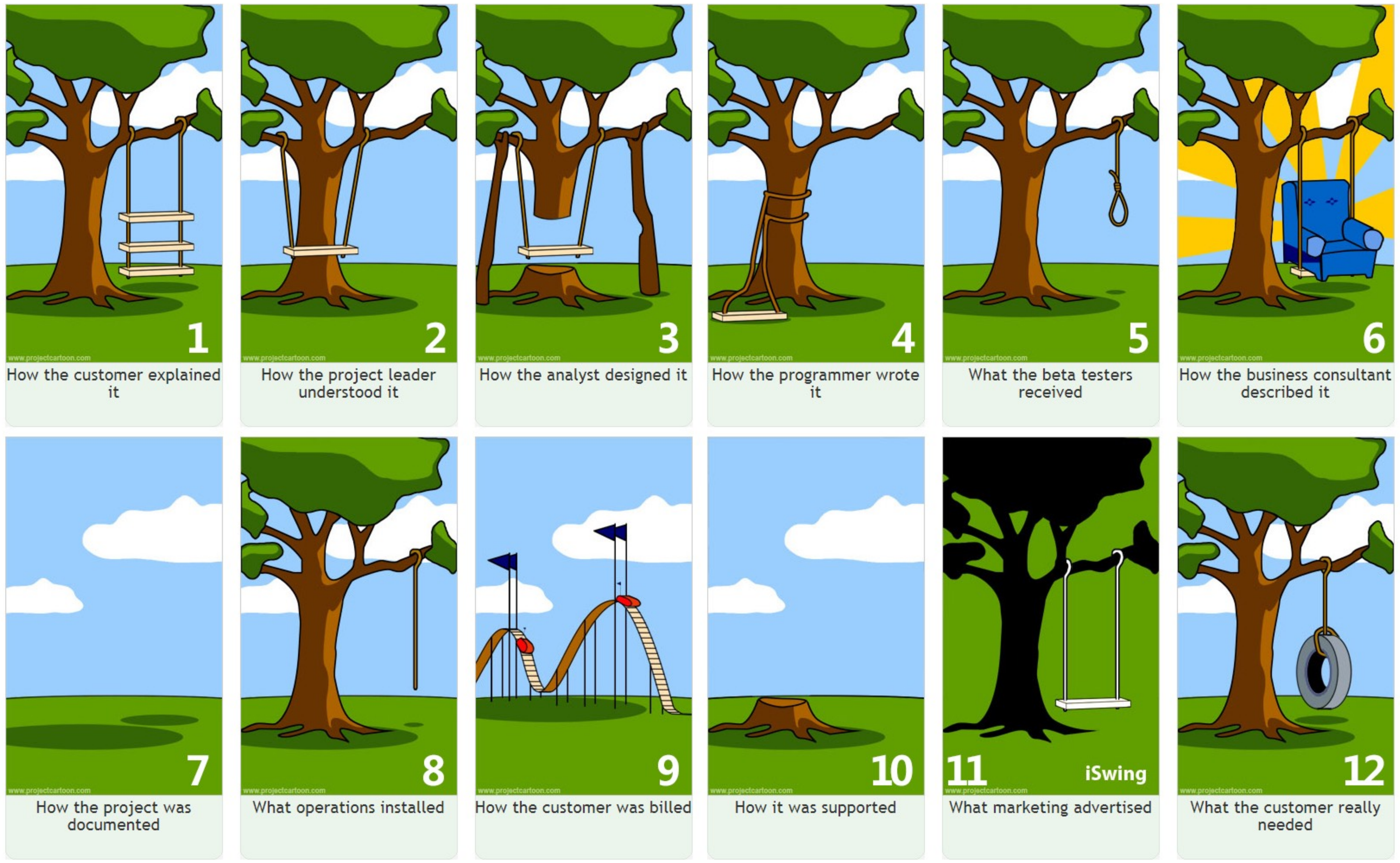
Requirement Engineering

CS350 Introduction to Software Engineering

Shin Yoo

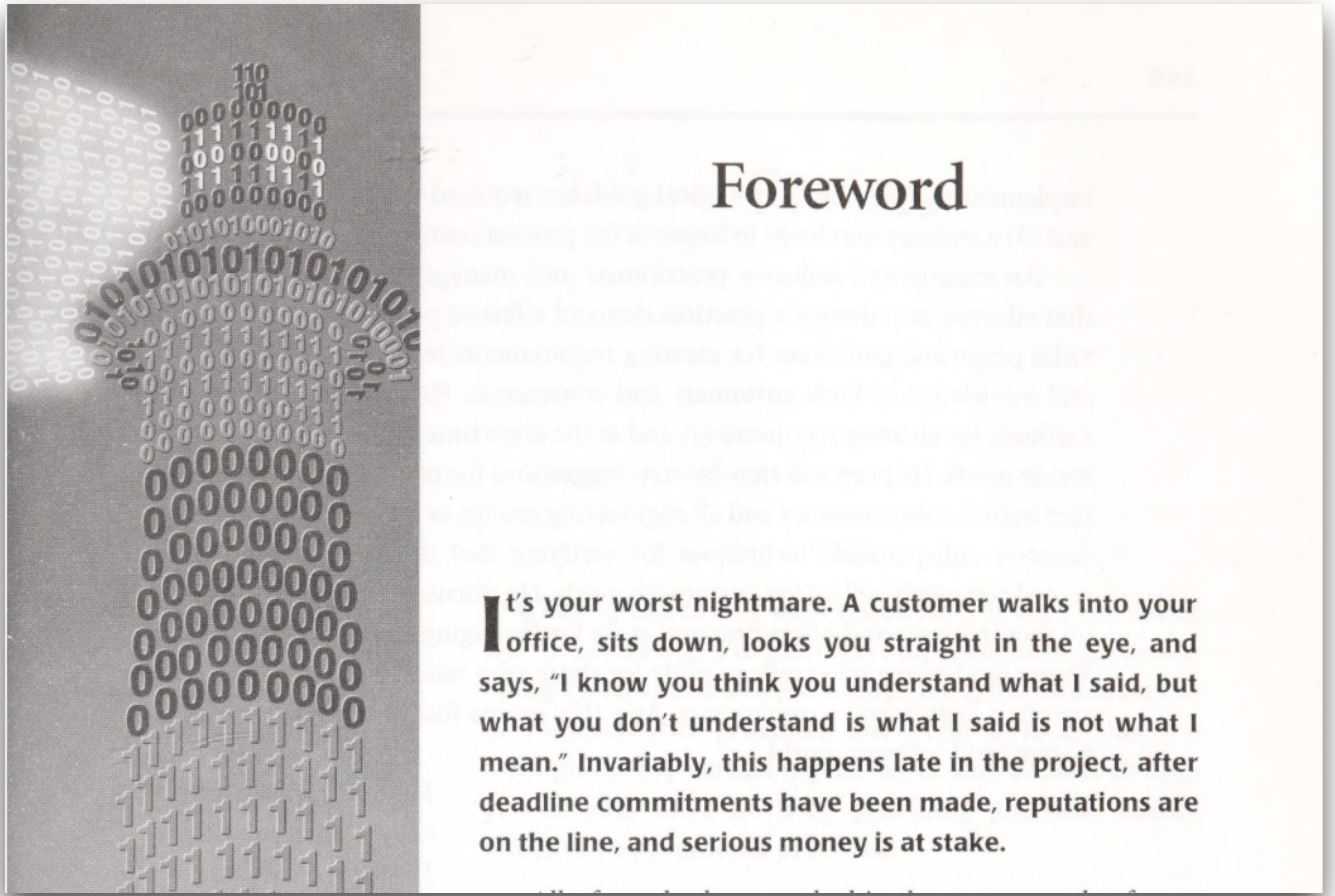
The Tree Swing Story

(from 1970s, apparently)


1
How the customer explained it
2
How the project leader understood it
3
How the analyst designed it
4
How the programmer wrote it
5
What the beta testers received
6
How the business consultant described it
7
How the project was documented
8
What operations installed
9
How the customer was billed
10
How it was supported
11
iSwing
What marketing advertised
12
What the customer really needed
www.projectcartoon.com

“Engineer” requirements?

- Mistakes are easier and cheaper to fix earlier in the project.
- Requirement Engineering is the earliest stage in the lifecycle of a software system.



Foreword

It’s your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, “I know you think you understand what I said, but what you don’t understand is what I said is not what I mean.” Invariably, this happens late in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

Airbus 320 Accident, Warsaw 1993

Lufthansa 2904 from Frankfurt to Warsaw

Airbus 320 Accident, Warsaw 1993

Lufthansa 2904 from Frankfurt to Warsaw

- In bad weather ands strong winds, the software controlled braking system did not deploy in time, resulting in insufficient runway to brake on and, eventually, a collision into a building. Two people killed, 54 injured.
- Multiple factors:
  - Pilots were given incorrect (outdated) information about the wind.
  - Crews failed to notice that the on-board information about the wind is inconsistent with those from the control tower.
  - Breaking software specifications did not anticipate this specific condition.

Airbus 320 Accident, Warsaw 1993

Lufthansa 2904 from Frankfurt to Warsaw

- The software system behaved exactly as specified: no bug in implementation —> the problem lies in specification, i.e., the requirements.
- Airplane braking: spoilers (flats on the wing) + reverse thrust (running engines backward)
  - It is crucial that neither is activated mid-air!
  - How should we determine whether an aircraft is mid-air?
  - Spec used the weight sensor in the landing gear: the aircraft has landed if there are weights on both wheels.
  - Spec also used rotation sensor on wheels: if speed of wheel rotation is higher than 72 knots, the airplane is landed.

Airbus 320 Accident, Warsaw 1993

Lufthansa 2904 from Frankfurt to Warsaw

```
if weight_on_both_wheels() or (is_left_wheel_turning() or is_right_wheel_turning()):
    can_brake = True
```

Airbus 320 Accident, Warsaw 1993

Lufthansa 2904 from Frankfurt to Warsaw

- In the incident:
  - Due to strong winds, only one wheel touched the runway initially.
  - Due to the wet condition, the wheel skidded (aquaplaned) without rotating.
- One-wheel landing is actually the standard landing technique when there is cross-wind; however, at the moment of landing, the wind changed to tailwind. The unnecessary one-wheel landing became even more dangerous due to the increased speed.
- Without crosswind to press down the other wheel, the plan skidded without braking for 9 seconds.

We need to understand what we’re building.

- Requirements: “…a condition or a capability that must be met or possessed by a system (or system component) to satisfy a contract, standard, specification, or other formally imposed documents.” (IEEE 610.12-1990)
- Can be about products or process; can be functional or non-functional.
- Clearly capturing requirements is the absolute necessary condition for a successful project.

Descriptive vs. Prescriptive Requirements

- Descriptive statements talk about the system properties that always hold (often due to physical constraints).
    - A train door is either open, or closed.
    - A book cannot be borrowed by two different library users.
    - A person cannot attend two different meetings at the same time.
- Prescriptive statements talks about desirable properties that may or may not hold depending on how the system behaves (our goal is to implement a system that makes these hold)
    - While the train is moving, all doors must remain closed.
    - One library user can borrow only up to three books at any time.
    - A meeting should always be scheduled in a time slot that is free for all participants.

User vs. System Requirements

- User requirements are statements of what service the system is expected to provide to the users, as well as the constraints under which the system operates.
- System requirements are detailed descriptions of the system functionality, and typically should define what is exactly to be implemented.
- Both are needed for different (business) reasons; mixing them up is not a good idea.

User vs. System Requirements

- “I need a page on my website - it should have text boxes for name, email address, and the message the user wants to send to my company. There should be a submit button - when the user clicks the button, the message should be delivered to my email address.”
- “I need a page on my website, where the users can tell me about any questions they may have. When they ask questions, I need their contact information as well so that I can follow up the inquiries. Please make it as attractive as possible.”

Stages in Requirement Engineering

- Elicitation: understand and articulate the user's needs
- Analysis: identify any conflicts
- Specification: document the user/system requirements
- Validation: make sure that the specification captures the actual needs
- Management: version control, traceability

Elicitation

- Understand the target system
- Identify stakeholders (i.e., domain experts)
- Acquire the domain knowledge and the needs of the stakeholders
- Define the project scope
- Challenges
  - Scope: the boundary of what to include may not be explicit or clear
  - Understanding: stakeholders may not know what they really need
  - Volatility: user needs may change over time

Elicitation Methods: Interviews

- Formal or informal interviews with stakeholders, where you talk to people about what they do
    - Closed (predefined questions), or open (no predefined agenda)
    - Good for getting an overall understanding of the target system
- Challenges
    - Jargons: domain experts will use their own language
    - Familiarity: some domain specific concepts are so familiar to stakeholders that they will find it actually difficult to articulate them

Elicitation Methods: Ethnography

- Ethnographic Observation: you become an observer, and simply watch the stakeholders perform their tasks.
- Requirements are derived from the actual way people work, rather than through the way domain concepts dictate they should work.
- Can reflect group dynamics and cooperation in workplaces.
- However, it may not always foster innovation, as it aims to capture what is currently going on.
    - Example: Nokia vs. Apple :)

Elicitation: Anti-patterns

- Talking about implementation details instead of requirements.
- Projecting developer's own models and ideas.
- Feature creep, i.e., gradual accumulation of features over time.
  - Remember, users may not know what they want, which also means that they may think they want something they actually do not need.
  - Developers like to write features (because it means more code, why not)
  - But this will add unnecessary complexity to the system, eventually slowing down the development (delays, bugs, etc)

Analysis

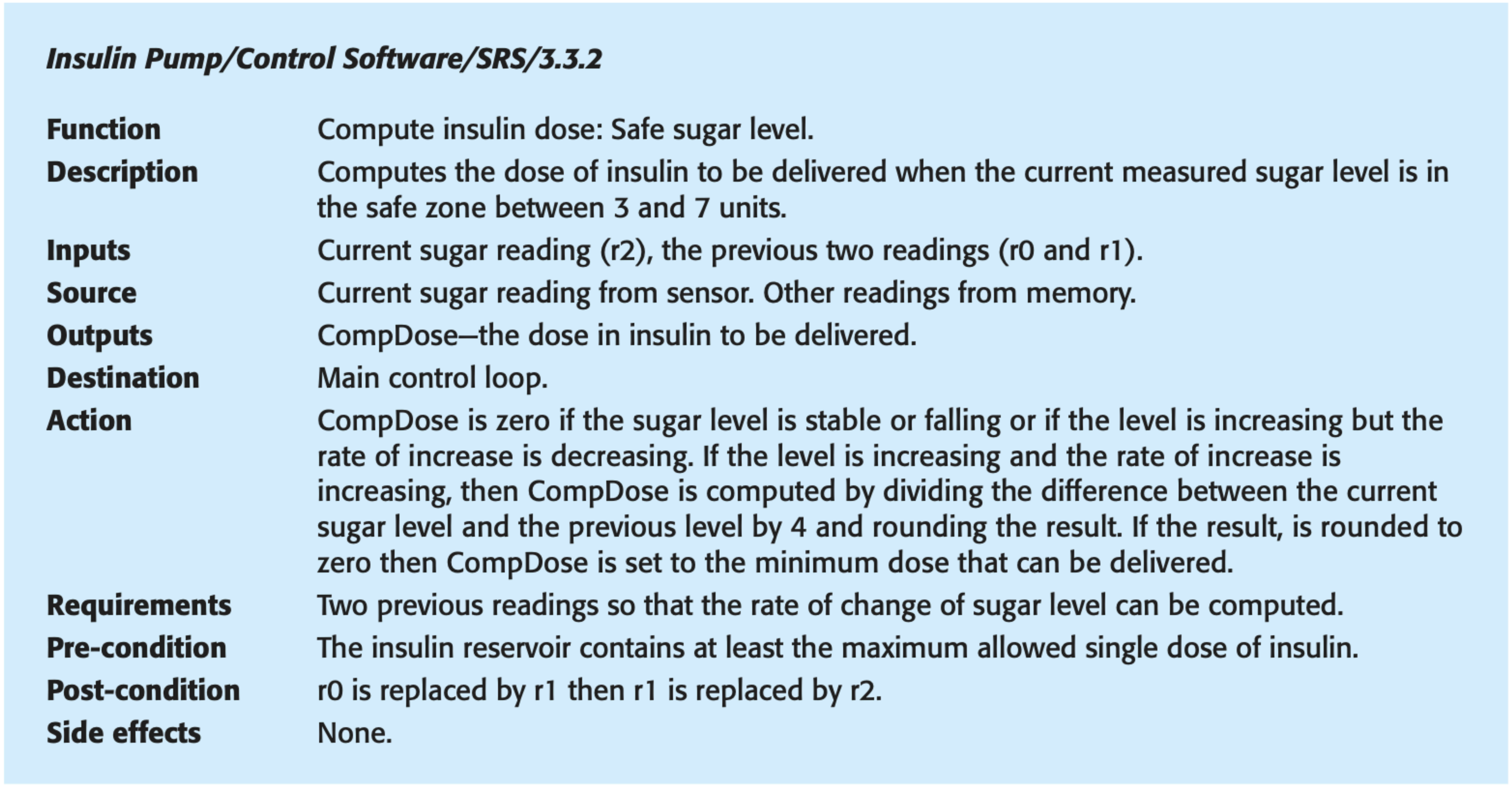
- Classify the requirements according to their scopes and priorities
- Identify conflicts and feasibility concerns, and negotiate
  - Different stakeholder needs may be in conflict with each other, requiring resolution based on priority
  - Some requirements may be infeasible due to many different reasons: replace or rescope.

Specification

- The process of writing down the user/system requirements. But in what language/format/notation?
    - Natural language: expressive, intuitive, and natural. But also vague, ambiguous, and open to interpretations. Try to have a standard format and use consistent language.
    - Structured Specifications: impose a systematically designed format for each requirement item (formal use cases).
    - Modeling Languages: use a formal modeling language (such as UML) to capture requirements (use case diagrams).

An example entry in structured req.

(taken from “Software Engineering” 9th ed., by Sommerville)

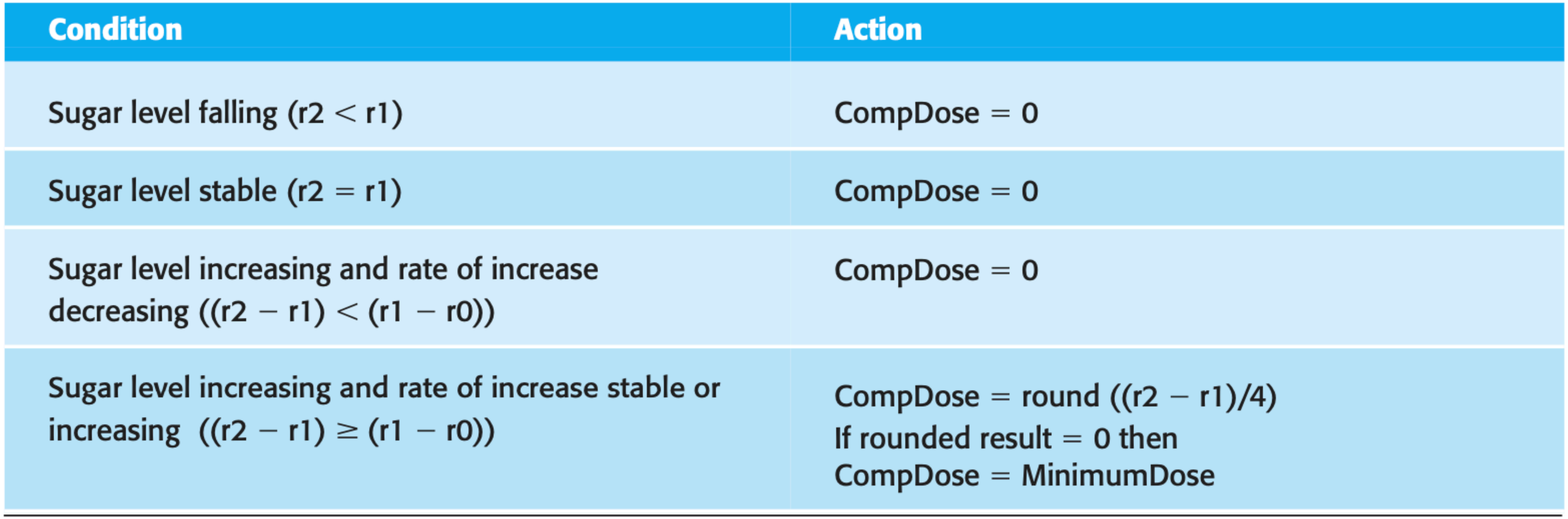


Insulin Pump/Control Software/SRS/3.3.2

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.

An example entry in structured req.

(taken from “Software Engineering” 9th ed., by Sommerville)



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 ($(r2 - r1) \geq (r1 - r0)$)	$CompDose = round((r2 - r1)/4)$ If rounded result $= 0$ then $CompDose = MinimumDose$

An example Use Case diagram

(taken from “Software Engineering” 9th ed., by Sommerville)

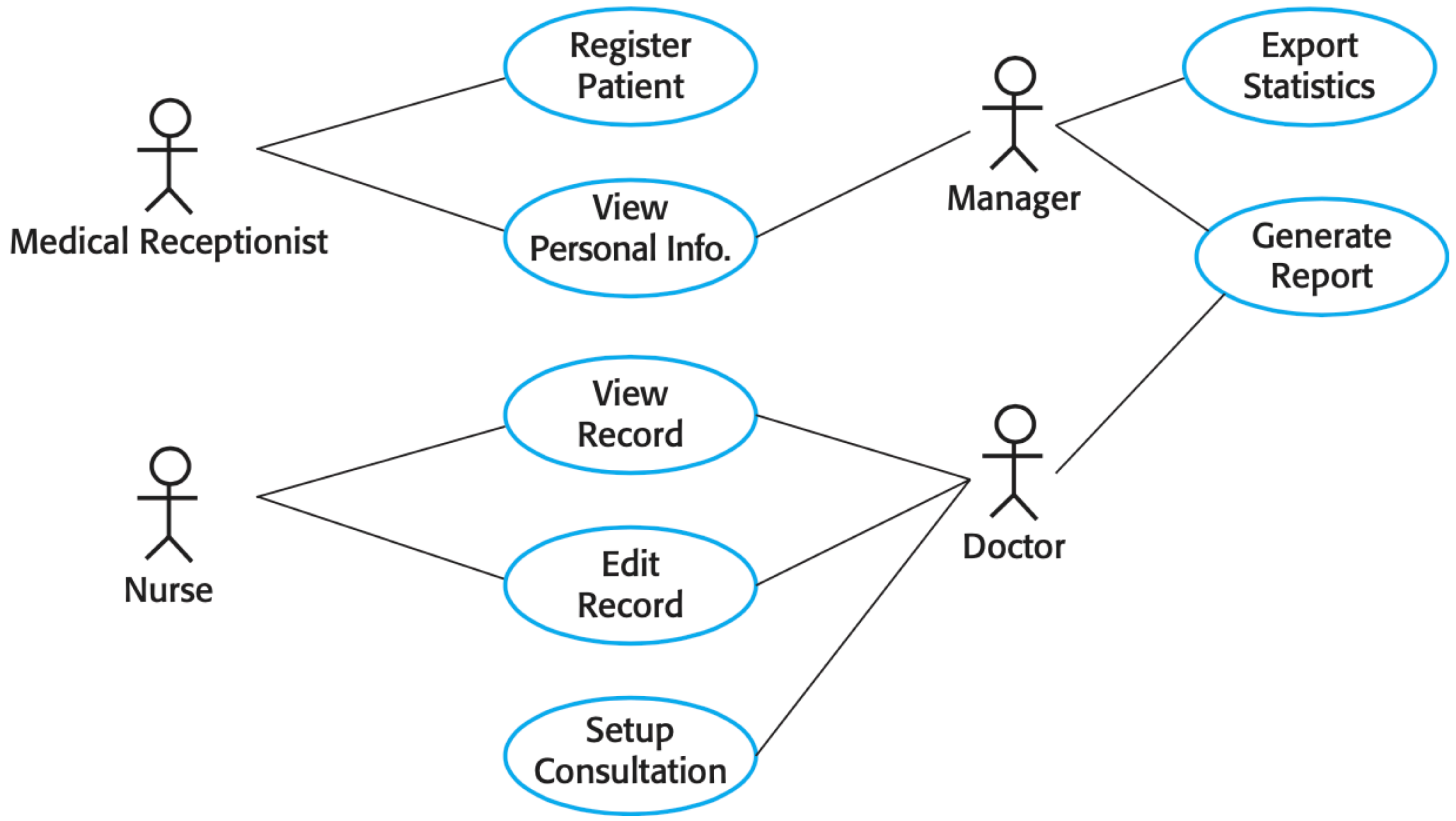


Figure 4.15 Use cases for the MHC-PMS

(Will talk more about this later)

Validation

- The process of checking that requirements actually define the system that the users really want.
    - Validity: is the specified functionality really the one users think they need?
    - Consistency: is there any conflicts between specified requirements?
    - Completeness: does it capture all functionalities and constraints from users?
    - Realism: is the specified requirement feasible?
    - Verifiability: is the specified requirement verifiable? That is, can we check its completion?

Validation: Techniques

- Review: systematic checks performed by a team of domain experts
- Prototyping: actually try to build an executable model and demonstrate it to the end users, who will experiment with the model
- Test case generation: try to generate test cases for the specified requirements - this will often reveal underlying problems in the requirements (this is a core element of extreme programming)

Management

- Requirements (especially for large software systems) always change: sometimes across its lifetime, sometimes within a single development cycle
- Many real-world problems that these systems are built for <u>cannot be completely defined:</u> at any given moment, the system is handling some defined parts of the entire problem. With changing understanding of the problem, comes changed definition, resulting in changed requirements.
  - Business/technical environment change (tech stack, other services, regulations, etc)
  - Financial and other organizational constraints change
  - Compromises between different sub user-group change

Management: Traceability and Version Control

- It is critical to have well documented iterations of “analyze - specify - change”: ad-hoc management can only result in fragmented and inconsistent system.
- Traceability: you can link all stages in any system change, from requirement change, through specification change, via implementation, to test results.
- Version control: crucial that you can look at the entire history of requirement changes.

Use Case

- A scenario that describes one particular interaction between users and the system performed to achieve a specific goal (i.e., an example behavior)
- An effective way for developers and users to capture and refine requirements.
  - Intuitive for the users
  - Corresponds to test cases
  - High-level description that is independent from implementation details
  - Decomposition of system functionalities

An example use case

- **Sale at POS**: A customer arrives at a checkout with items to purchase. The cashier uses the POS to record each purchased item. The system keeps a running total, as well as per-item details. The customer enters payment information, which the system validates and records. The system also updates the shop inventory. The customer receives a receipt from the system, and leaves with the purchased items.

Actors, Goals, Scenarios

- Actor: someone (or another system) that is interacting with the system
    - Primary Actor: the person who initiates the scenario with an action
- Goal: the desired outcome of the primary actor
- Success Scenario: the desired functionality achieved
- Extension Scenario: a possible branch in use case that may be triggered by unusual conditions (e.g., error or edge cases)

Use Case Extensions

- Consider how individual actions can fail
- Provide reasonable response for each of the extension: either jump to another scenario, or end the interaction
- Avoid unreasonable assumptions (e.g., user never forgets the password)
- Do not go beyond the scope of the original scenario (e.g., natural disaster or power cut)

Steps of creating use cases

- Identify actors and goals
  - Actor: which users and subsystems interact with our system?
  - Goal: what is required by each actor?
- Describe the main success scenario
- List the potential failure extensions

Steps of creating use cases

- Identify actors and goals
- Describe the main success scenario
    - Imagine the “happy ending” story: everything else is an exception to the success
    - Each actor contributes to the main happy ending scenario by providing information
- List the potential failure extensions

Steps of creating use cases

- Identify actors and goals
- Describe the main success scenario
- List the potential failure extensions
    - Imagine which steps can fail
    - Provide a recovery path if possible (i.e., go back to the main happy ending scenario); otherwise the failure is non-recoverable (an exceptional ending)

Software Requirement Specification (SRS)

- "a description of the software system to be developed" (Wikipedia)
- Just a natural language document - you cannot and will not compile it, there is no fixed formal structure, etc.
- **However, getting this right is of crucial importance.** You will get back to this document to decide whether your system is correct or not.
- Use concise and consistent language; use diagrams (UML?) for clearer understanding.
- In reality, people with many different roles will read your SRS: clients, managers, system engineers, test engineers, maintenance engineers, etc…
- As an example, IEEE recommended format is linked in the course webpage.

SRS for CS350 Projects

- There is no set length, but try to be precise about your project: your SRS will be the main source of information for the developers.
    - If something is unspecified, the only likely consequence is…
- A document can be arbitrarily long without containing much information, but being shorter does not guarantee being concise and effective!

Conclusion

- Incorrectly captured requirements can result in serious failures; taking care to capture requirements early in the lifecycle will save a lot of future troubles.
- Requirement engineering should focus on goals and interactions; NOT on internal system activities, implementation choices, and user interface.
- Try to be as clear and concise as possible: we use modeling languages to achieve this.