Requirements Engineering

2020/2021
Course 3

Course 3 outline

- Types of Requirements
- Structure of System Specification Document
- Examples of SRS
- Requirements Specification
    - Engineered Use Cases

Course 3 Bibliography

1. Ian Sommerville, Software Engineering (9th edition), Addison-Wesley, 2011
2. Suzanne Robertson, James Robertson, Mastering the Requirements Process (3nd Edition), Addison-Wesley Professional, 2012
3. C. Williams, M. Kaplan, T. Klinger, A. Paradkar, "Toward Engineered, Useful Use Cases", in Journal of Object Technology, Vol. 4, No. 6, Special Issue: Use Case Modeling at UML-2004, Aug 2005 , pp. 45-57 http://www.jot.fm/issues/issue_2005_08/article4
4. Klaus Pohl, Requirements Engineering, Springer, 2010

Types of Requirements

- There are many different classifications for requirements.
  - Business Requirements
  - Market Requirements
  - User requirements
  - System requirements
  - Functional requirements
  - Non-functional requirements
  - Domain requirements
  - UI Requirements
  - etc.

Types of Requirements

- User requirements:
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints.
  - Written for customers.
  - Eg. The software must provide a means of representing and accessing external files created by other tools.
  - Readers: client managers, system end-users, client engineers, contractor managers, system architects
- System requirements:
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
  - Defines what should be implemented, so it may be part of a contract between client and contractor.
  - Eg. The user shall be provided with facilities to define the type of external file. Each external file may have an associated tool which may be applied to the file.
  - Readers: system end-users, client engineers, system architects, software developers

Types of Requirements

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
- Domain requirements
  - Requirements that come from the application domain of the system and that reflect characteristics of that domain.

Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.
- Eg:
  - A user shall be able to search the appointments lists for all clinics.
  - The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
  - Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Eg. The term ‘search’ in requirement 1
  - User intention - search for a patient name across all appointments in all clinics;
  - Developer interpretation - search for a patient name in an individual clinic. User chooses clinic then search.
- Requirements should be both complete and consistent.
  - Complete: they should include descriptions of all facilities required.
  - Consistent: there should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is almost impossible to produce a complete and consistent requirements document.

Non-functional requirements

- They define system properties and constraints.
- E.g. reliability, response time and storage requirements, I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional requirements implementation

- Non-functional requirements may affect the overall architecture of a system rather than the individual components.

  E.g., to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define what system services are required.

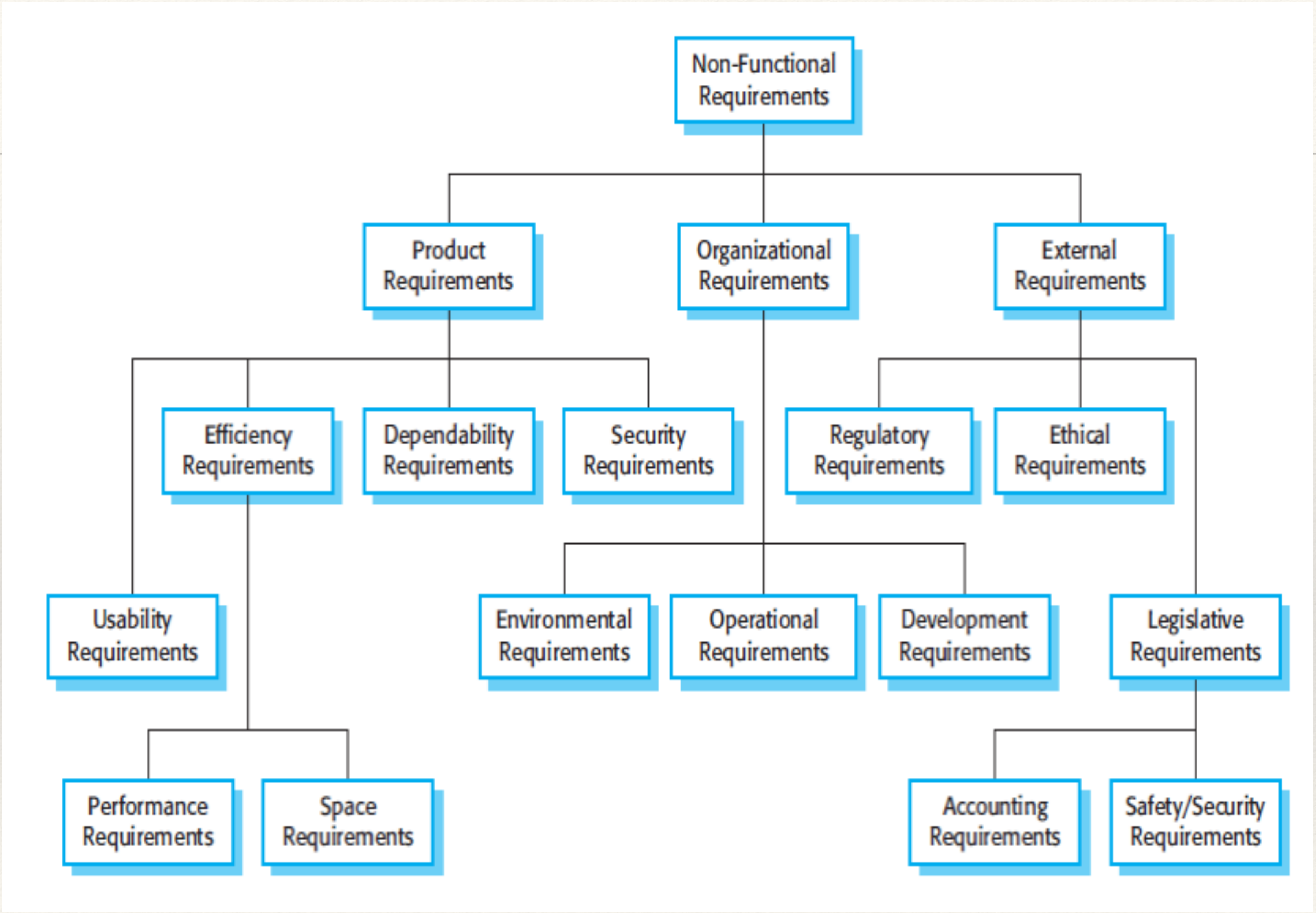
  It may also generate requirements that restrict existing requirements.

Non-functional classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements

- Product requirement
  - The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.
- Organisational requirement
  - Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.
- External requirement
  - The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.


Non-Functional Requirements
Product Requirements
Organizational Requirements
External Requirements
Efficiency Requirements
Dependability Requirements
Security Requirements
Regulatory Requirements
Ethical Requirements
Usability Requirements
Environmental Requirements
Operational Requirements
Development Requirements
Legislative Requirements
Performance Requirements
Space Requirements
Accounting Requirements
Safety/Security Requirements

Goals and requirements

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal:
  - A general intention of the user such as ease of use.

  Eg. The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.
- Verifiable non-functional requirement:
  - A statement using some measure that can be objectively tested.

  Eg. Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.
- Goals are helpful to developers as they convey the intentions of the system users.

Property	**Measure**
Speed	• Processed transactions/second • User/event response time • Screen refresh time
Size	• K Bytes • Number of RAM chips
Ease of use	• Training time • Number of help frames
Reliability	• Mean time to failure • Probability of unavailability • Rate of failure occurrence • Availability
Robustness	• Time to restart after failure • Percentage of events causing failure • Probability of data corruption on failure
Portability	• Percentage of target dependent statements • Number of target systems

Requirements interaction

- Conflicts between different non-functional requirements are common in complex systems.
- Eg. Spacecraft system
  - To minimize weight, the number of separate chips in the system should be minimized.
  - To minimize power consumption, lower power chips should be used.
  - However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?

Domain requirements

- Derived from the application domain and describe system characteristics and features that reflect the domain.
- Domain requirements may be new functional requirements, constraints on existing requirements or define specific computations.
- If domain requirements are not satisfied, the system may be unworkable.
- Eg. Library system:

  There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.

  Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Domain requirements problems

- Understandability
  - Requirements are expressed in the language of the application domain;

  Eg. The deceleration of the train shall be computed as:

  $D_{train} = D_{control} + D_{gradient}$

  where $D_{gradient}$ is 9.81ms2 * compensated gradient/alpha and where the values of 9.81ms2 /alpha are known for different types of train.

  - This is often not understood by software engineers developing the system.
- Implicitness
  - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

Other classifications*

- Functional requirements
  - specify the functionality the system shall provide to its users (persons or other systems)

  Eg. If a sensor detects that a glass pane is damaged or broken, the system shall inform the security company.
- Quality requirements
  - defines a quality property of the entire system or of a system component, service or function
- Constraints
  - A constraint is an organizational or technological requirement that restricts the way in which the system shall be developed.

***Klaus Pohl, Karl Wiegers**

Quality requirements

- Important primarily to users:
  - Availability, Efficiency, Flexibility, Integrity
  - Interoperability, Reliability, Robustness, Usability
- Important primarily to developers:
  - Maintainability, Portability, Reusability, Testability
- Non-functional req. vs quality req.
  - Non-functional requirements are either:
    - Underspecified functional requirements
    - Quality requirements

Quality requirements

Eg. Underspecified functional requirement:

“The system shall be secure”

- Each user must log in to the system with his user name and password prior to using the system. (functional req.)
- The system shall remind the user every four weeks to change the password. (functional req.)
- When the user changes his password, the system shall validate that the new password is at least eight characters long and contains alphanumerics characters. (functional req.)
- The user passwords stored in the system must be protected against password theft. (quality req. – integrity)

Constraints - examples

- Constraints affecting the system:
    - A fire protection requirements demands that the terminals in the sales rooms do not exceed the size: 120 cm (height) x 90 cm (width) x 20 cm (depth)
- Constraints affecting the development process:
    - The effort for the development of the system must not exceed 480 person-months.
    - The system must be developed using the Rational Unified Process.

Requirements Recap

- User Requirements:
  - Should describe functional and non-functional requirements in such a way that they are understood by system users who do not have detailed technical knowledge.
  - User requirements are defined using natural language, tables and diagrams so that they can be understood by all users.
- System Requirements:
  - More detailed specifications of system functions, services and constraints than user requirements.
  - They are intended to be a basis for designing the system.
  - They may be incorporated into the system contract.
  - System requirements may be defined or illustrated using system models.

The software requirements document

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is not a design document.
- It should set what the system should do rather than how it should do it.
- Information in requirements document depends on the type of system and the approach to development used.
- Systems developed incrementally will, typically, have less detail in the requirements document.
- Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

Users of a requirements document

- System customers
  - Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
- Managers
  - Use the requirements document to plan a bid for the system and to plan the system development process.
- System engineers
  - Use the requirements to understand what system is to be developed.
- System test engineers
  - Use the requirements to develop validation tests for the system.
- System maintenance engineers
  - Use the requirements to understand the system and the relationships between its parts.

Structure of a requirements document

- Different templates:
  - IEEE 830 Standard
  - Volere Template
  - US Department of Defense
  - etc.

IEEE requirements standard

- Defines a generic structure for a requirements document that must be instantiated for each specific system.

1 Introduction

1.1 Purpose of the Software Requirement Specification

1.2 Scope of the Product

1.3 Definitions, Acronyms and Abbreviations

1.4 References

1.5 Overview of the Rest of the Software Requirement Specification

2 General Description

2.1 Product Perspective

2.2 Product Functions

2.3 User Characteristics

2.4 General Constraints

2.5 Assumptions and Dependencies

3 Specific Requirements

Appendices

Index.

Requirements document structure

- Preface
  - Should define the expected readership of the document and describe its version history.
- Introduction
  - Should describe the need for the system. It should briefly describe the system's function and explain how it will work with other systems.
- Glossary
  - Should define the technical terms used in the document. It should not make any assumptions about the experience or expertise of the reader.
- User requirements definition:
  - Describe the services provided for the user, including non-functional system requirements.
- System architecture
  - Should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules.

Requirements document structure (cont.)

- System requirements specification
    - Should describe the functional and nonfunctional requirements in more detail
- System models
    - May include graphical system models showing the relationships between the system components, the system and its environment.
- System evolution
    - Should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, etc.
- Appendices
    - Should provide detailed, specific information that is related to the application being developed (i.e. hardware and database descriptions)
- Index
    - Several indexes to the document may be included (alphabetic index, index of diagrams, index of functions, etc.)

Volere Template Specification v16

Project Drivers

1. The Purpose of the Project
2. The Stakeholders

Project Constraints

3. Mandated Constraints
4. Naming Conventions and Terminology
5. Relevant Facts and Assumptions

Functional Requirements

6. The Scope of the Work
7. Business Data Model & Data Dictionary
8. The Scope of the Product
9. Functional Requirements

Non-functional Requirements

10. Look and Feel Requirements
11. Usability and Humanity Requirements
12. Performance Requirements
13. Operational and Environmental Requirements
14. Maintainability and Support Requirements
15. Security Requirements

Volere Template Specification v16 (cont.)

16. Cultural Requirements
17. Legal Requirements

Project Issues

18. Open Issues
19. Off-the-Shelf Solutions
20. New Problems
21. Tasks
22. Migration to the New Product
23. Risks
24. Costs
25. User Documentation and Training
26. Waiting Room
27. Ideas for Solutions

Examples Software Requirements Specification

- A few examples of SRS can be found on: labor/Romana/Master/is258/ReqEngineering/courses
- MS Teams Class Materials

Requirements Specification

- The process of writing down the user and system requirements in a requirements document.
- User requirements have to be understood by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development. It is important that these are as complete as possible.
- Ways of writing a system requirements specification:
  - Natural language
  - Structured natural language
  - Design description languages
  - Graphical notations
  - Mathematical specifications
  - others

Natural language specification

- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Each sentence should express one requirement.
- Used for writing requirements because it is expressive, intuitive and universal.
- The requirements can be understood by users and customers.
- Guidelines:
  - Invent a standard format and use it for all requirements.
  - Use language in a consistent way. Use *shall* for mandatory requirements, *should* for desirable requirements.
  - Use text highlighting to identify key parts of the requirement.
  - Include an explanation (rationale) of why a requirement is necessary.
  - Avoid putting more than one requirement in a paragraph: often indicated by the presence of the word "and".
  - Avoid vague words: usually, generally, often, normally, typically;
  - Avoid vague terms: user friendly, versatile, flexible;
  - Avoid wishful thinking: 100% reliable, please all users, safe, run on all platforms, etc.

Natural language specification

- Problems with natural language specification:
- Lack of clarity:
    - Precision is difficult without making the document difficult to read.
- Requirements confusion
    - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
    - Several different requirements may be expressed together.

Eg.

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured specifications

- An approach to writing requirements where the freedom of the requirements writer is limited, and requirements are written in a standard way.
- The terminology used in the description may be limited.
- The advantage is that most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.
- DOORS (Dynamic Object Oriented Requirements System):
  - Uses boilerplates to standardize the language used for requirements.

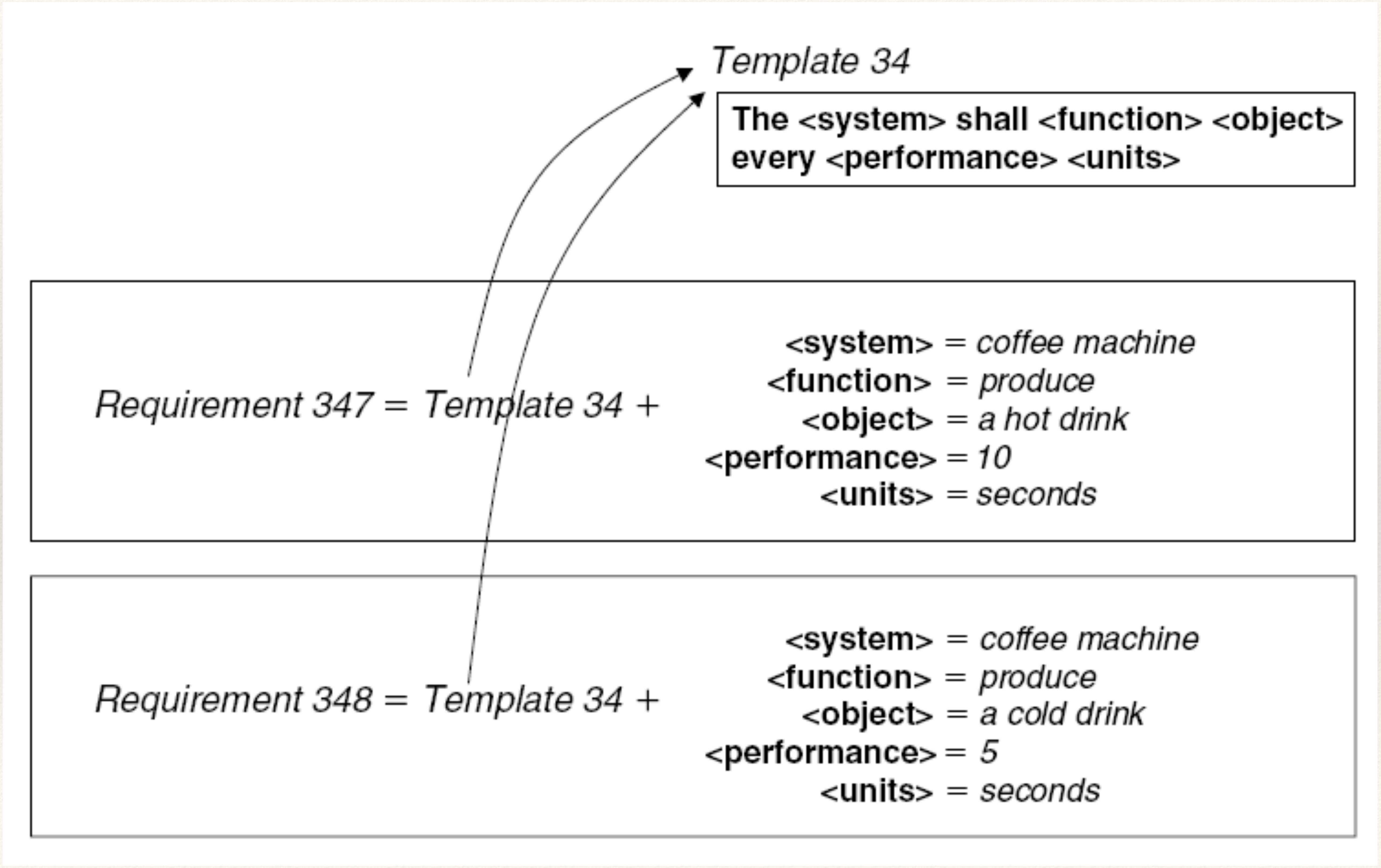
Eg.

The <system> shall be able to <function> <object> within <performance> <units> from <event>

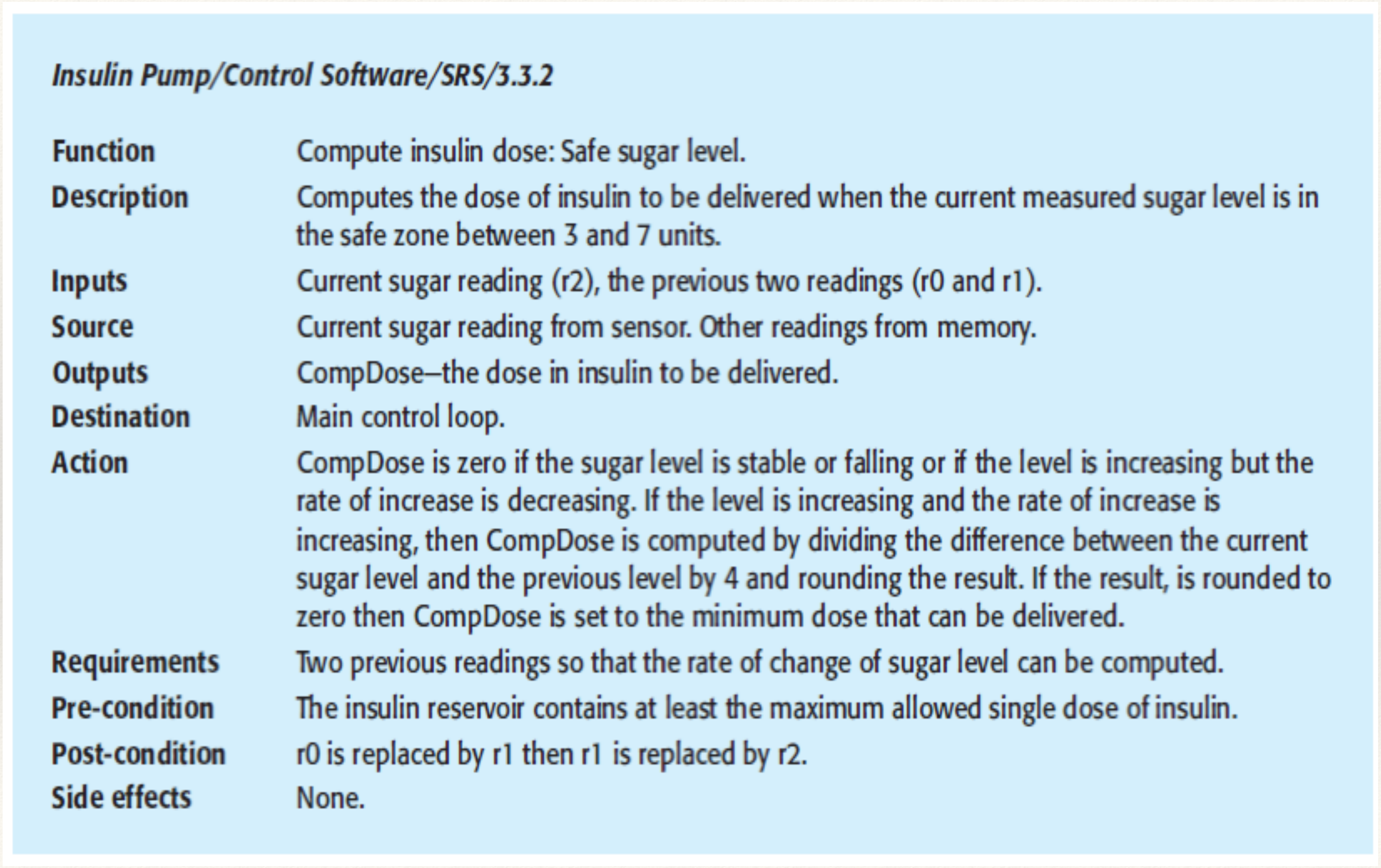
The <system> shall be able to <function> <object> while

<operational condition>

Structured specifications


Template 34
The <system> shall <function> <object> every <performance> <units>
Requirement 347 = Template 34 +
<system> = coffee machine
<function> = produce
<object> = a hot drink
<performance> = 10
<units> = seconds
Requirement 348 = Template 34 +
<system> = coffee machine
<function> = produce
<object> = a cold drink
<performance> = 5
<units> = seconds

Structured specifications

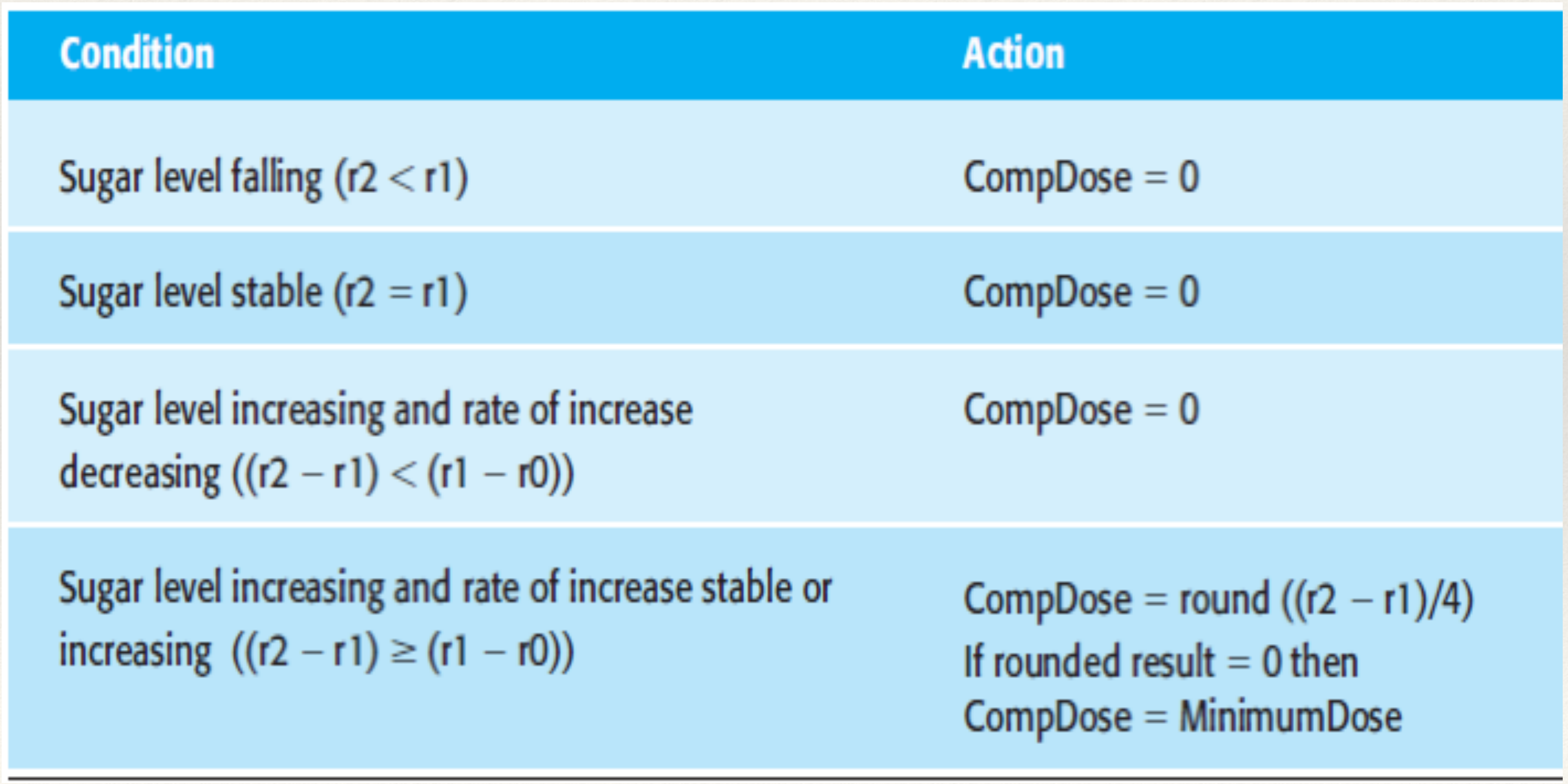


Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose–the dose in insulin to be delivered.
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Tabular specification

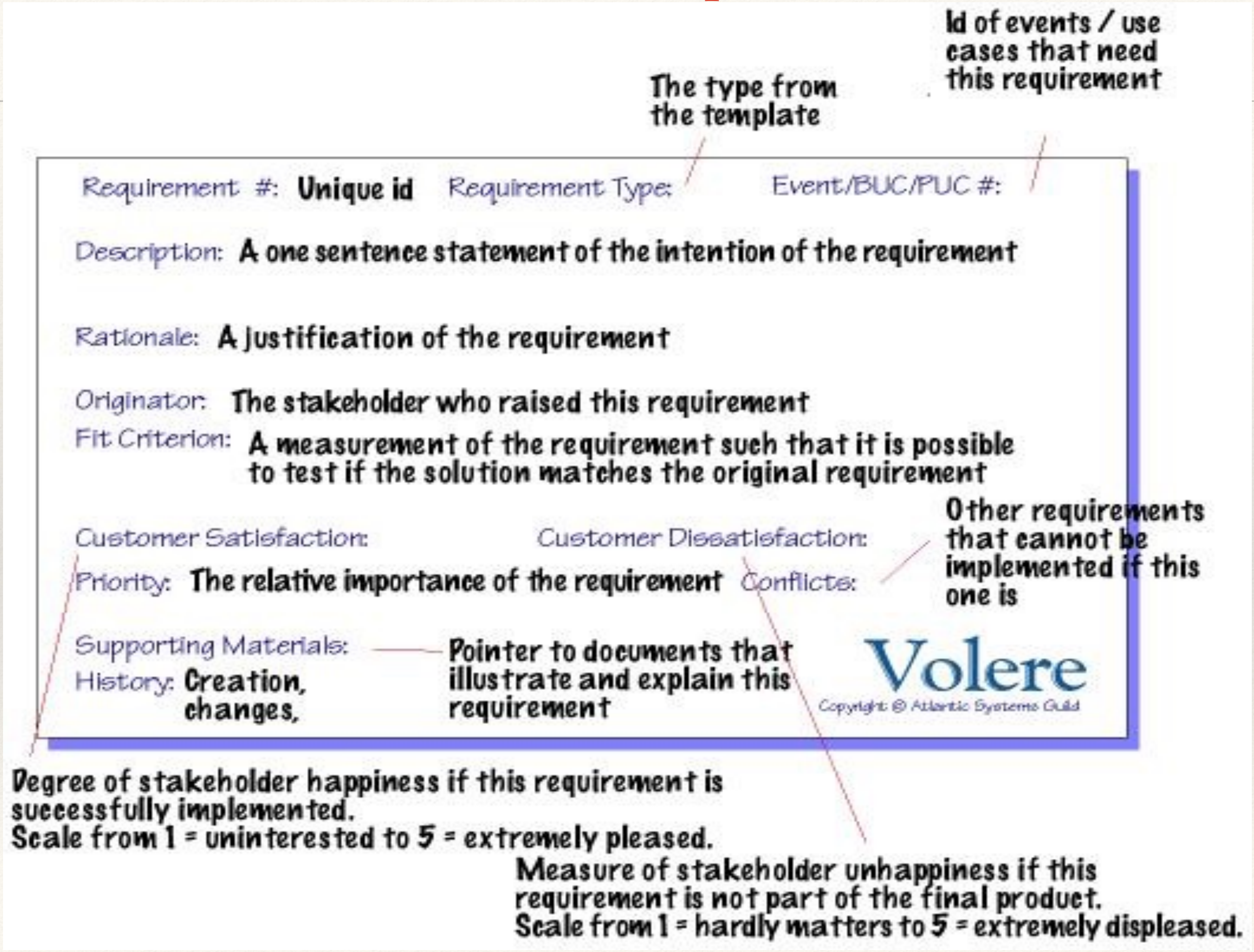
- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.



Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0

Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4)
	If rounded result = 0 then
	CompDose = MinimumDose

Volere atomic requirement shell



Id of events / use cases that need this requirement
The type from the template
Requirement #: Unique id
Requirement Type:
Event/BUC/PUC #:
Description: A one sentence statement of the intention of the requirement
Rationale: A justification of the requirement
Originator: The stakeholder who raised this requirement
Fit Criterion: A measurement of the requirement such that it is possible to test if the solution matches the original requirement
Customer Satisfaction:
Customer Dissatisfaction:
Other requirements that cannot be implemented if this one is
Priority: The relative importance of the requirement
Conflicts:
Supporting Materials:
Pointer to documents that illustrate and explain this requirement
History: Creation, changes,
Volere
Copyright © Atlantic Systems Guild
Degree of stakeholder happiness if this requirement is successfully implemented. Scale from 1 = uninterested to 5 = extremely pleased.
Measure of stakeholder unhappiness if this requirement is not part of the final product. Scale from 1 = hardly matters to 5 = extremely displeased.

Volere atomic requirement shell

Description and Rationale
Examples:

A. Description: The product shall record roads that have been treated.
   Rationale: To be able to schedule untreated roads and highlight potential danger.

B. Description: The product shall record the start and end time of a truck’s scheduled activity.

   Rationale A: The truck depot foreman wants to know which trucks are being most used.

   Rationale B: Trucks are to be scheduled a maximum of 20 out of 24 hours to allow for maintenance and cleaning.

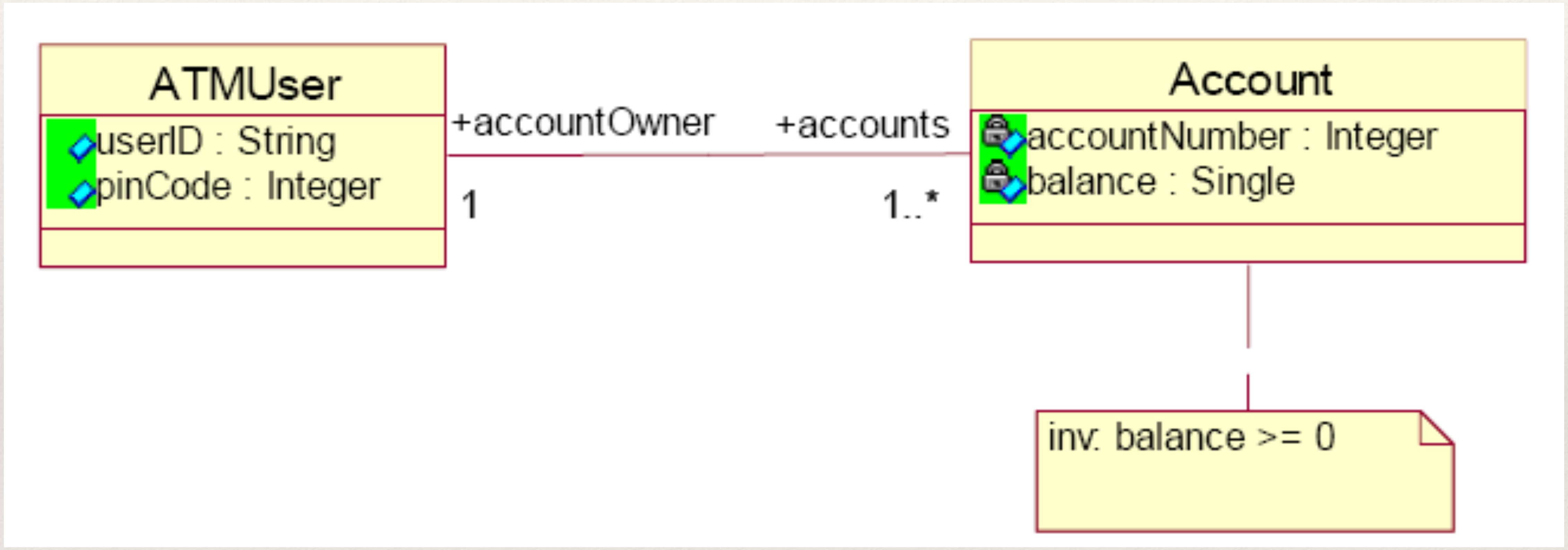
Use Cases

- Use cases were developed as a technique for capturing the required behavior of a software system.
- They allow the system requirements to be specified by saying what the system will do without saying how it does it.
- Their usage is not as widespread as it could be.
- Several reasons:
    - Confusion regarding the meaning of the term use case. Different approaches advocate graphical versus textual methods.
    - Misaligned characterizations of use cases in the UML metamodel. In UML, use cases are characterized as BehavioredClassifiers, which does not correctly support the best practices for representing use cases that have been proposed.
    - Use case semantics are poorly defined, limiting their usefulness primarily for communication. While this is an important use for them, wider industrial use requires that they support machine based processing and analysis.

Use Cases

- Basic use case content:
  - Preconditions,
  - Main (success) sequences,
  - Alternative/exceptional sequences,
  - Postconditions.
- For engineered use cases more details are needed to define what the sequences contain, i.e., domain model
- A domain model is a model of the significant elements from the system's application domain: the elements that will be created, modified, and used in the application.
- They are represented as UML classes with attributes.
- In the domain model are also present the domain rules (or invariants) which capture constraints that must hold for instances of the domain objects to be valid.

Use Cases


ATMUser
userID : String
pinCode : Integer
+accountOwner
+accounts
1
1..*
Account
accountNumber : Integer
balance : Single
inv: balance >= 0

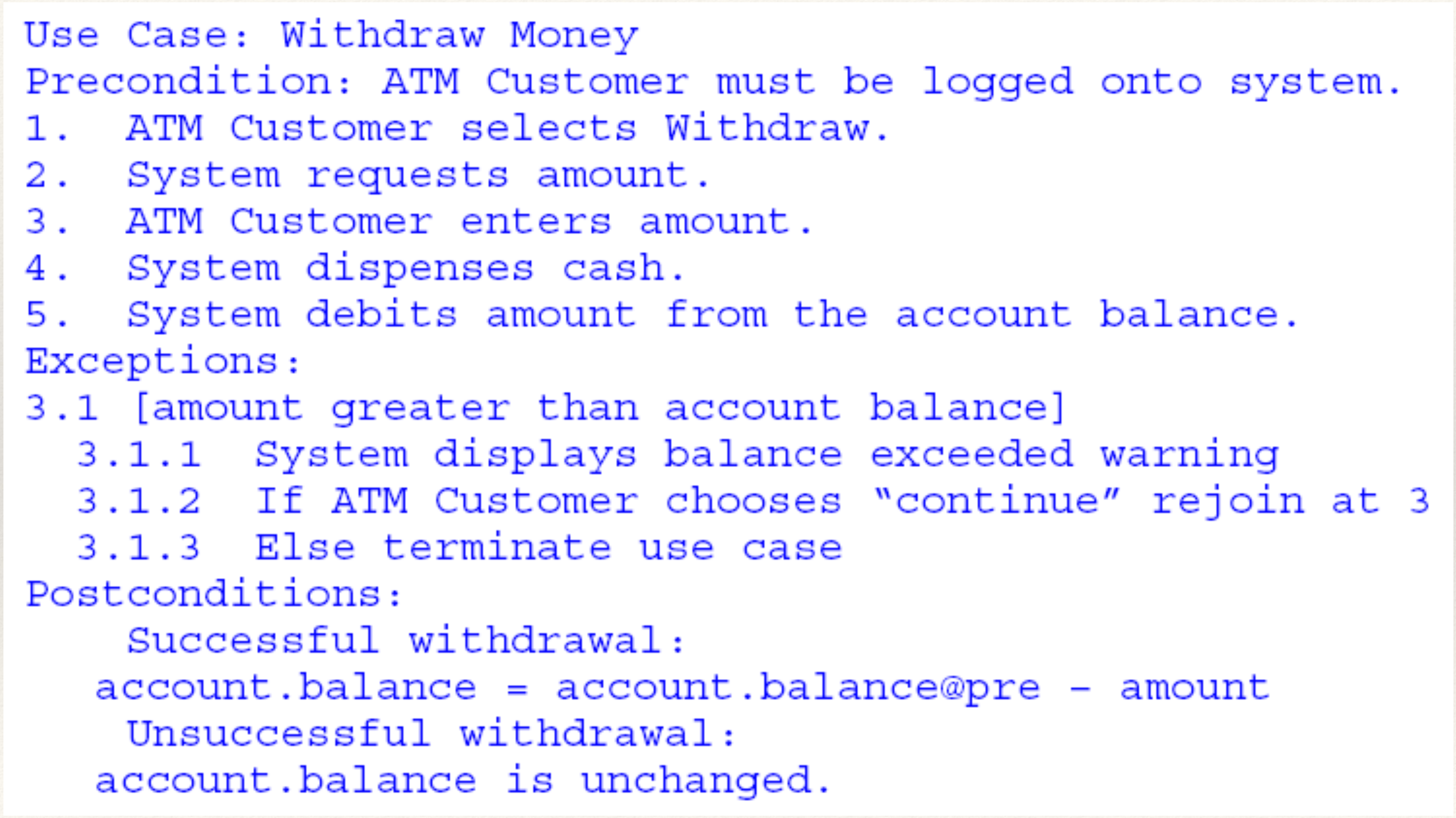
Use Cases

- In UML use cases are typically defined as sequences of actions that occur between the system and one or more actors, but the specifics of the action types that are supported are left undefined.
- For engineered UC:
  - Use cases support 4 basic actions and 4 flow-of-control actions, each of which has an associated statement for use in detailing the use case.
- Basic Actions:
  - Input – an actor provides input to the use case
  - Output – the use case returns output to an actor
  - Computation – a computation is performed using provided input and domain instance information. Computation entails creating, modifying, or deleting instances of domain classes.
  - Exception Handling – the system responds to an issue with the input or state of the domain model instances.

Use Cases

- Flow of Control Actions:
  - Selection – allows the use case to conditionally execute actions
  - Iteration – allows the use case to repeatedly execute an action sequence
  - Inclusion – allows the use case to include the behavior of another use case
  - Extension – defines the extension point for an extending use case.
- Engineered use cases also have preconditions and postconditions, and both are written in terms of the domain model.
- The representation (use cases, a high level domain model, and domain invariants) serves as a basis for engineered use cases.
- The use cases consist of very specific types of actions.
- They specify, in terms of the domain model, what occurs in an interaction with the system being specified without saying how it is done.

Engineered Use Case - Example



```
Use Case: Withdraw Money
Precondition: ATM Customer must be logged onto system.
1.   ATM Customer selects Withdraw.
2.   System requests amount.
3.   ATM Customer enters amount.
4.   System dispenses cash.
5.   System debits amount from the account balance.
Exceptions:
3.1 [amount greater than account balance]
   3.1.1   System displays balance exceeded warning
   3.1.2   If ATM Customer chooses "continue" rejoin at 3
   3.1.3   Else terminate use case
Postconditions:
      Successful withdrawal:
    account.balance = account.balance@pre - amount
      Unsuccessful withdrawal:
    account.balance is unchanged.
```

Precise Use Case Semantics

- In order to support an engineering approach to creating precise use cases, semantic issues in addition to the structural/content issues must be addressed.
- Execution Semantics:
  - For use cases to be precise, we must be able to describe what they mean when they are executed.
  - The execution of a use case is performed in the context of a state consisting of instances of domain model classes.
  - Domain objects may be created, modified, or deleted.
  - Links may be added, or deleted.
  - Attributes may be modified.

Precise Use Case Semantics

- Capabilities that describe the execution semantics of use cases:
  - Input / Output statements – instantiate formal parameter(s) specified in the statement with value(s) of the appropriate type(s).
  - Computation – create/delete an instance of a domain object, create/delete a link between existing domain object instances, modify the attribute of a domain object.
  - Exception – check exception conditions and fires if true.
- For a use case, execution proceeds as follows:
  - Each statement in the main scenario is executed.
  - During its execution, the alternatives / exceptions defined for that statement are checked for execution eligibility.
  - If one or more is eligible, the one that takes priority must be determined and that one must be executed. (The semantics for determining priority must be defined in the language.)
  - If the alternative specifies where control returns to, that is the point where execution continues, otherwise, it resumes at the next statement in the main interaction course.

Engineered Use Cases

- Applications:
    - prototyping, estimation, refinement to design, test generation.
- Prototyping: precise, engineered use cases can be used as the foundation for an executable prototype of the system.
- The steps toward building a useful prototype:
    1. Build the domain and use case models.
    2. Create mocked-up user interface (UI) elements for the input and output statements.
    3. Associate the UI elements with the input and output statements.
    4. Execute the prototype with all involved stakeholders, gathering important feedback.
    5. Modify the model according to the feedback and repeat the exercise until the requirements are agreed upon by all stakeholders.
- Advantage: it requires almost no effort beyond normal analysis and design activities to have an executable prototype for validating user requirements.

Engineered Use Cases

- Refinement to Design: refine the use cases to a detailed design of the system:
  1. The domain model is refined to a full analysis model: identification of boundary and controller classes that serve as additions to the model classes already present in the domain model. The boundary classes can be inferred from the UI elements.
  2. Operations for all of the classes in the analysis model are identified. These are informed by the computation actions in the use case model, as well as required controller and UI computations for the system.
  3. The input statements are mapped to operation invocations on the boundary classes.
  4. The output statements are mapped to returned parameters from or operation invocations on the boundary classes.
  5. Interaction diagrams are used to show how use cases are realized in the system using the mappings defined in steps 3-4 as a basis.
  6. Further refinement to a detailed design and implementation can proceed from the analysis model and use case realizations.

Engineered Use Cases

- Test Case Creation/Generation:
  - Precise test cases can be created from the engineered use cases, domain model, and domain invariants.
- Steps in test generation:
  1. For each use case, ensure that there are test cases for each path through the use case.
  2. For each condition in a use case (either selection or exception), ensure that condition coverage is achieved using a condition coverage technique.
  3. For each domain rule in the domain model, determine all use cases which update any variable in the domain rule. Determine the ways in which the implementation could be tested to ensure the rule holds, and perform those tests.
- Advantages:
  - Reduction in test suite size (fewer generated tests than manually written)
  - Time decreased to setup/run/verify test suite
  - Some defects detected by generated tests, but not by manually written tests