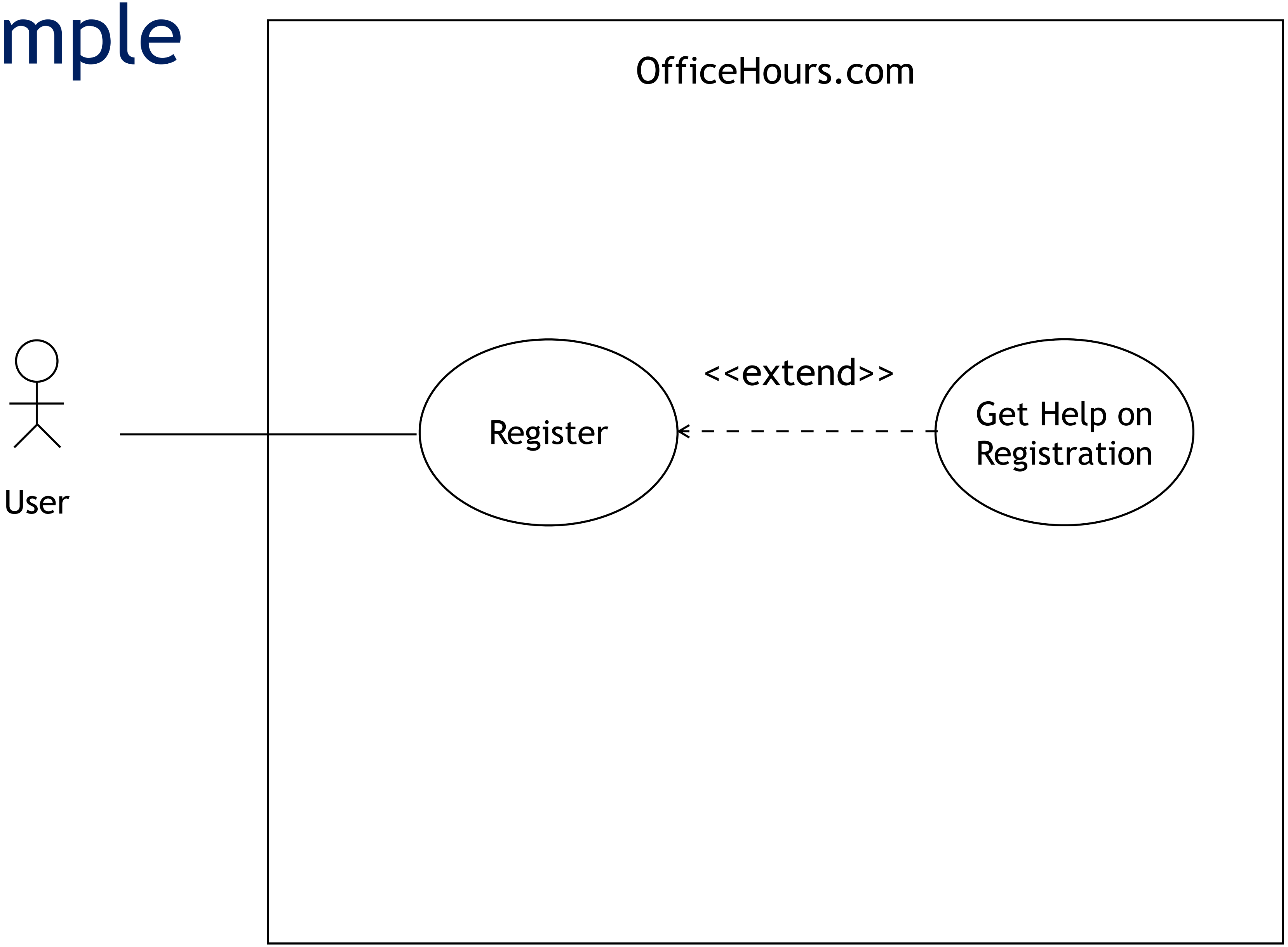
Example


The Loo Store
Take Customer Order
<<include>>
Sales Assistant
Identify Customer
Return Faulty Goods
<<include>>

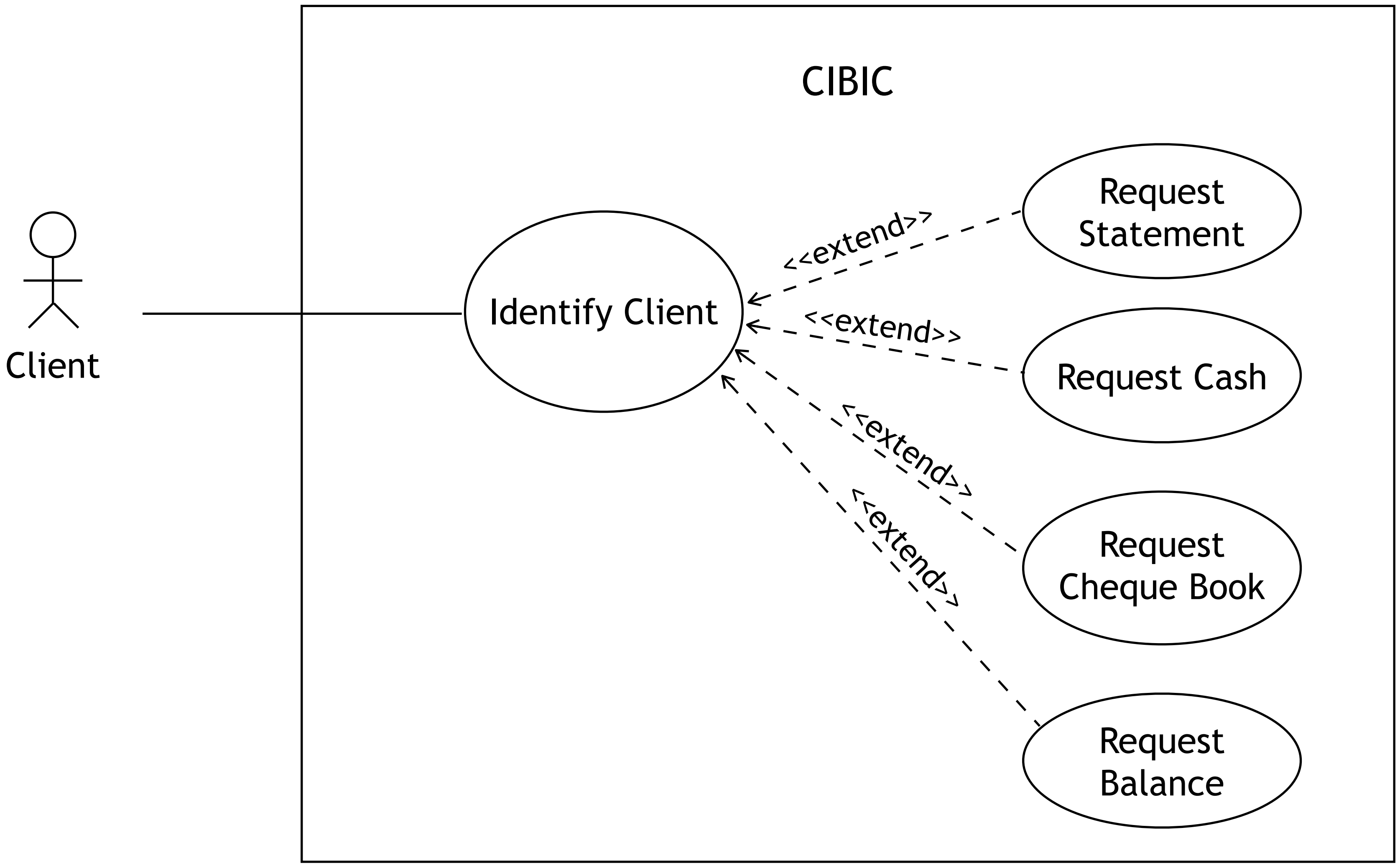
<<extend>>

- A subcase that extends or replaces the end of an existing use case.
- A use-case is similar to another one but does a little bit more
- Put the typical behaviour in one use case and the **extended behaviour** somewhere else
  - Capture the normal behaviour
  - Try to figure out what can go wrong in each step
  - Capture the extended cases in separate use-cases
- Makes it a **lot** easier to understand

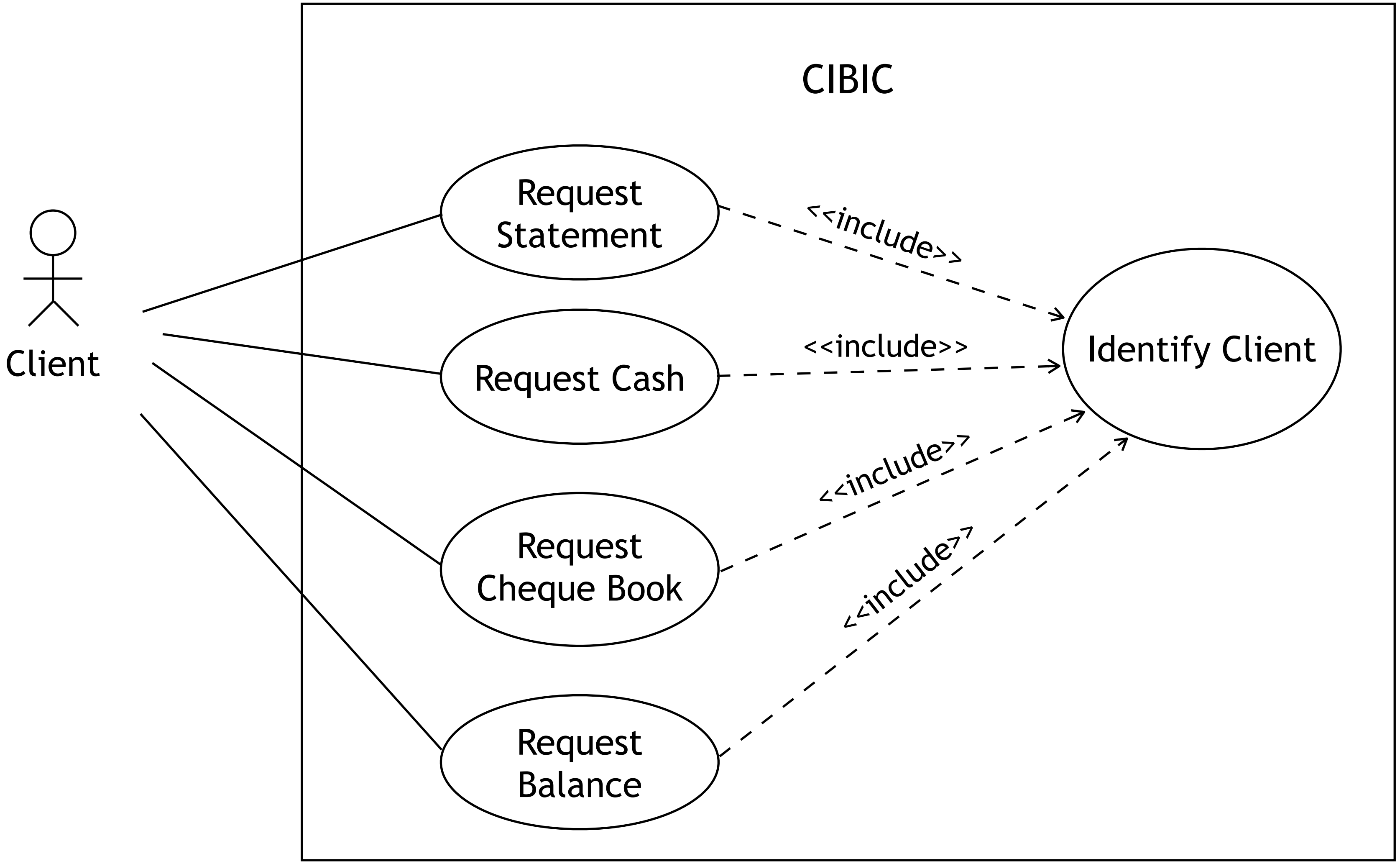
Example



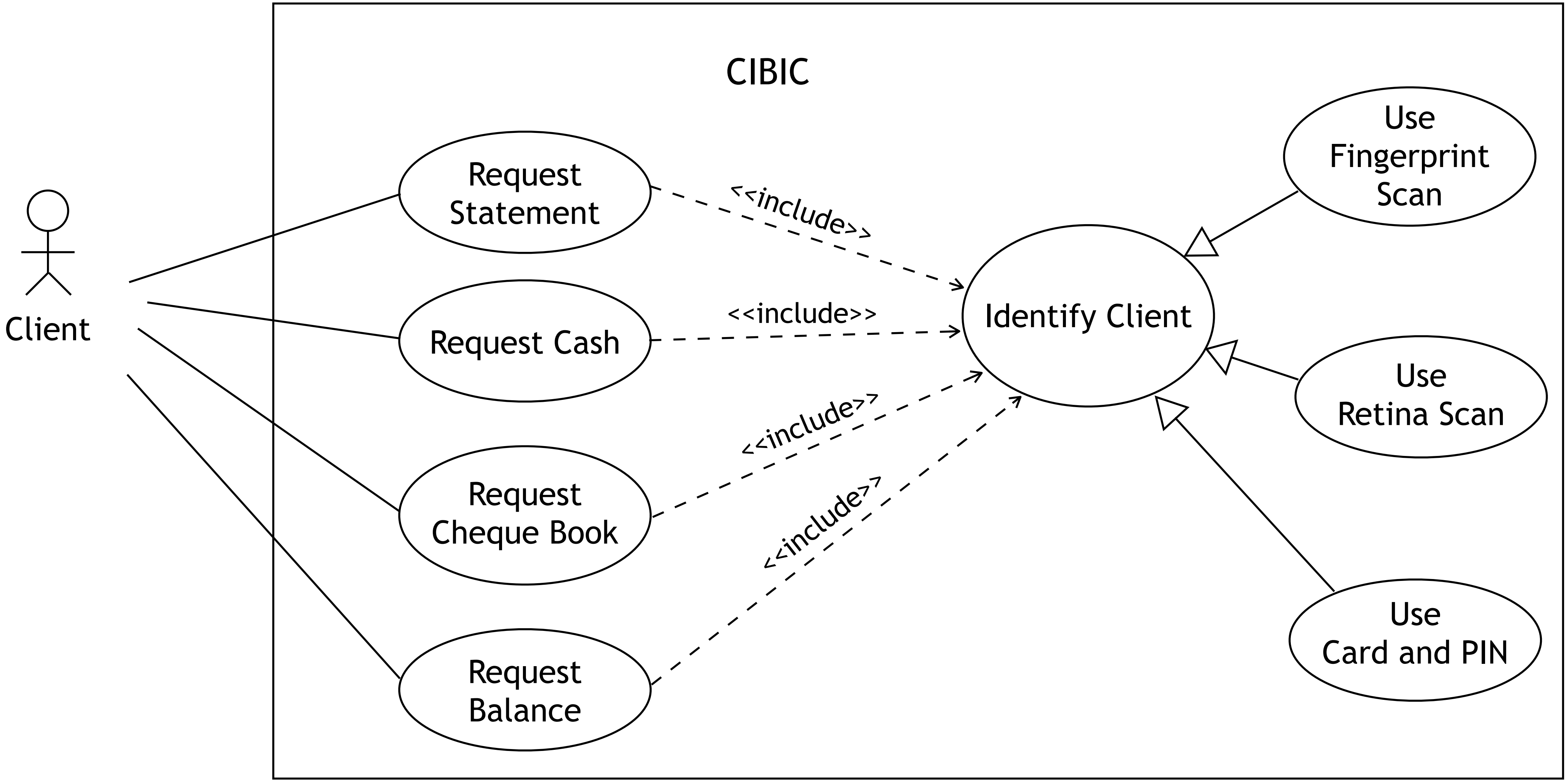
What is wrong with this use case diagram?



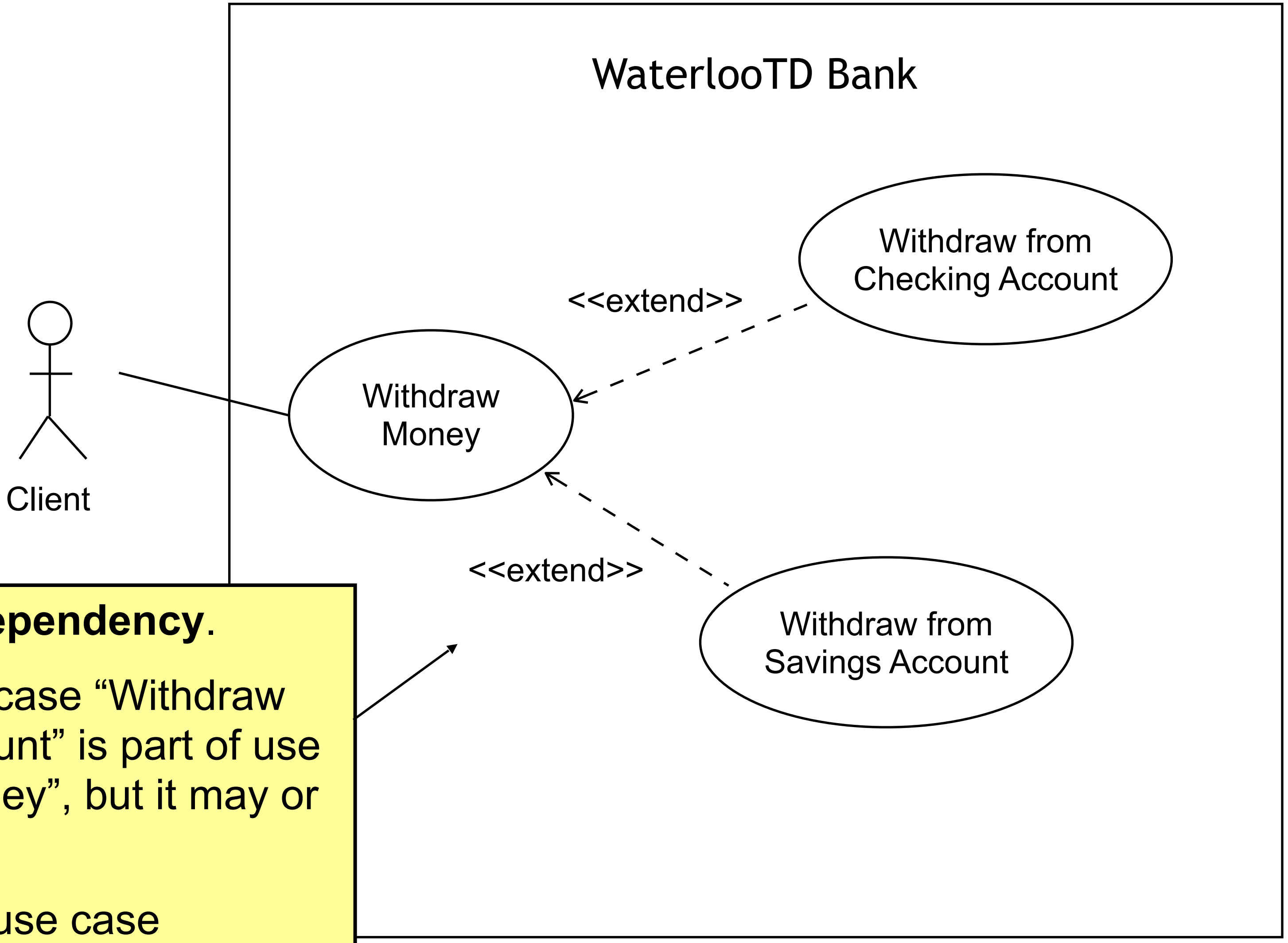
What is wrong with this use case diagram?



What is wrong with this use case diagram?



Another example:

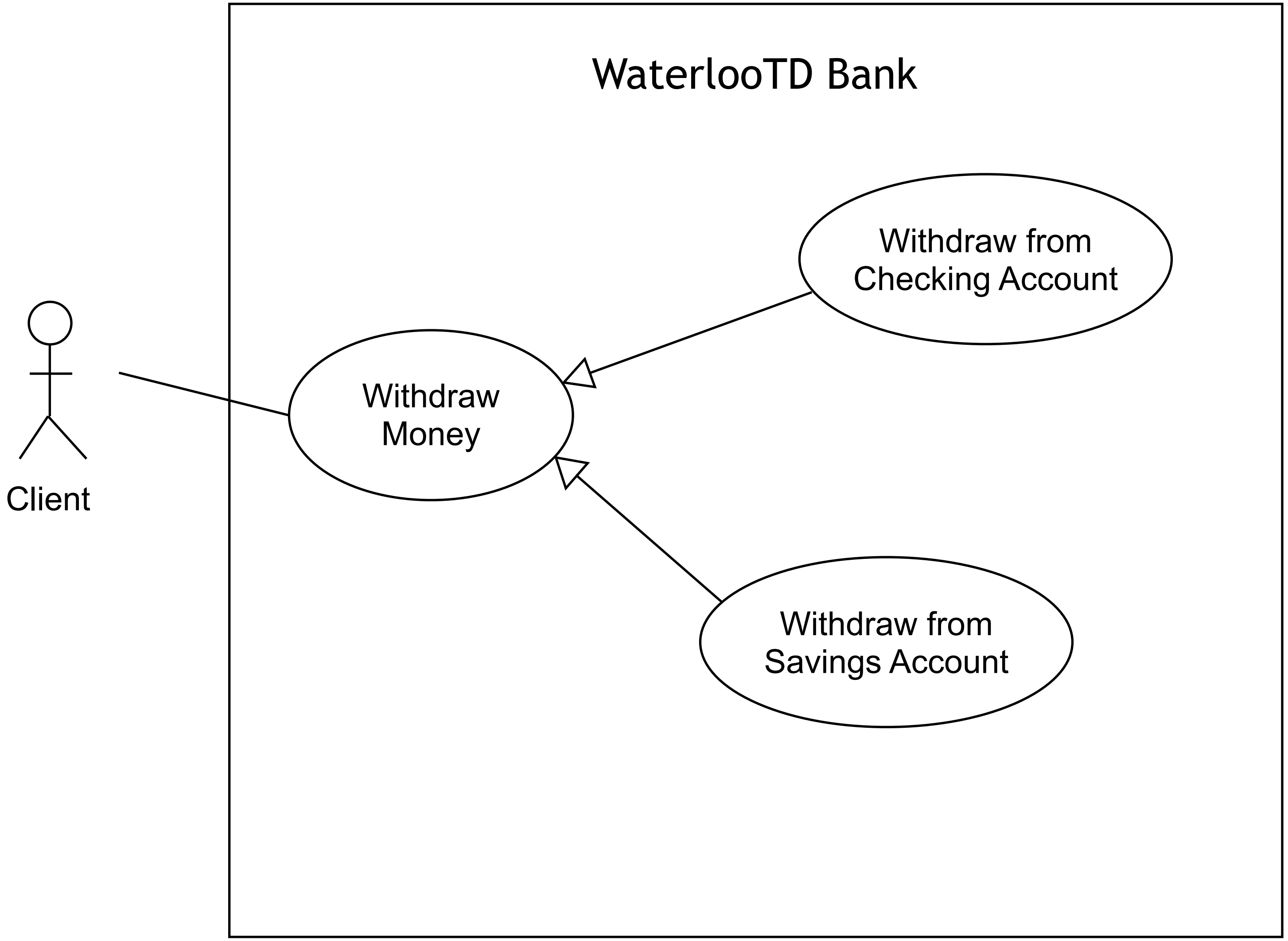


This is an **extend dependency**.

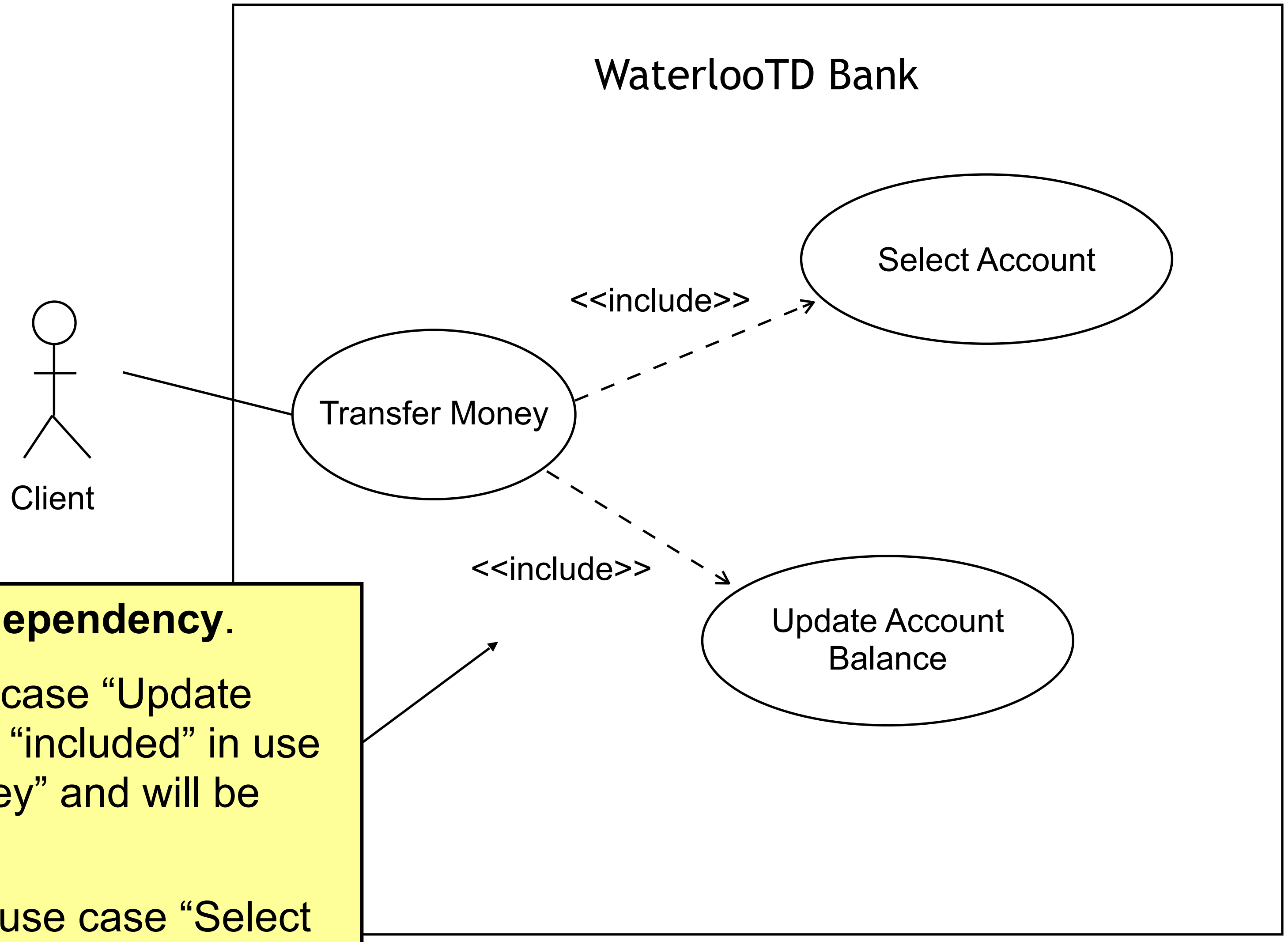
It indicates that use case “Withdraw from Checking Account” is part of use case “Withdraw Money”, but it may or may not be invoked.

The same is true of use case “Withdraw from Savings Account”.

Another example:



Another example:

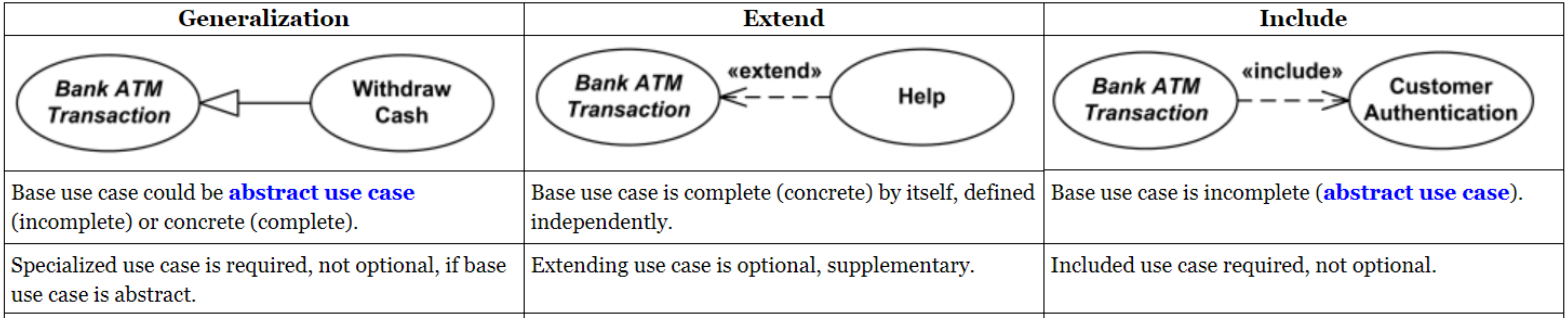


This is an **include dependency**.

It indicates that use case “Update Account Balance” is “included” in use case “Transfer Money” and will be invoked.

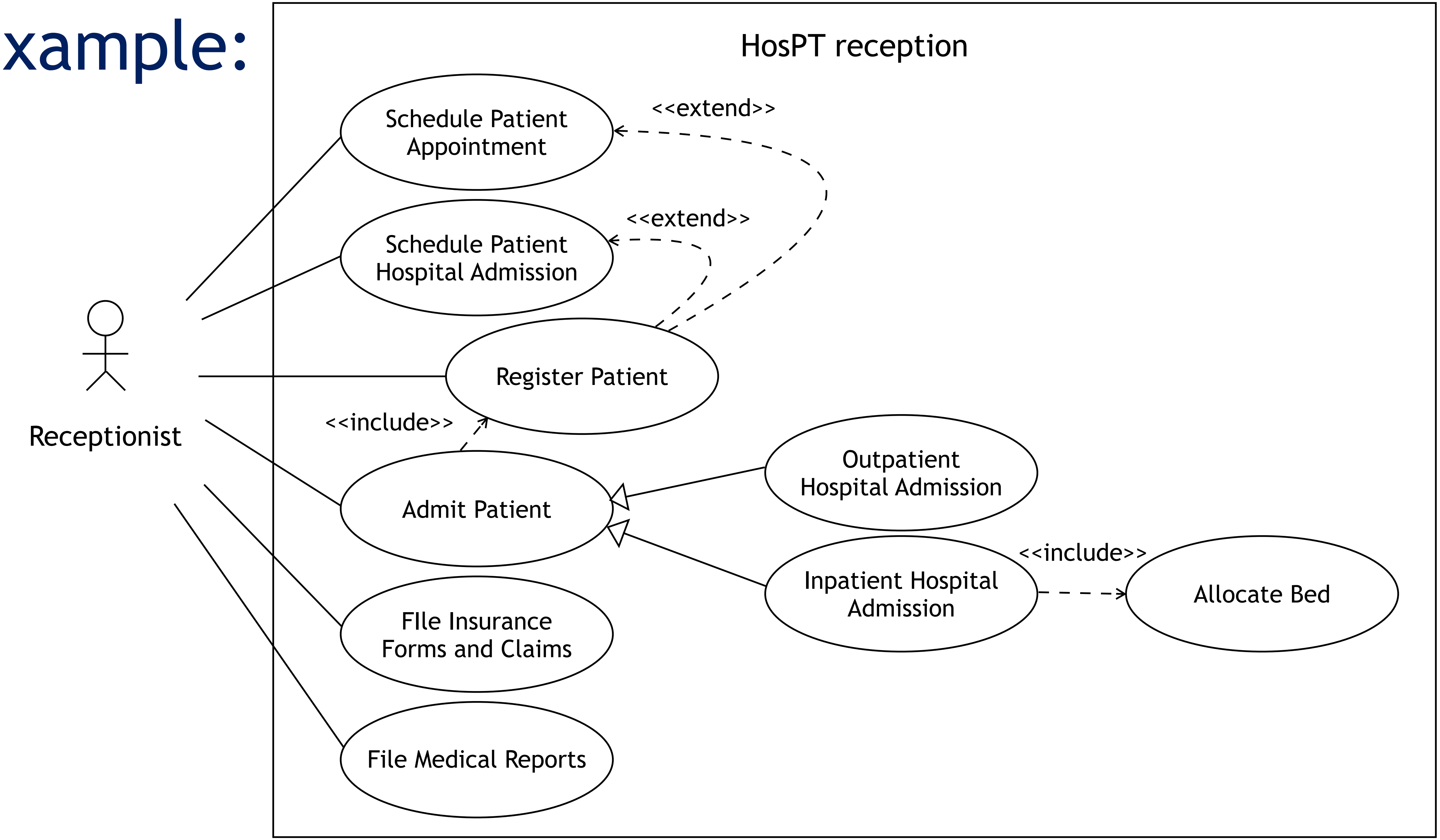
The same is true of use case “Select Account”.

Summary



Generalization	Extend	Include
Bank ATM Transaction Withdraw Cash	*Bank ATM Transaction* «extend» Help	*Bank ATM Transaction* «include» Customer Authentication
Base use case could be **abstract use case** (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (**abstract use case**).
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case required, not optional.

Example:


HosPT reception
Receptionist
Schedule Patient Appointment
Schedule Patient Hospital Admission
Register Patient
Admit Patient
FIle Insurance Forms and Claims
File Medical Reports
Outpatient Hospital Admission
Inpatient Hospital Admission
Allocate Bed
<<extend>>
<<extend>>
<<include>>
<<include>>

Use-Case description

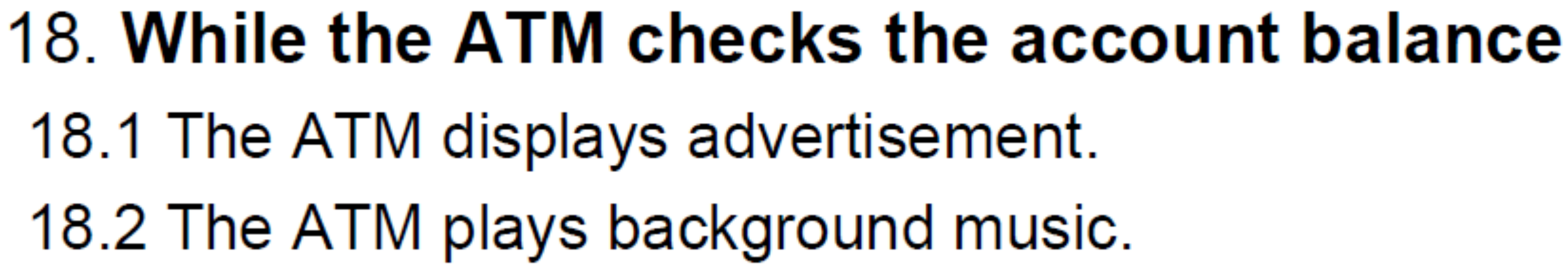
- Describe use cases in a table format.

Name: Order Blood			**ID:** UC25
Authors: Steve Doe **Goal:** Process blood order and payment. **Trigger:** Customer submits blood order payment information. **Preconditions:** Customer is registered in the system. **Notes:**			
Main Scenario:			
Customer	System	Blood Database	Credit Card Authorization System
1. Customer submits blood order.			
	2. Checks availability of blood.		
		3. Requested Blood is available.	
	4. Prompts customer for Payment type: credit or invoice.		

More Complex Actions

- If => Conditional statement
- For => iteration expression
- While => conditional iteration
- Go To UCn

Example:



18. **While the ATM checks the account balance**

18.1 The ATM displays advertisement.

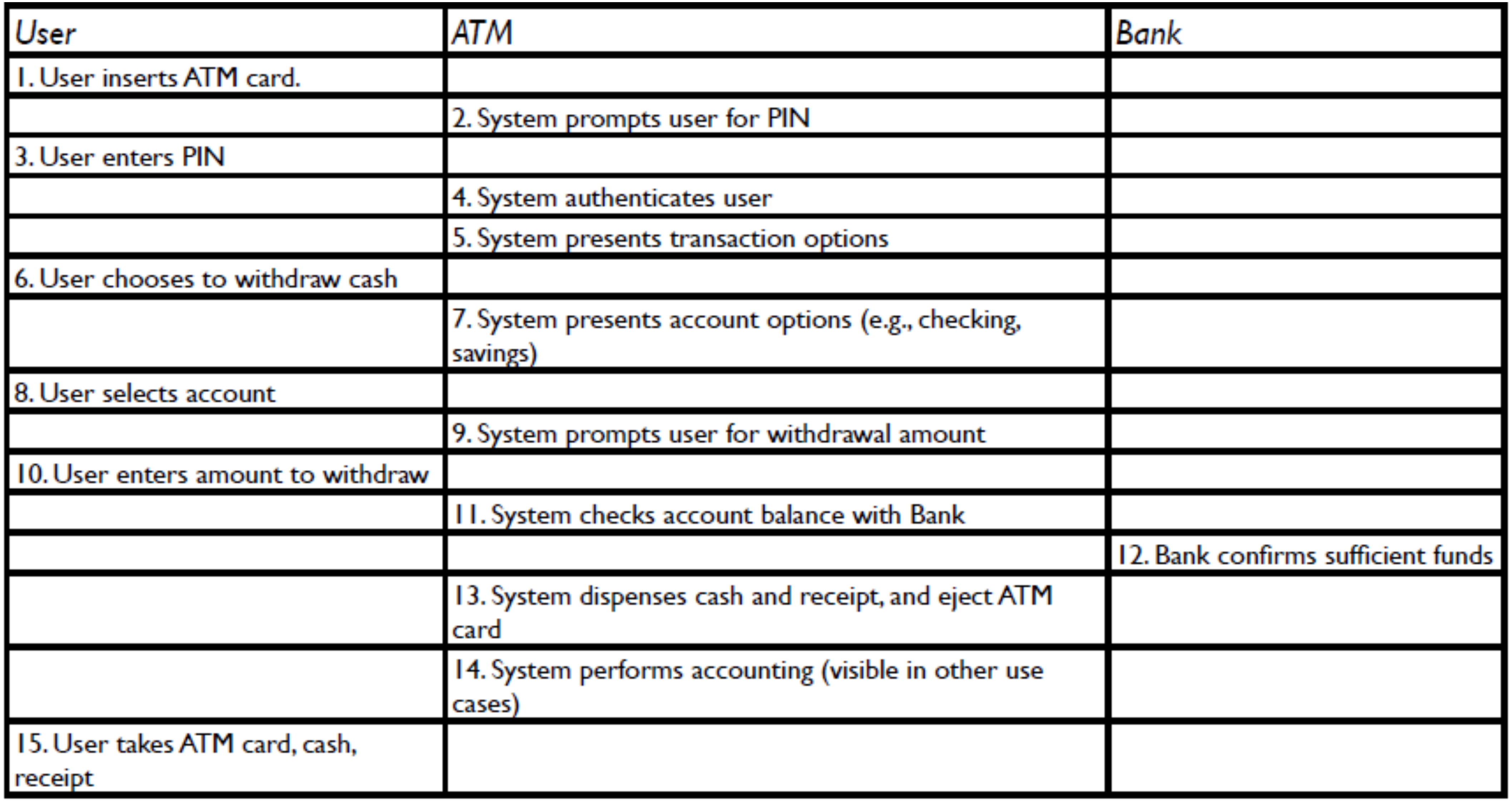
18.2 The ATM plays background music.

These are not needed very often and maybe a sign that the use case is becoming too detailed or too much like pseudo-code.

Scenarios

- Scenario is a one full execution path through a use case, listing only observable actions of the system and actors.
- A single-use case contains many scenarios

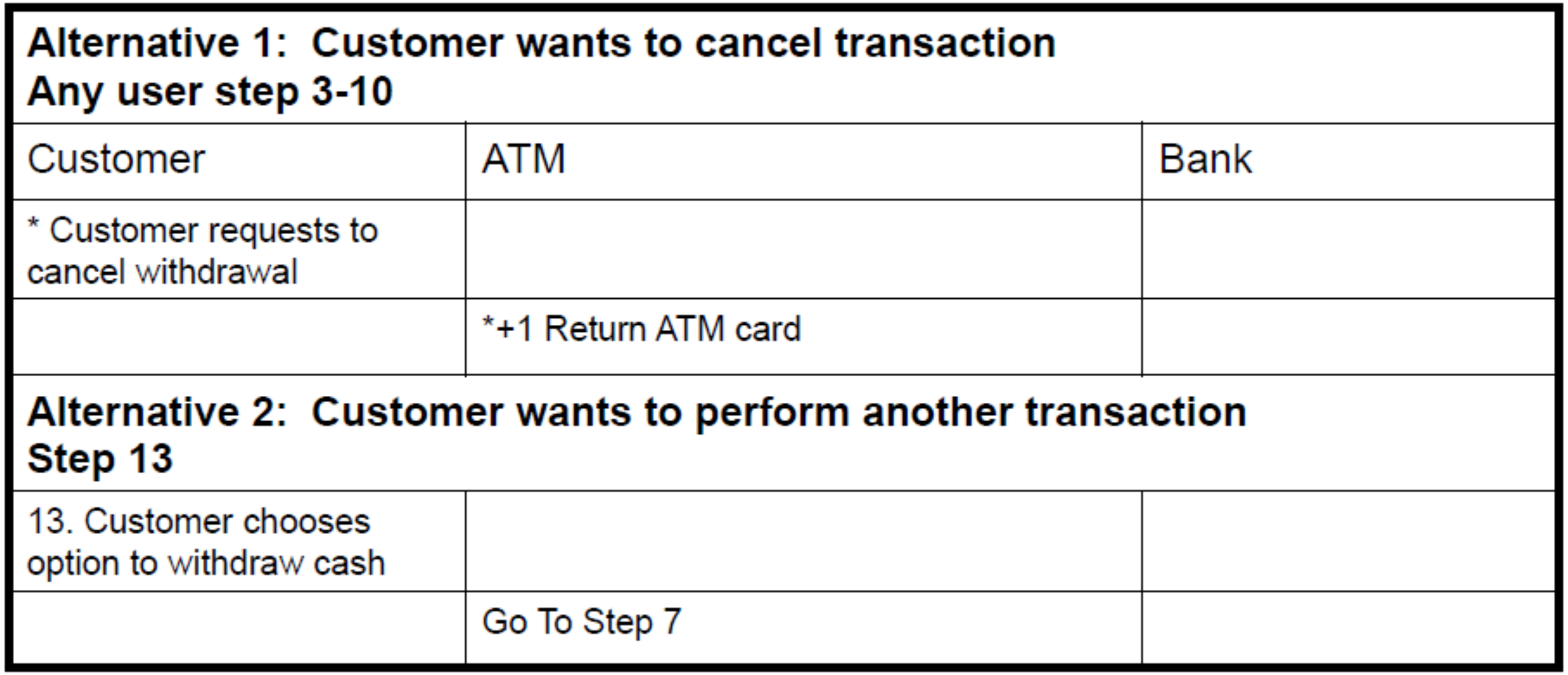
Main Scenario



User	ATM	Bank
1. User inserts ATM card.		
	2. System prompts user for PIN	
3. User enters PIN		
	4. System authenticates user	
	5. System presents transaction options	
6. User chooses to withdraw cash		
	7. System presents account options (e.g., checking, savings)	
8. User selects account		
	9. System prompts user for withdrawal amount	
10. User enters amount to withdraw		
	11. System checks account balance with Bank	
		12. Bank confirms sufficient funds
	13. System dispenses cash and receipt, and eject ATM card	
	14. System performs accounting (visible in other use cases)	
15. User takes ATM card, cash, receipt		

Alternative Scenario

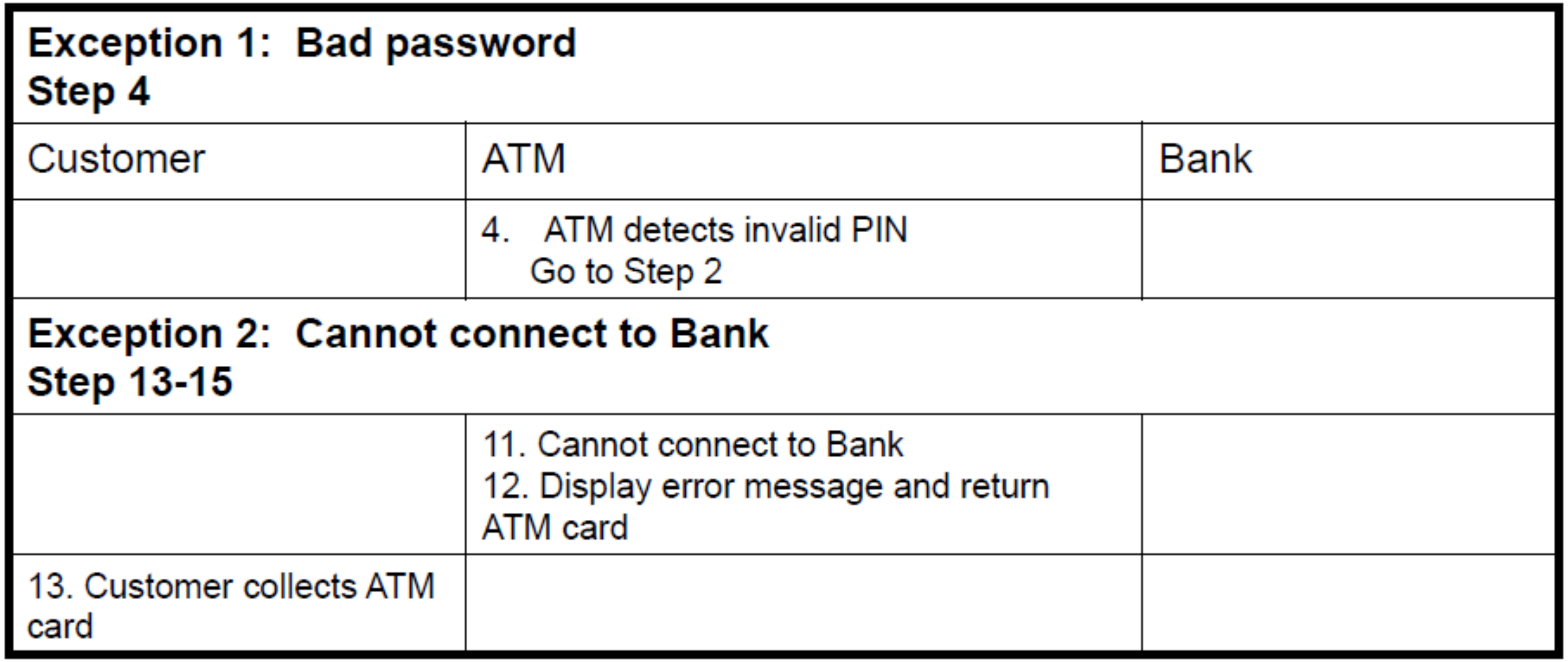
- A sub-case that achieves the primary goal of UC through different sequences of steps/actions



Alternative 1: Customer wants to cancel transaction **Any user step 3-10**		
Customer	ATM	Bank
* Customer requests to cancel withdrawal		
	*+1 Return ATM card	
Alternative 2: Customer wants to perform another transaction **Step 13**		
13. Customer chooses option to withdraw cash		
	Go To Step 7	

Exception

- A sub-case that captures a special case



Exception 1: Bad password **Step 4**		
Customer	ATM	Bank
	4. ATM detects invalid PIN Go to Step 2	
Exception 2: Cannot connect to Bank **Step 13-15**		
	11. Cannot connect to Bank 12. Display error message and return ATM card	
13. Customer collects ATM card		

Use case traps to avoid

- **Too many use cases:**
  - If you are caught in a use case explosion, you might not write them at the appropriate level of abstraction.
  - Do not create a separate use case for every possible scenario.

- **Highly complex use cases:**
  - You cannot control the complexity of the business tasks, but you can control how you represent them in use cases.
  - Select one success path through the use case and call that the main flow. Use alternative flows for the other logic branches that lead to success, and use exceptions to handle branches that lead to failure.
  - You might have many alternatives, but each one will be short and easy to understand.

- **Including design in the use cases:**
  - Use cases should focus on what the users need to accomplish with the system's help, not how the screens will look.
  - Emphasize the conceptual interactions between the actors and the system.
    - For example, say "System presents choices" instead of "System displays a drop-down list."
  - Don't let the UI design drive the requirements exploration.
- **Including data definitions in the use cases:**
  - Store data definitions in a project-wide data dictionary and data model
- **Use cases that users do not understand:**
  - If users cannot relate a use case to their business processes or goals, there is a problem.
  - Write use cases from the user's perspective, not the system's point of view, and ask users to review them.
  - Keep the use cases as simple as possible while still achieving clear and effective communication.

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Use Cases and Scenarios

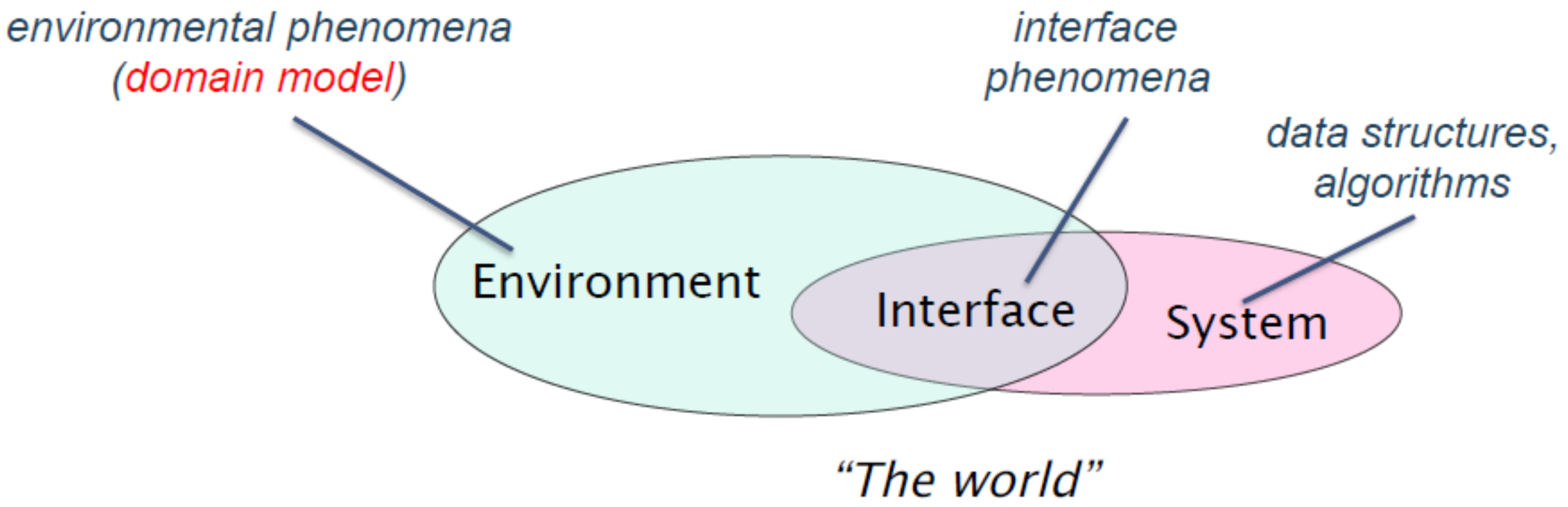
CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Domain Model

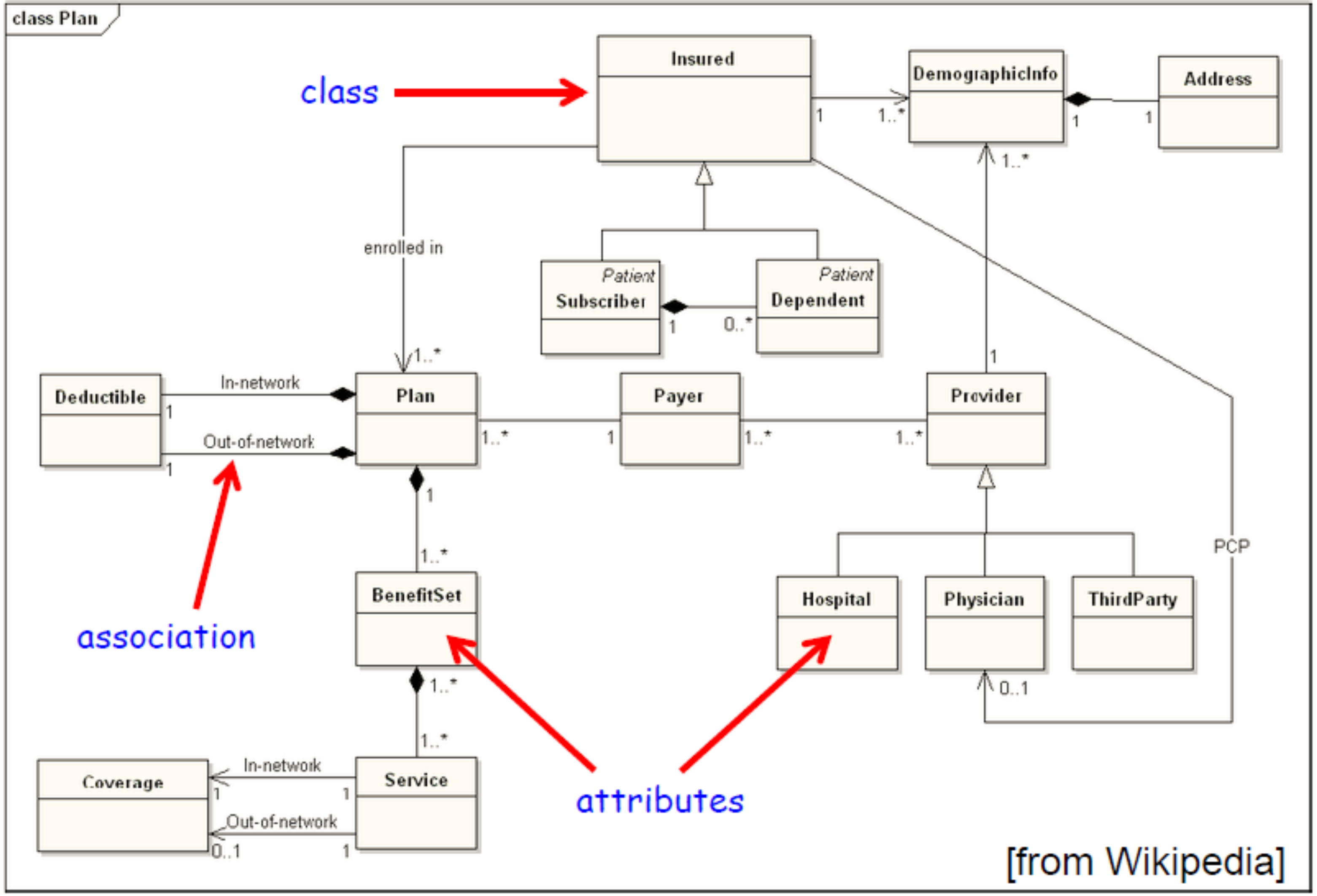
How are class diagrams, and, indeed, how are all of UML, used in requirements engineering to help arrive at a specification of requirements?

Do you remember?



A <u>domain model</u> is a model of the operating environment of our proposed system

Just a quick peek



[from Wikipedia]

The object model

- **Structural** view of the system being modeled (*as-is* or *to-be*)
- Roughly, shows how relevant system concepts are structured and interrelated
- Represented by UML class diagram
  - "objects", classes **not** in the OO design sense: RE is concerned with the *problem world* only!
  - classes with **no** operations: data encapsulation is a design concern; no design decisions here!
- Multiple uses
  - precise definition of system concepts involved in other views, their structure and **descriptive** properties
  - state variables manipulated in other views
  - common vocabulary
  - basis for generating a glossary of terms

Outline

- What is a conceptual object?
- Entities
- Associations and multiplicities
- Attributes
- Specialization
- Aggregation
- More on class diagrams
  - derived attributes, OR associations, associations of associations
- Building object models: heuristic rules

What is a conceptual object?

- Set of instances of a domain-specific concept manipulated by the modelled system. These **instances**
  - are distinctly identifiable
  - can be enumerated in any system state
  - share similar features
    - common name, definition, type, domain properties,
    - common attributes, associations
  - may differ in their individual states and state transitions

Types of conceptual object

1. **Agent**: active, autonomous object
   - instances have individual behavior =

     sequence of state transitions for state variables they control
   - e.g. Patron, Staff; TrainController, TrainDriver
   - represented as UML class (if attributes, associations needed)

2. **Entity**: autonomous, passive object
   - instances may exist in system independently of instances of other objects
   - instances cannot control behavior of other objects
   - e.g. Book, Journal; Train, Platform
   - represented as UML class

3. **Event**: instantaneous object

- instances exist in single system state
- e.g. BookRequest; StartTrain
- represented as UML class (if attributes, associations needed)

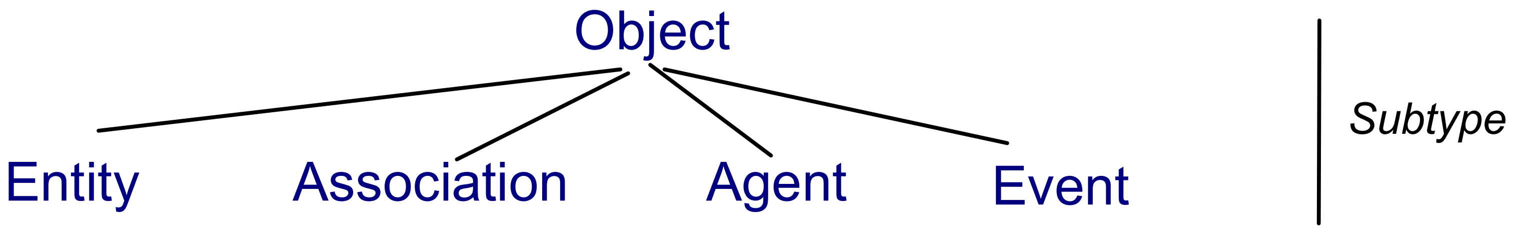
4. **Association**: object dependent on objects it links

- instances are conceptual links among object instances
- e.g. Loan linking Patron and BookCopy

  Copy linking BookCopy and Book

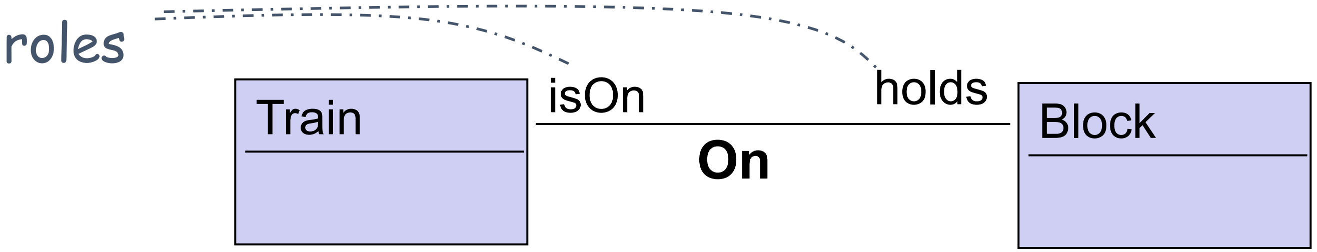
  At linking Train and Platform

  On linking Train and Block
- represented as UML association



Associations

- Association = conceptual object linking other objects, each playing specific role
  - dependent on objects it links
  - linked objects may be entities, associations, events, agents

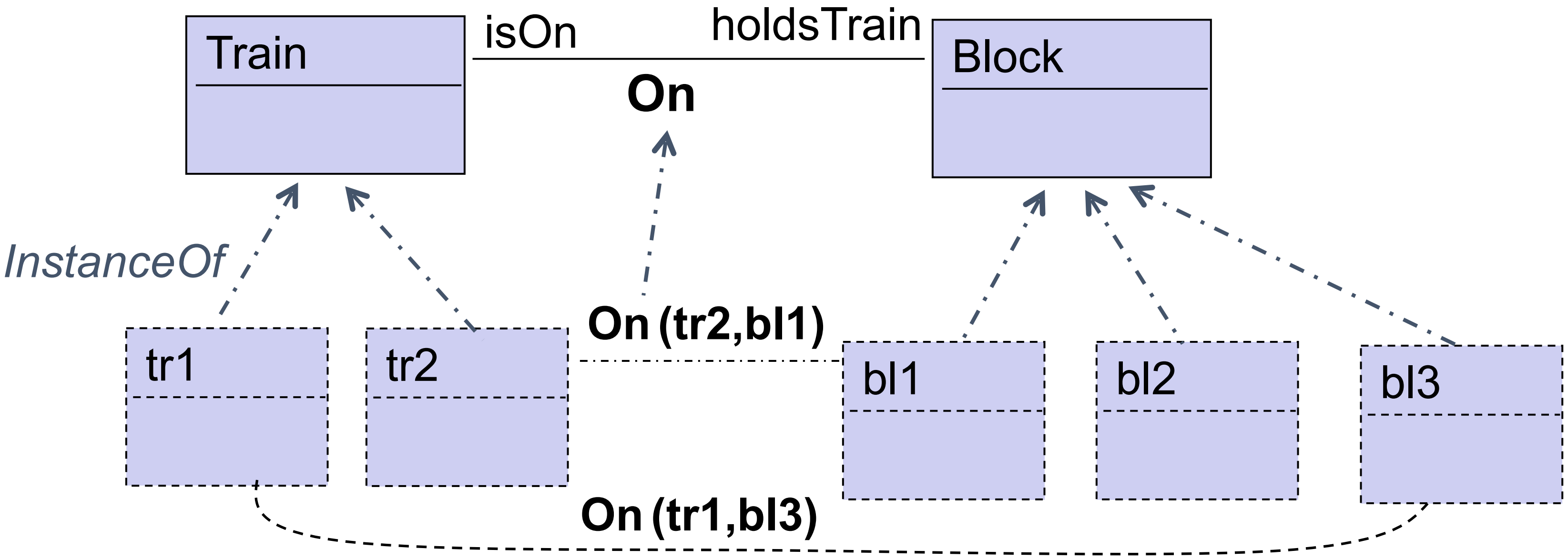


- **Reflexive** association = same object appears under different roles
- Arity of association = number of objects linked by it
  - In 2 slides

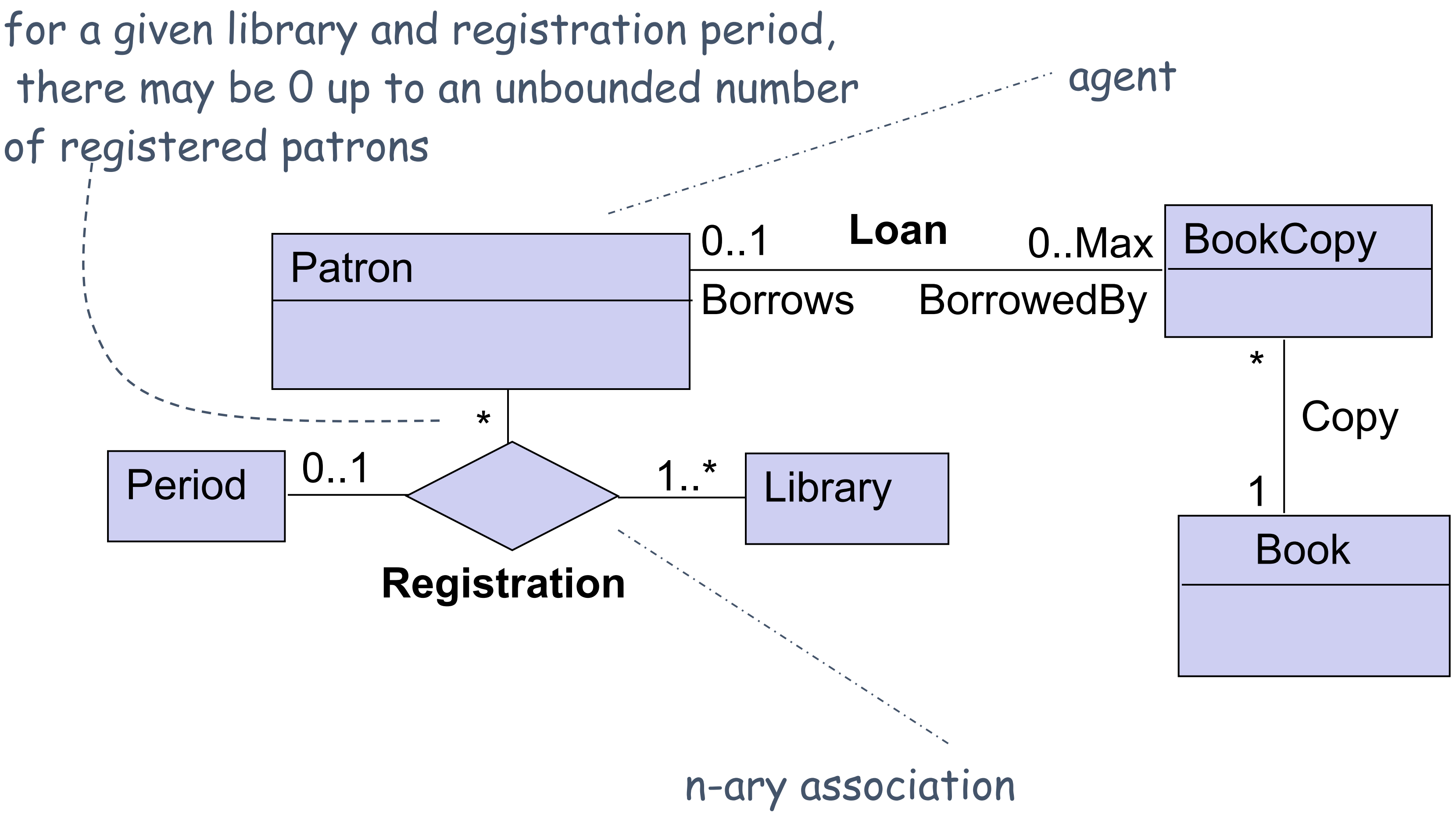
Association instances

- Association instance = tuple of linked object instances,

each playing corresponding role



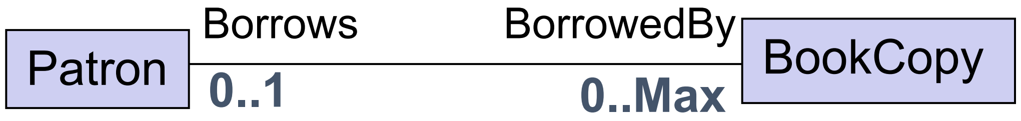
N-ary associations: arity > 2



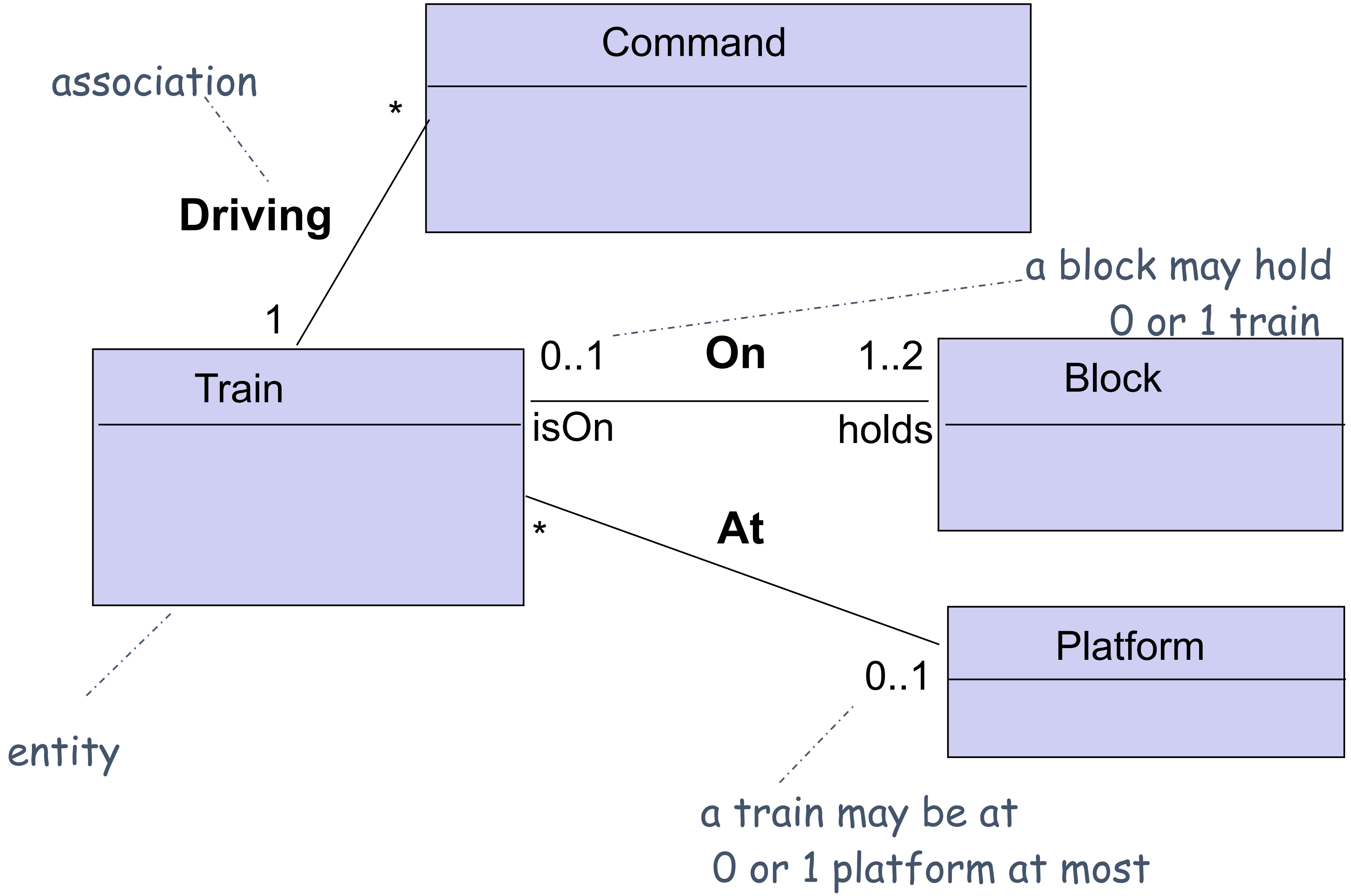
Multiplicities of n-ary association

- From fixed **source** (n-1)-tuple of currently linked instances: **min/max** number of linked **target** instances
    - attached to the role of the **target** instance
- For binary associations, express standard constraints
    - min = 0: optional link (possibly no link in some states)
    - min = 1: mandatory link (at least one link to target in any state)
    - max = 1: uniqueness (at most one link to target in any state)
    - max = *: arbitrary number *N* of target instances linked to source instance, in any state (*N* >=0)

  Notation: "k" for "k..k", "*" for "0..*"

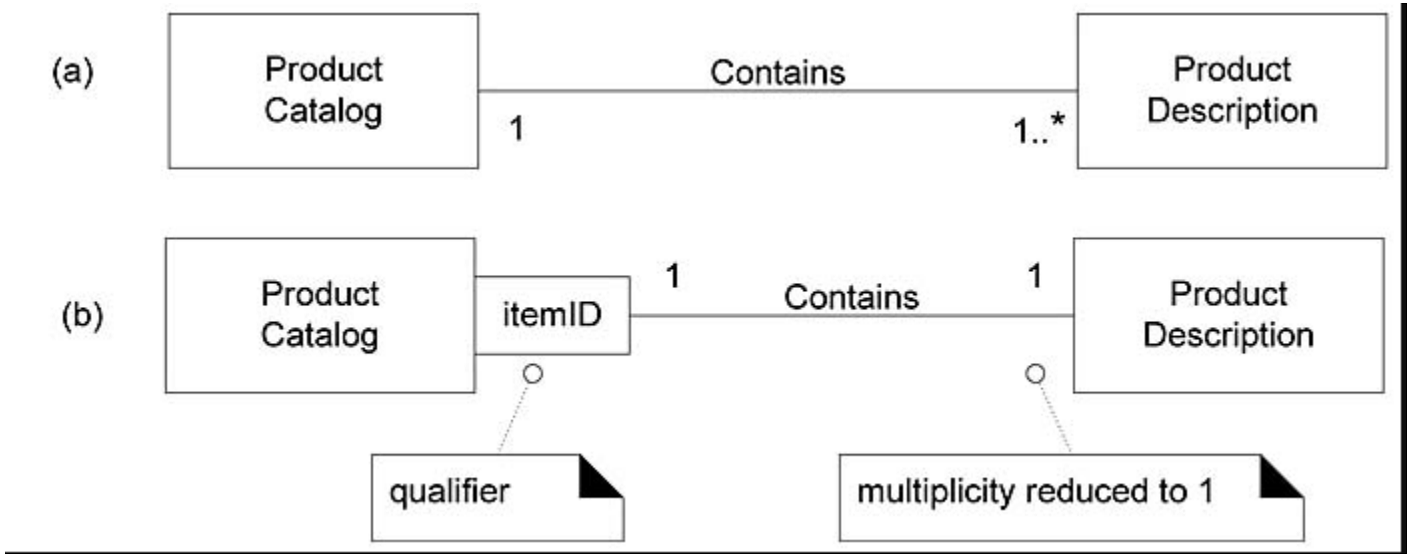


Entities, associations in UML

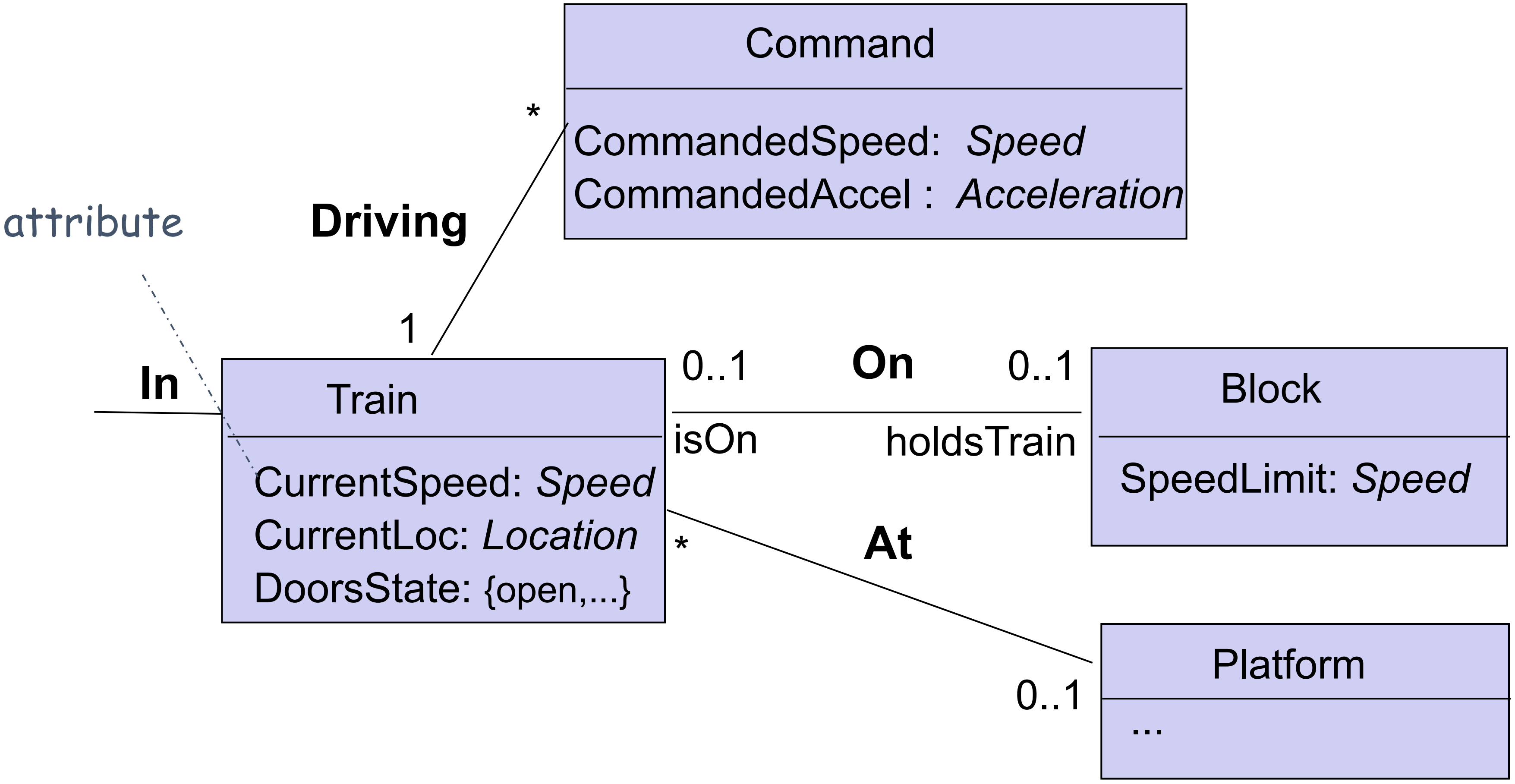


Association Qualifiers

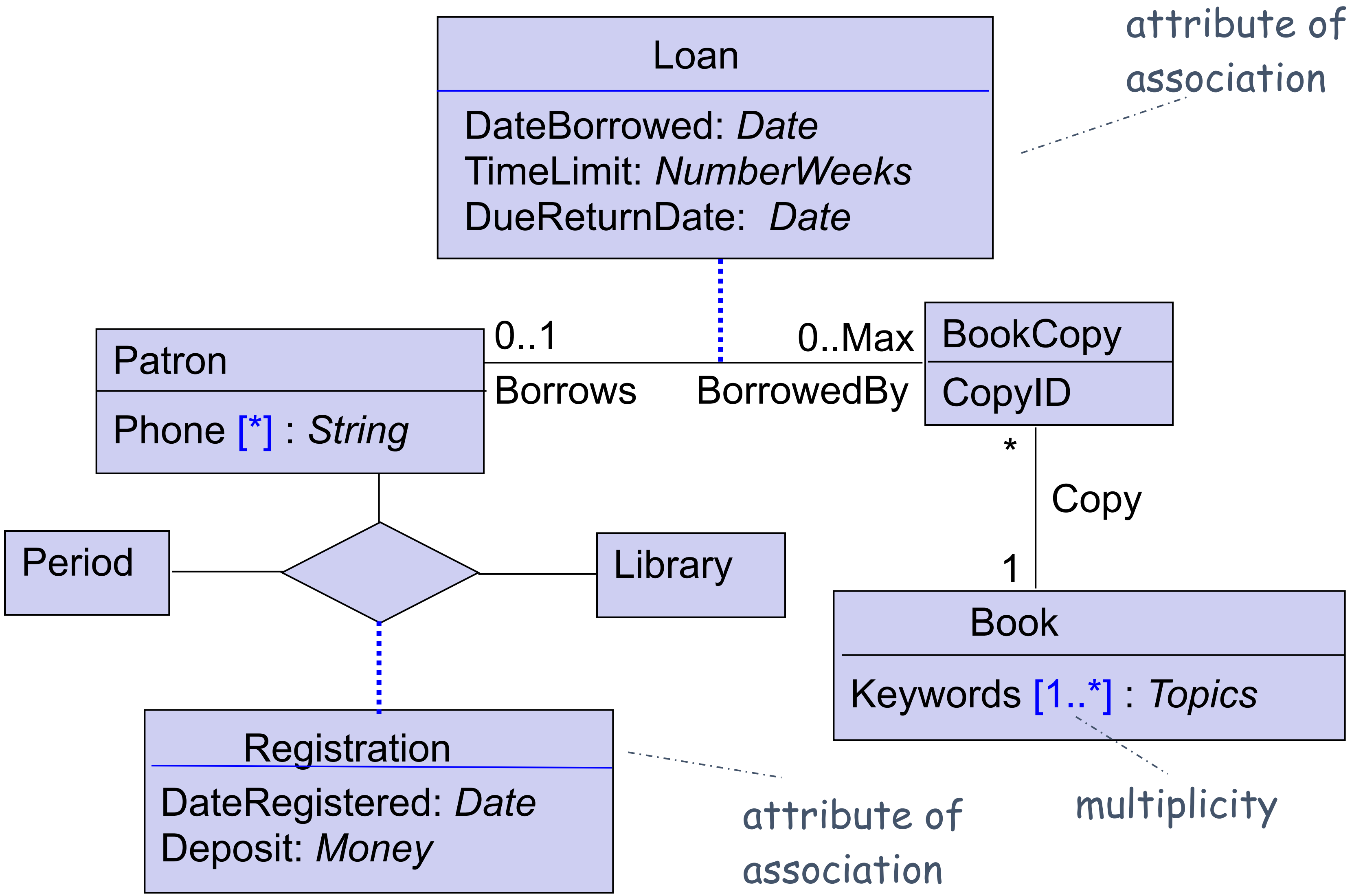
- A qualifier is a unique association used at one end of the association to distinguish among the set of objects at the other end of the association.
  - “uniquely identifies.”



Entities, associations, attributes in UML



Entities, agents, associations, attributes in UML



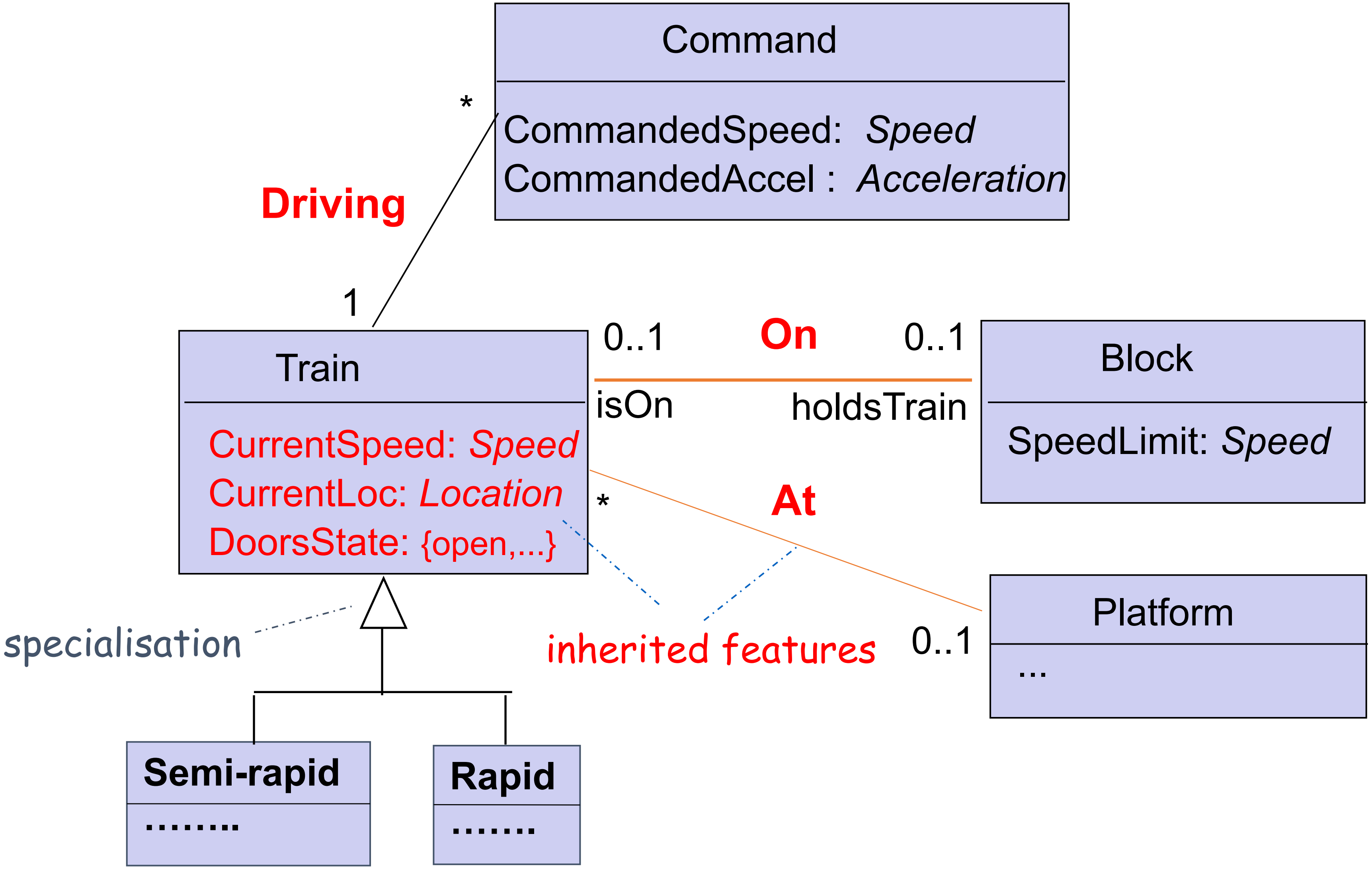
Built-in associations for structuring object models

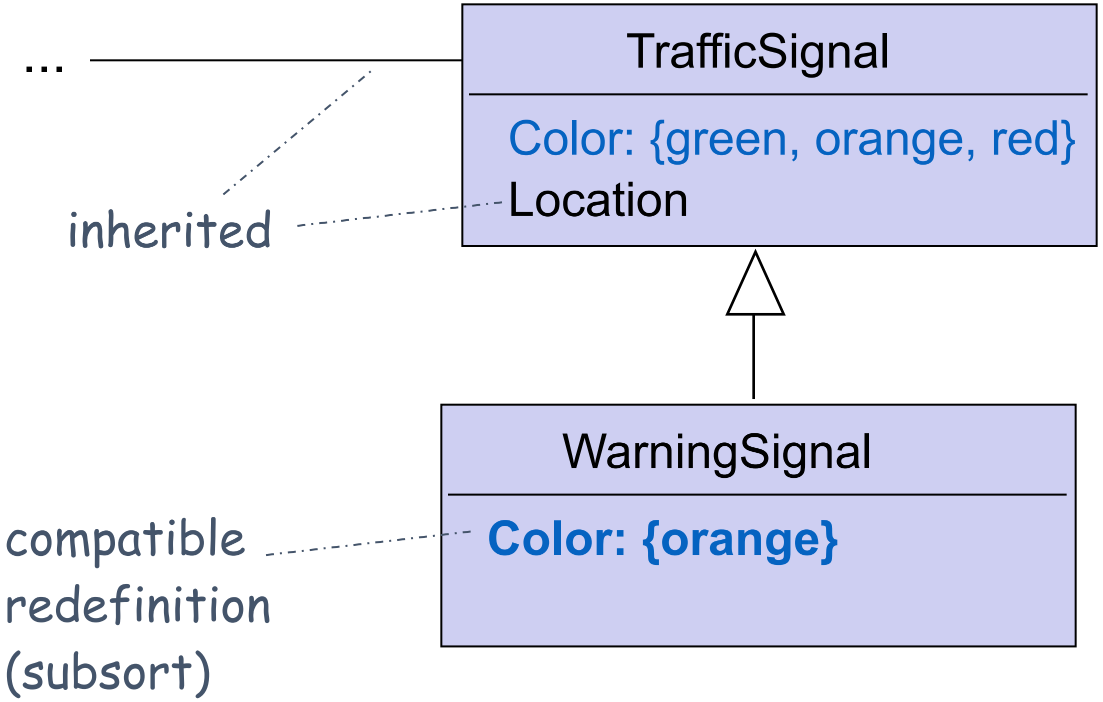
- Object specialization/generalization, decomposition/aggregation
  - applicable to entities, agents, events, associations
- Specialization = subclassing: object *SubOb* is a specialization of object *SuperOb* **iff** for any individual *o*:

  InstanceOf (*o*, SubOb) => InstanceOf (*o*, SuperOb)

  - SubOb **specializes** SuperOb, SuperOb **generalizes** SubOb
  - amounts to set inclusion on set of current instances
- **Feature inheritance** as a consequence
  - by default, *SubOb* inherits from *SuperOb* all its attributes, associations, domain properties while have its own distinguishing features
  - may be inhibited by compatible redefinition of feature with same name within specialized SubOb (“override”)

Example: Object specialization with inheritance



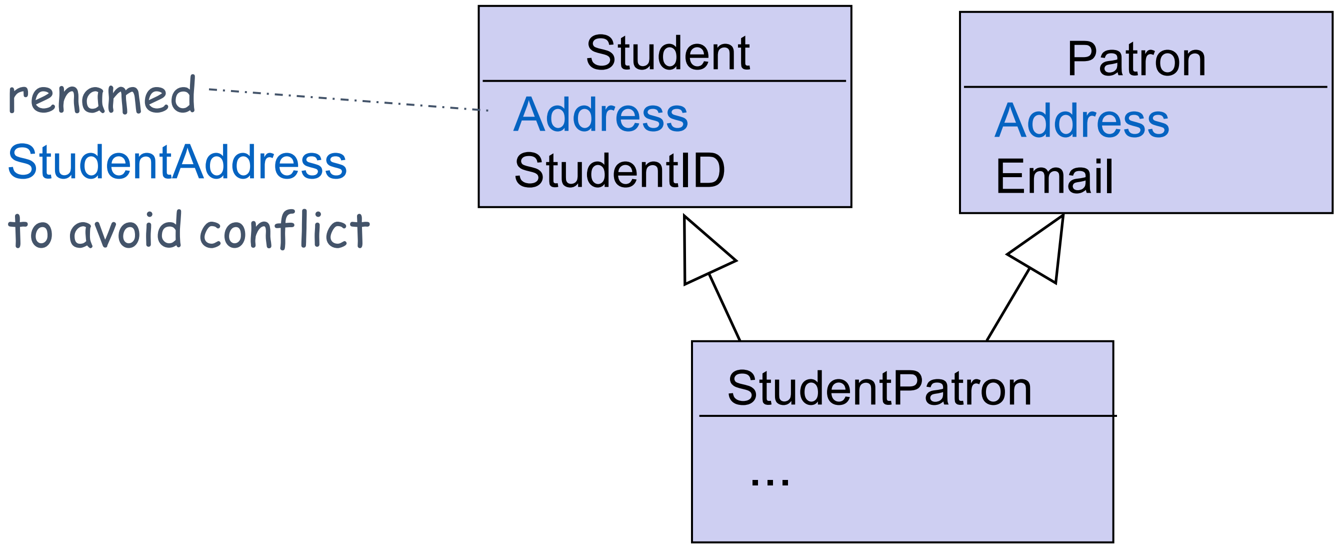


The more specific feature always overrides the more general one.

Multiple inheritance

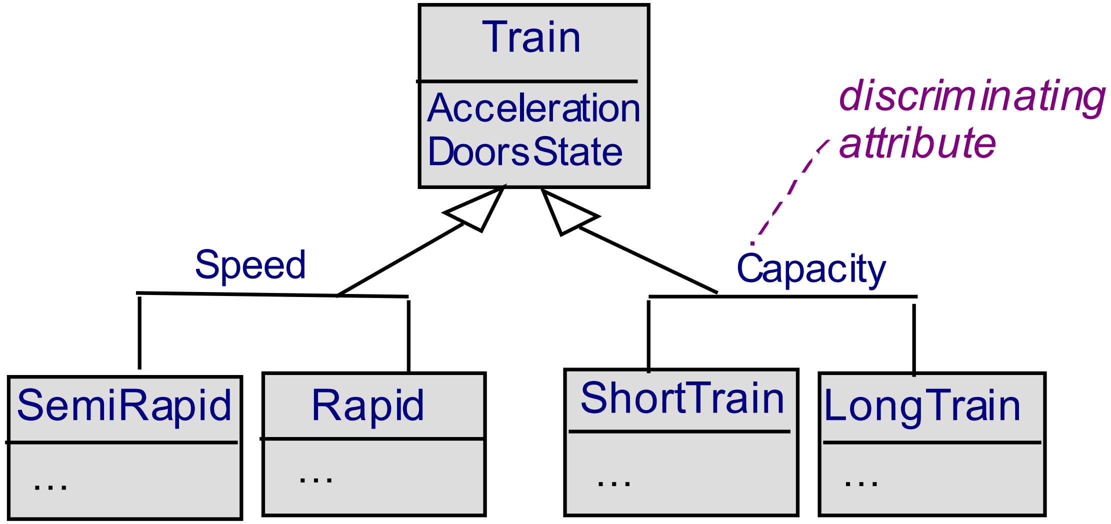
- The Same object may be a specialization of multiple super-objects
  - by default, inheritance of all features from all super-objects
- Can result in inheritance conflicts
  - different features with the same name inherited from different super-objects

  => conflicting features first renamed to avoid this

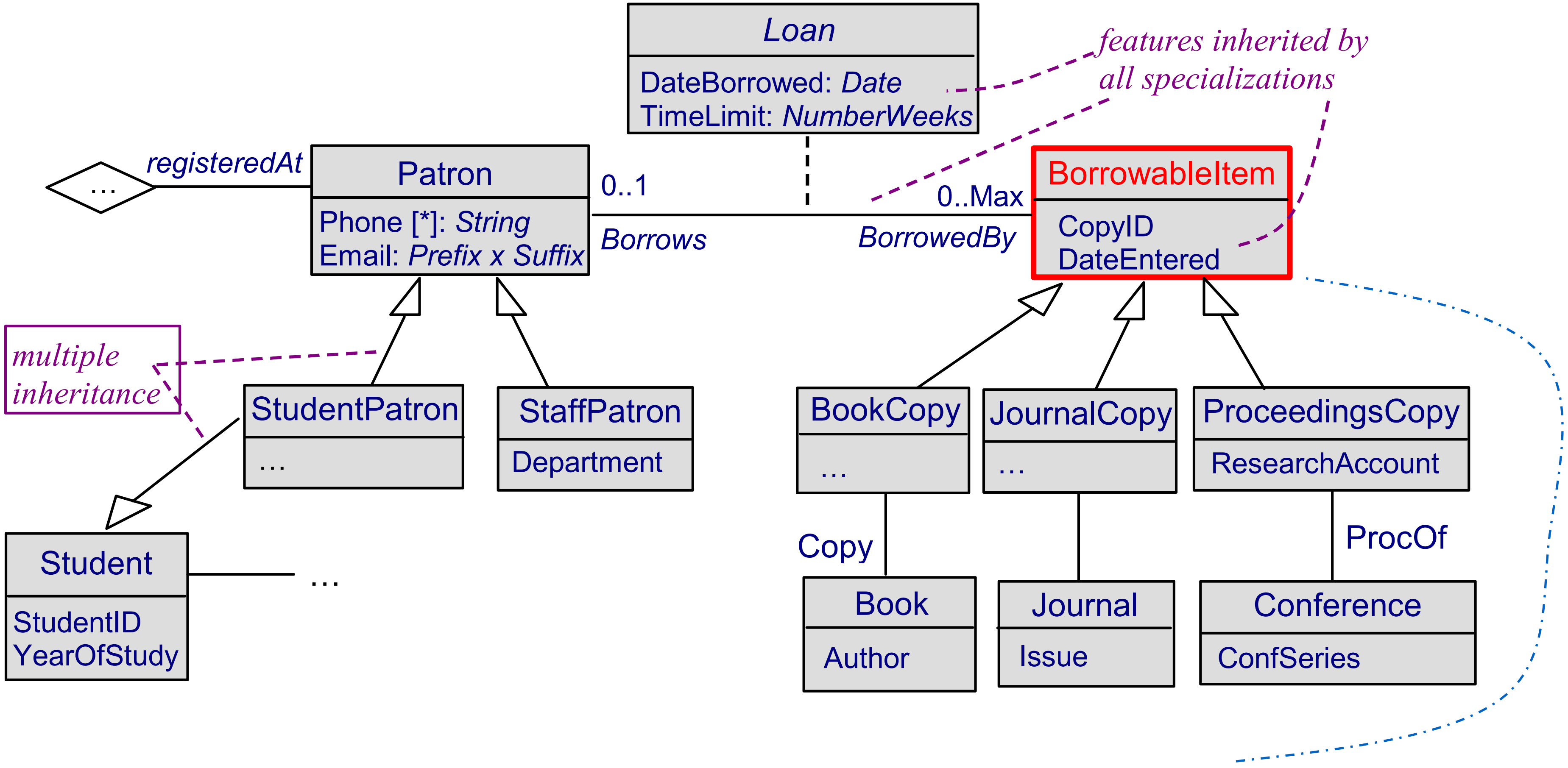


Multiple specializations

- The Same object may have multiple specializations
  - Different subsets of object instances associated with different criteria
  - Same object instance may be a member of different subsets (one per criterion)
- **Discriminator** = attribute of super-object whose values define different specializations (differentiation criterion)



Object generalization

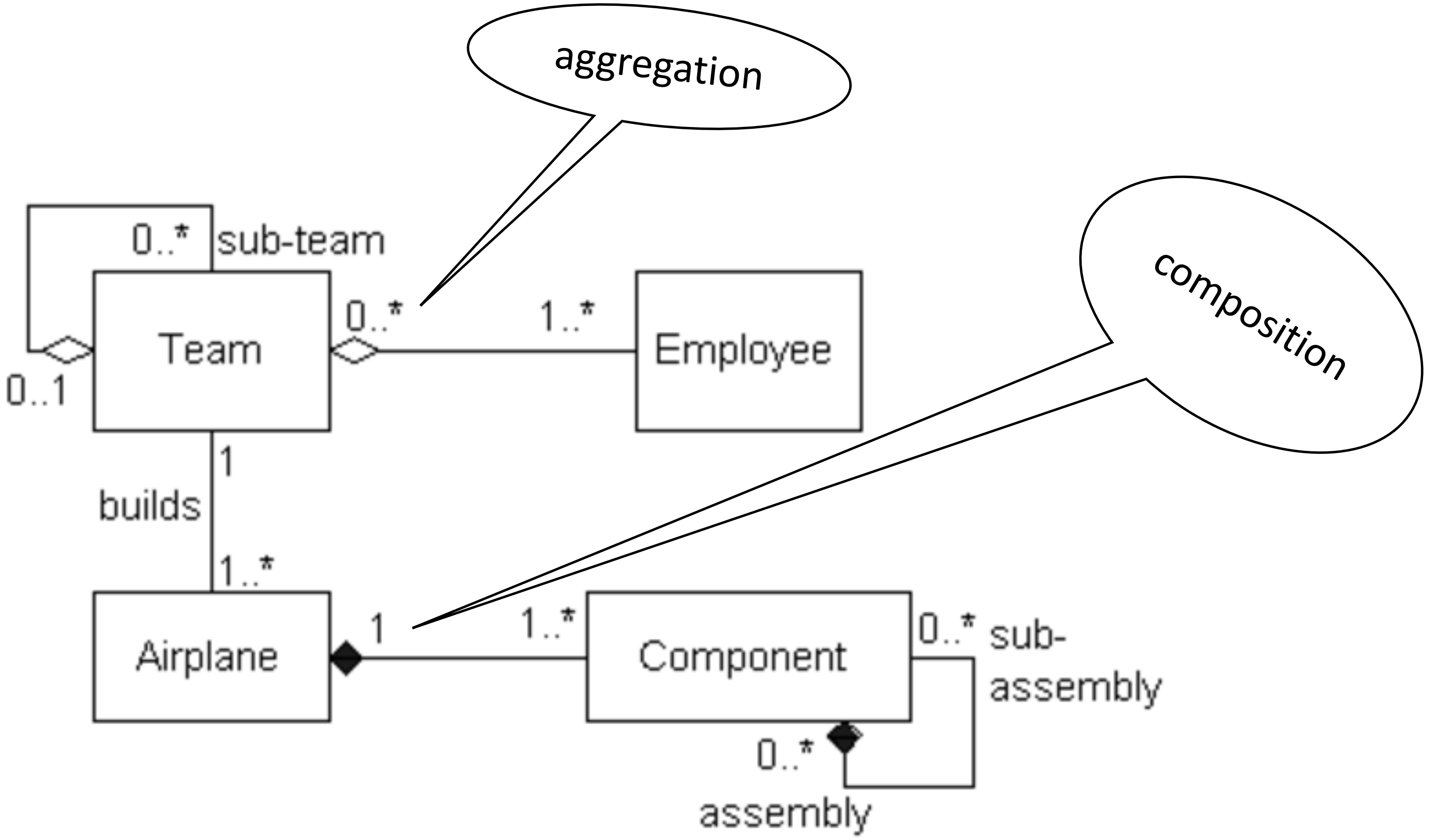


generalization is
not necessarily apparent
in problem world

Object aggregation/composition

- **Aggregation**: an object may belong <u>weakly</u> to several containers
  - A has an aggregation relationship with B and C if they are parts of A
- **Composition**: an object may <u>strongly</u> be a part of at most one container
- Strong form of aggregation
  - Parts only belong to one whole
  - If the whole is deleted, parts get deleted
- Fuzzy distinctions between
  - Aggregation
  - Composition
  - Association

Examples of aggregation and composition

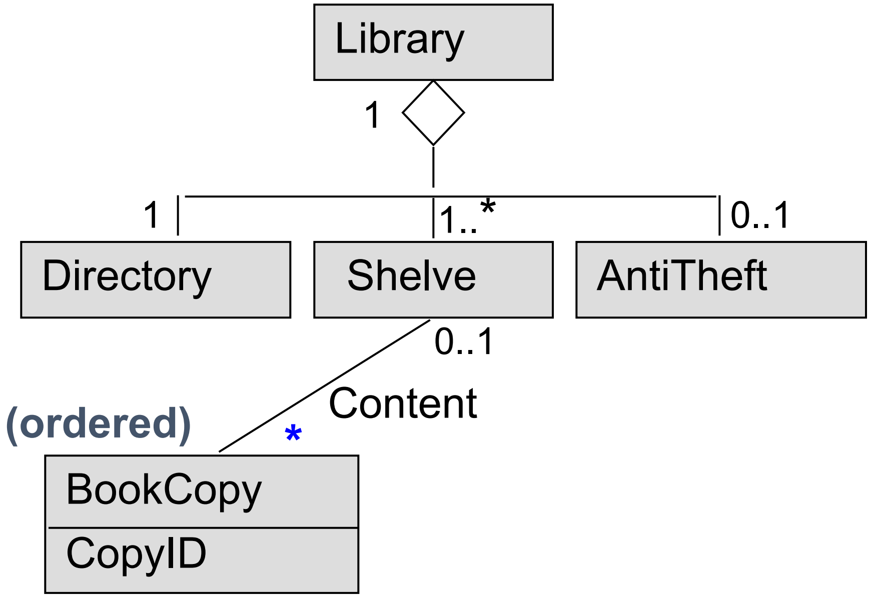


Tips:

1. You Should Be Interested In Both The Whole And The Part
2. Depict the Whole to the Left of the Part
3. Apply Composition to Aggregates of Physical Items
4. Apply Composition When the Parts Share The Persistence Lifecycle With the Whole
5. Don't Worry About Getting the Diamonds Right ☺

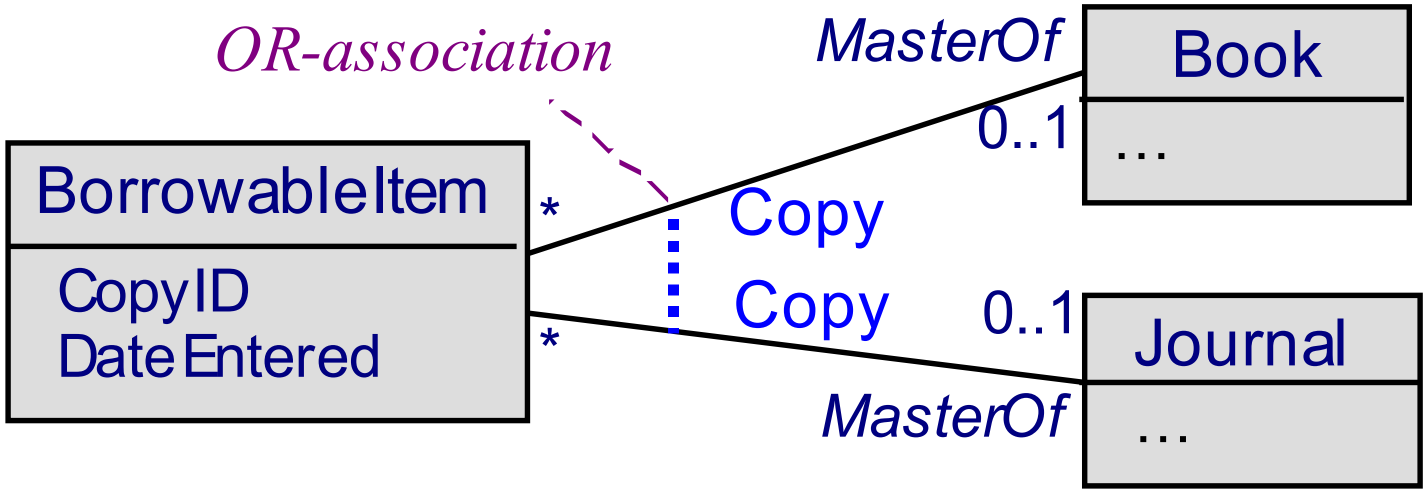
More on UML class diagrams

- **Ordered** association: multiple target instances from the source instance (or tuple of instances) are ordered.



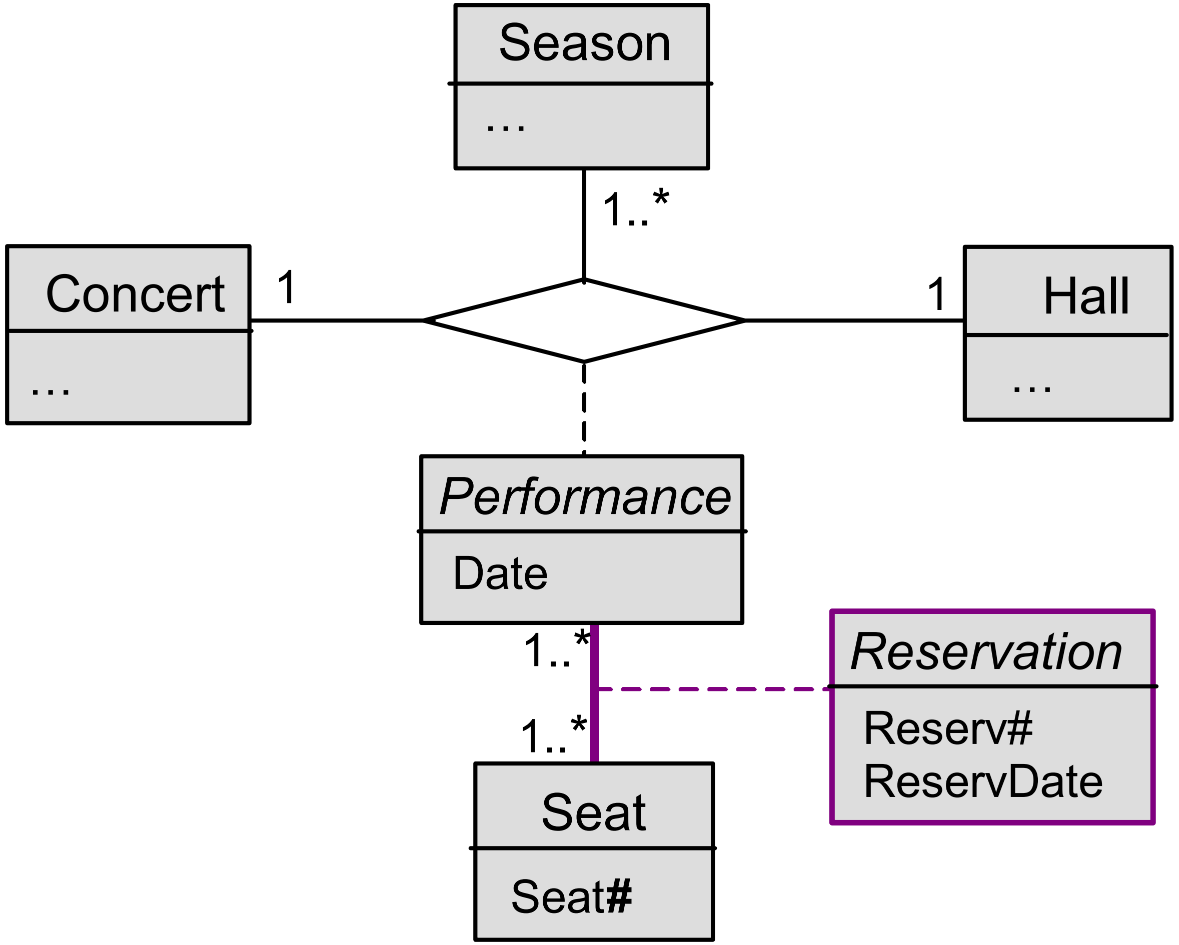
More on UML class diagrams

- **OR association** = same role played by alternative objects
  - set of object instances in this role =
    union of alternative sets of object instances



More on UML class diagrams

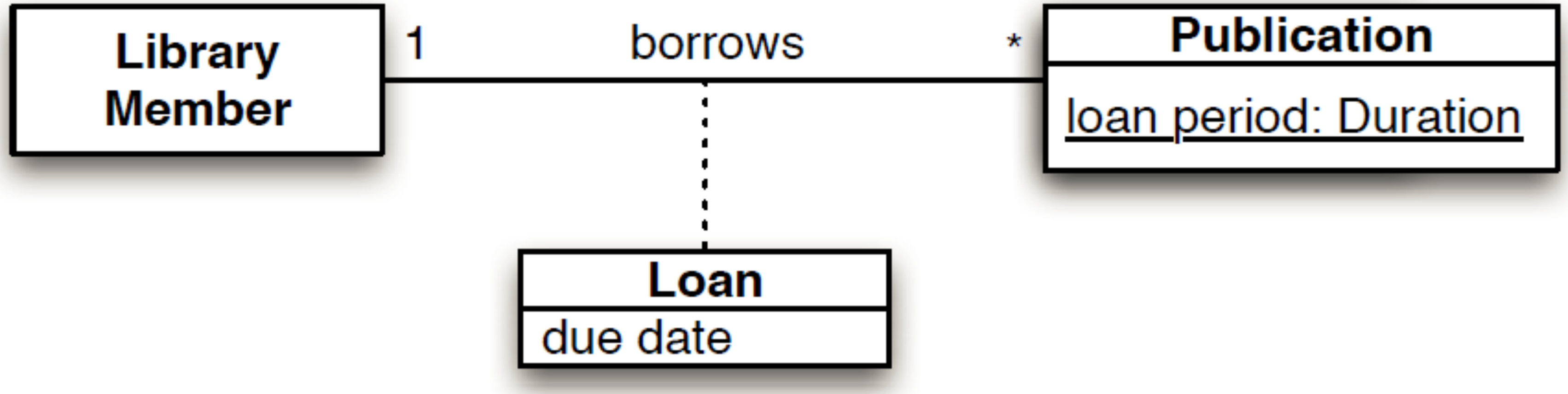
- **Association of associations**: one of the linked objects is an association



By default, every object has its own copy of attributes, and may have its own attribute values.

A class-scope attribute is an attribute whose value is changeable, but is shared by all of the class's object instances. Thus, all objects will have the same class-scope attribute value.

Syntax: underlined attribute declaration



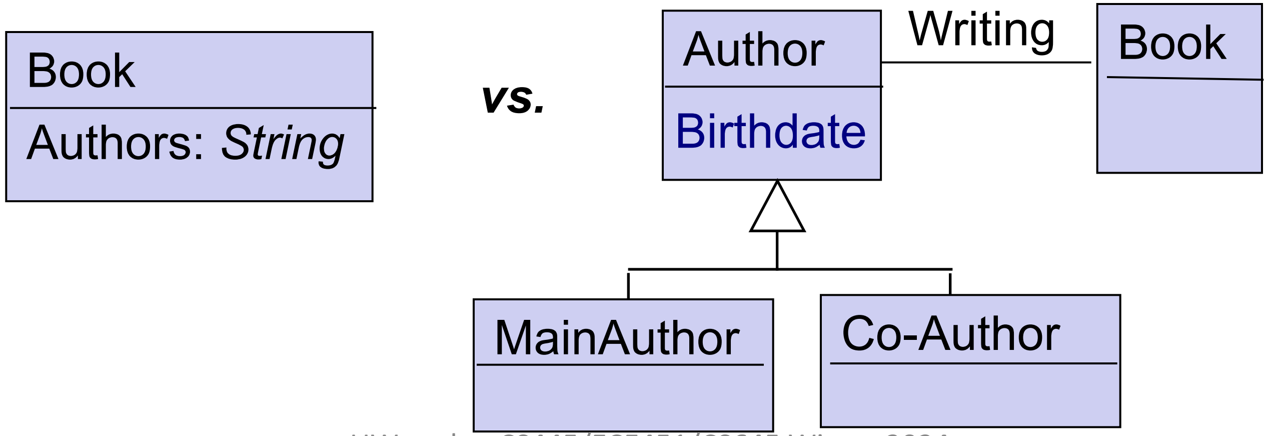
Building object models: heuristic rules

- Deriving pertinent and complete object models from goal models
    - deriving objects, associations, attributes
- From goal model to object model
    - Object or attribute?
    - Entity, association, agent, or event?
    - Bad smells

Object *or* attribute ?

For *X*: conceptual item in goal specs, make *X* an **attribute** if

- *X* is a function: yielding one single value (possibly structured) when applied to conceptual instance
- instances of *X* need not be distinguished
- you don't want to attach attributes/associations to *X*, specialize it, or aggregate/ decompose it
- its range is not a concept you want to specialize or attach attributes/associations



Entity, association, agent, *or* event ?

For *X*: conceptual object in goal specs

- instances of *X* are defined in one single state

  ⇒ event e.g. StartTrain

- instances of *X* are active: control behaviors of other object instances

  ⇒ agent e.g. DoorsActuator

- instances of *X* are passive, autonomous

  ⇒ entity e.g. Train

- instances of *X* are passive, dependent on other, linked object instances

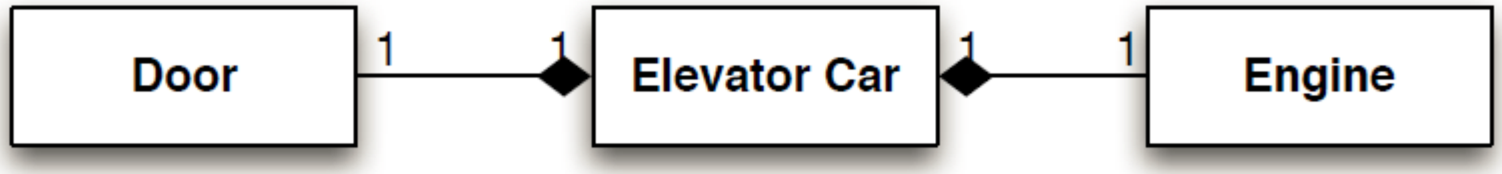
  ⇒ association e.g. Following (Train, Train)

  N-ary if each of the N parties ...

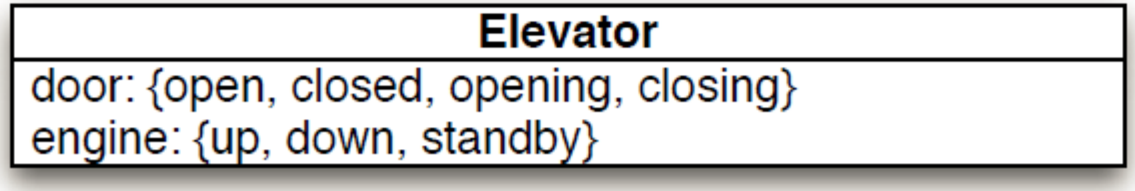
  - need be considered as objects
  - yields tuples to be distinguished

Building object models: bad smells

Distinct classes:

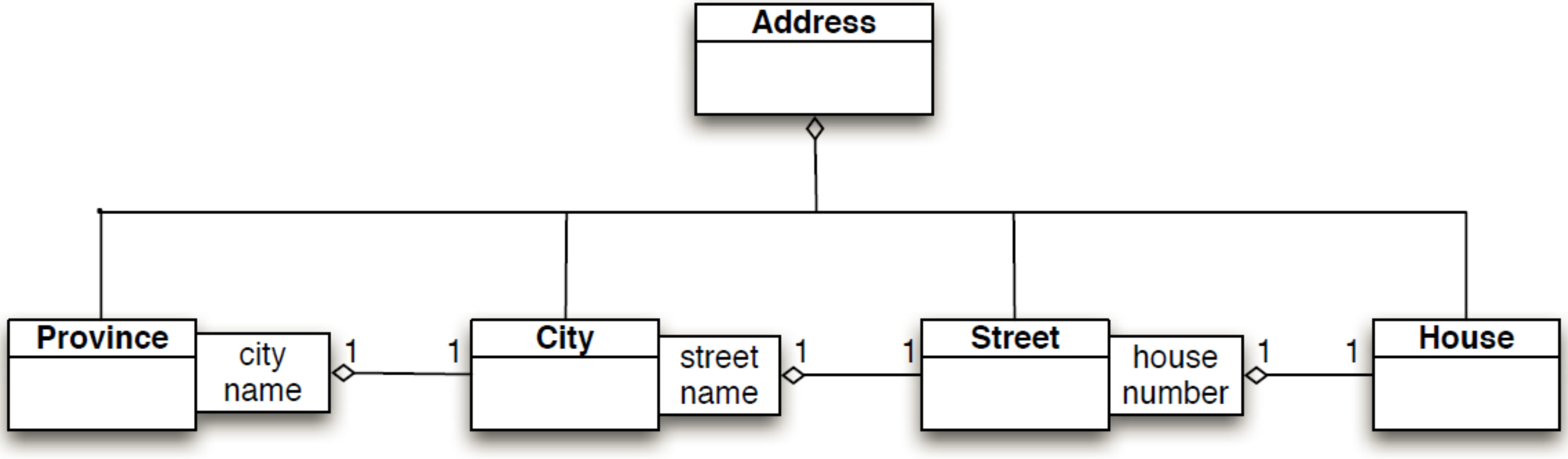


Single class:

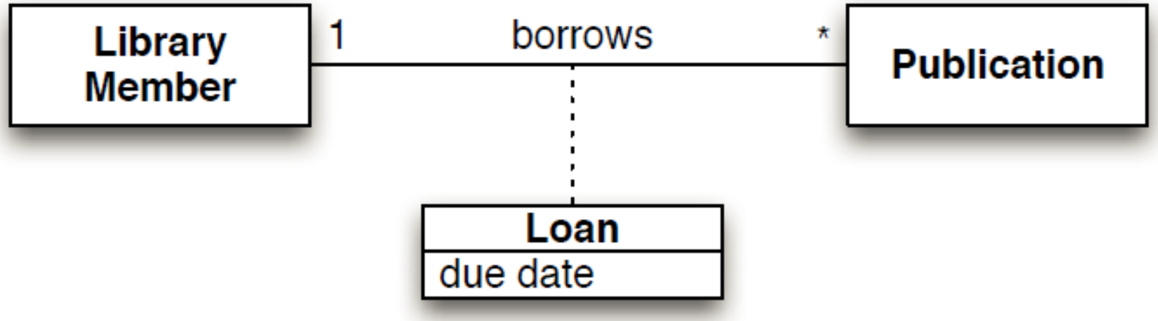


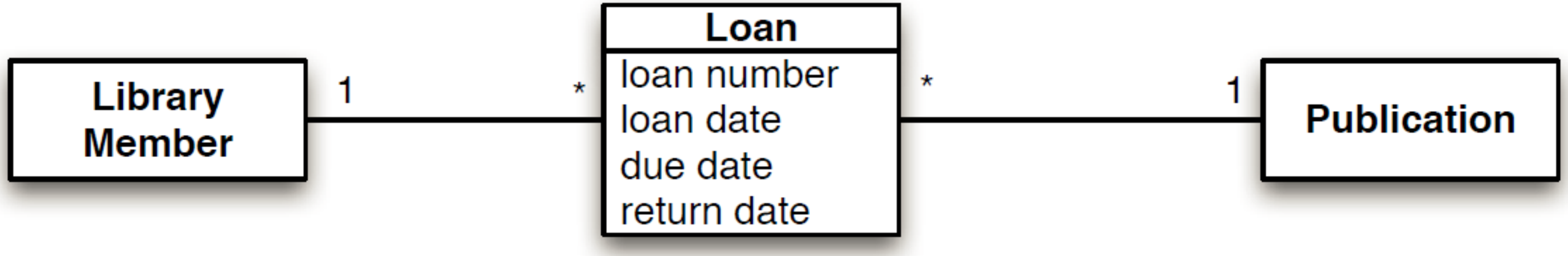
door: {open, closed, opening, closing}
engine: {up, down, standby}

Building object models: bad smells



Building object models: bad smells

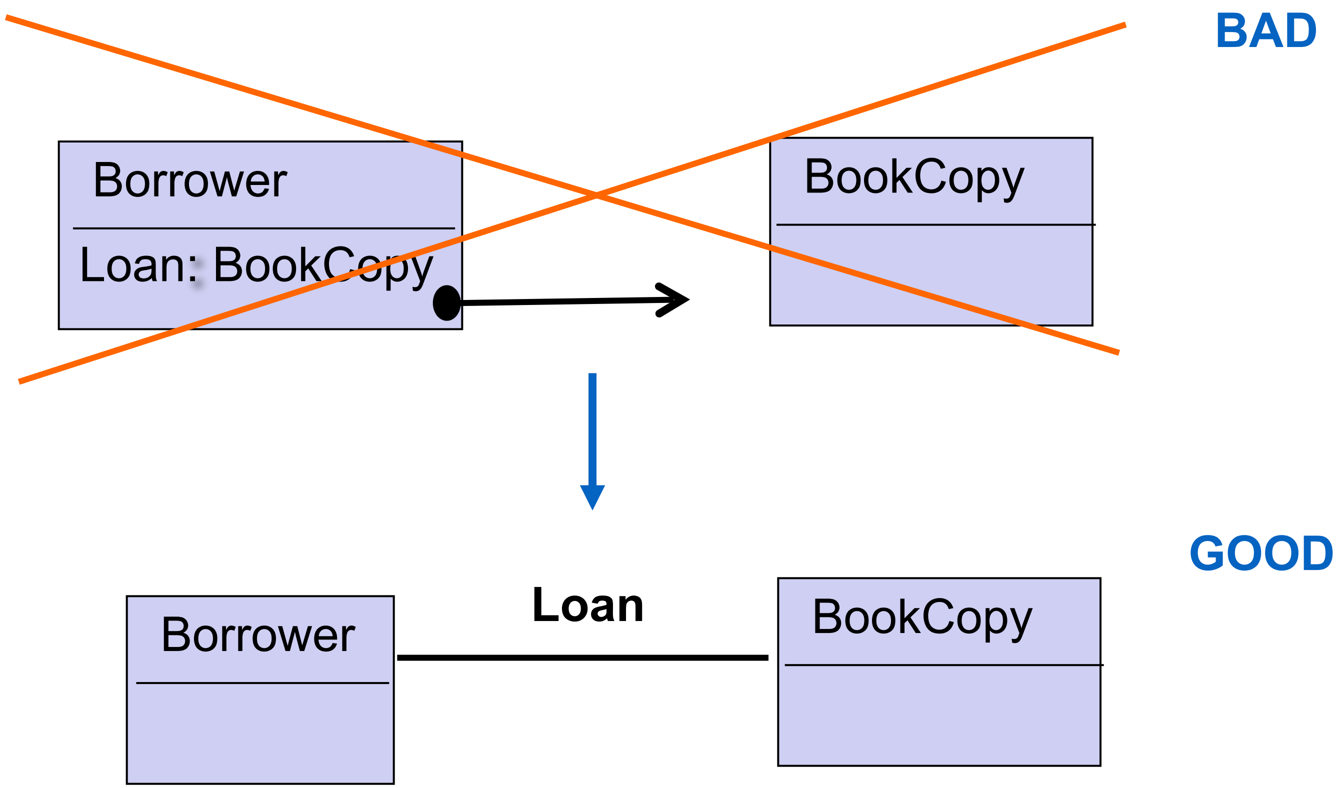




Building object models: bad smells

Avoid “pointers” to other objects as attributes

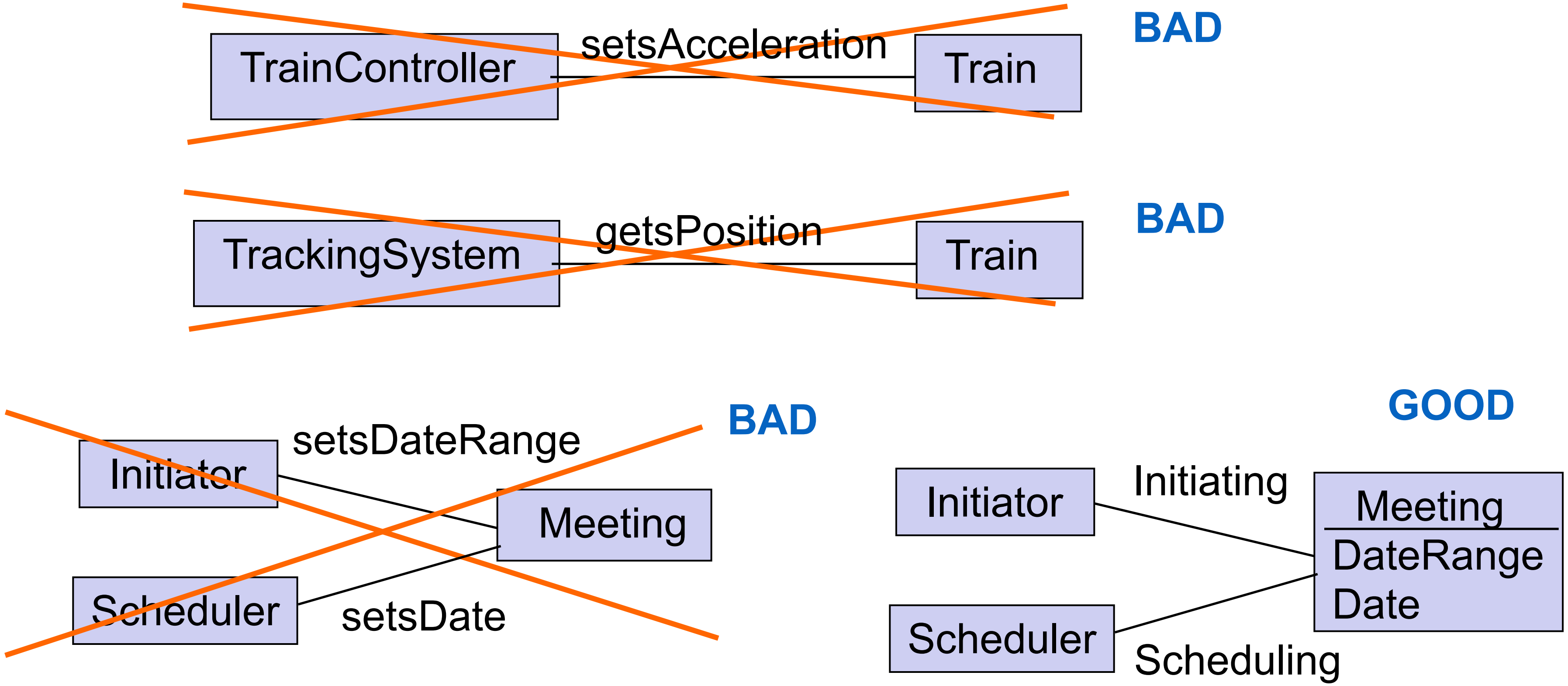
- use binary associations instead



Building object models: bad smells (2)

Avoid non-structural links pertaining to other views

- **monitoring/control** links from agent model (context diagram)



TrainController — setsAcceleration — Train

BAD

TrackingSystem — getsPosition — Train

BAD

Initiator — setsDateRange — Meeting

Scheduler — setsDate — Meeting

BAD

GOOD

Initiator — Initiating — Meeting

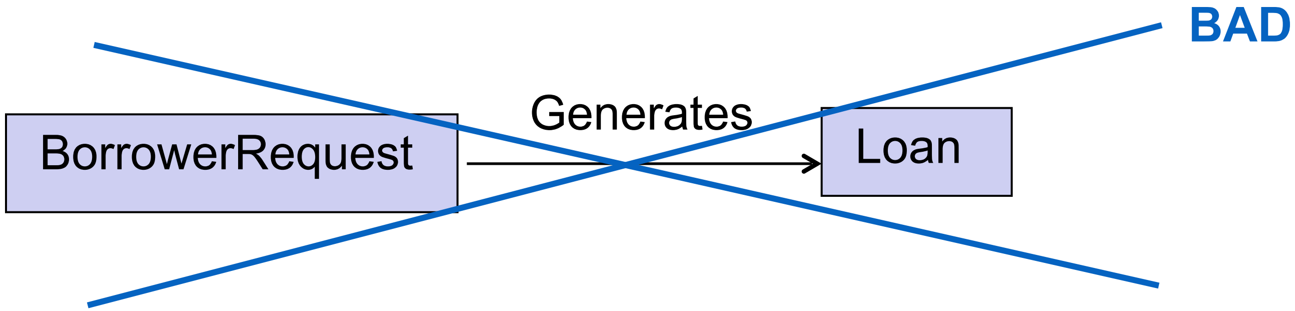
Scheduler — Scheduling — Meeting

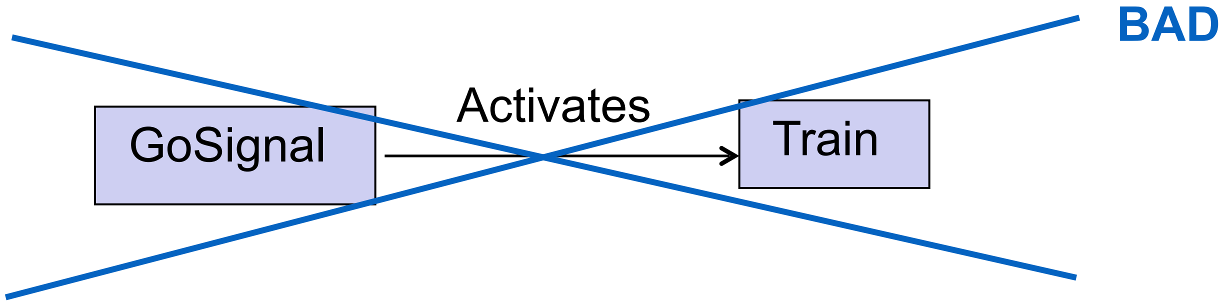
Meeting: DateRange, Date

Building object models: bad smells (3)

Avoid non-structural links pertaining to other views

- dynamic links from behavior model (state diagram)

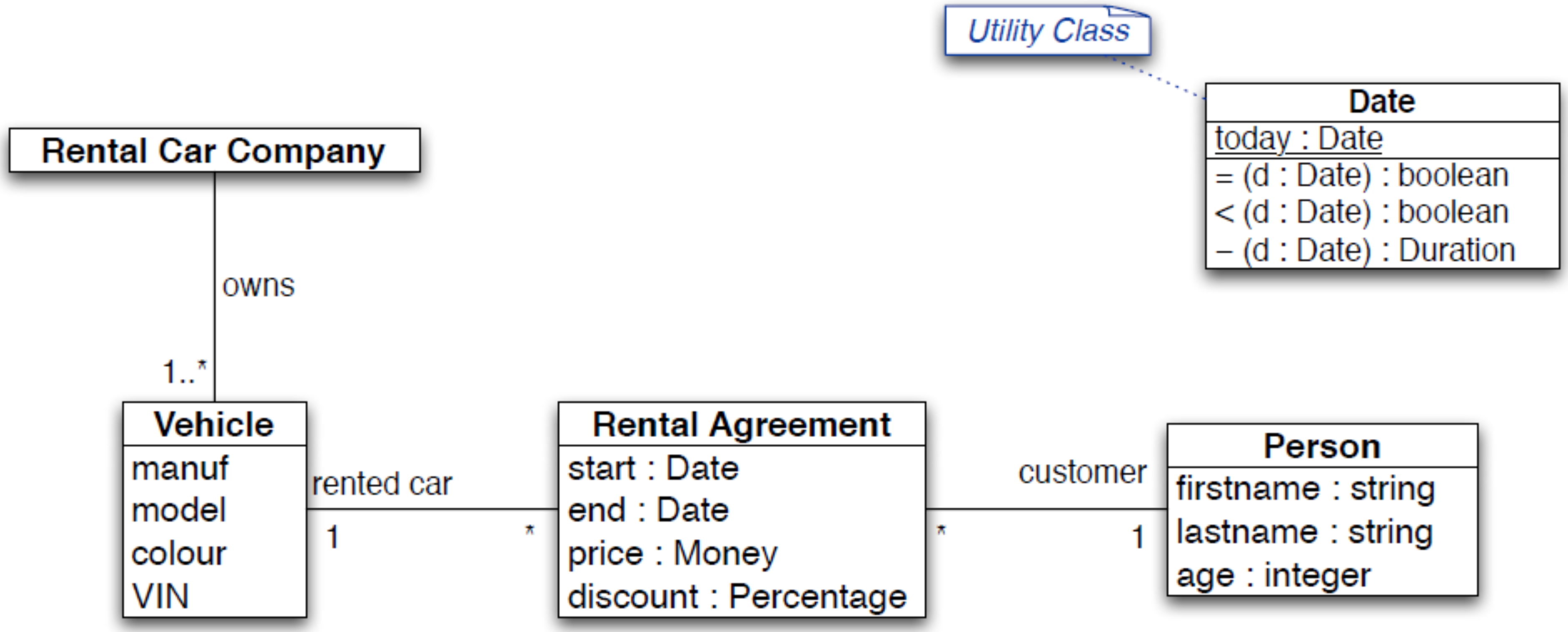




Building object models: bad smells (4)

Avoid obscure names for objects & attributes

- suggestive shortcut of their annotated definition
  - *don't forget precise definition!*
  - *don't confuse terms !* e.g. Book *vs.* BookCopy
- from problem world, NOT implementation-oriented
  - Bad JPEG_File , Book_File
  - Good Picture , Directory
- specific, NOT vague
  - Bad Person , Form
  - Good Patron , RegistrationForm
- commonly used, NOT invented
  - Bad PersonalIdentificationCard, ConferenceBook
  - Good StudentCard, Proceedings


Utility Class
Date
today : Date
= (d : Date) : boolean
< (d : Date) : boolean
– (d : Date) : Duration
Rental Car Company
owns
1..*
Vehicle
manuf
model
colour
VIN
rented car
1
*
Rental Agreement
start : Date
end : Date
price : Money
discount : Percentage
customer
*
1
Person
firstname : string
lastname : string
age : integer

Domain Model should have

- Attributes and their types
- Multiplicities on all associations (including “1” multiplicities)
- Association names, or role names, for all non-trivial associations
- Qualifiers to simplify multiplicities in associations
- Actors showed as stick figures or as classes with «actor» stereotype
  - This requires you to show multiplicities between actors and classes, which can be a valuable requirement detail.

Domain Model should NOT have…

- Class-level operations or methods
- Visibility annotations (i.e., private, protected, public)
- Navigability arrows
- Initial attribute values (unless you need them for model correctness)
- Object construction and destruction functions

Behave! Watch your language

- The goal is to create a *conceptual* model:
  - Models of real-world entities (customers, accounts, bills) and not of system entities (databases, SW components)
- Focus on the information/artifacts that the system will input, transform, analyze, display, etc.; physical and conceptual

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Domain Model

CS445/ECE 451/CS645

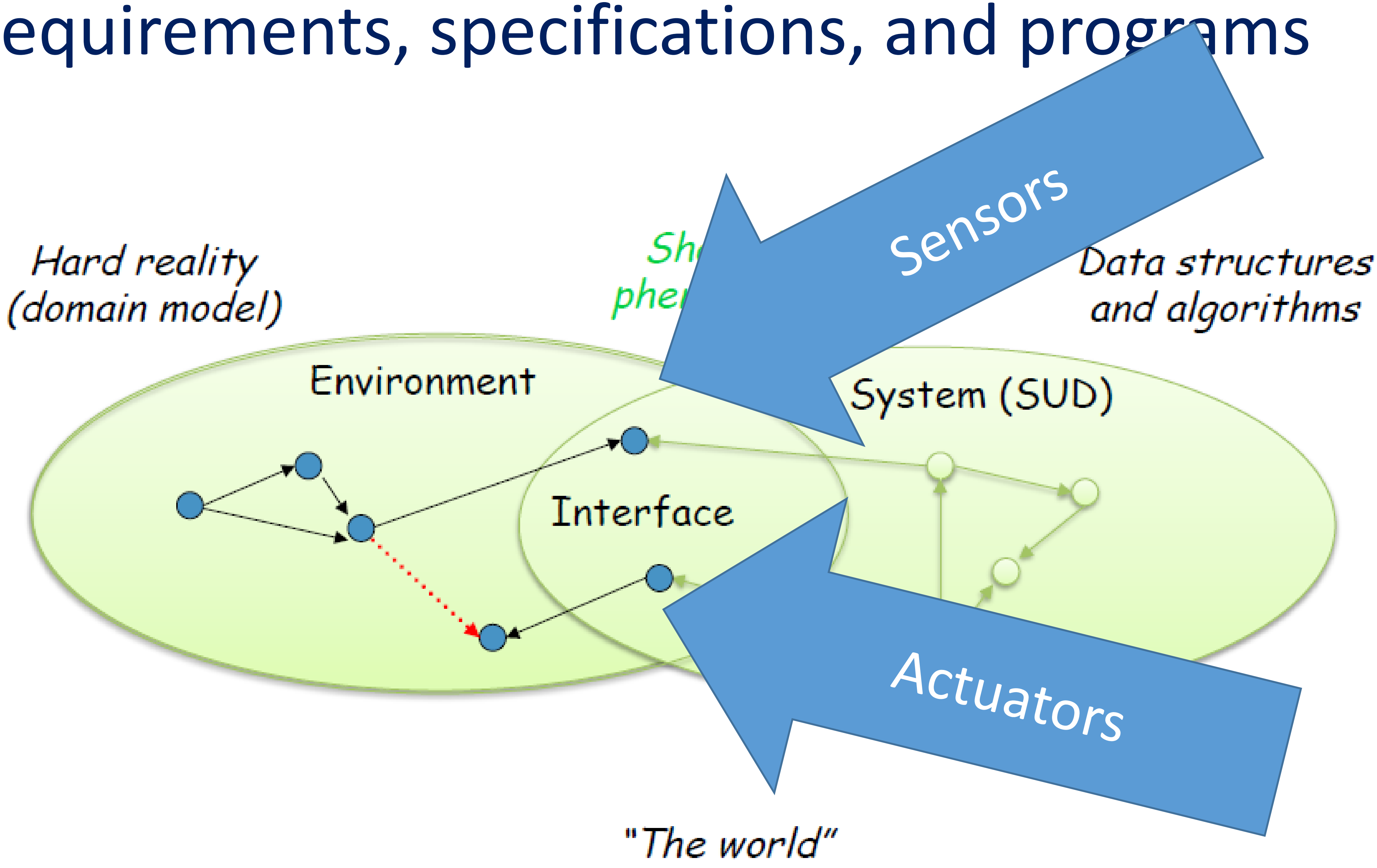
Software Requirements Specifications & Analysis

Requirements Engineering Reference Model

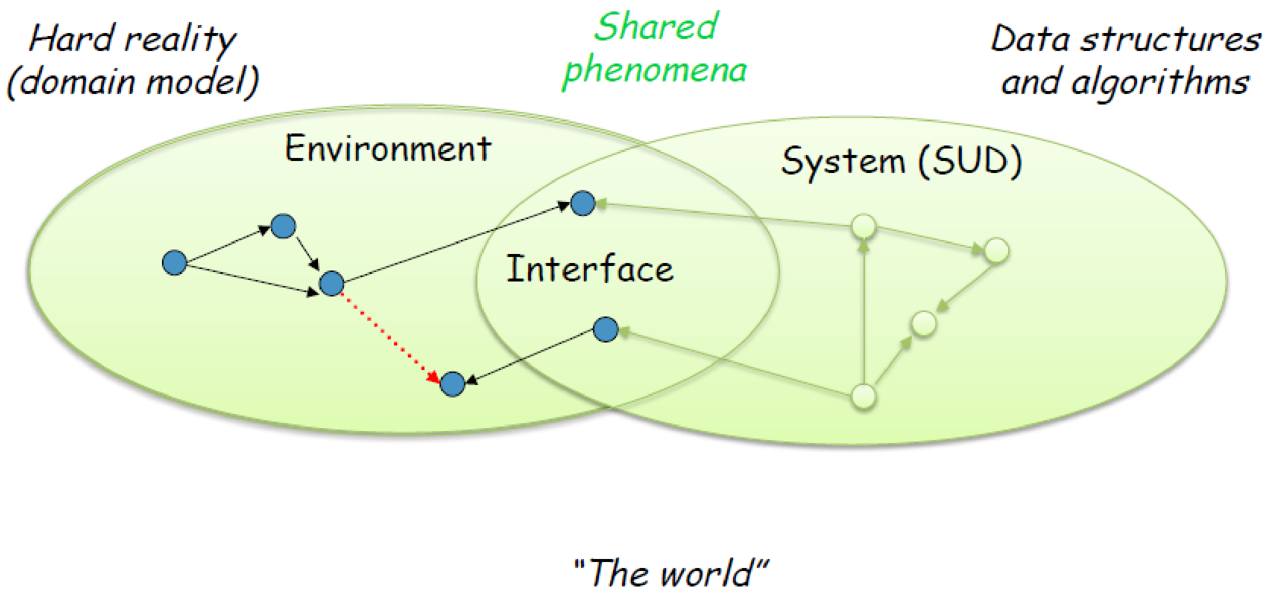
Overview

Goal: A clear understanding of what requirements are, what specifications are, and what the relationship between them is.

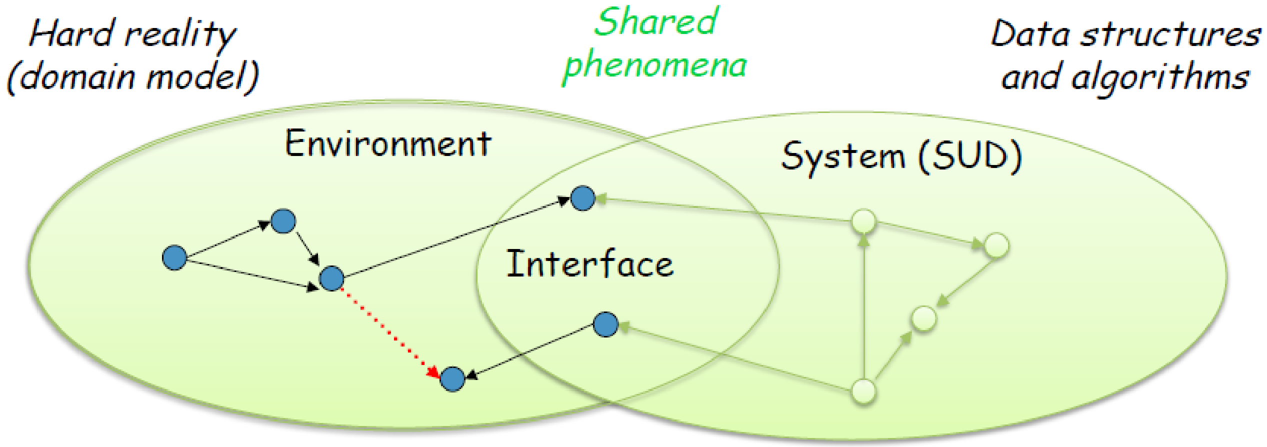
Requirements, specifications, and programs



- A *system* can be a socio-technical artifact to be constructed; it can be composed of some mix of software and hardware, humans and processes.
- We scope the *Environment* to include only those aspects of the real world that are relevant to the particular problem at hand.
- The generalized environment is sometimes called the *application domain*.
- A *domain model* is a diagram that shows how domain entities are related to each other. (I will talk about it later)



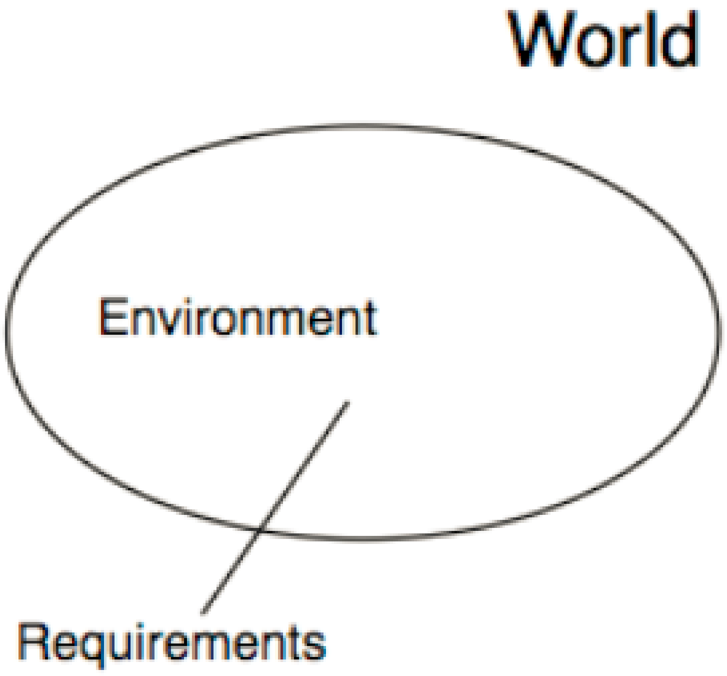
- Shared *Phenomena* are visible to both the Environment and the System and form the *Interface* between the two.
- Interface serves as a communication bridge between the environment and the system.



"The world"

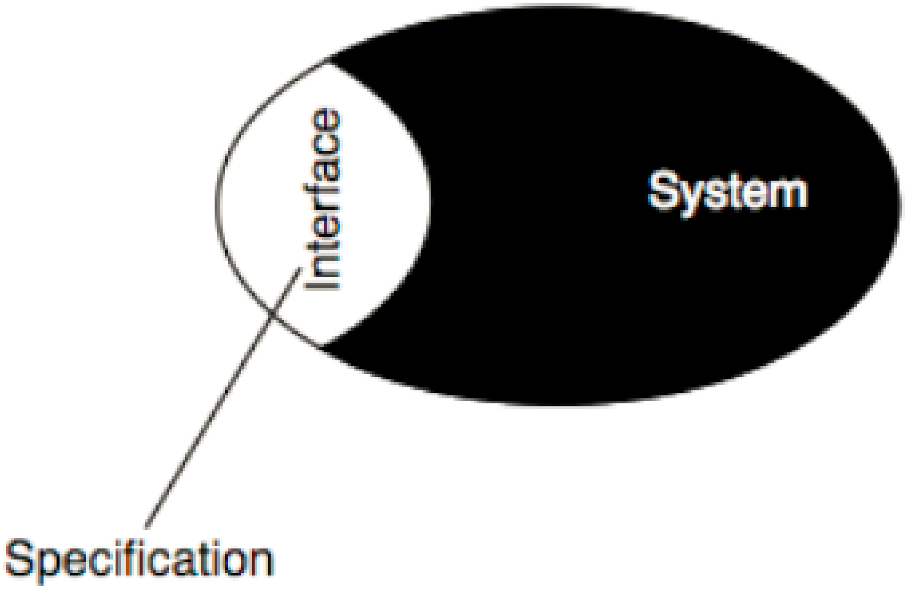
Requirements

- Are desired changes to the World.
- Are expressed in terms of environmental phenomena.
- Are statements of *desired* properties:
  - Often high level
  - May need to be elaborated, organized, analyzed
  - Heard during elicitation



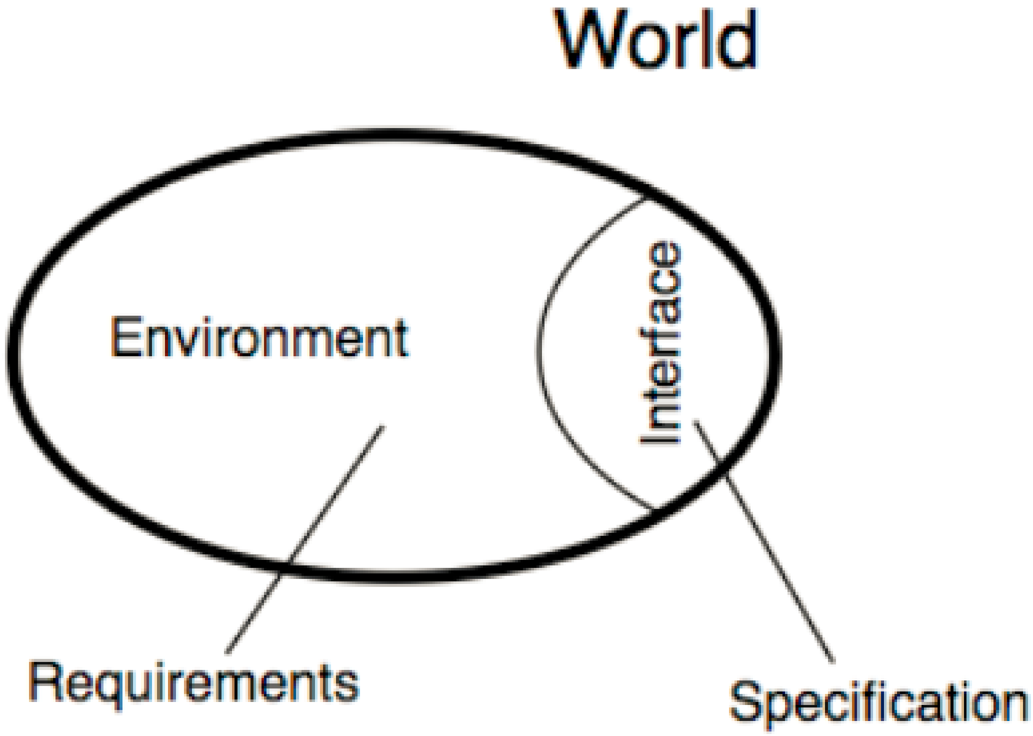
Specification

- Is a description of the proposed system.
- Should describe *what* the system is supposed to do, without indicating *how* the system will be realized.



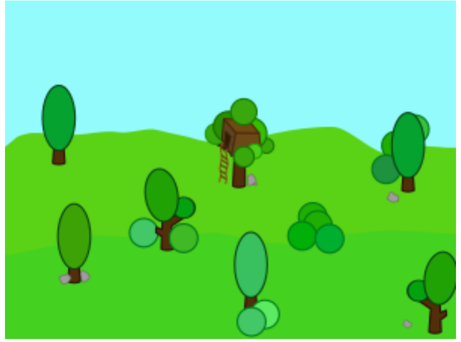
Scoping the environment

- It is a subset of the world
- Want to model only as much of the world as is necessary to express the requirements and the specification



Example

Suppose that the city of Waterloo decides to raise funds by instituting users fees for public parks.



Requirements:

R1: Collect $1 fee from each user on entry to the park.

R2: Ensure that anyone who has paid may enter the park.

R3: Ensure that no one may enter the park without paying.

Solutions: Human fee collectors vs. turnstiles with automated coin collection.

Turnstile Example

Requirement	Interface	Specification
Collect $1 fee from each user on entry to the park	Coin slot	(Env) coin inserted into slot (Sys) senses coin
Ensure that anyone who has paid may enter the park	Barrier	(Sys) unlocks barrier upon sensing a new coin (Env) visitor can detect that barrier is unlocked, can push barrier
Ensure that no one may enter the park without paying.	Barrier	(Sys) detects entry (Sys) relocks barrier

Domain Knowledge

- Ideally, we want to be able to show that the specifications imply the requirements:

$$\text{Spec} \models \text{Req}$$

- Often, we cannot do so without making some assumptions about how the environment behaves.

$$\text{Dom} \subseteq \text{Env}$$

- Domain Knowledge is thus the set of properties that we know (or assume) to be true of the Environment that is relevant to the problem.

- An *assumption* is a statement that is believed to be true in the absence of proof or definitive knowledge.

- Business assumptions are specifically related to the business requirements. Incorrect assumptions can potentially keep you from meeting your business objectives.

RE Reference Model

The fundamental law of requirements:

$$\text{Dom, Spec} \models \text{Req}$$

Must be able to argue that the specification of the system plus the assumptions are enough to satisfy the requirements.

Deriving Specifications

For each requirement (Req)

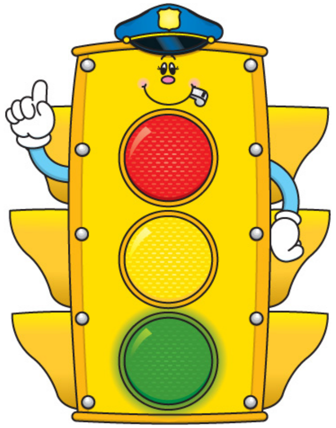
- Determine how the system will monitor/control the environment
- Determine whether Req constrains environmentally-controlled phenomena (if so, identify domain assumptions (Dom))
- Check that $\text{Dom, Spec} \models \text{Req}$

Example: Traffic Light

R: Allow car traffic to cross an intersection safely, without colliding with traffic travelling in other directions.

D: drivers behave legally and cars function correctly

S: spec of traffic light that guarantees that perpendicular directions do not show green/yellow at the same time.



Correctness

- To evaluate a specification, you must be able to argue that the SUD spec plus the domain assumptions are enough to satisfy the requirements. Dom, Spec $\models$ Req
- What do you need to do if you couldn't make this argument successfully?

R: Allow car traffic to cross an intersection safely without colliding with traffic travelling in other directions.

D: drivers behave legally, and cars function correctly

S: spec of a traffic light that guarantees that perpendicular directions do not show green/yellow simultaneously.

Uncertainty in D, S ⊨ R

- D, S ⊨ R tries to describe what happens ultimately formally.
- One would expect computers and software and their combination to be formal in this sense.
- But, the real world intervenes to make this formula a guideline, not an accurate, precise model.

RE Reference Model

The fundamental law of requirements:

$$\text{Dom, Spec} \models \text{Req}$$

Must be able to argue that the specification of the system plus the assumptions are enough to satisfy the requirements.

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

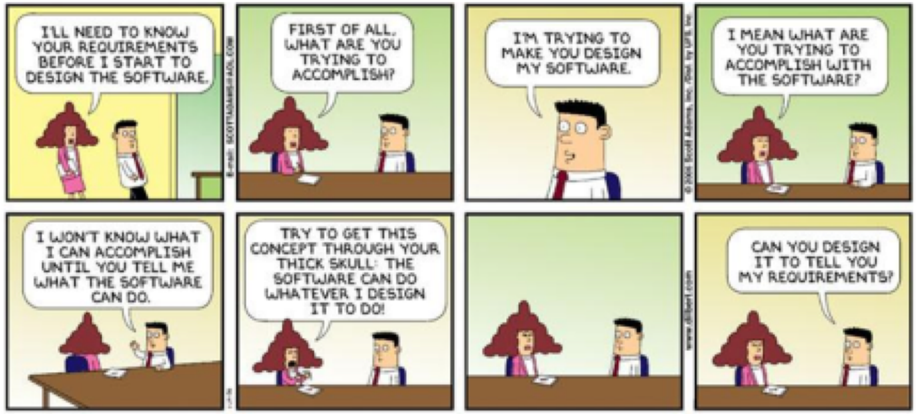
Requirements Engineering Reference Model

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Elicitation

To elicit means “to bring out, to evoke, to call forth”


I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

© Scott Adams, Inc./Dist. by UFS, Inc.

Purpose

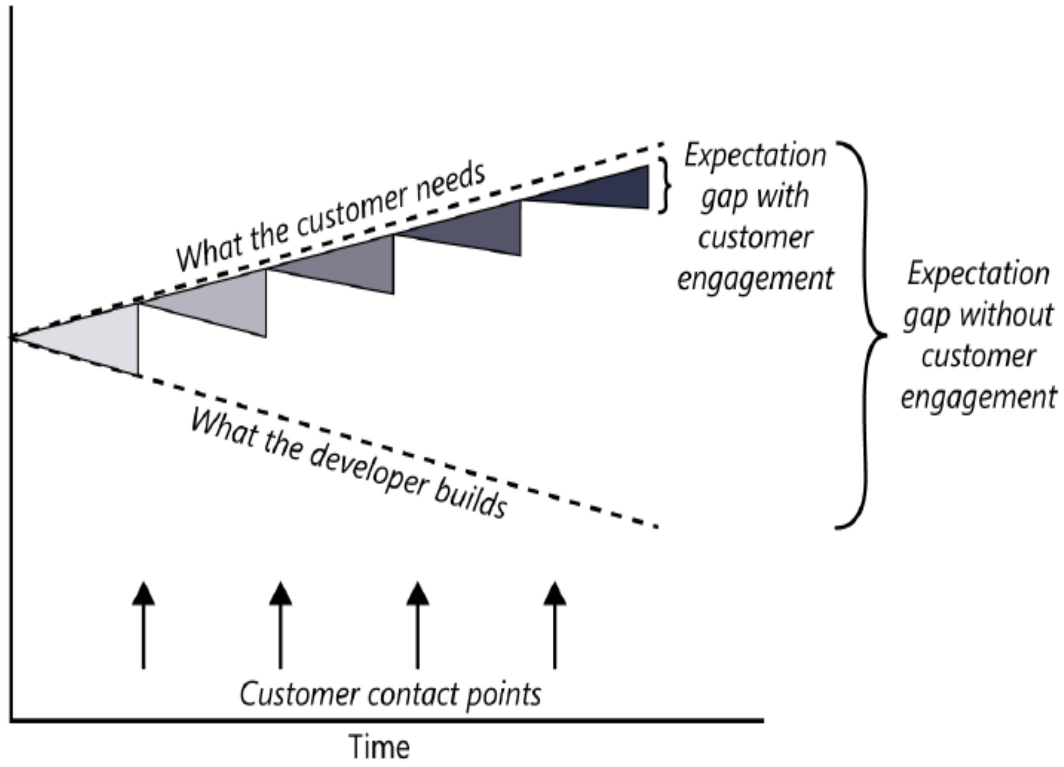
The purpose of elicitation is to get information about:

- Current work and current problems
- The requirements of the system
- The environment in which the system will operate

In order to:

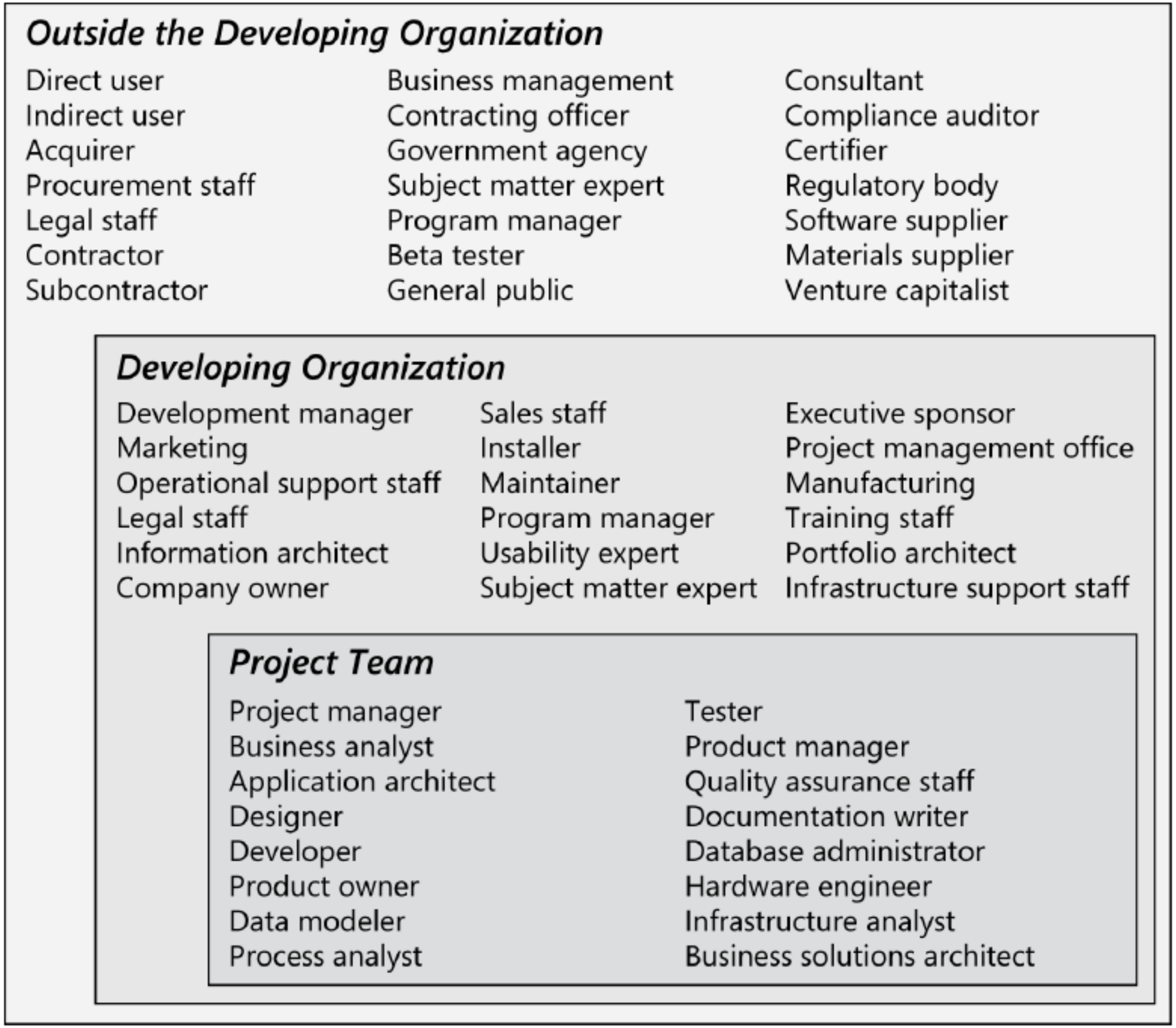
- identify relevant requirement sources
- elicit existing requirements from the identified sources
- develop new and innovative requirements

The expectation gap


What the customer needs
What the developer builds
Expectation gap with customer engagement
Expectation gap without customer engagement
Customer contact points
Time

Who is a stakeholder?

A *stakeholder* is a person, group, or organization actively involved in a project, is affected by its process or outcome, or can influence its process or outcome. Stakeholders can be internal or external to the project team and the developing organization.



Outside the Developing Organization

Direct user
Indirect user
Acquirer
Procurement staff
Legal staff
Contractor
Subcontractor
Business management
Contracting officer
Government agency
Subject matter expert
Program manager
Beta tester
General public
Consultant
Compliance auditor
Certifier
Regulatory body
Software supplier
Materials supplier
Venture capitalist

Developing Organization

Development manager
Marketing
Operational support staff
Legal staff
Information architect
Company owner
Sales staff
Installer
Maintainer
Program manager
Usability expert
Subject matter expert
Executive sponsor
Project management office
Manufacturing
Training staff
Portfolio architect
Infrastructure support staff

Project Team

Project manager
Business analyst
Application architect
Designer
Developer
Product owner
Data modeler
Process analyst
Tester
Product manager
Quality assurance staff
Documentation writer
Database administrator
Hardware engineer
Infrastructure analyst
Business solutions architect

Who is the customer?

Customers are a subset of stakeholders. A *customer* is an individual or organization that derives either direct or indirect benefits from a product.

Software customers could request, pay for, select, specify, use, or receive the output generated by a software product.

The customer-development partnership

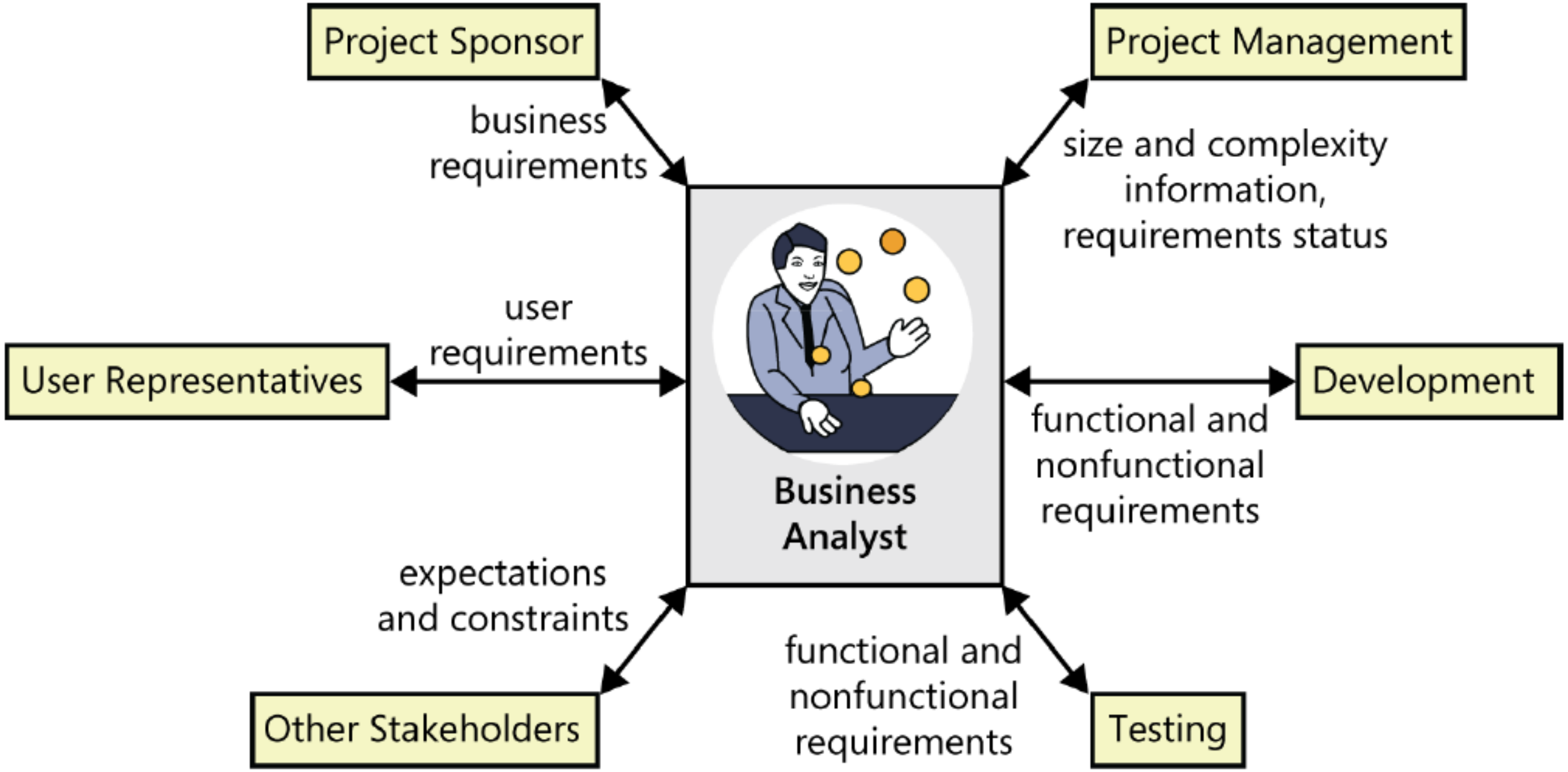
- An excellent software product: a well-executed design based on excellent requirements.
- Excellent requirements: effective collaboration between developers and customers.
- A collaborative effort: all parties involved know what they need to be successful and when they understand what their collaborators need to be successful.
- As project pressures rise, it's easy to forget that all stakeholders share a common objective: to build a product that provides adequate business value and rewards to all stakeholders.
- The business analyst typically is the point person who has to forge(form/make) this collaborative partnership.

Requirements Bill of Responsibilities for BA

You have the responsibility to

1. speak customer language.
2. learn about customer’s business and their objectives.
3. record requirements in an appropriate form.
4. provide explanations of requirements, practices and deliverables.
5. accept change of requirements.
6. maintain an environment of mutual respect.
7. provide ideas and alternatives for customers’ requirements and their solutions.
8. describe characteristics that will make the product easy to use.
9. provide ways to adjust requirements to accelerate development through reuse.
10. provide a system that meets customers’ functional needs and quality expectations.

The business analyst role


Project Sponsor
business requirements
Project Management
size and complexity information, requirements status
user requirements
User Representatives
Business Analyst
Development
functional and nonfunctional requirements
expectations and constraints
Other Stakeholders
functional and nonfunctional requirements
Testing

The business analyst bridges communication between customer and development stakeholders.

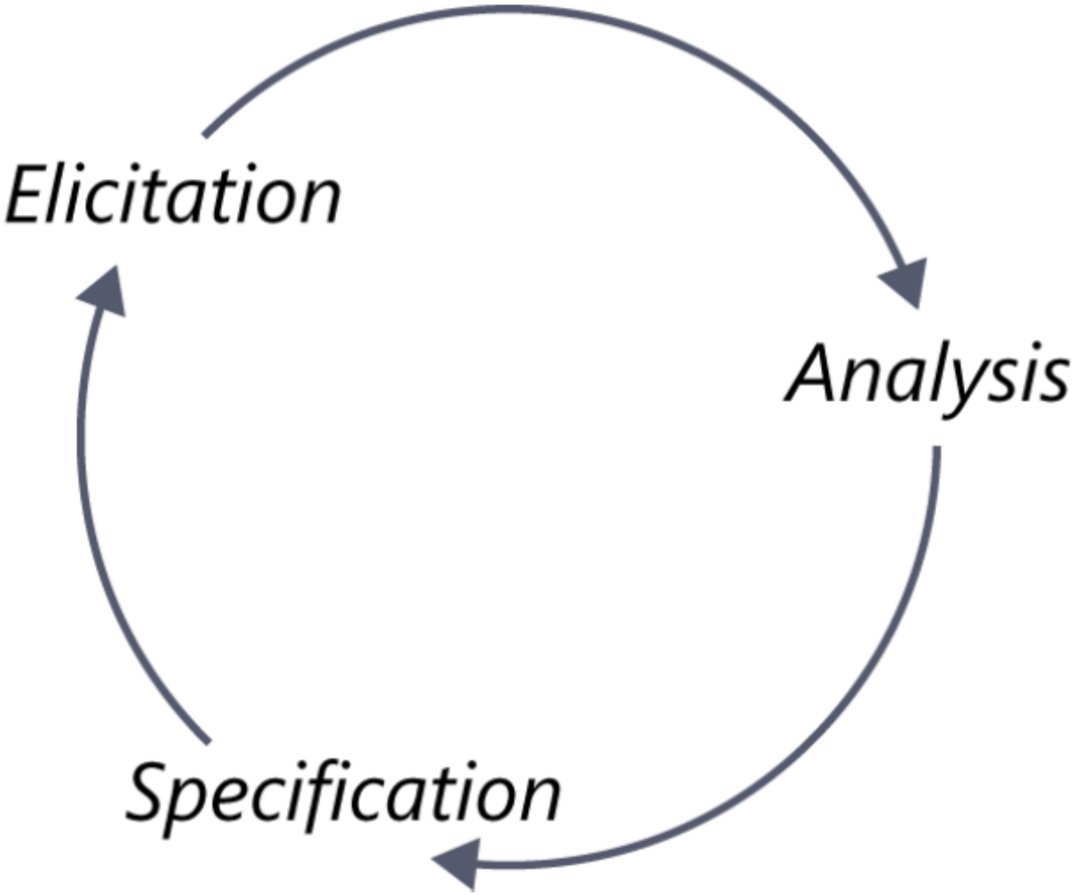
The business analyst’s tasks

- Define business requirements.
- Plan the requirements approach.
- Identify project stakeholders and user classes.
- Elicit requirements.
- Analyze requirements.
- Document requirements.
- Communicate requirements.
- Lead requirements validation.
- Facilitate requirements prioritization.
- Manage requirements.

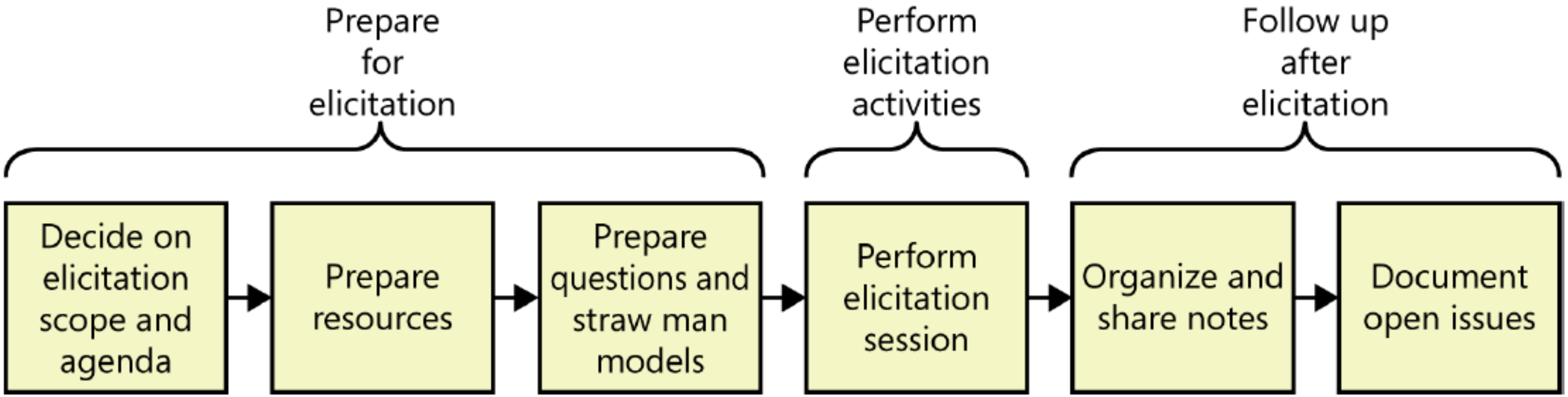
Essential analyst skills

1. Listening skills
2. Interviewing and questioning skills.
3. Thinking on your feet.
4. Analytical skills.
5. Systems thinking skills.
6. Learning skills.
7. Facilitation skills.
8. Leadership skills.
9. Observational skills.
10. Communication skills.
11. Organizational skills.
12. Modeling skills.
13. Interpersonal skills.
14. Creativity

The cyclic nature of requirements elicitation, analysis, and specification.



Activities for a single requirements elicitation session.


Prepare for elicitation
Perform elicitation activities
Follow up after elicitation
Decide on elicitation scope and agenda
Prepare resources
Prepare questions and straw man models
Perform elicitation session
Organize and share notes
Document open issues

Before we walk through this process, though, let’s explore some of the requirements elicitation techniques you might find valuable.

Requirements elicitation techniques

1. Interview
2. Workshop
3. Focus Groups
4. Observations
5. Questionnaires
6. System Interface Analysis
7. User Interface Analysis
8. Document Analysis

Requirements elicitation techniques

1. Interview (The most obvious way to find out what the users of a software system need is to ask them):
   - Establish rapport (a close and harmonious relationship)
   - Stay in scope
   - Prepare questions (closed and opened)
   - Suggest ideas
   - Listen actively
   - Summing up
   - Follow up

2. Workshop (Workshops encourage stakeholder collaboration in defining requirements):

- Establish and enforce ground rules
- Fill all of the team roles
- Plan an agenda
- Stay in scope
- Timebox discussions
- Keep the team small but include the right stakeholders
- Keep everyone engaged

3. Focus groups:
    - A focus group is a representative group of users who convene in a facilitated elicitation activity to generate input and ideas on a focused product’s functional and quality requirements.
    - Focus group sessions must be interactive, allowing all users to voice their thoughts.
    - Focus groups help explore users’ attitudes, impressions, preferences, and needs
4. Observations:
    - When you ask users to describe how they do their jobs, they will likely have a hard time being precise; details might be missing or incorrect
    - Observations are time-consuming.
    - Observations can be silent or interactive.

5. Questionnaires:

- To survey large groups of users/stakeholders to understand their needs.
- They are inexpensive, making them a logical choice for eliciting information from large user populations, and they can be administered easily across geographical boundaries.
- The analyzed results of questionnaires can be used as input for other elicitation techniques.
- You can also use questionnaires to survey commercial product users for feedback.

- Preparing well-written questions (answer options) is the biggest challenge with questionnaires:
  - complete the set of possible responses.
  - mutually exclusive and exhaustive.
  - don’t imply a “correct” answer.
  - consistent with scales.
  - open-ended questions vs. closed questions.

6. System interface analysis:

- Reveals functional requirements regarding the exchange of data and services between systems.

- For each system that interfaces with yours, identify functionality in the other system that might lead to requirements for your system.

7. User interface analysis:

- To study existing systems to discover user and functional requirements.
- Can help you identify a complete list of screens to help you discover potential features.
- It’s a great way to get up to speed on how an existing system works (unless you need much training).
- Instead of asking users how they interact with the system and what steps they take, perhaps you can reach an initial understanding yourself.

8. Document analysis:

- The most helpful documentation includes requirements specifications, business processes, lessons learned collections and user manuals for existing or similar applications.
- Documents can describe corporate or industry standards that must be followed or regulations with which the product must comply.
- Comparative reviews point out shortcomings in other products that you could address to gain a competitive advantage.
- Problem reports and enhancement requests collected from users by the help desk and field support personnel can offer ideas for improving the system in future releases.

How do you know when you’re done?

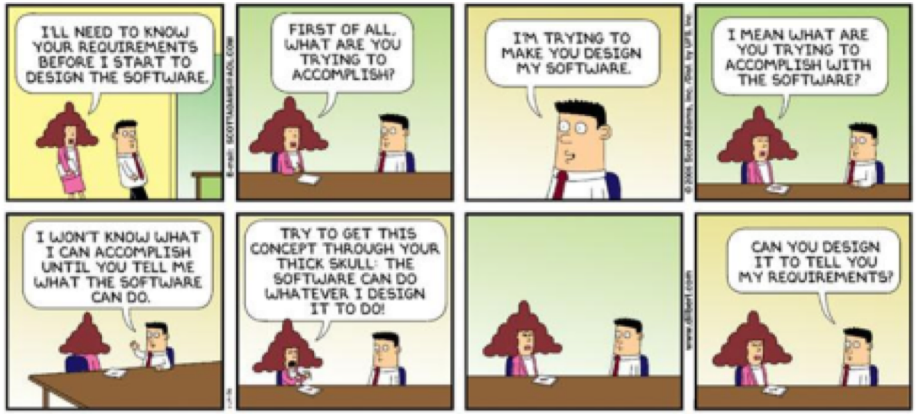
- Can’t think of any more use cases or user stories.
- Users propose new scenarios but don’t lead to any new functional requirements.
- Users repeat issues they already covered in previous discussions.
- Suggested new features, user requirements, or functional requirements are all deemed to be out of scope.
- Proposed new requirements are all low priority.
- The users are proposing capabilities that might be included “sometime in the product’s lifetime” rather than “in the specific product we’re talking about right now.”
- Developers and testers who review the requirements for an area raise a few questions.

CS445/ECE 451/CS645

Software Requirements Specifications & Analysis

Elicitation

To elicit means "to bring out, to evoke, to call forth"


I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

© Scott Adams, Inc./Dist. by UFS, Inc.

CS 445 / ECE 451 / CS 645

Software Requirements Specifications & Analysis

Introduction

Grad Student

- …. Who is taking this as CS 645: Please send the instructor an e-mail during January!
- You will be required to do a 20-30 minute lecture and a written report on a topic related to the course material
  - I am very open to possible topics
  - It’s worth 10% of your final grade
- [NOTE TO ALL] Grad lecture material can be on the exam!

Course project

- To be done in groups of 5, self-chosen
  - You will all get the same project grades
    - unless your partners think you're …..
  - By 6 am on **Tuesday, January 16**, each team must e-mail the instructor with
    - Member names
    - Project name
    - Project description.

Course project – cont.

- Doing so will allow you to apply the requirements engineering principles and techniques discussed in lectures to the problems of eliciting, documenting, and validating the specification of a non-trivial software system.

Working in groups

- Don’t just pick your friends
- Consider:
  - work habits
  - goals
  - good organization skills for project coordination
  - good writing skills
- Ideally:
  - Equitably distribute the workload
  - Minimize resentment

- The purpose of working in groups is to get you used to work in groups

- Working in groups is a crucial skill for success in the industry.
  - It’s also tough to teach or lecture about, so we try to ensure that you have some exciting experiences.

- Your project is not a collection of little independent tasks; instead, there will be several work stages for each deliverable:
  - Discuss and allocate tasks to group members
  - Work on tasks (alone or not)
  - Distribute draft solutions
  - Meet to discuss drafts, evaluate, iterate, and plan
  - Revise, evaluate, iterate
  - Stitch together final draft and submit

- Specifying something real is much more interesting than a mocked-up example.

- Your group will be assigned a TA, who will serve as your customer and will grade all of your submissions
    - And thus, you'll get some consistency in marking too

- Your job:
  - to create detailed models of the various entities and processes,
  - to decide what features should be there,
  - to decide the correct functionality of these features,
  - eventually, use these models and decisions to create an SRS describing your software,
  - You should be thinking of these as you progress through the project.
  - In other words, you will have to modernize the system

Master the basics

- You will participate in brainstorming meetings to identify requirements.
- Your group will decide on a consistent set of requirements and then model and specify a system with these requirements in the form of an SRS.

Notes

- The final deliverable is an SRS
- You will be expected to be serious, creative, and consider the project as a real case.
- Remember that you don’t have to implement them.

Important notes when emailing us:

- Number questions and have only one question per number.
- **Do not** draw and scan deliverables. Using a tool is part of the difficulties of software engineering.
- Identify yourself (the group name) **both** on the e-mail (either subject or body, or both)

Sources for all lectures

- Previous offering of course notes
- Fowler, *UML Distilled*, Addison-Wesley, 2004, 3rd edition. (available electronically at the DC library)
- Jackson, *Software Requirements and Specification*, ACM Press, 1995.
- Gause and Weinberg, *Exploring Requirements*, Dorset House, 1989.
- Van Lamsweerde, *Requirements Engineering: From Systems Goals to UML Models to Software Specification*, Wiley, 2009.
- Lauesen, *Software Requirements: Styles and Techniques*, Addison-Wesley, 2002.
- Robertson and Robertson, *Mastering the Requirements Process*, 2nd ed., Wiley, 2006.
- Klaus Pohl . Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2010.
- Karl Wiegers and Joy Beatty, Software Requirements, Third Edition.

Questions

What is “Requirement” ?

Requirement:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

[IEEE 610.12-1990 standard]

What is “requirement”? Cont.

[*Requirements encompass both the user’s view of the external system behavior and the developer’s view of some internal characteristics. They include both the behavior of the system under specific conditions and those properties that make the system suitable—and maybe even enjoyable—for use by its intended operators.]*

Ian Sommerville and Pete Sawyer (1997)

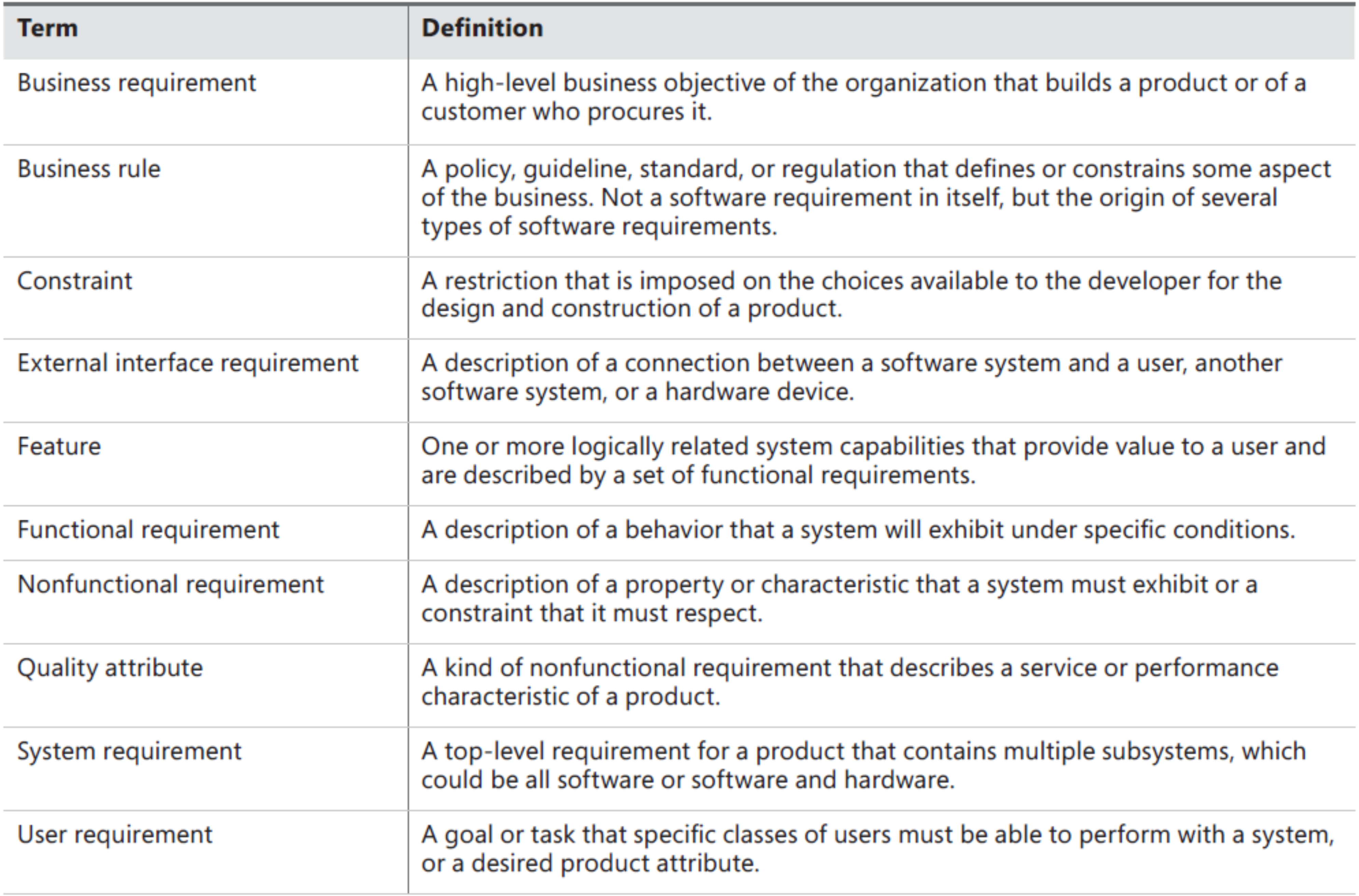
What is “requirement”? Cont.

“Requirements are a specification of what should be implemented. They are descriptions of how the system should behave or of a system property or attribute. They may be a constraint on the development process of the system.”

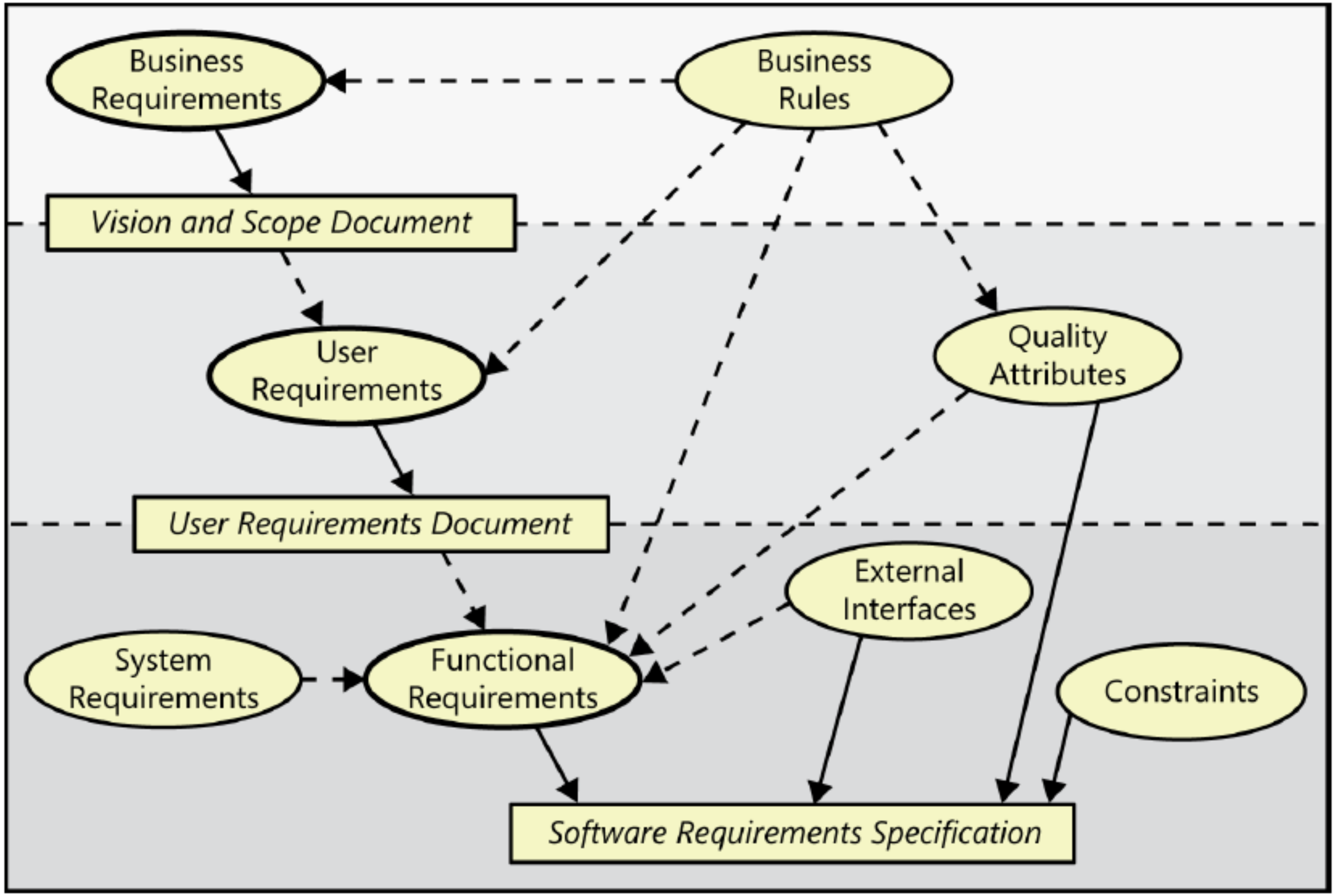
Wiegers Karl E. And Beatty Joy

Software Requirements (Developer Best Practices)

Levels and types of requirements

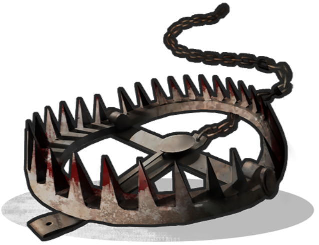


Term	Definition
Business requirement	A high-level business objective of the organization that builds a product or of a customer who procures it.
Business rule	A policy, guideline, standard, or regulation that defines or constrains some aspect of the business. Not a software requirement in itself, but the origin of several types of software requirements.
Constraint	A restriction that is imposed on the choices available to the developer for the design and construction of a product.
External interface requirement	A description of a connection between a software system and a user, another software system, or a hardware device.
Feature	One or more logically related system capabilities that provide value to a user and are described by a set of functional requirements.
Functional requirement	A description of a behavior that a system will exhibit under specific conditions.
Nonfunctional requirement	A description of a property or characteristic that a system must exhibit or a constraint that it must respect.
Quality attribute	A kind of nonfunctional requirement that describes a service or performance characteristic of a product.
System requirement	A top-level requirement for a product that contains multiple subsystems, which could be all software or software and hardware.
User requirement	A goal or task that specific classes of users must be able to perform with a system, or a desired product attribute.



Relationships among several types of requirements information. Solid arrows mean "are stored in"; dotted arrows mean "are the origin of" or "influence."

Trap:

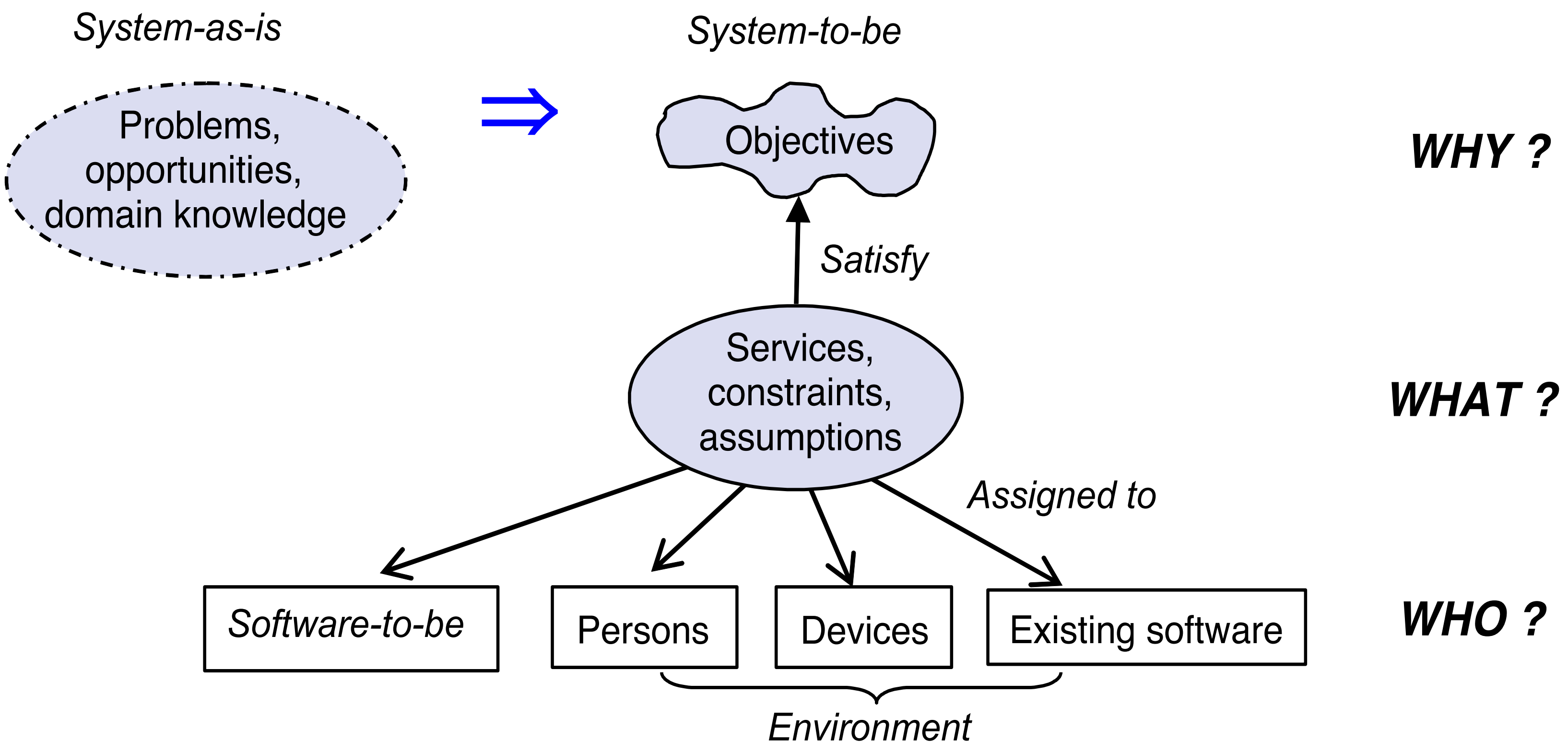


Don’t assume that all your project stakeholders share a common notion of what requirements are. Establish definitions up front so that you’re all talking about the same things.

What is Requirements engineering?

- To make sure that a software solution correctly solves a particular problem, we must first correctly understand and define what problem needs to be solved.

- Figuring out what is the right problem is can be surprisingly difficult.
  - *What* problem should be solved
  - *Why* such a problem needs to be solved
  - *Who* should be involved in the responsibility of solving that problem



System-as-is

Problems, opportunities, domain knowledge

⇒

System-to-be

Objectives

Satisfy

Services, constraints, assumptions

Assigned to

Software-to-be

Persons

Devices

Existing software

Environment

WHY ?

WHAT ?

WHO ?

Why Software Fails…

- We waste billions of dollars each year on entirely preventable mistakes.
- Most IT experts agree that such failures occur far more often than they should.
- The failures are universal: they happen in every country; to large companies and small; in commercial, nonprofit, and governmental organizations; and without regard to status or reputation.
- The business and societal costs of these failures--in terms of the wasted taxpayer and shareholder dollars and investments that can't be made--are now well into the billions of dollars a year.

Why do projects fail so often?

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

Top software failures

Real life examples of software development failures

12 famous ERP disasters, dustups and disappointments

10 Biggest Software Bugs and Tech Fails of 2021

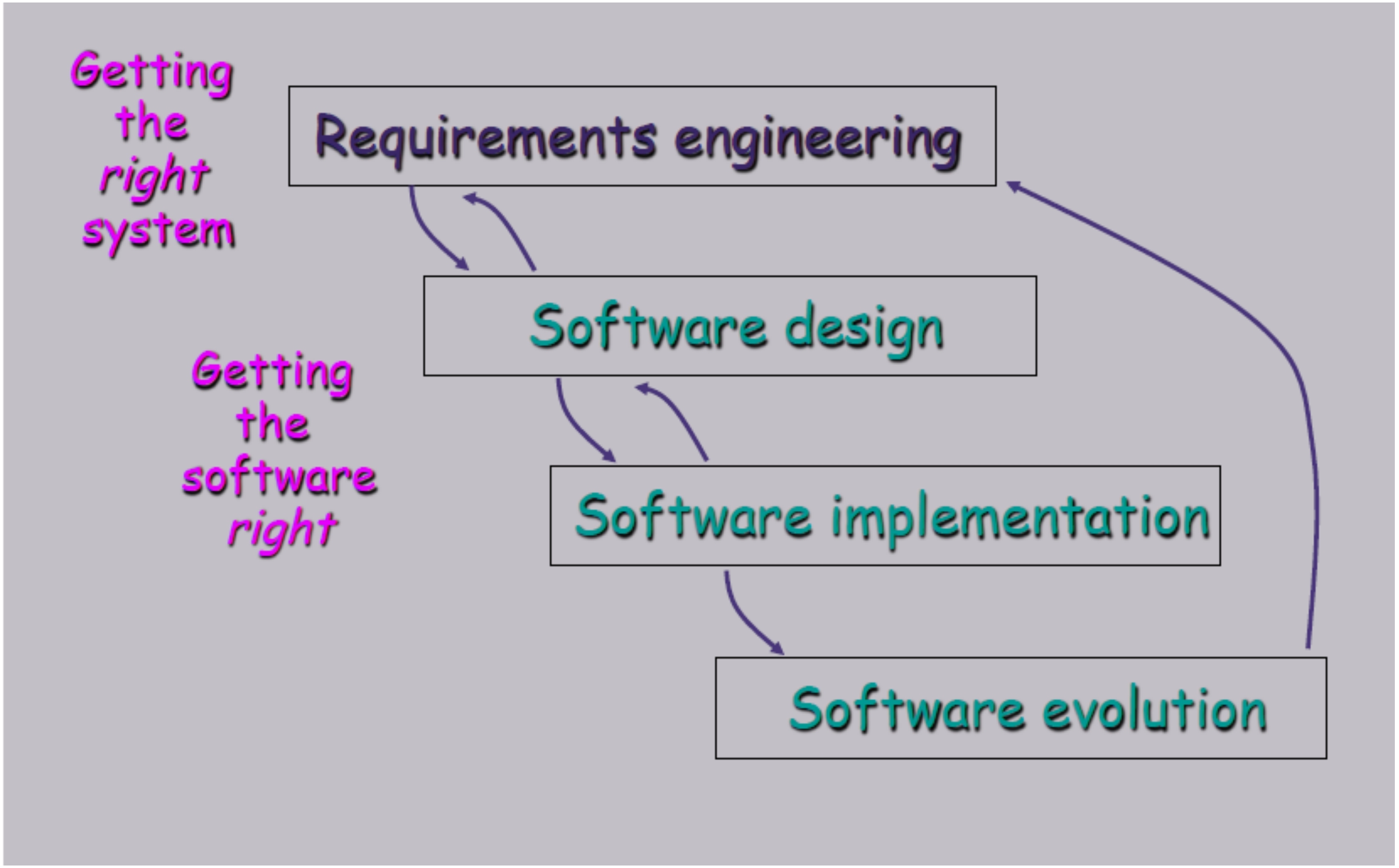
11 of the most costly software errors in history

Requirements Engineering, roughly…

- Analyze problems with an existing system (*system-as-is*)
- Identify objectives & opportunities for new system (*system-to-be*)
- Define functionalities of, constraints on, responsibilities in system-to-be,
- Specify all of these in a *requirements document*

System = software + environment

Requirements in the software lifecycle



Getting the *right* system

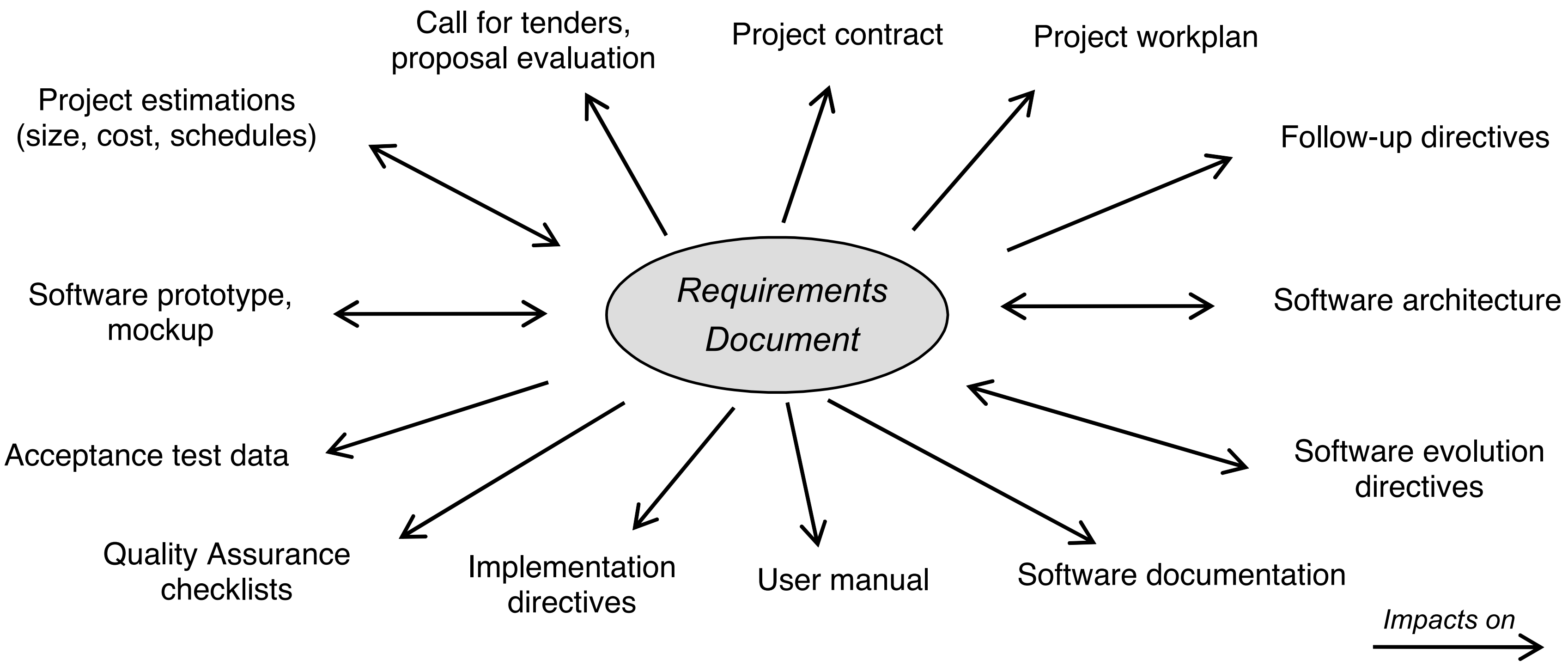
Requirements engineering

Software design

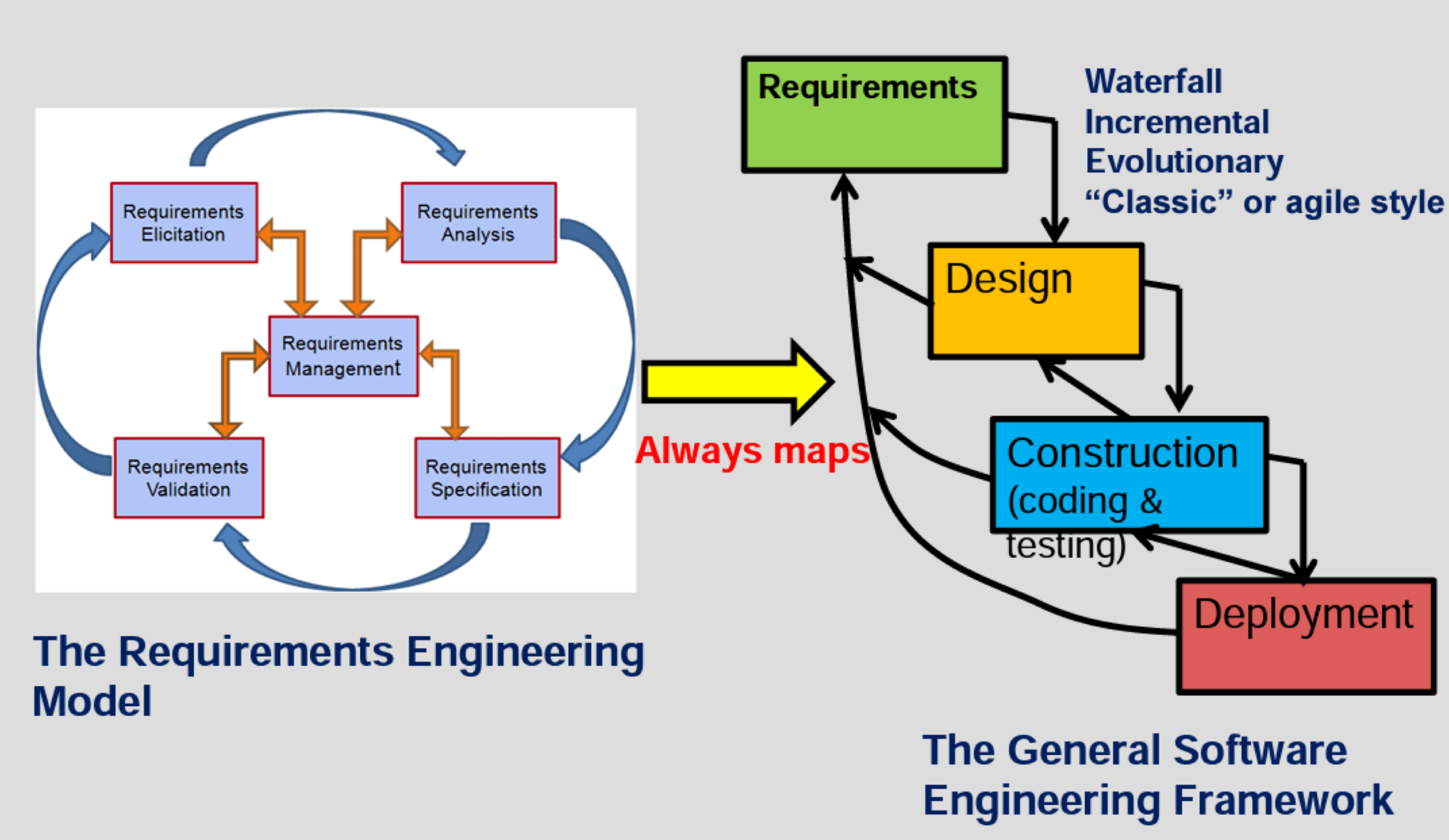
Getting the software *right*

Software implementation

Software evolution


Call for tenders, proposal evaluation
Project contract
Project workplan
Project estimations (size, cost, schedules)
Follow-up directives
Software prototype, mockup
Requirements Document
Software architecture
Acceptance test data
Software evolution directives
Quality Assurance checklists
Implementation directives
User manual
Software documentation
Impacts on

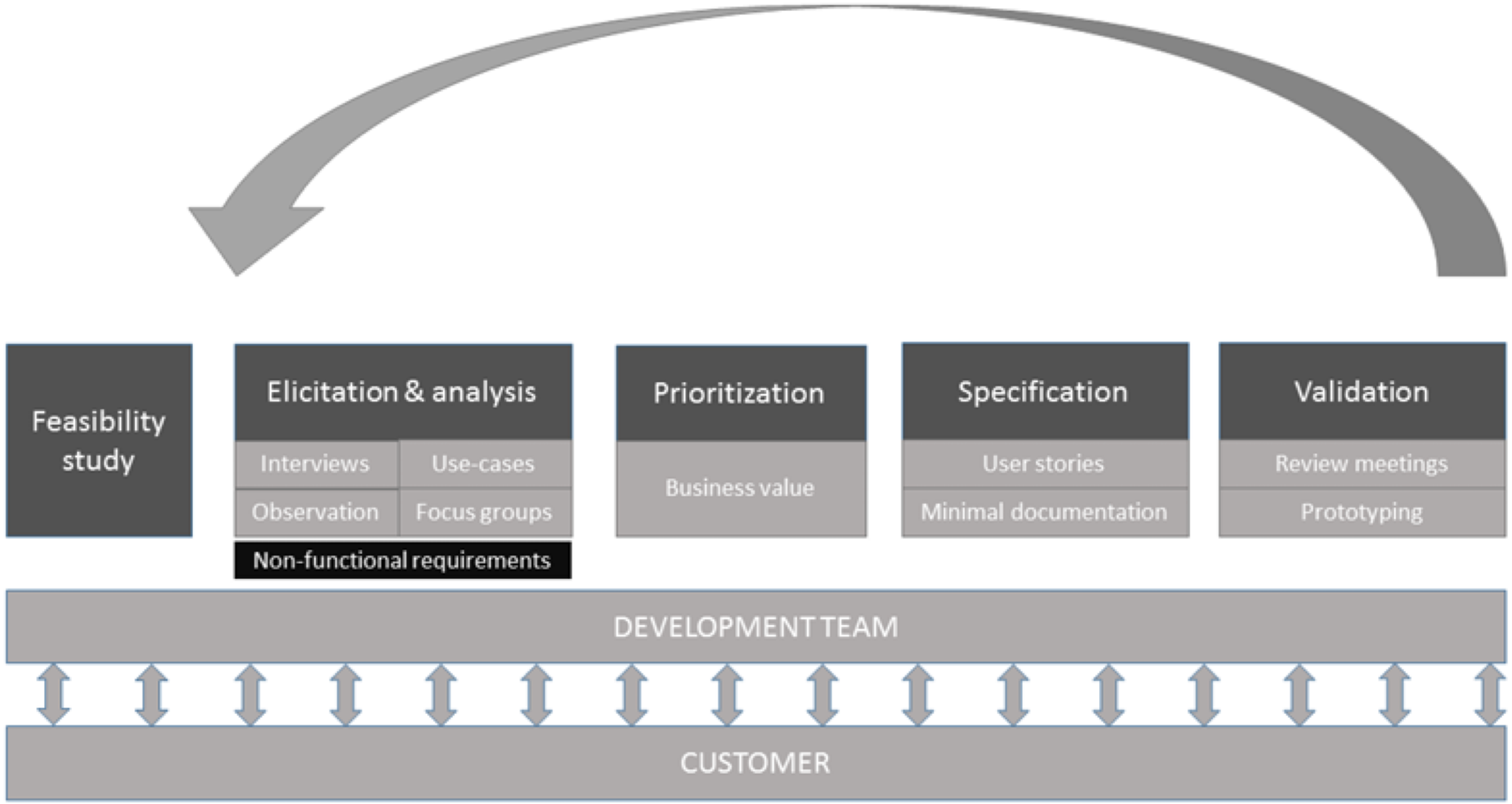
Waterfall



The Requirements Engineering Model

The General Software Engineering Framework

Agile



Feasibility study

Elicitation & analysis

Interviews

Use-cases

Observation

Focus groups

Non-functional requirements

Prioritization

Business value

Specification

User stories

Minimal documentation

Validation

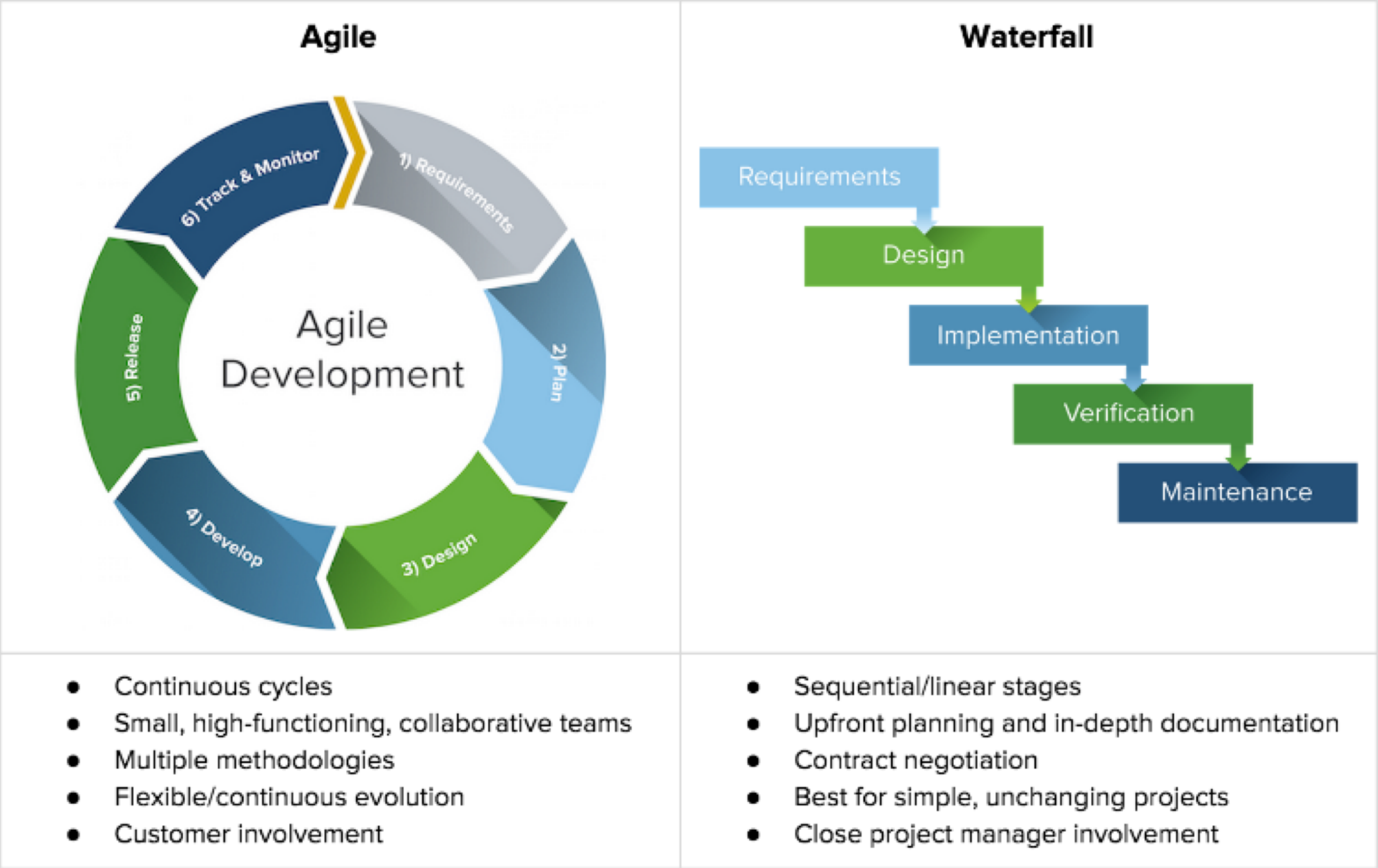
Review meetings

Prototyping

DEVELOPMENT TEAM

CUSTOMER

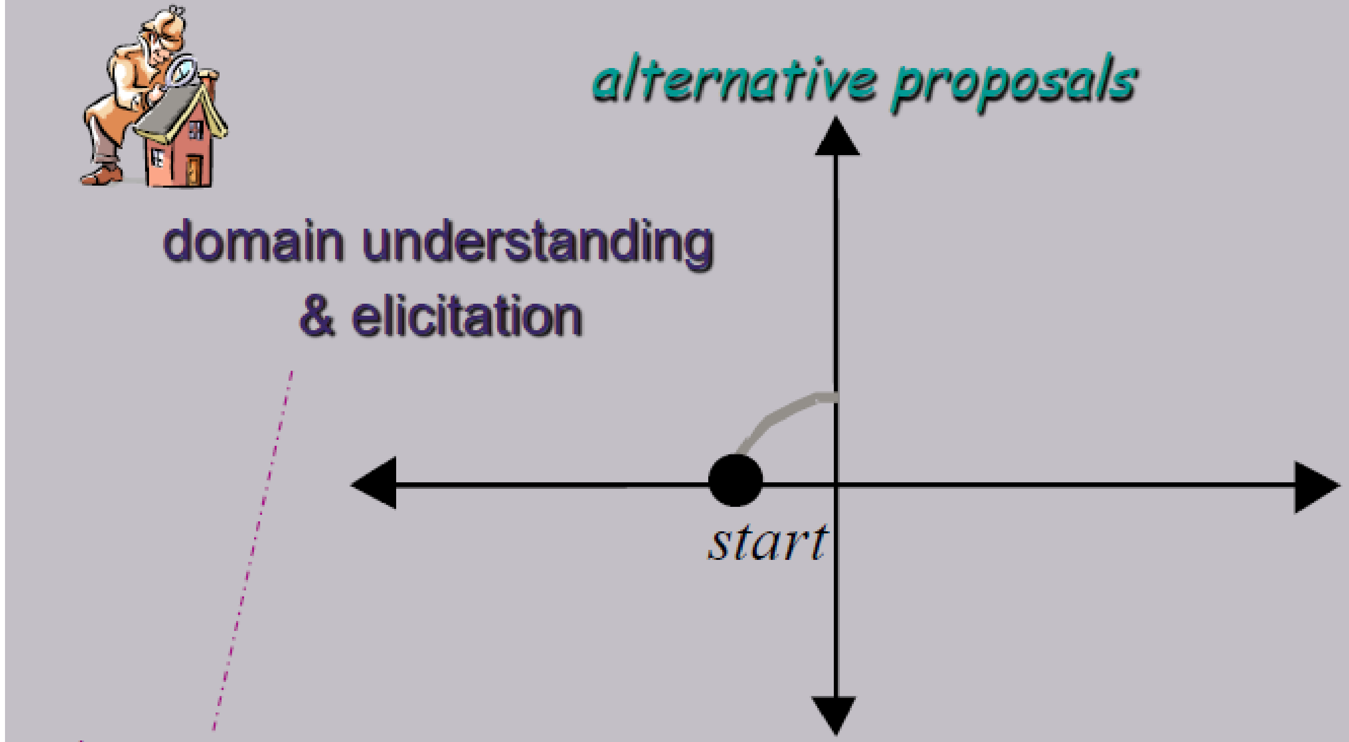
RE: Waterfall vs. Agile



Agile	Waterfall
• Continuous cycles • Small, high-functioning, collaborative teams • Multiple methodologies • Flexible/continuous evolution • Customer involvement	• Sequential/linear stages • Upfront planning and in-depth documentation • Contract negotiation • Best for simple, unchanging projects • Close project manager involvement

Source: officialconsumerreport.com

The RE process

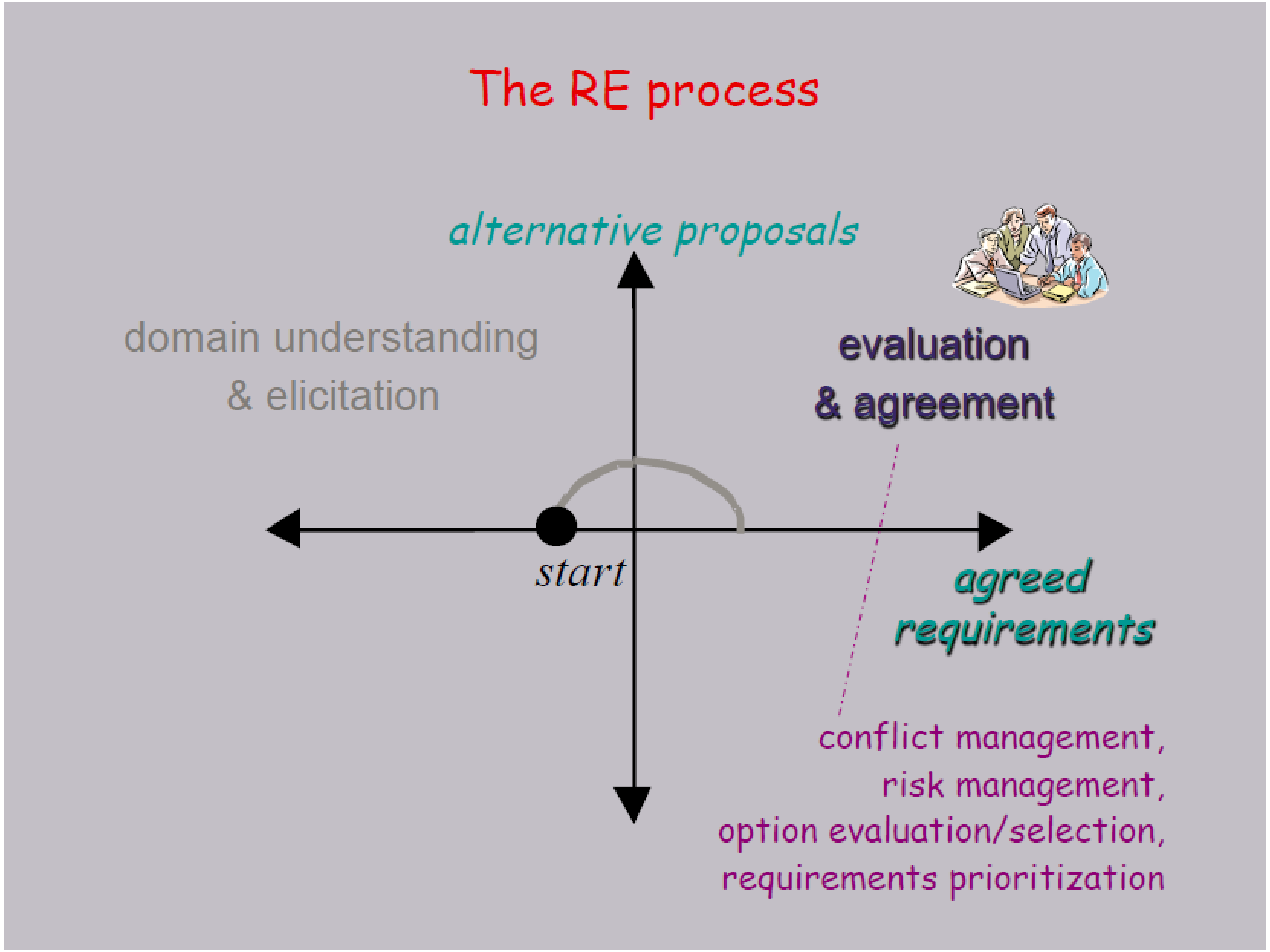


alternative proposals

domain understanding
& elicitation

start

business organization,
application domain, stakeholders
problems, improvement objectives, constraints,
alternative options, preliminary requirements



The RE process

alternative proposals

domain understanding
& elicitation

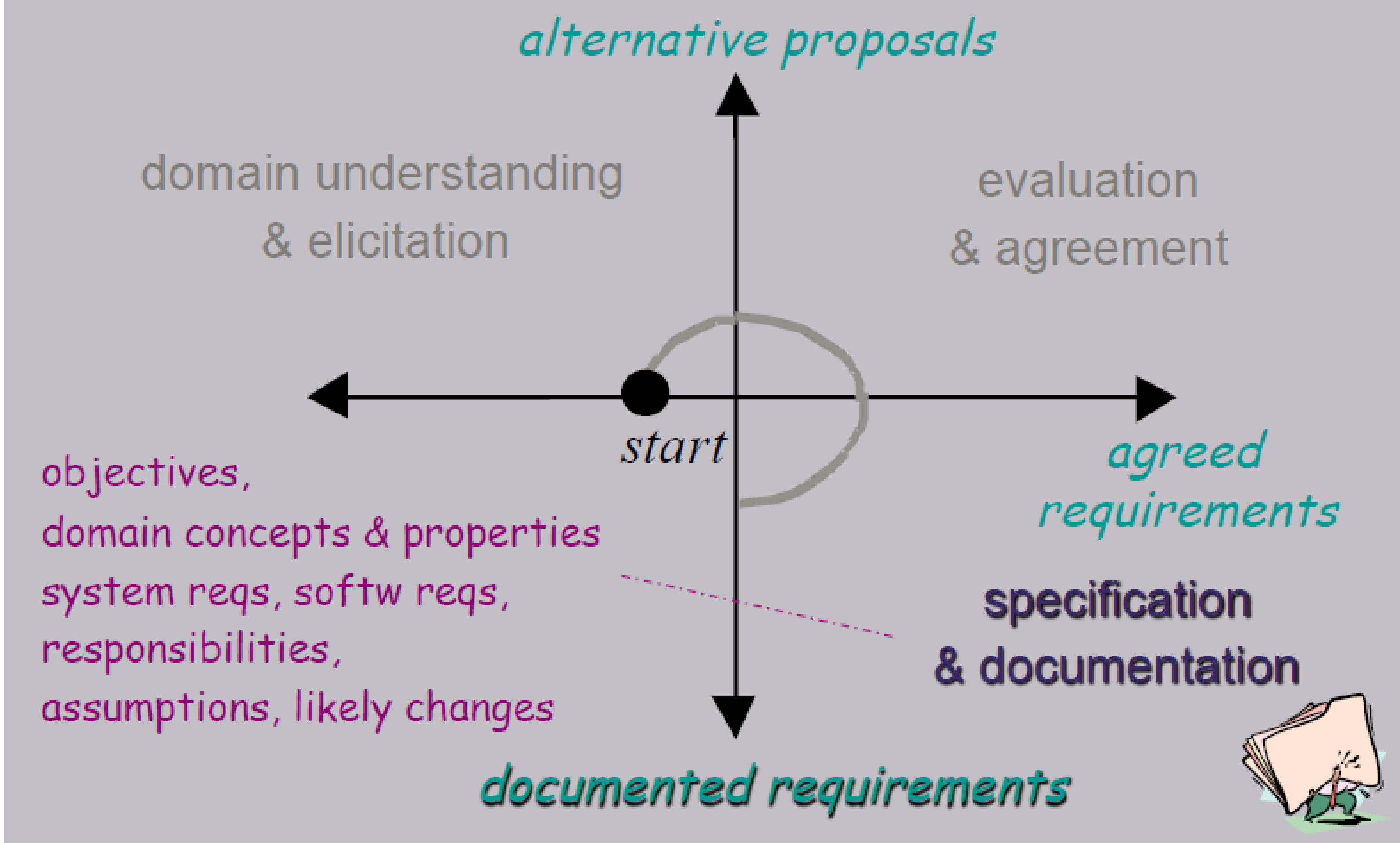
evaluation
& agreement

start

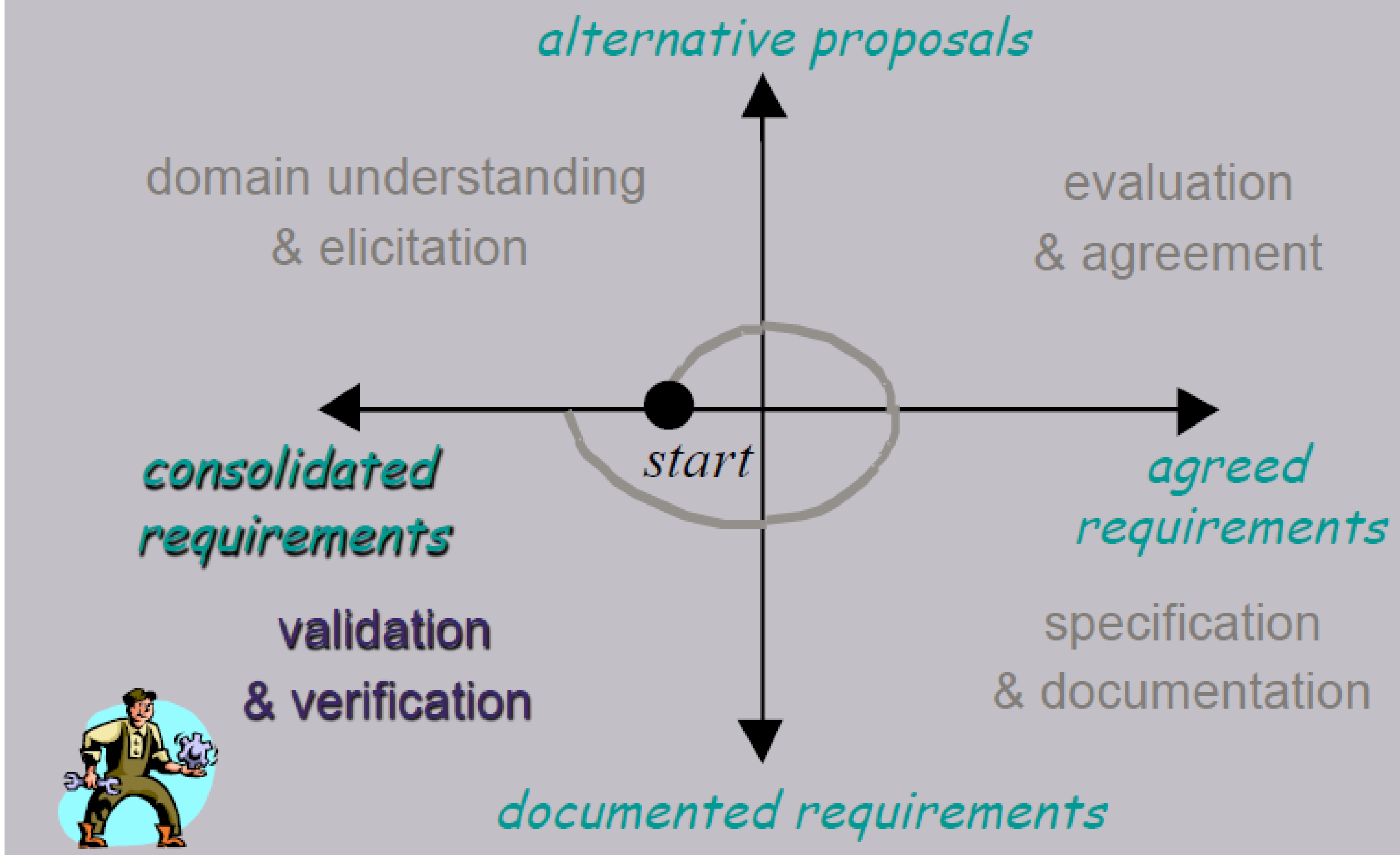
agreed
requirements

conflict management,
risk management,
option evaluation/selection,
requirements prioritization

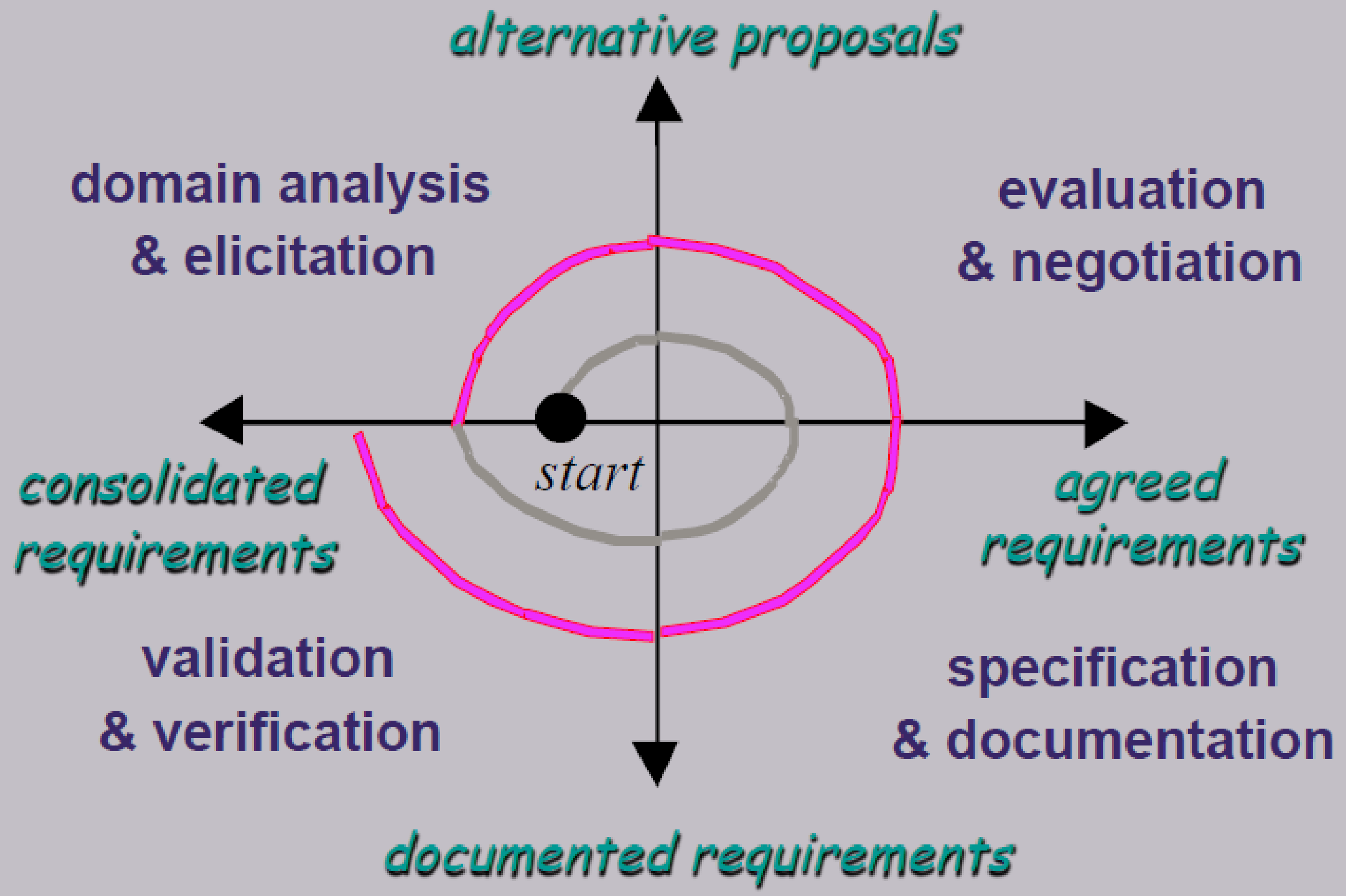
The RE process



The RE process


alternative proposals
domain understanding & elicitation
evaluation & agreement
start
consolidated requirements
agreed requirements
validation & verification
specification & documentation
documented requirements

RE is an iterative process



alternative proposals

domain analysis & elicitation

evaluation & negotiation

start

consolidated requirements

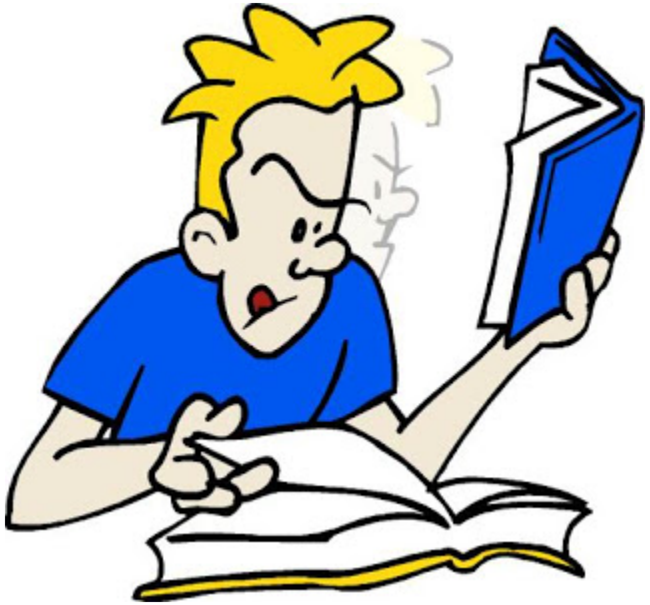
agreed requirements

validation & verification

specification & documentation

documented requirements

Why RE is hard?



- Broad scope
- Multiple concerns
- Multiple abstraction levels
- Multiple stakeholders
- Additional activities during the process

Why RE is important

- Legal impact
- Social impact
- Technical impact
- Impact on certification
- Impact on economy, security, and safety

Requirements error are the most dangerous software errors

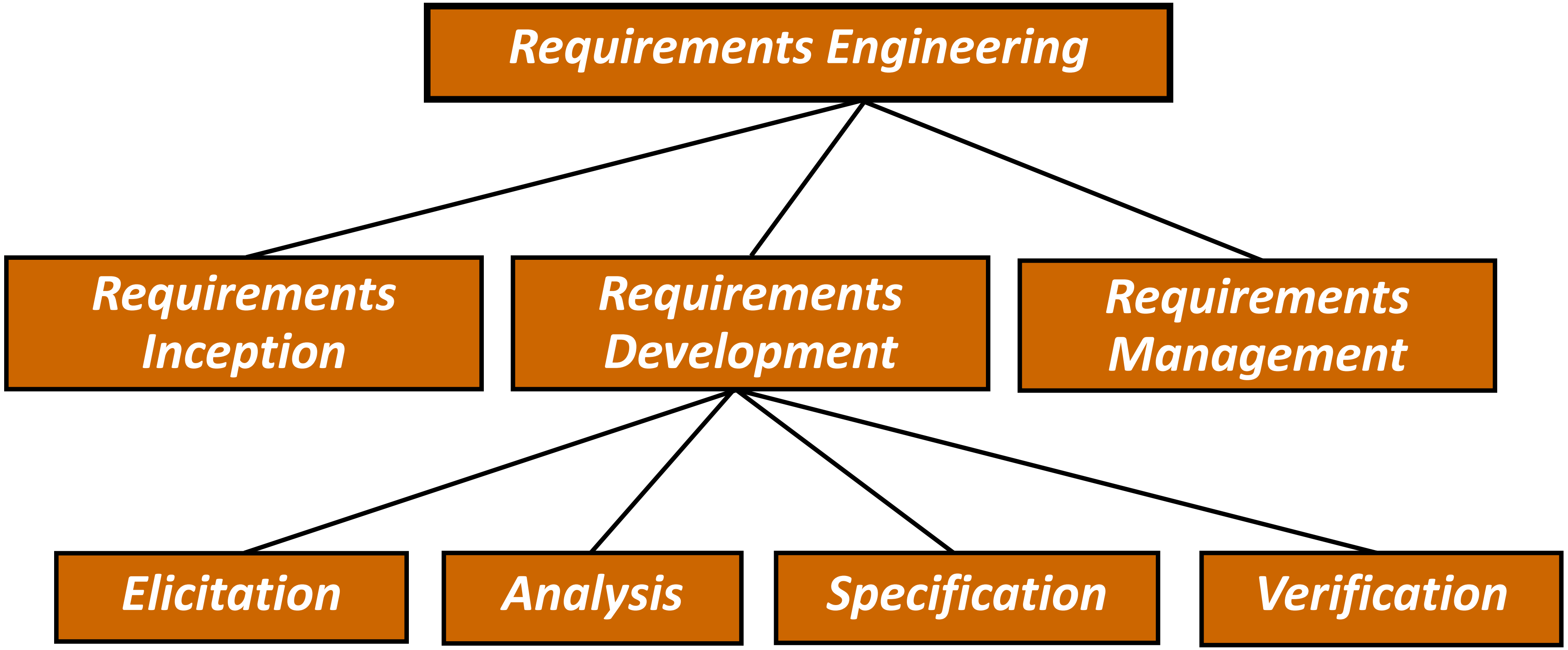
1. IranAir A300 Airbus was shot by US Vincennes in July 1988
2. First version of London ambulance dispatching system, with two tragic failures of the system (Oct-Nov 1992)
3. The Crash of an American Airlines Boeing 757 in Cali on Dec 1995
4. New York subway crash on June 1995
5. Ariane 5 Rocket failure

Statistics from NIST Report

- NIST (National Institute of Standards and Technology) has published a report on project statistics and experiences based on data from a large number of software projects[1]
  - 70% of the defects are introduced in the specification phase
  - 30% are introduced later in the technical solution process
  - Only 5% of the specification inadequacies are corrected in the specification phase
  - 95% are detected later in the project or after delivery where the cost for correction on average is 22 times higher compared to a correction directly during the specification effort
  - The NIST report concludes that extensive testing is essential, however testing detects the dominating specification errors late in the process

[1] http://www.nist.gov/public_affairs/releases/n02-10.htm (May 2002)

Requirements Engineering Activities



Functional vs. Non-functional requirements

- A functional requirement is a requirement defining functions of the system under development
  - Describes what the system should do
- A non-functional requirement is a requirement that is not functional. This includes many different kinds of requirements.

Functional Requirements

- What inputs should the system accept
- What outputs should the system produce
- What data should the system store other systems might use
- What computations should the system perform
- The timing and synchronization of the above

- Depend on the type of software, expected users, and the type of system where the software is used

Non-Functional Requirements (NFR)

- Non-functional requirements are important
  - If they are not met, the system is useless
  - Non-functional requirements may be challenging to state precisely (especially at the beginning), and imprecise requirements may be difficult to verify
- They are sometimes called quality requirements, quality of service, or extra-functional requirements.
- Will talk about them in detail later on in the term!

Errors in a requirements document (RD)

- **Omission**: problem world feature not stated by any RD item

  e.g. no requirements about the state of train doors in case of an emergency stop

- **Contradiction**: RD items stating a problem world feature in an incompatible way

  "Doors must always be kept closed between platforms."
  and "Doors must be opened in case of an emergency stop"

- **Inadequacy**: RD item not adequately stating a problem world feature

  "Panels inside trains shall display all flights served at the next stop."

- **Ambiguity**: RD item allowing a problem world feature to be interpreted in different ways

  "Doors shall be open as soon as the train is stopped at the platform."

- **Un-measurability**: RD item stating a problem world feature in a way precluding option comparison or solution testing

  "Panels inside trains shall be user-friendly."

Flaws in a requirements document (RD)

- **Noise**: RD item yielding no information on any problem world feature (Variant: uncontrolled redundancy)

  "Non-smoking signs shall be posted on train windows."

- **Overspecification**: RD item stating a feature not in the problem world but in the machine solution

  "The *setAlarm* method shall be invoked on receipt of an *Alarm* message."

- **Unfeasibility**: RD item not implementable within budget/schedule

  "In-train panels shall display all delayed flights at the next stop."

- **Unintelligibility**: RD item is incomprehensible to those needing to use it

  A requirement statement containing five acronyms

- **Poor structuring**: RD items are not organized according to any sensible or visible structuring rule

  Requirements that do not follow a structural rule (paragraphs, bullet points, subject-verb-object, "shall")

Flaws in a requirements document (2)

- **Forward reference**: RD item making use of problem world features not defined yet

  Multiple uses of the concept of *worst-case stopping distance* appear several pages after in the RD before its definition

- **Remorse**: RD item stating a problem world feature lately or incidentally

  After multiple uses of the undefined concept of *worst-case stopping distance*, the last one is directly followed by an incidental definition between parentheses

- **Poor modifiability**: RD items whose changes must be propagated throughout the RD

  Use of fixed numerical values for quantities subject to change

- **Opacity**: RD item whose rationale, authoring or dependencies are invisible

  "The commanded train speed must always be at least seven mph above physical speed" *without* any explanation of the rationale for this

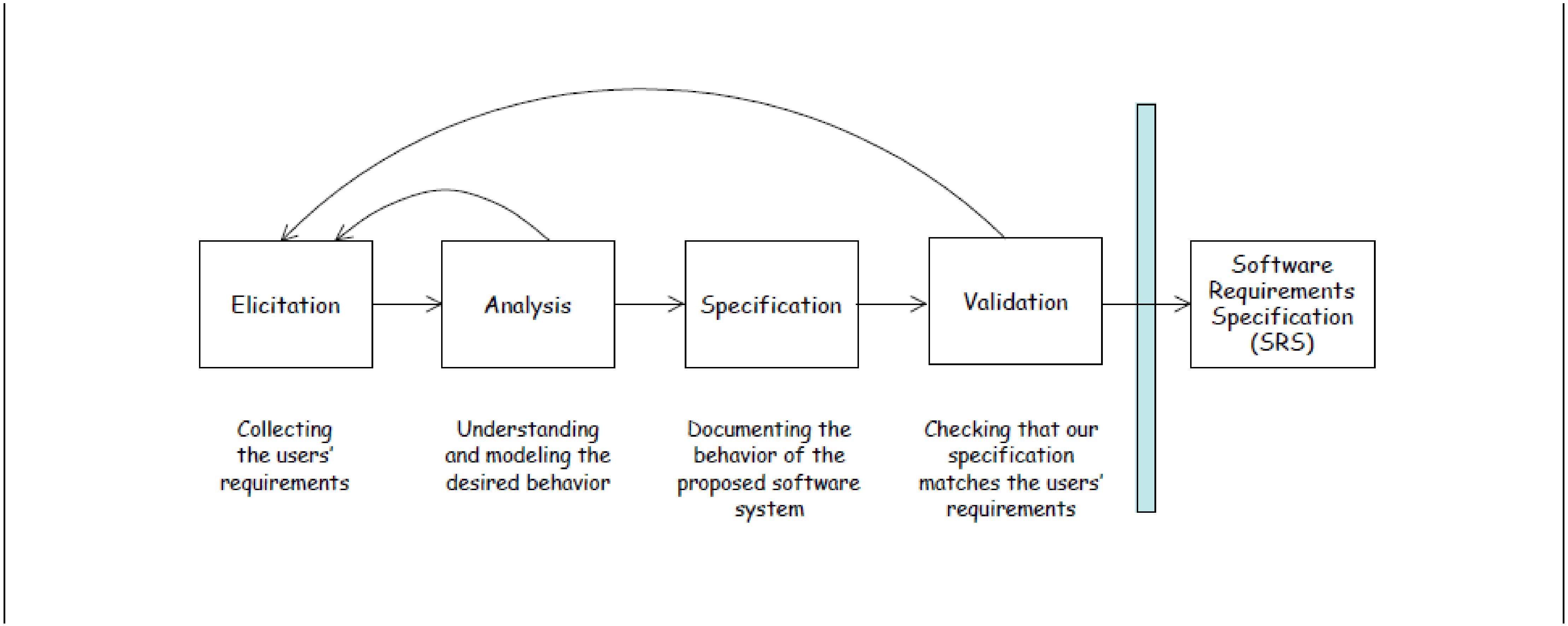
Course Goals

- To *understand* the stakeholders’ needs and expectations
  - Users, clients, etc.

- To *determine* the software system’s requirements

- To *document* the software system’s requirements

Requirements Engineering Process



Elicitation

Analysis

Specification

Validation

Software Requirements Specification (SRS)

Collecting the users' requirements

Understanding and modeling the desired behavior

Documenting the behavior of the proposed software system

Checking that our specification matches the users' requirements

CS 445 / ECE 451 / CS 645

Software Requirements Specifications & Analysis

Introduction