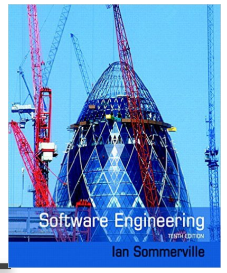
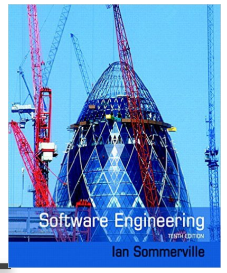


Chapter 2 – Software Processes

Lecture 1

The software process



- A structured set of activities used to develop a software system
- Many different software processes … but all involve:

Specification (a.k.a Requirements)

- defining what the system should do

Design

- defining the organization of the system as a whole
- defining the organization of a component part of the system

Implementation

- implementing the system

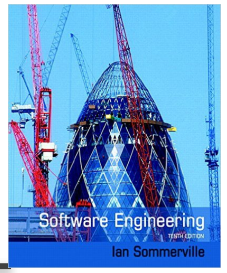
Testing and Validation

- checking that it performs as planned (*testing* aka *verification*)
- checking that it does what the customer wants (*validation*)

Evolution

- changing the system in response to changing customer needs

Software process descriptions



- When we discuss processes, we usually talk about
  - the activities in these processes such as:
    - specifying a data model
    - designing a user interface, etc.
  - the ordering of these activities
- Process descriptions may also include:

Products

- the outcomes of a process activity

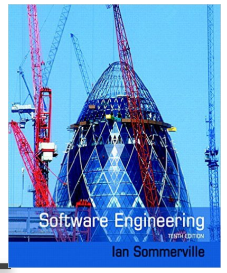
Roles

- the responsibilities of the people involved in the process

Pre- and post-conditions

- statements that are true before and after a process activity has been enacted or a product produced

Plan-driven and agile processes



Plan-driven processes

- processes where all of the process activities are planned in advance and progress is measured against this plan.

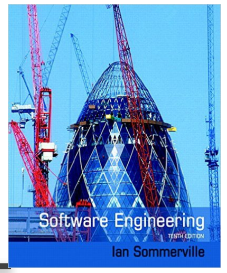
Agile processes

- planning is incremental and it is easier to change the process to reflect changing customer requirements.

◇ In practice

- most practical processes include elements of both plan-driven and agile approaches
- There are no right or wrong software processes

Software process models



The waterfall model

- Plan-driven model
- Separate and distinct phases of specification and development

Incremental development

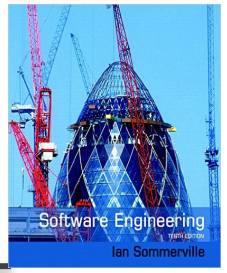
- Specification, development and validation are interleaved
- May be plan-driven or agile

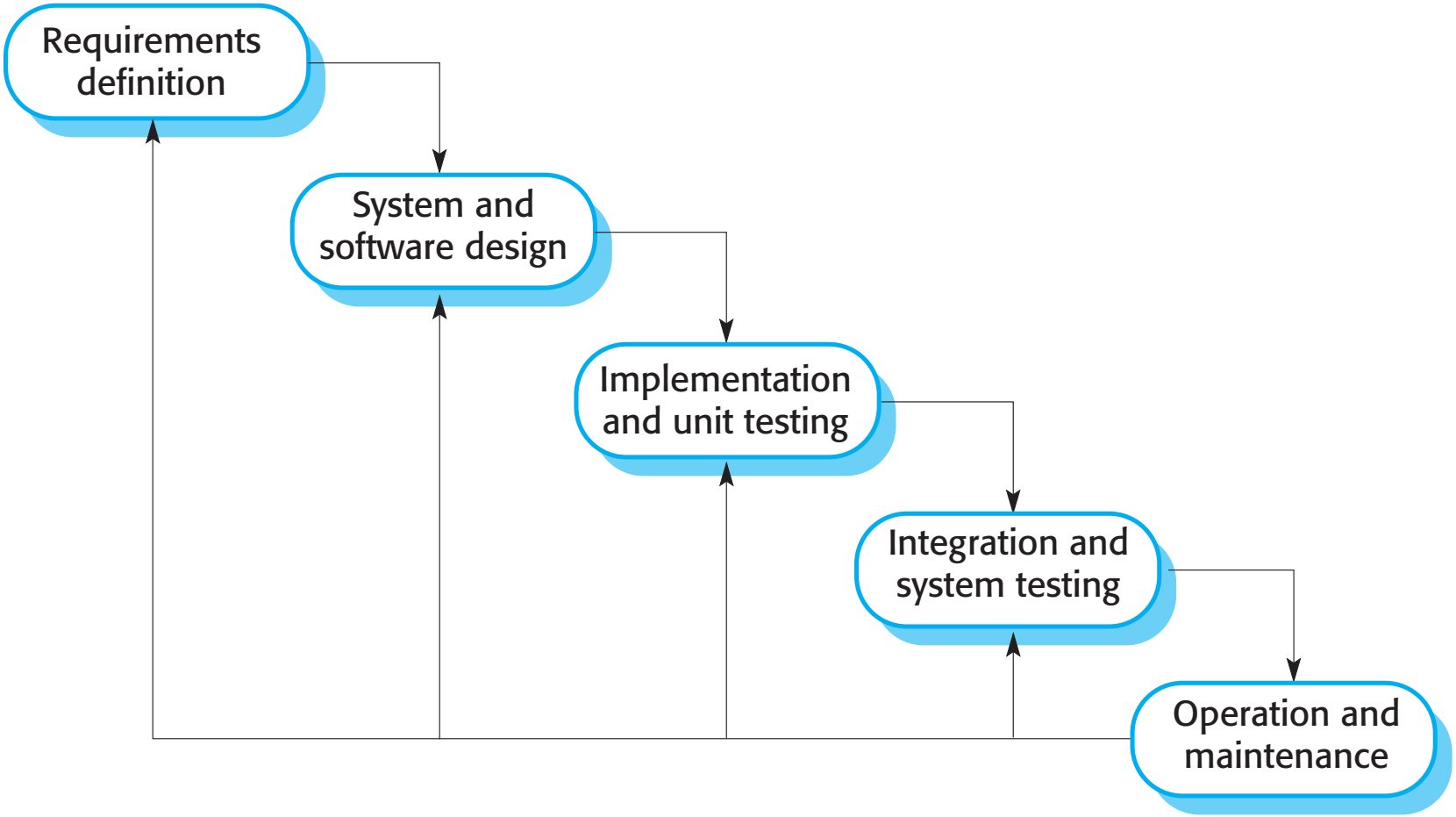
Reuse-oriented software engineering

- The system is assembled from existing components
- May be plan-driven or agile

◇ In practice, most large systems are developed using a process that incorporates elements from all of these models

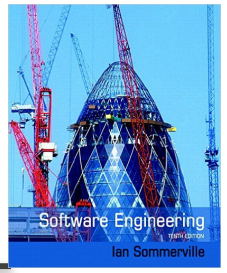
The waterfall model





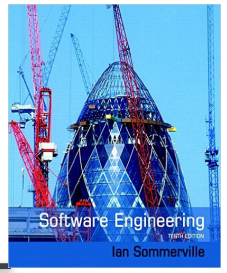
- a phase should be complete before moving onto the next

Advantages of the Waterfall model



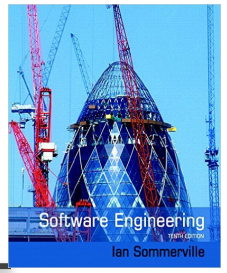
- Encourages “big picture” perspective
  - Minimizes chances that important features will be overlooked
- Easier to make changes in planning stage
  - Only changing a single diagram rather than 1000’s of lines of code
- Prevents “writing yourself into a corner”
  - If you don’t plan ahead, may structure code that prevents new/next feature from being added
  - Could cause massive rewrites or complete overhaul of the system
- Helps keep remote teams “on the same page”
  - People don’t think alike … neither do groups

Waterfall model problems



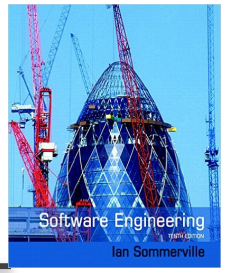
- Difficulty of accommodating change after the process is underway
  - In principle, a phase has to be complete before moving onto the next phase.
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements

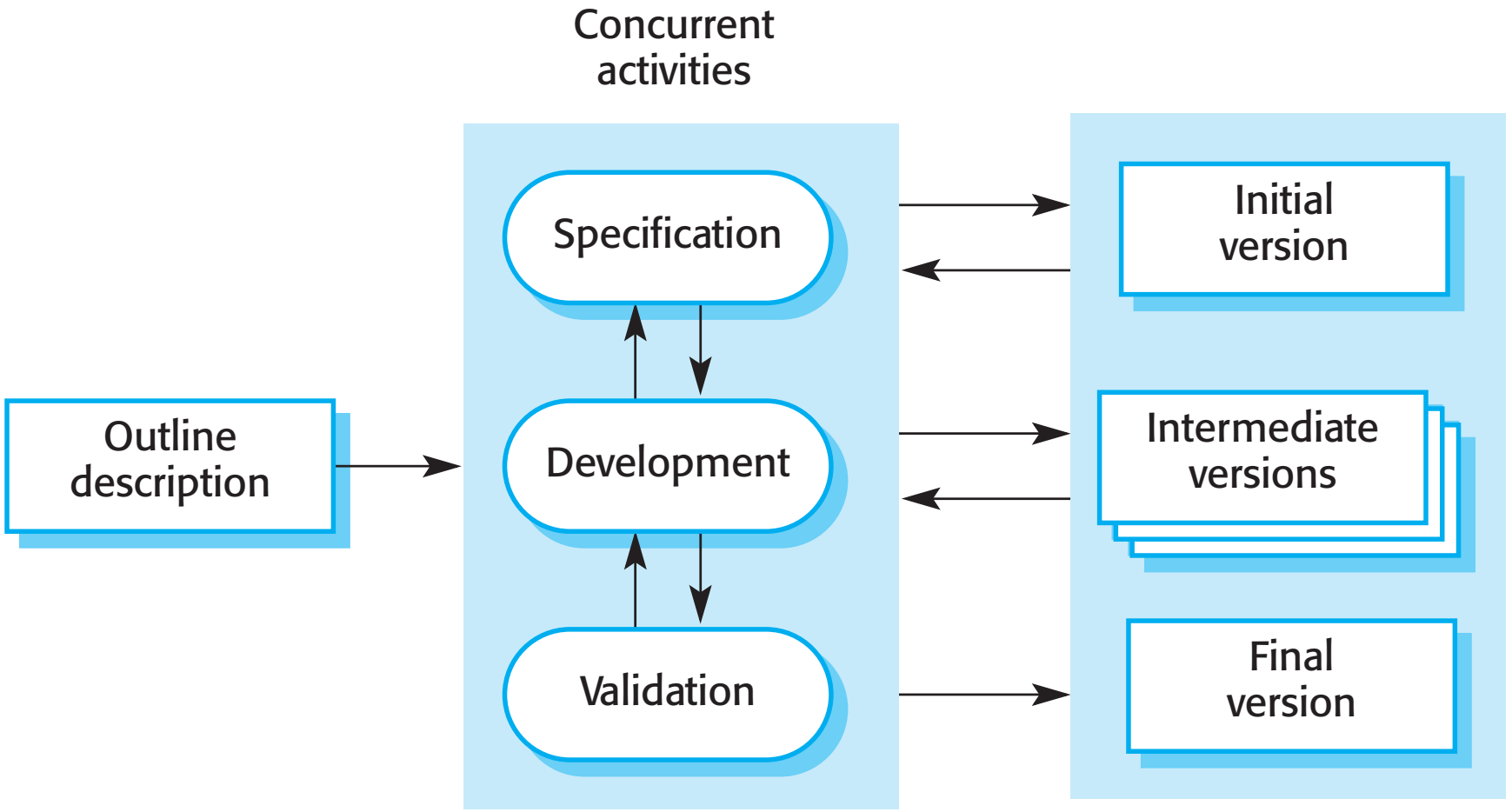
Where would Waterfall be used?



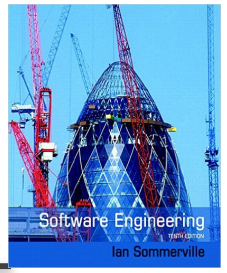
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Incremental development
(includes Agile processes)



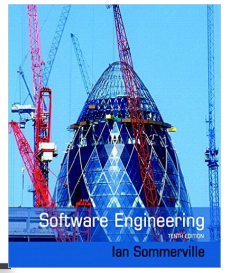


Incremental development benefits



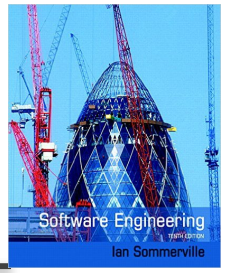
- Reduces the cost of accommodating changing customer requirements
  - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- Easier to get customer feedback on the development work that has been done
  - Customers can comment on demonstrations of the software and see how much has been implemented
- More rapid delivery and deployment of useful software to the customer is possible
  - Customers get the software earlier than is possible under waterfall
- Allows for the “debugging” of specifications & designs
  - The best “test” of a specification/design is the working softwares

Incremental development problems

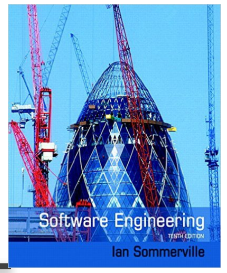


- The process is not visible.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

Process activities



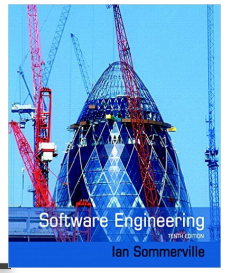
- Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.



Chapter 4 – Requirements Engineering

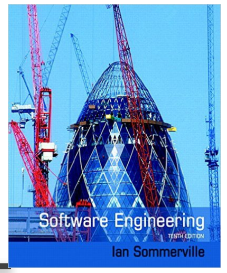
Lecture 1

Requirements



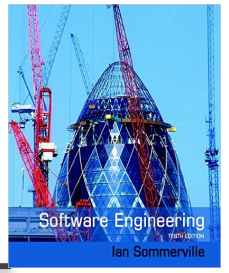
- Establish
  - the services that the customer requires from a system
  - the constraints under which it operates and is developed
- Ranges from
  - a high-level abstract statement of a service or of a system constraint
  - to a detailed mathematical functional specification.
- This range happens because they serve a dual function
  - the basis for a bid for a contract
    - therefore must be open to interpretation
  - the basis for the contract itself
    - therefore must be defined in detail
  - Both these statements may be called requirements.

The software requirements document



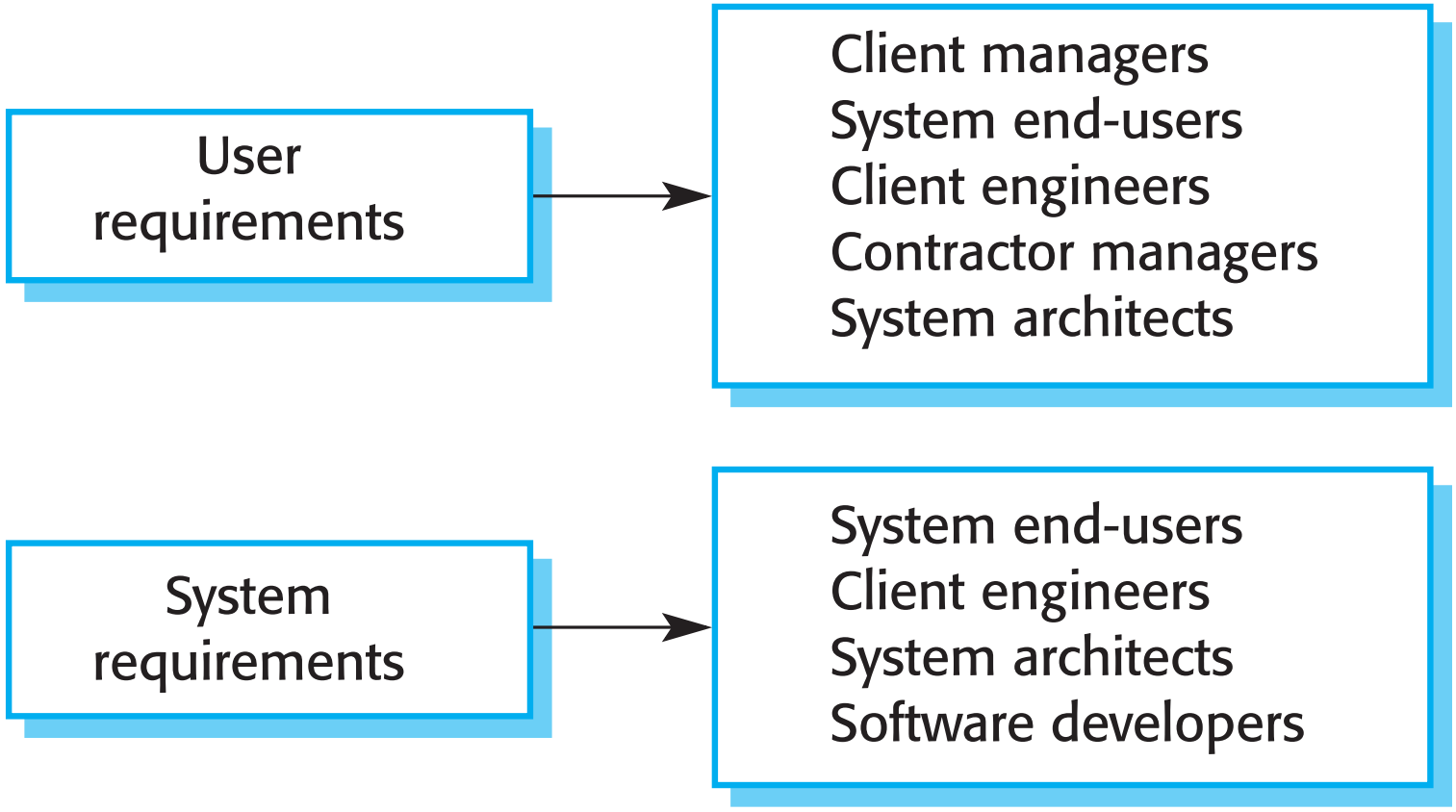
- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Types of requirement

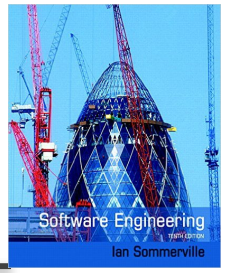


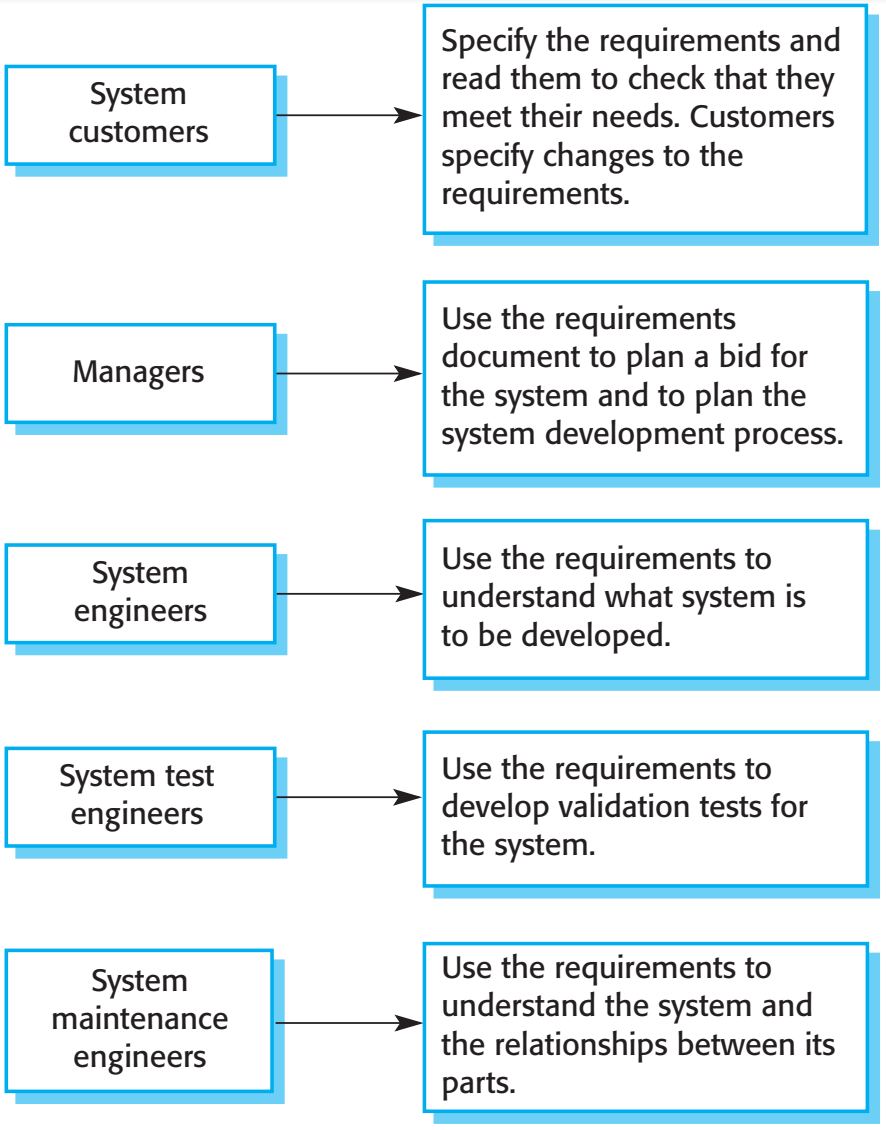
- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints.
  - Written for customers.
- System requirements
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
  - Defines what should be implemented so may be part of a contract between client and contractor.

Readers of different types of requirements specification

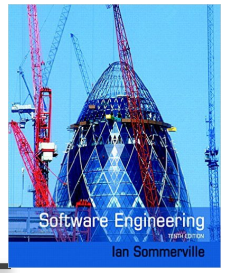


Users of a requirements document



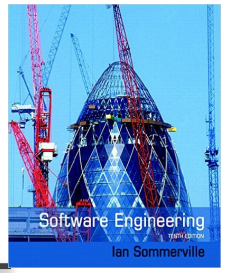


Requirements document variability



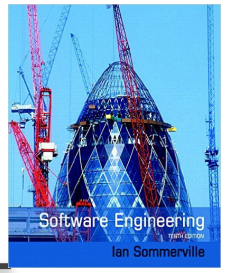
- Information in requirements document depends on type of system and the approach to development used.
- Systems developed incrementally will, typically, have less detail in the requirements document.
- Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

Requirements specification



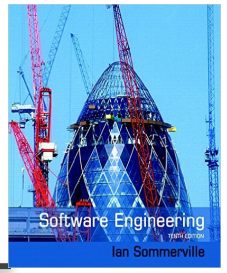
- The process of writing down the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification



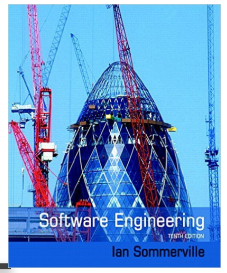
Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and design



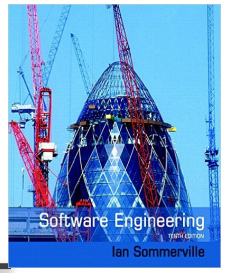
- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

Natural language specification



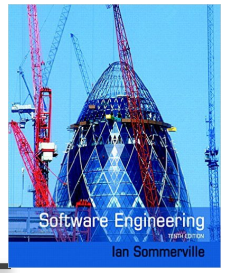
- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements



- 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.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

User and system requirements



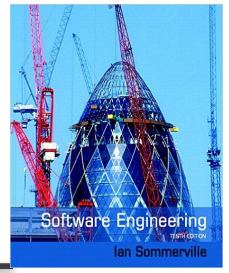
User requirement definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20 mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

A ‘prescribing medication’ story



Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

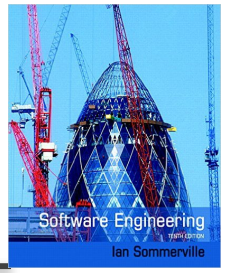
If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication



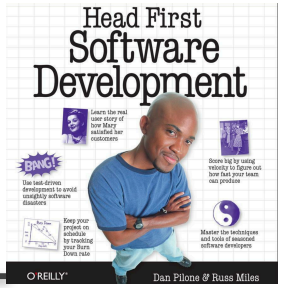
Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low.
If within the range, enable the 'Confirm' button.

How to get a requirement: User Stories



- Developed for OO and Agile
- Main idea:
  - one requirement per 'card' from the users perspective

User stories SHOULD...

- ☐ ... describe **one thing** that the software needs to do for the customer.
- ☐ ... be written using language that **the customer understands**.
- ☐ ... be **written by the customer**.
- ☐ ... be **short**. Aim for no more than three sentences.

You should be able to check each box for each of your user stories.

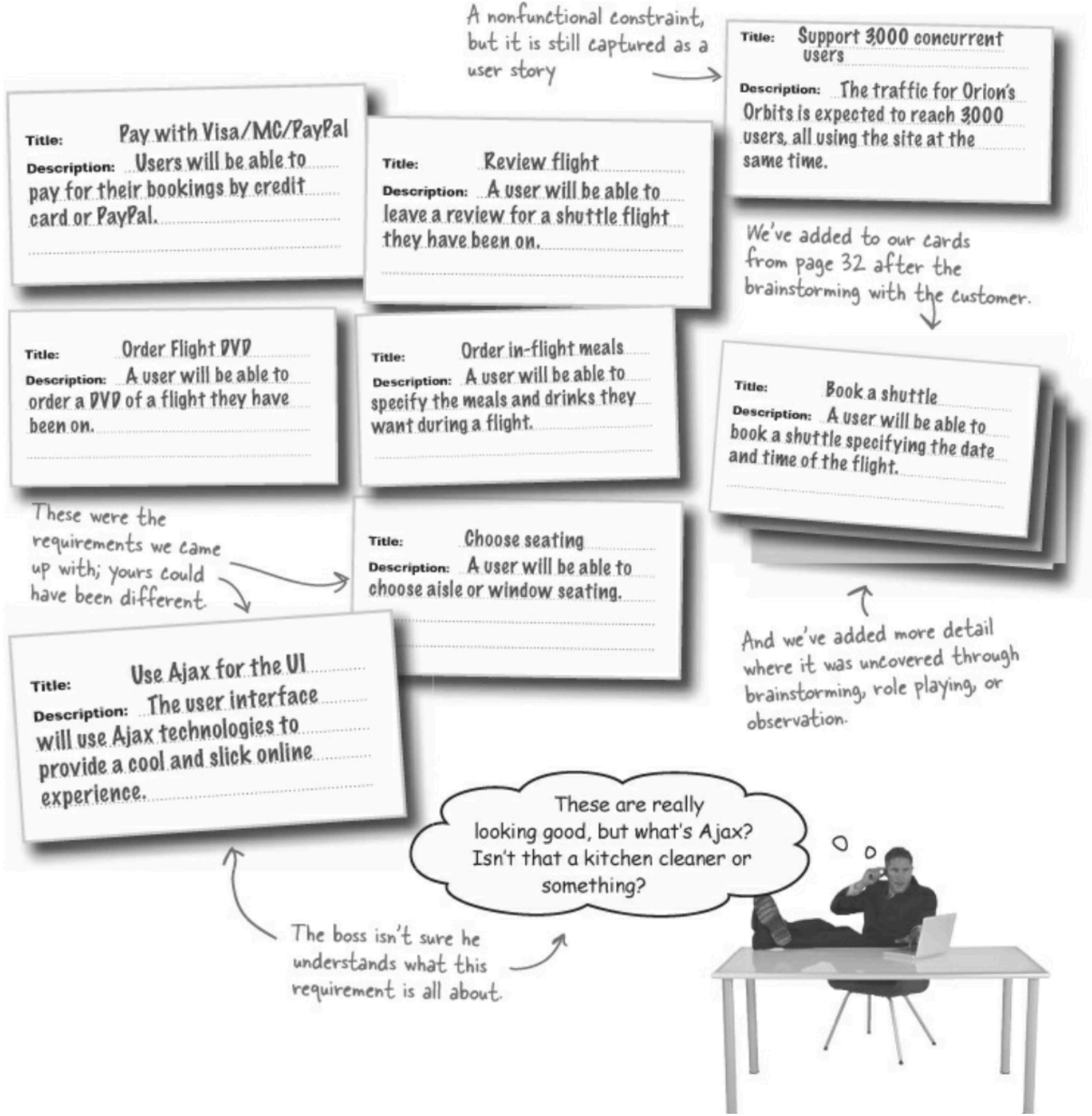
Think "by the customer, for the customer"

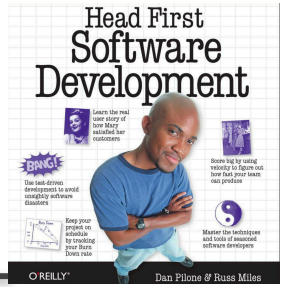
This means the customer drives each one, no matter who scribbles on a notecard.

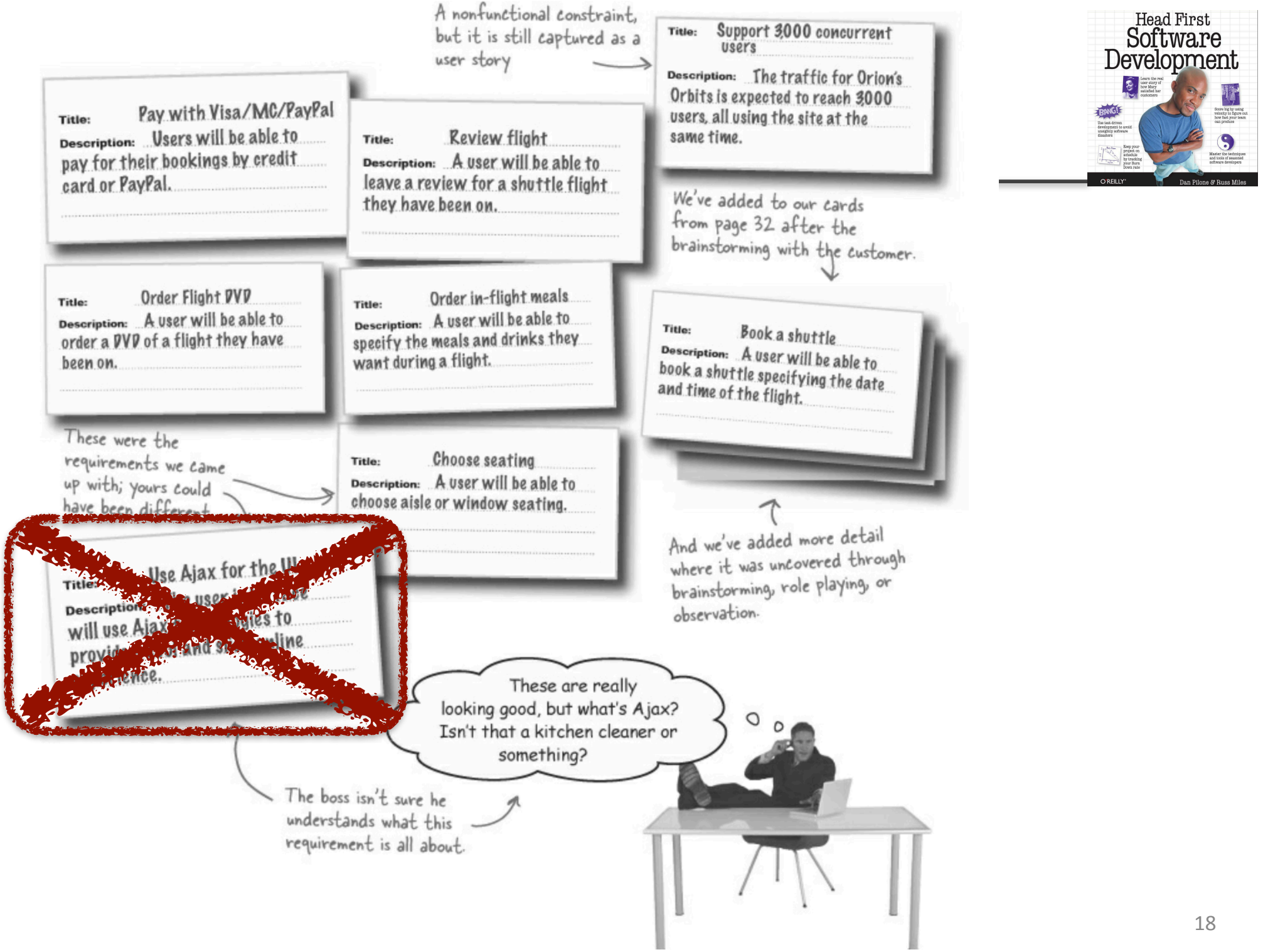
User stories SHOULD NOT...

- ☐ ... be a long essay.
- ☐ ... use technical terms that are unfamiliar to the customer.
- ☐ ... mention specific technologies.

If a user story is long, you should try and break it up into multiple smaller user stories (see page 54 for tips).


A nonfunctional constraint, but it is still captured as a user story
Title: Support 3000 concurrent users
Description: The traffic for Orion's Orbits is expected to reach 3000 users, all using the site at the same time.
Title: Pay with Visa/MC/PayPal
Description: Users will be able to pay for their bookings by credit card or PayPal.
Title: Review flight
Description: A user will be able to leave a review for a shuttle flight they have been on.
We've added to our cards from page 32 after the brainstorming with the customer.
Title: Order Flight DVD
Description: A user will be able to order a DVD of a flight they have been on.
Title: Order in-flight meals
Description: A user will be able to specify the meals and drinks they want during a flight.
Title: Book a shuttle
Description: A user will be able to book a shuttle specifying the date and time of the flight.
These were the requirements we came up with; yours could have been different.
Title: Choose seating
Description: A user will be able to choose aisle or window seating.
And we've added more detail where it was uncovered through brainstorming, role playing, or observation.
Title: Use Ajax for the UI
Description: The user interface will use Ajax technologies to provide a cool and slick online experience.
These are really looking good, but what's Ajax? Isn't that a kitchen cleaner or something?
The boss isn't sure he understands what this requirement is all about.


Head First
Software Development
BANG!
O'REILLY
Dan Pilone & Russ Miles


A nonfunctional constraint, but it is still captured as a user story
Title: Support 3000 concurrent users
Description: The traffic for Orion's Orbits is expected to reach 3000 users, all using the site at the same time.
Title: Pay with Visa/MC/PayPal
Description: Users will be able to pay for their bookings by credit card or PayPal.
Title: Review flight
Description: A user will be able to leave a review for a shuttle flight they have been on.
We've added to our cards from page 32 after the brainstorming with the customer.
Title: Order Flight DVD
Description: A user will be able to order a DVD of a flight they have been on.
Title: Order in-flight meals
Description: A user will be able to specify the meals and drinks they want during a flight.
Title: Book a shuttle
Description: A user will be able to book a shuttle specifying the date and time of the flight.
These were the requirements we came up with; yours could have been different.
Title: Choose seating
Description: A user will be able to choose aisle or window seating.
And we've added more detail where it was uncovered through brainstorming, role playing, or observation.
Title: Use Ajax for the UI
Description: The user interface will use Ajax technologies to provide a fluid and streamline experience.
These are really looking good, but what's Ajax? Isn't that a kitchen cleaner or something?
The boss isn't sure he understands what this requirement is all about.

Head First
Software Development
BANG!
O'REILLY
Dan Pilone & Russ Miles

User Story Outline

Examples from:

Justinmind

Title:	Priority:	Estimate:
User Story: As a [description of user], I want [functionality] so that [benefit].		
Acceptance Criteria: Given [how things begin] When [action taken] Then [outcome of taking action]		

User Story Outline

Examples from:

Justinmind

Title:	Priority:	Estimate:

As a *<type of user>*

I want to *<perform some task>*

so that I can *<achieve some goal>*

Acceptance criteria

Given *<some context>*

When *<some action is carried out>*

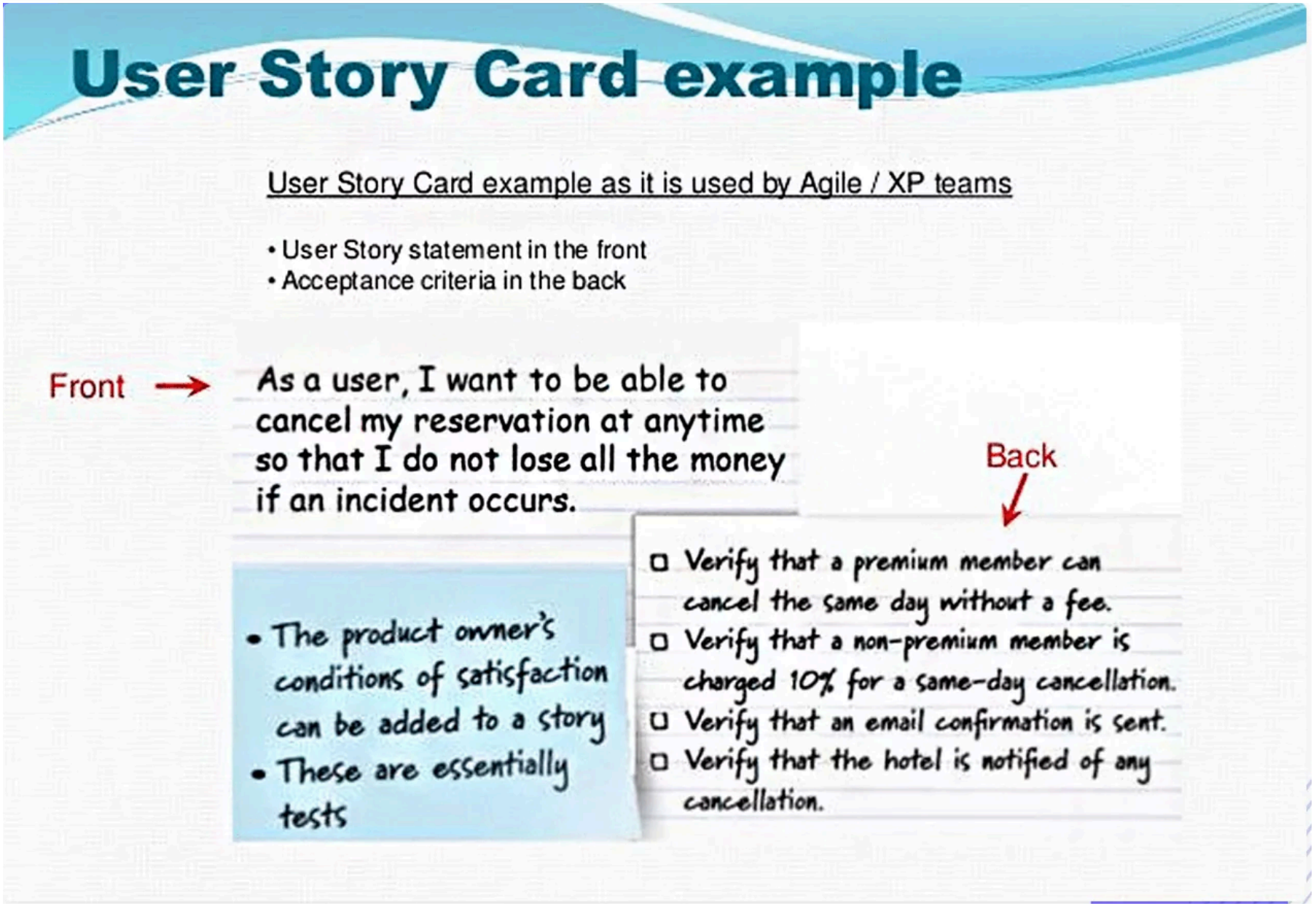
Then *<a set of observable outcomes should occur>*

Template from MS Word

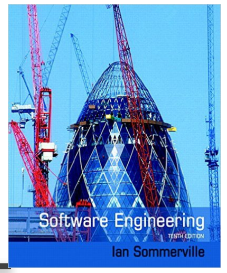
User Story Outline

Examples from:

Justinmind

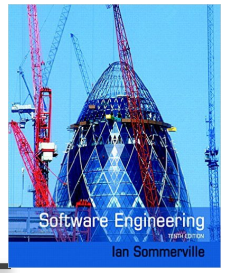


Functional and non-functional requirements



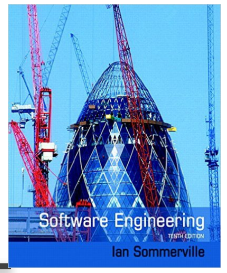
- Functional requirements
  - Statements of services the system should provide,
  - how the system should react to particular inputs
  - how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or 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
  - Constraints on the system from the domain of operation

Functional requirements



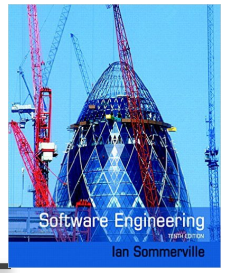
- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

Functional requirements for the MHC-PMS



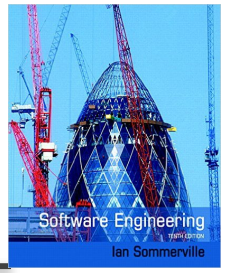
- A user shall be able to search the appointments lists for all clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements imprecision



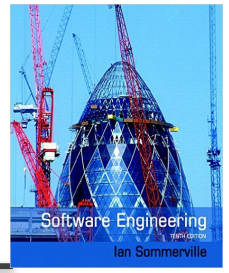
- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency



- In principle, requirements should be both:
- 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 impossible to produce a complete and consistent requirements document.

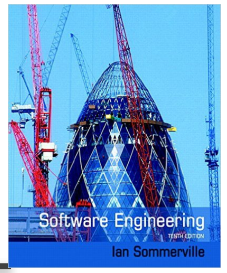
Example requirements for the insulin pump software system



3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes.
(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)

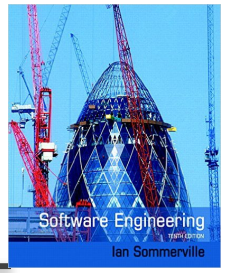
3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.
(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured specifications



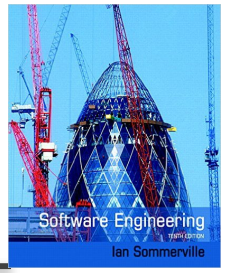
- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications



- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

A structured specification of a requirement for an insulin pump



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.

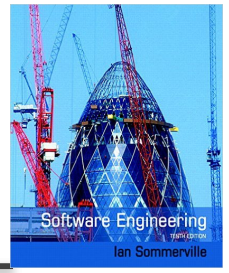
Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A structured specification of a requirement for an insulin pump



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.

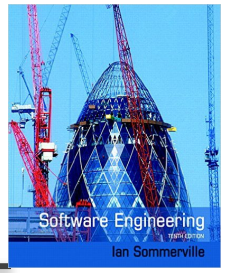
Pre-condition

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.

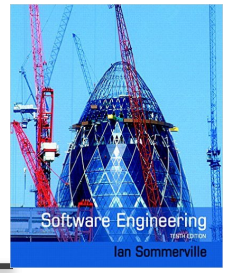
Side effects None.

Tabular specification



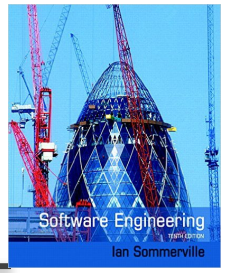
- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Tabular specification of computation for an insulin pump



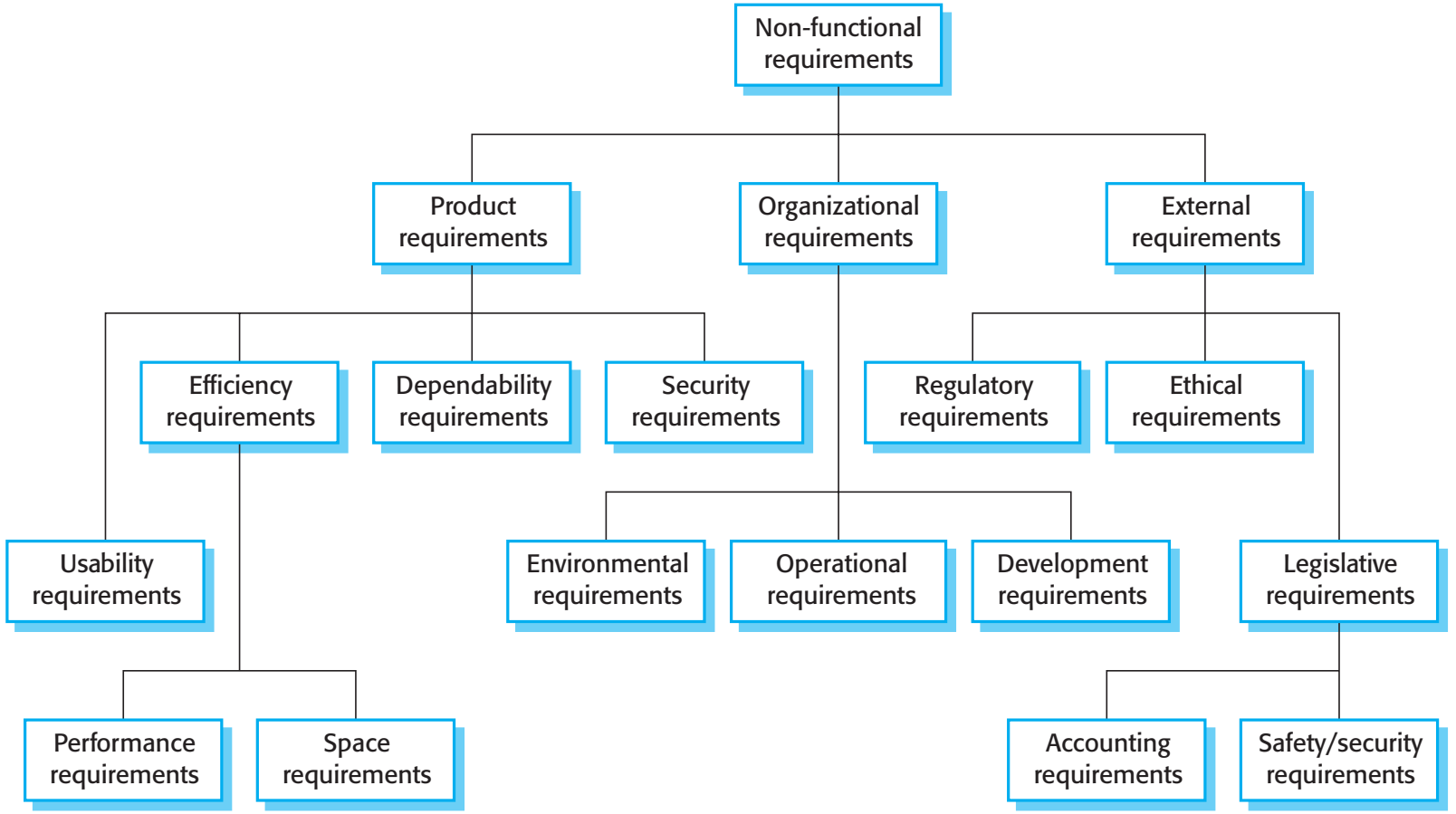
Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2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

Non-functional requirements

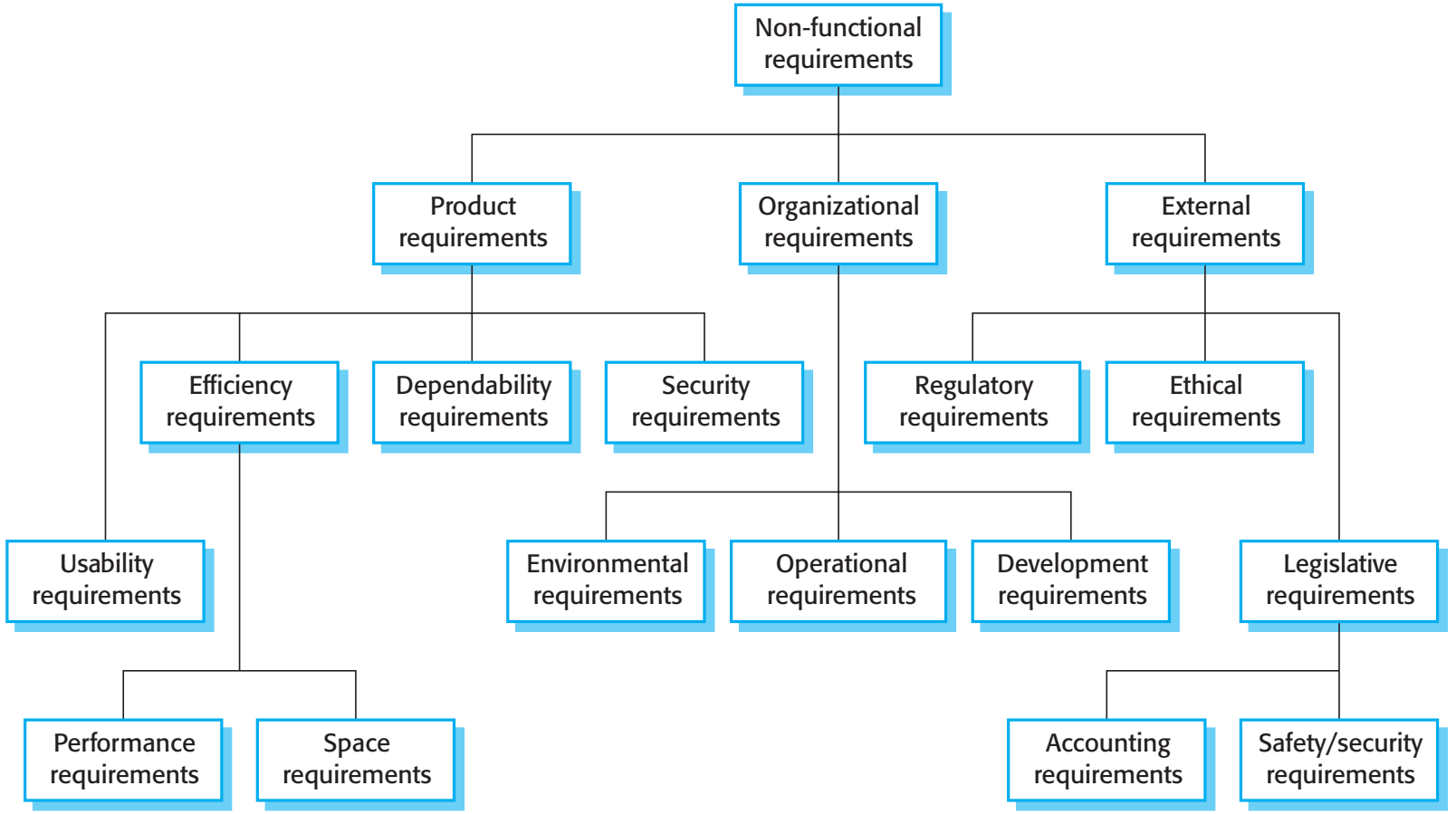


- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- Non-functional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

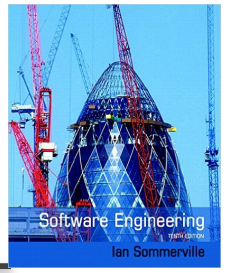
Types of nonfunctional requirement



Types of nonfunctional requirement



Examples of nonfunctional requirements in the MHC-PMS

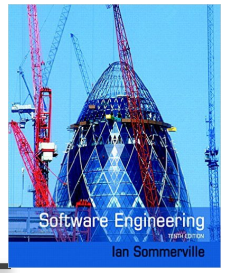


Product requirement
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational 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.

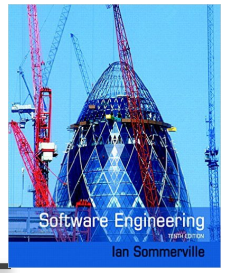
Non-functional requirements implementation



- Non-functional requirements may affect the overall architecture of a system, not just the individual components
  - Eg., to ensure that performance requirements are met,
    - you may have to organize the system to minimize communications between components

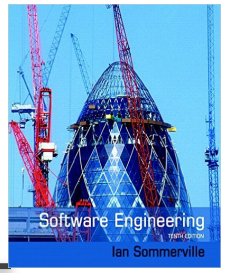
- Non-functional requirements may generate functional ones
  - e.g., security may necessitate authentication (login screens etc.)
  - It may also generate requirements that restrict existing requirements

Goals and requirements



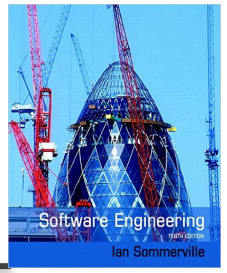
- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - A general intention of the user such as ease of use
  - Goals are helpful to developers as they convey the intentions of the system users
- Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested

Usability requirements



- (Goal)
  - The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.
- (Testable non-functional requirement)
  - Medical staff shall be able to use all the system functions after four hours of training
  - After this training, the average number of errors made by experienced users shall not exceed two per hour of system use

Metrics for specifying nonfunctional requirements

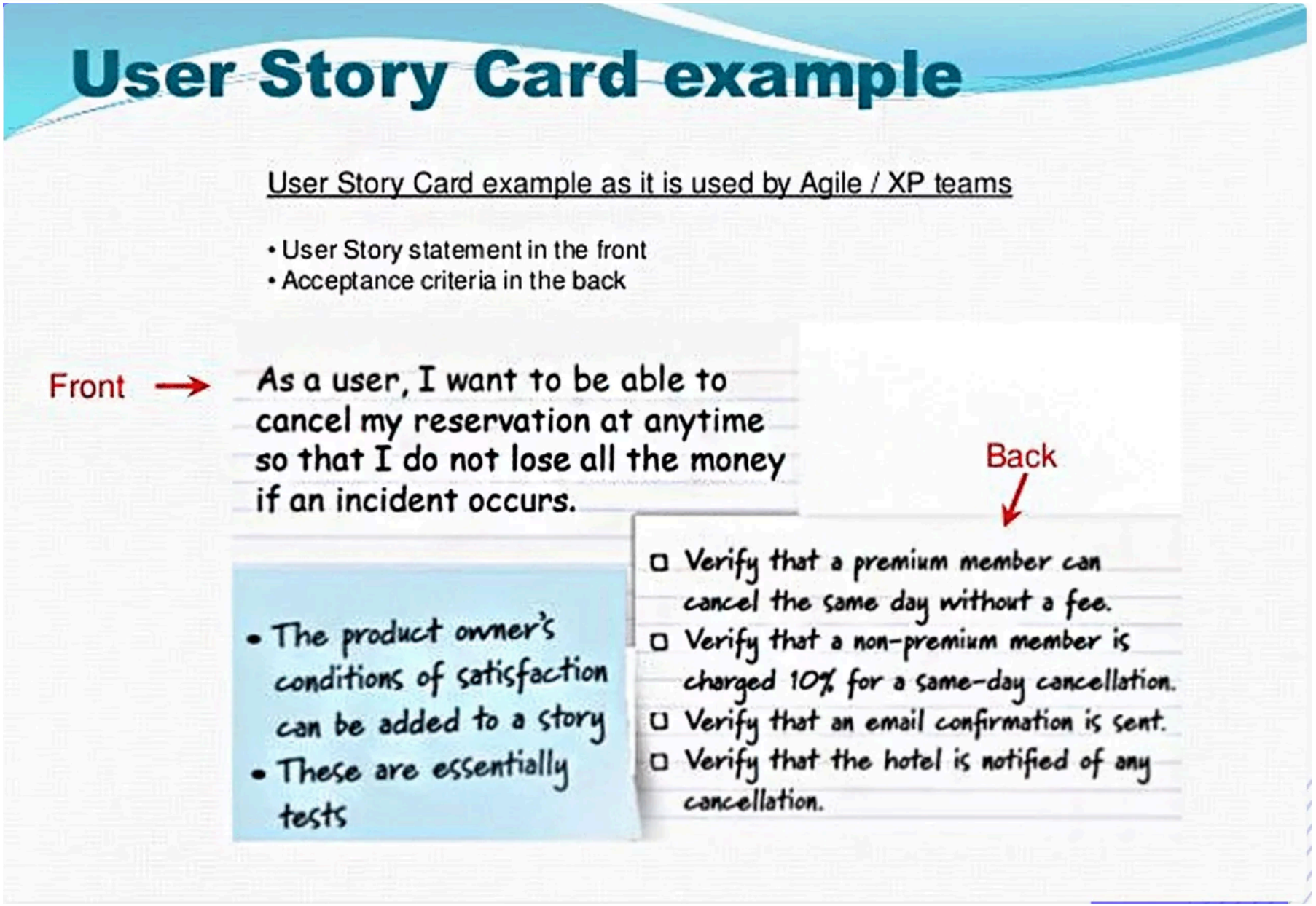


Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart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

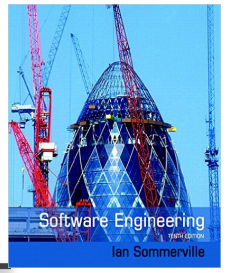
User Story Outline

Examples from:

Justinmind

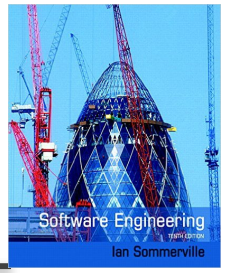


Requirements discovery



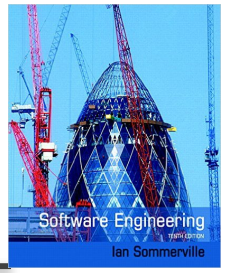
- The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- Interaction is with system stakeholders from managers to external regulators.
- Systems normally have a range of stakeholders.

Stakeholders in the MHC-PMS



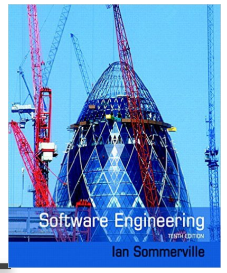
- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.

Stakeholders in the MHC-PMS



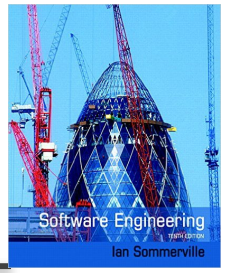
- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Health care managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Interviewing



- Formal or informal interviews with stakeholders are part of most RE processes.
- Types of interview
  - Closed interviews based on pre-determined list of questions
  - Open interviews where various issues are explored with stakeholders.
- Effective interviewing
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

Interviews in practice



- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn’t worth articulating.

Object Oriented Design

A Concept Overview

Basis of OO Design

Encapsulation in an object

- Information hiding
- Decoupling of the use of an object (code) with the implementation of an object (code)

Behavior

- No default use of “getters” or “setters”

Small methods

- one method does one thing

Basis of OO Design

Message passing paradigm

- Event/User driven programming (*no handholding*)
- "distributed" processing

Main line almost empty

- only used to "set the stage" and setoff first event

Delegation

- Do not ask for information and then do a computation
- Let object do the computation itself and ask for the answer

Documenting Design

(*Now Very Important*)

Need documentation to "hold" application together

- No through-line anymore
- Need to know what is calling what

Two features facilitate this

- Requirements level: Use Cases
- Design level: Sequence Diagrams

  (*works with use cases*)

Signs that you are thinking procedurally

In Java

- over reliance of static methods
- ideally only main should be static

Existence of:

- God objects
- Large methods

Not fully delegating

- Asking for data to work on instead of asking for results

In Concept

Procedural	*Object Oriented*
Centralized	*decentralized*
Guided	*Emergent*
Main flow	*Event driven*
Purpose focused	*Usage focused*
Brittle	*Flexible*
Smaller scale	*larger scale*
Algorithmic	*Interactive*

In Practice

Procedural	*Object Oriented*
Large procedures	*Small methods*
Small to medium number of function call	*Large number of function (method) calls*
Data passing and updating	*Data encapsulation in objects with object passing and object state changes*

Applicability

Procedural	*Object Oriented*
Single (small number of) complex task(s)	*Large complex systems composed of interacting "simple" parts (entities)* • *i.e. systems that are composed of multiple entities that have behaviour and interact*
Decomposable problems with non-interacting parts	*Emphasis on user activity* • *User interfaces* • *Distributed / Internet / Web applications*
	Aids in testing/debugging large programs

Use Cases

Use Cases

- use case is an example of the system-to-be “in action”
  - “action” as seen from “outside the system looking in”, not the internal workings of the system
- use cases are text, not diagrams
- use cases describe interactions with the system
  - but written in “third person” as an observer factorial number of paths through system
- each pathway through a system called a scenario
  - can’t enumerate (let alone describe) all of them
- choose typical paths through system as 'main success' and then think about the alternates

A Use Case

- Brief format

  Process Sale:

  A customer arrives at a checkout with items to purchase.
  The cashier uses the POS system to record each purchased item.
  The system presents a running total and line-item details.
  The customer enters payment information,

  which the system validates and records.

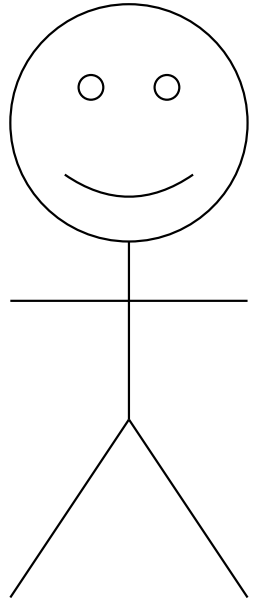
  The system updates inventory.
  The customer receives a receipt from the system,

  and then leaves with the items.

- Casual format the same except has *Main Success Scenario* along with *Alternate Scenarios*

Actors

- Properties of 'Actors'
  - Have behaviour
  - interact with the system
  - Exists:
    - outside of the system (common)
    - inside the system as ***subsystems***
- Example
  - person
  - computer system or sub-system
  - organization



subsystems == modules

Actors: Three Types

- Primary
  - Main users of system services
  - For whom the system was built to “serve”
- Supporting
  - Provide a service to the system
  - Often another computer system external to the system under discussion (e.g. Credit Card information, PayPal, , etc.)
- Offstage
  - have interest in use case but not directly involved
  - e.g. government tax department

A Use Case Template

Use Case: **<**UC ID> <UC concise name>: <optional - UC full name>

Primary Actor: <has goals/needs satisfied by the UC>

Goal: <high-level description of use case purpose>

Stakeholders List: <list of anyone affected by the use case along with their goals>

Initiating Event: <the event(s) that must have happened before Use Case can be executed >

Pre Conditions: <list of conditions that are true about the "world" when the scenario starts >

Main Success Scenario: <process statements> *this + alt. flows are the UC's largest part*

Alternate Flows or Exceptions: <more process statements>

Post Conditions: <list of conditions that must be true when the Scenario is complete >

Use Cases Utilized: < list of other use cases used>

Scenario Notes: < description of supporting actors, concurrency of actions,
and any additional information such as requirements related to the UC.>

Use Case Example: Fully Dressed

UC1: Get paid for car accident

Stakeholders List:

- Claimant wants to be reimbursed
- Insurance Company wants to ensure that accident is genuine and insurable
- Agent wants to meet with as many clients as possible

Primary Actor: the claimant

Initiating Event:

- Claimant has a car accident and wants to file an insurance claim

Basic Flow: **(aka Main Success Scenario)**

1. Claimant submits claim with substantiating data.
2. Insurance Company verifies claimant owns a valid policy
3. Insurance Company assigns agent to examine case
4. Agent verifies all details are within policy guidelines
5. Insurance Company pays claimant

Use Case Example: Fully Dressed

Alternate Flows: **(aka Alternate Scenarios, aka Extensions)**

1a. Submitted data is incomplete:

1a.1. Insurance company requests missing information

1a.2. Claimant supplies missing information

2a. Claimant does not own a valid policy:

2a.1. Insurance Company declines claim,

2a.2. Insurance Company notifies claimant,

2a.3. Insurance Company records all this,

2a.4. Insurance Company terminates proceedings.

continued next page …

Use Case Example: Fully Dressed

Alternate Scenario: continued

3a. No agents are available at this time

3a.1. (What does the insurance company do here?)

4a. Accident violates basic policy guidelines:

4a.1. Insurance Company declines claim,

4a.2. Insurance Company notifies claimant,

4a.3. Insurance Company records all this,

4a.4. Insurance Company terminates proceedings.

4b. Accident violates some minor policy guidelines:

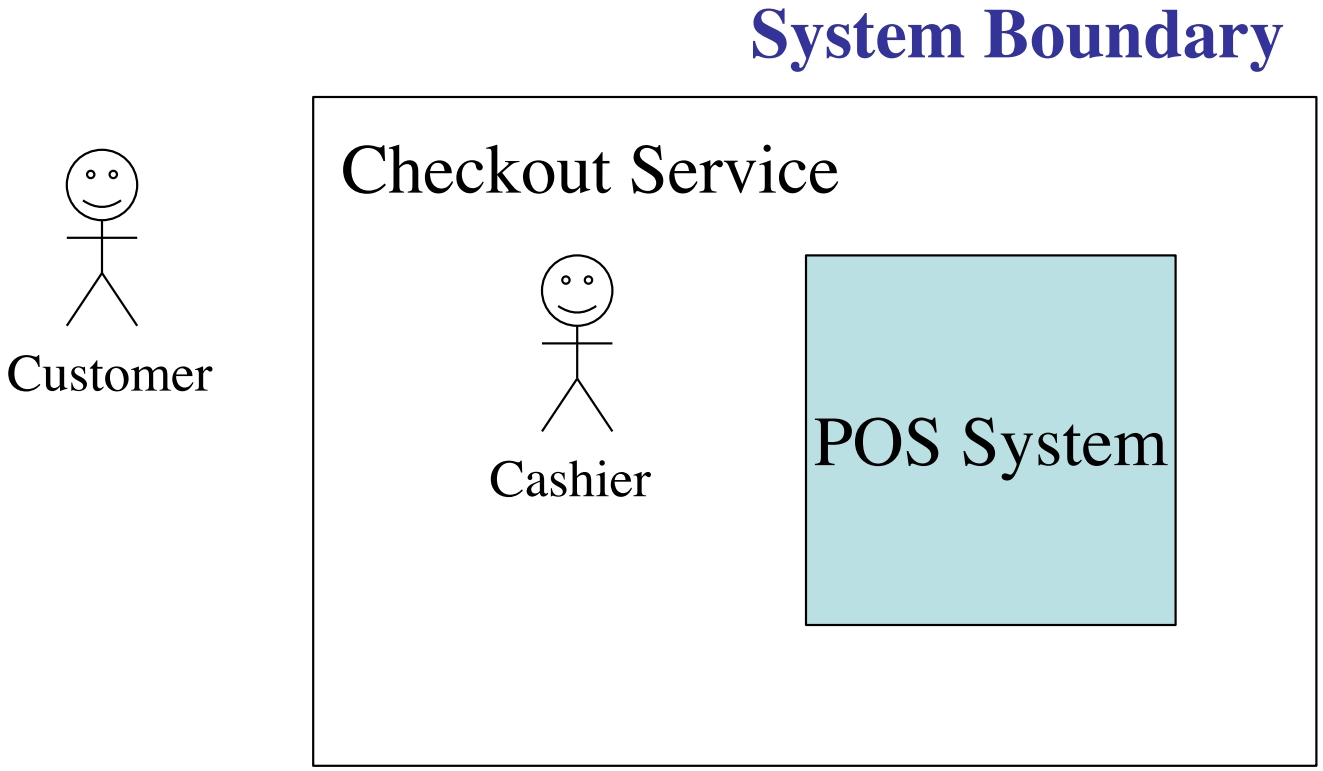
4b.1. Insurance Company begins negotiation with claimant as to degree of payment to be made.

Guidelines for Writing

- as many formats as authors it seems
- essential style writing
  - users intent, system responsibilities, no actions
  - keep the user interface out
- write tersely- avoid noise
- black box use cases
  - responsibilities, not components or design
    - the system records the sale
    - NOT the system generates an SQL INSERT statement
- Actor and Actor-Goal perspective

How to find Use Cases

1. Choose the System Boundary
2. Find Primary Actors and Goals
3. Define Use Cases



Find Actors and Goals

- Who starts and stops the system?
- Who does system administration?
- Who does user and security management?
- Does the system respond to time events?
  - *Time might be an actor then*
- Does the system respond to events in another system?
- Who evaluates the system activity
- Is there a monitoring process that restarts the system?

Tasks, Goals and Use Cases

- Actors have goals and will you your software as a tool to accomplish them.
- The achievement of the goals produces a result that the actor values
  - rather than ask 'what do you do' which gets you a description of what already exists
  - ask what are your goals and what are the measurable results
- Use case names often is a verb associated with the goal

Finding 'good' use cases

- big enough to form a useful picture of the purpose of an important part the system
- small enough that you suspect it can be implemented using half a dozen “objects”
- Boss Test (importance test)
  - logging in as use case?
  - tell boss “today I logged in to the system” as test
- Size test
  - Not a single action
  - should be complex enough so basic flow should have 5 to 15 steps

Use Cases Diagrams

Use Case Diagrams: The bits

Figure C.115
Use case

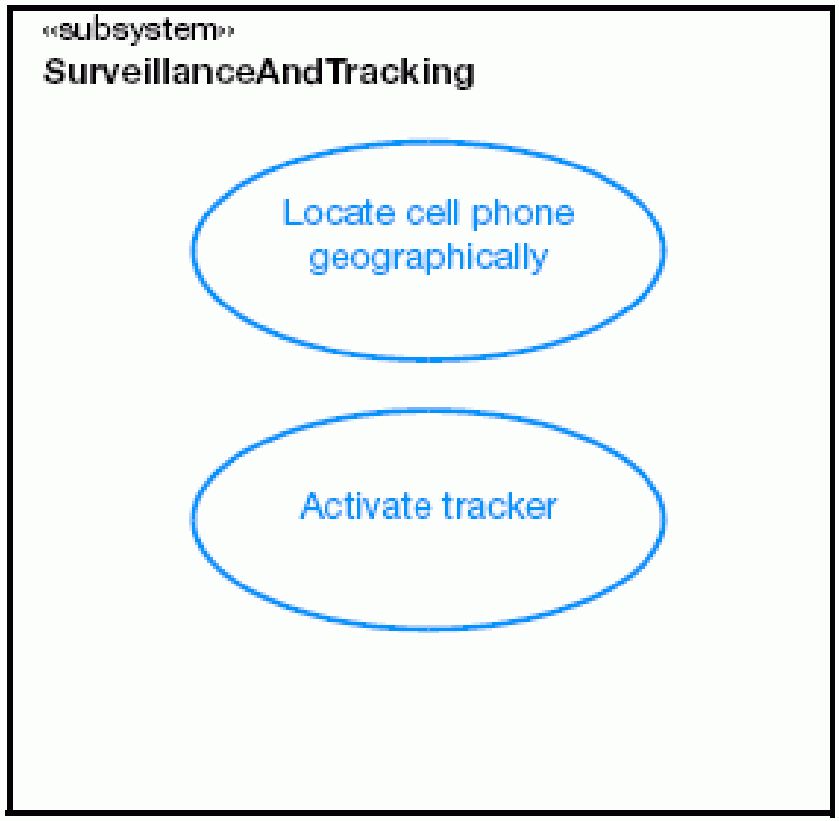
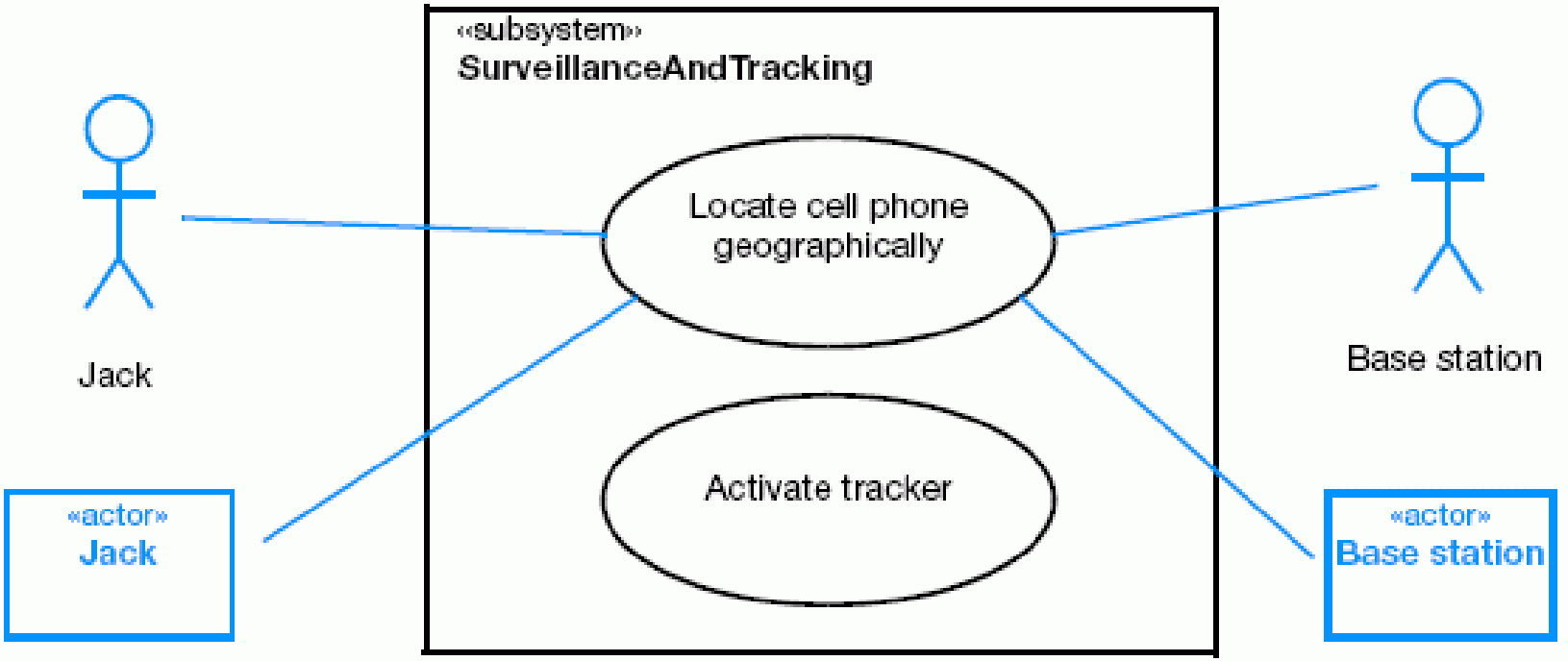


Figure C.116
Interactors



Relationships

Figure C.117
Generalization relationship

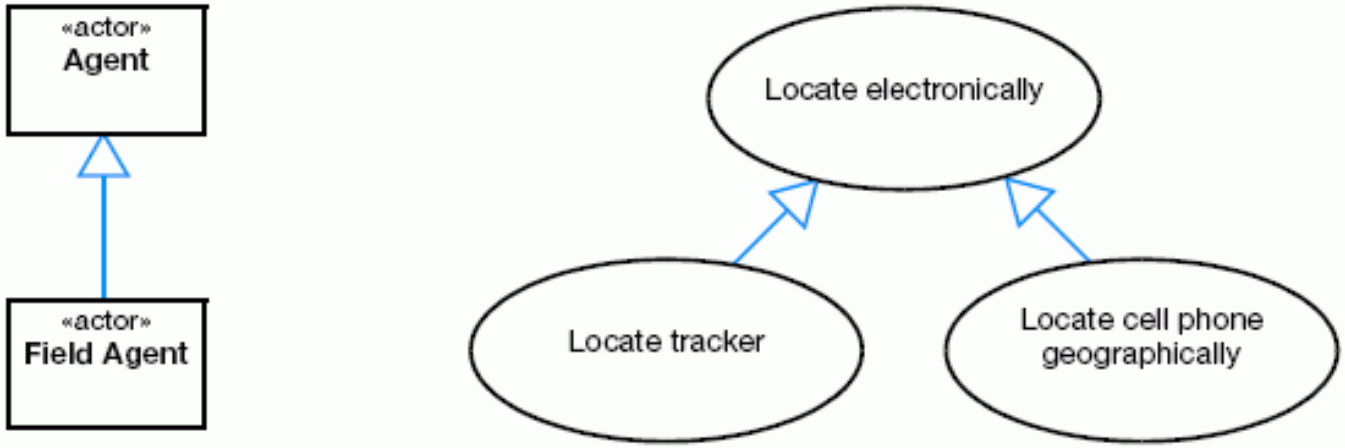


Figure C.118
Extend relationship

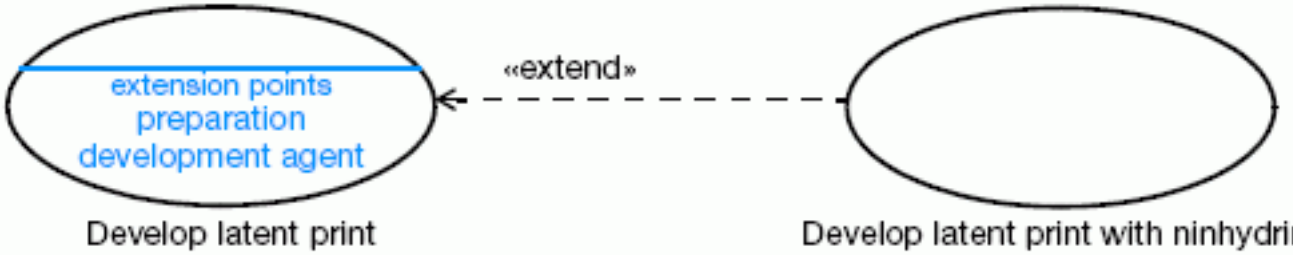
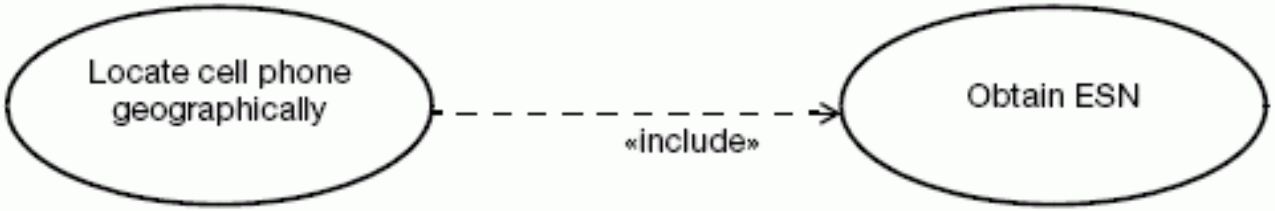
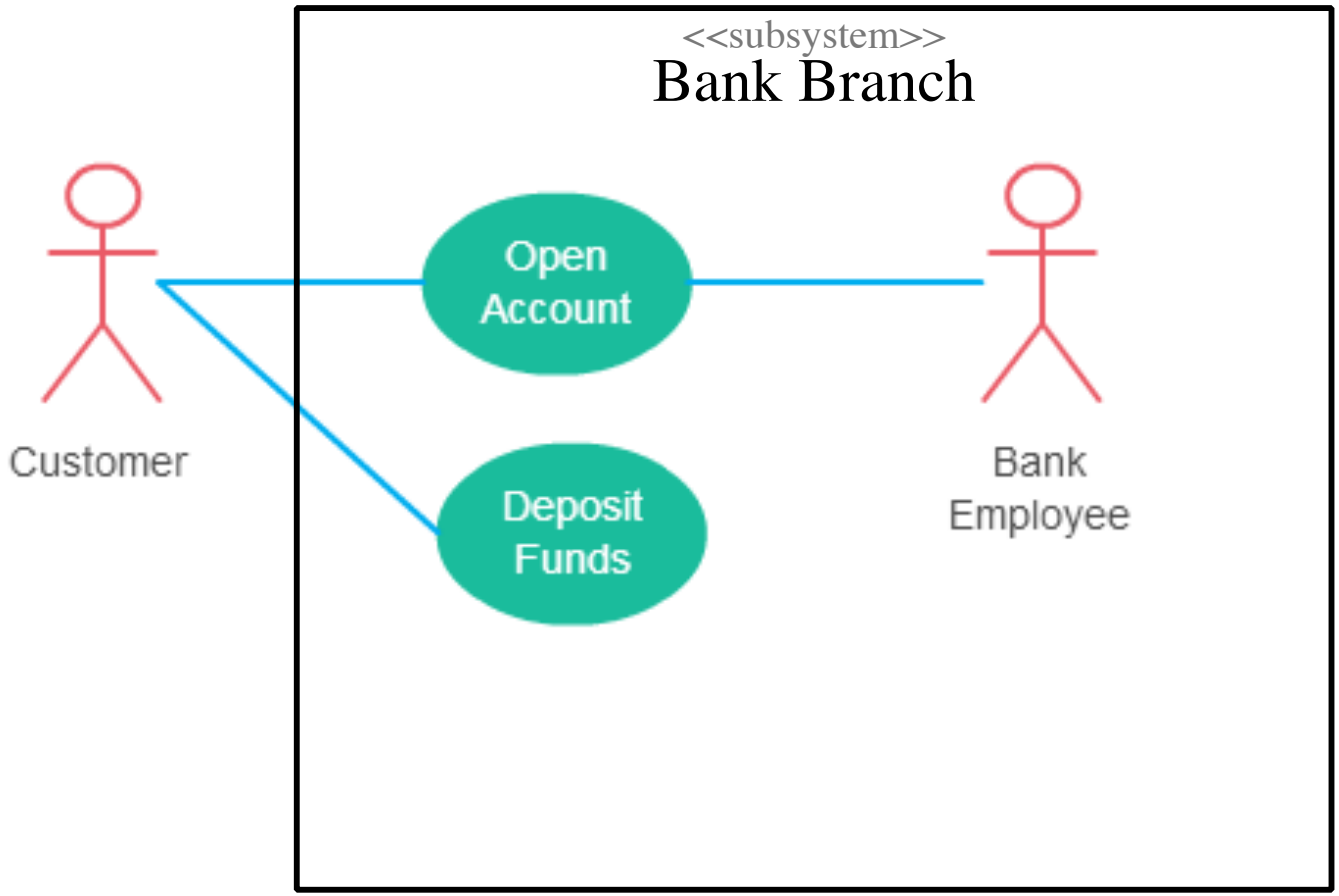


Figure C.119
Include relationship

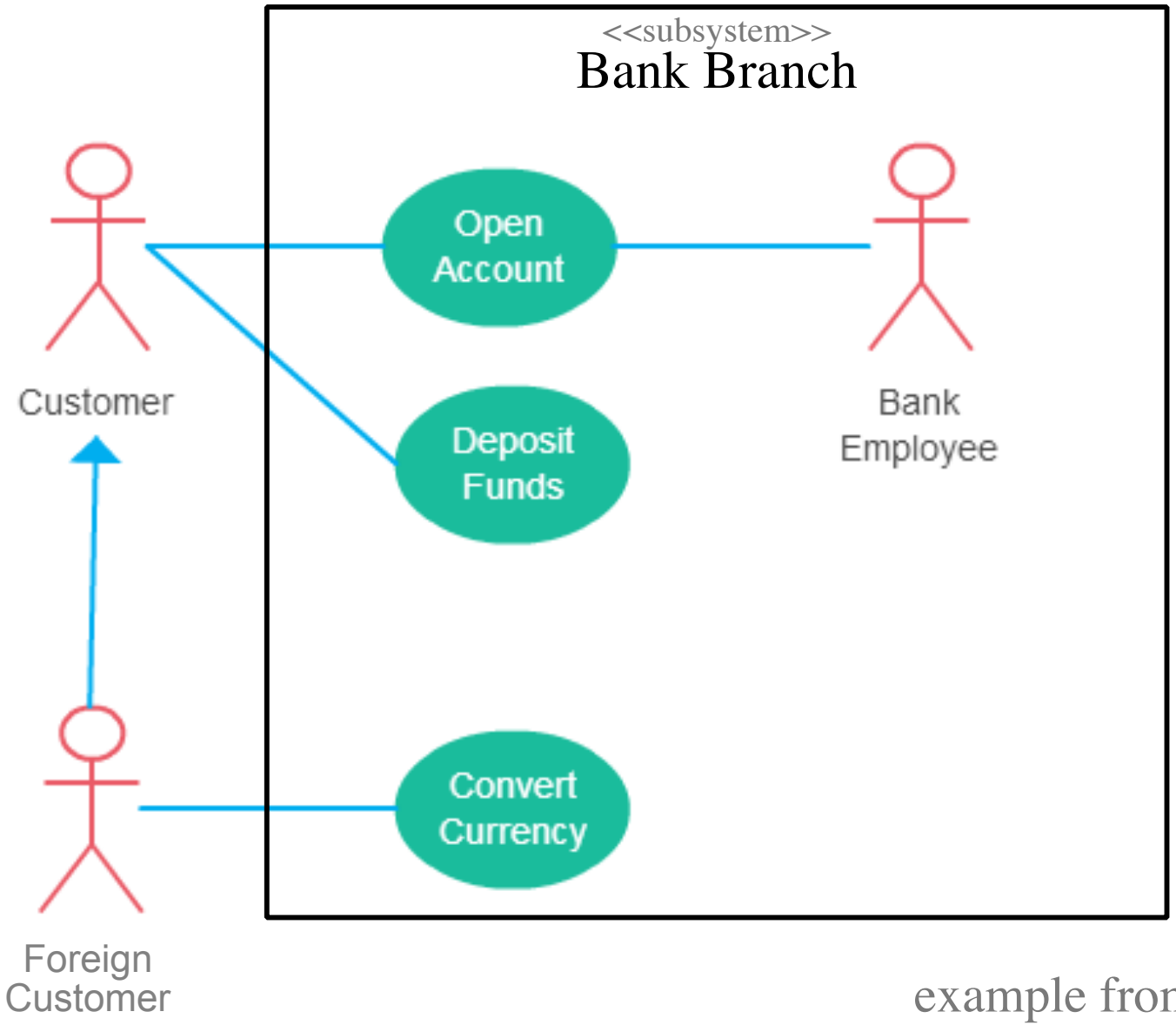


Use Case Diagrams: Example



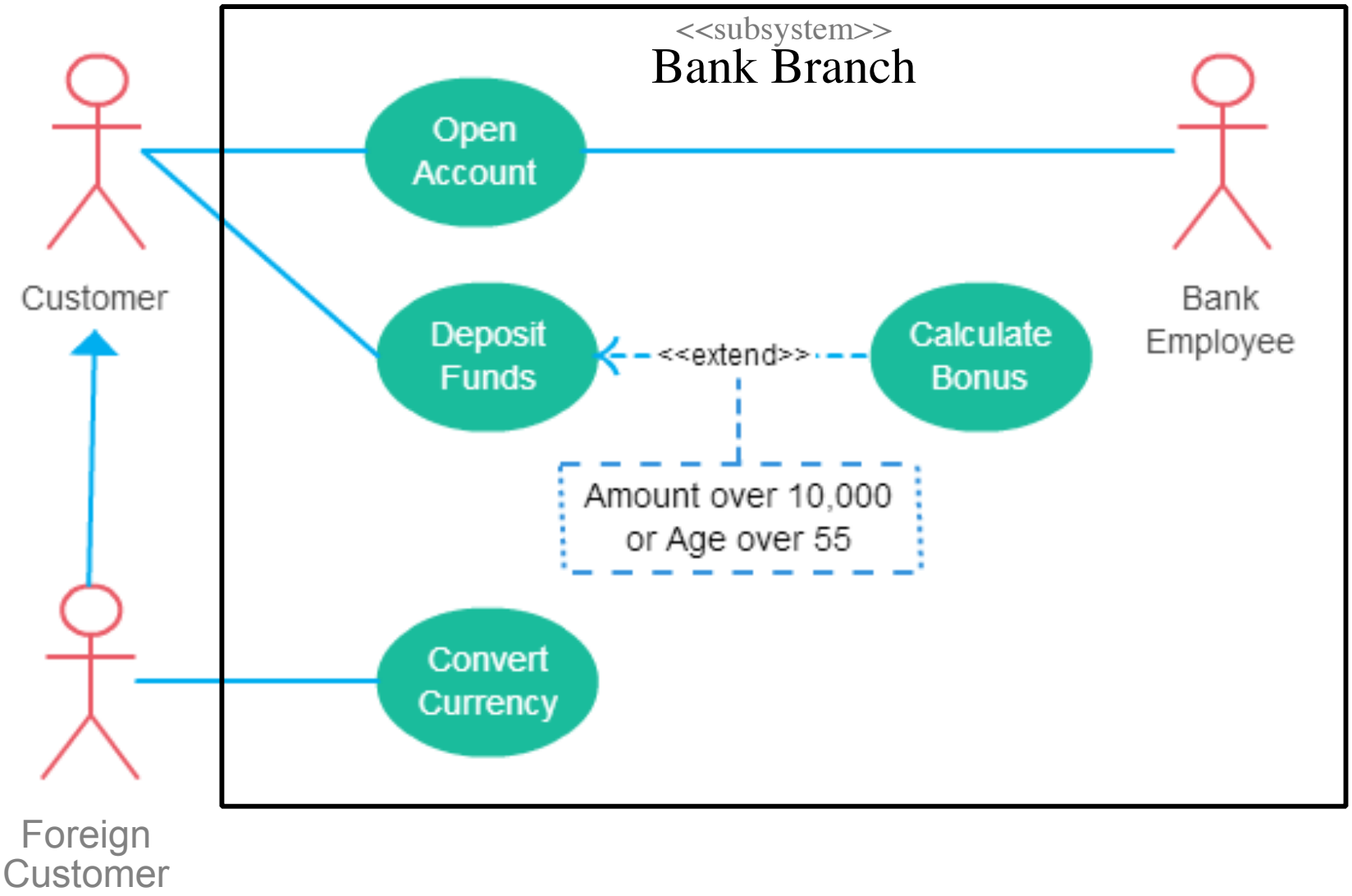
example from Creately

Use Case Diagrams: Generalize Actor



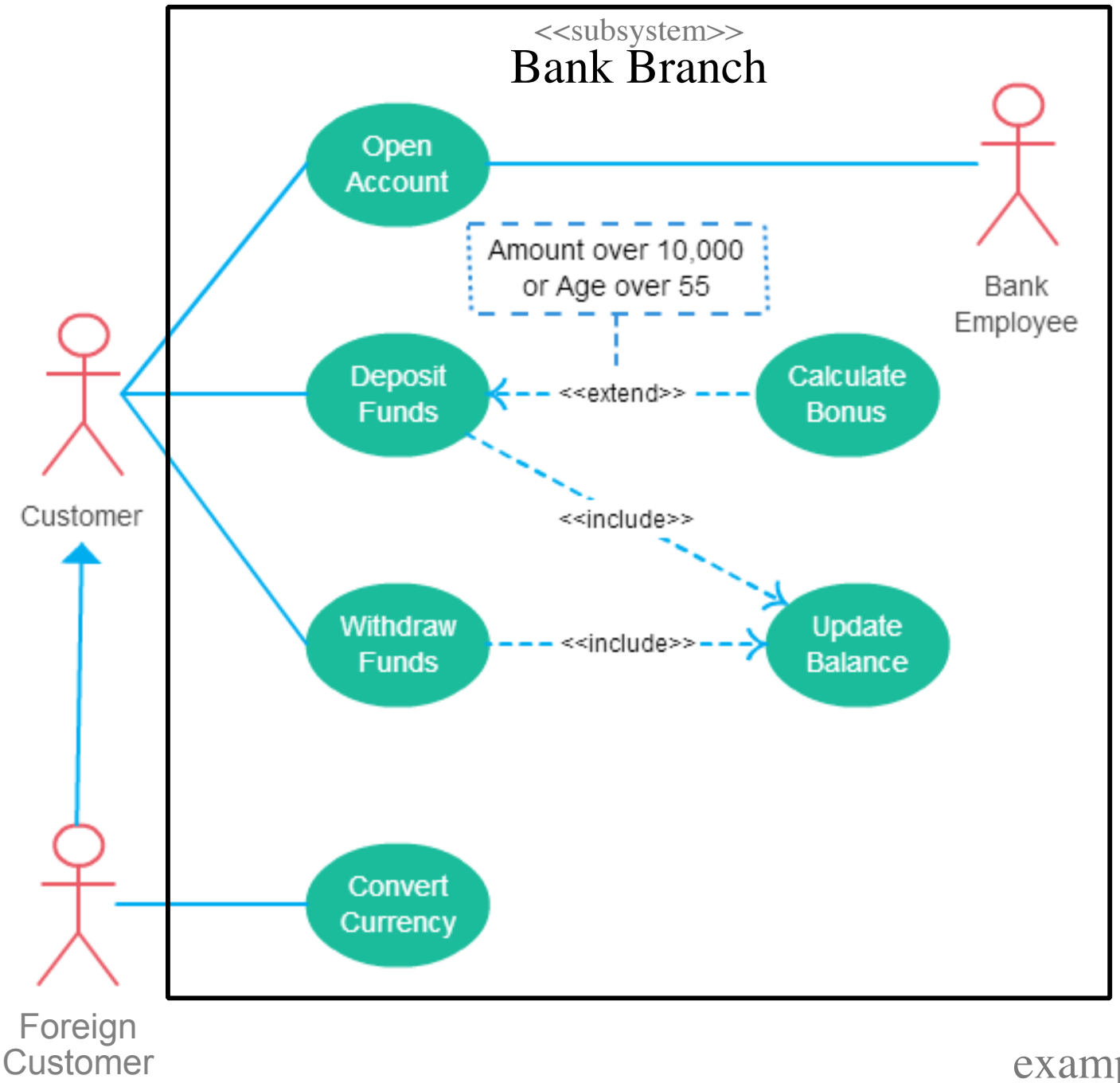
example from Creately

Use Case Diagrams: Extend Use Case



example from Creately

Use Case Diagrams: Include Use Case



example from Creately

Use Case Diagrams: Generalize Use Case