Requirements Engineering

Seminar 3:

* Lauesen chapters: 6: Quality Requirements 7: Requirements in the product life cycle 9: Checking and validation 10: Techniques at work

* [QUPER]: Regnell, Björn, Richard Berntsson Svensson, and Thomas Olsson. "Supporting roadmapping of quality requirements." IEEE software 25.2 (2008). doi: 10.1109/MS.2008.48

* [RP]: Ruhe, Gunther, and Moshood Omolade Saliu. "The art and science of software release planning." IEEE software 22.6 (2005): 47-53. doi: 10.1109/MS.2005.164

Funktionella krav:

- Vad som görs
- Ofta antingen/eller

- Indata – Utdata
- Funktioner

Kvalitetskrav,
(kallas även icke-funktionella krav, extrafunktionella krav):

- Hur bra det görs
- Mäts ofta på en skala
- Sätter begränsningar på systemet (eller utvecklingsprocessen)
- Kan ofta slå tvärs över många funktioner

Prestanda
Tillförlitlighet
Användbarhet
Säkerhet
Interoperabilitet
Underhållsbarhet
…

Men uppdelningen är inte svartvit...

Functional reqs FR:

- What the system shall do
- Often intended to be implemented as a whole or else not implemented at all
- Often regards input/output **data** and **functions** that process the input data to produce the output

Quality Requirements QR,
(also known as: Non-Functional Reqs (NFR) or Extra-Functional Reqs)

- How **good** the system shall do it
- Often measured on a scale
- Often put constraints on the system (or the development process)
- Often cross-cutting: may impact many functions or even the whole system

Performance
Reliability
Usability
Safety, Security
Interoperability
Maintainability
…

But the division is not black and white…

FR & QR are often tightly coupled

In practice it is often difficult to separate functional and quality requirements as quality requirements often are manifested into extra functionality.

Example: **Quality** requirement on security requires a log-in **function**.

Difficult trade-offs among QR

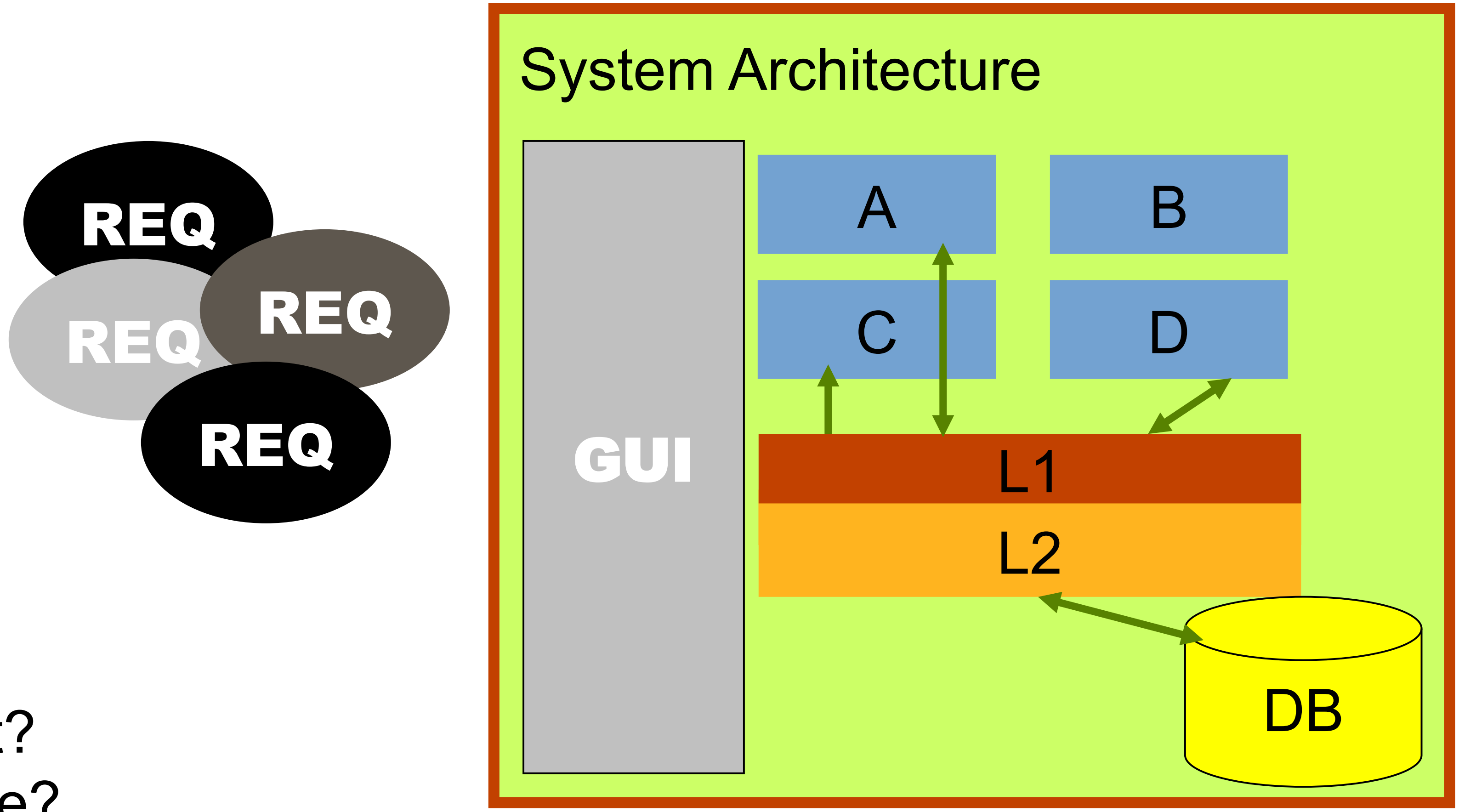
Quality requirements often **counteract** each other.

Common examples:

- Higher performance
  -> lower maintainability
- Higher security
  -> lower usability

Requires carefully considered trade-offs!

Quality requirements often determine choice of architecture



Cost?
Value?
Long-term vs short-term?

Paper [QUPER]

Supporting Roadmapping of Quality Requirements

Björn Regnell, Richard Berntsson Svensson, Thomas Olsson, IEEE Software 25(2) pp 42-47 March-April 2008

https://vimeo.com/10581781

Quality Requirements challenge in market-driven RE

Systematic prioritization of **FEATURES** is state-of-art in roadmapping and platform/product scoping

…but…

Prioritisation of **QUALITIES** is often handled ad hoc with no specific support for roadmapping

One FR imply many different qualities.

How to scope both FR and QR together?

Improving Quality Requirements

It's 3D **Cost &Benefit &Quality**

Problem:

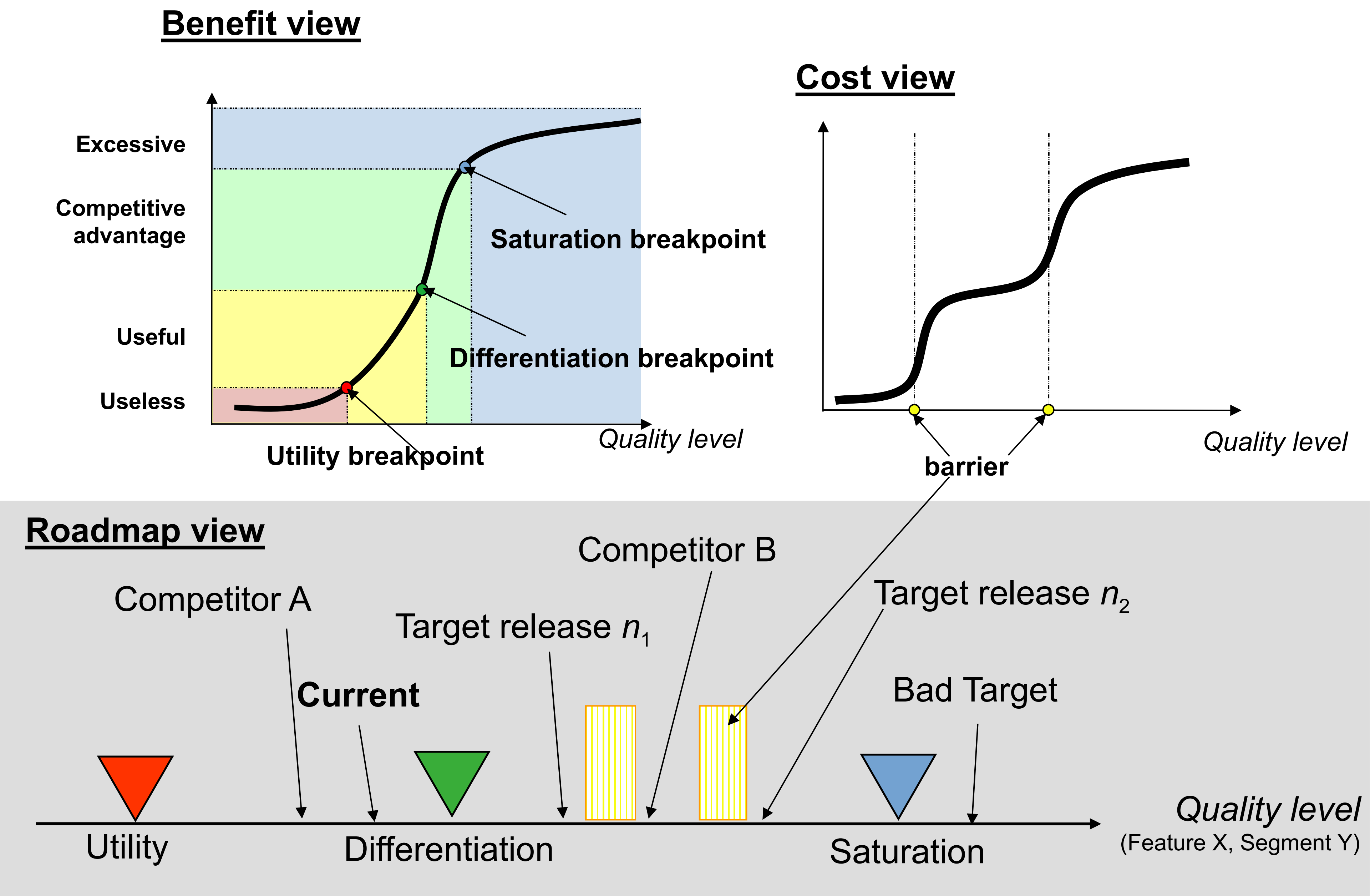
Quality requirements such as performance are often given without explanation

- Would just a little less still be almost as valuable?
- Would just a little less be very much cheaper?

One proposed solution:

Estimate cost-benefit breakpoints and barriers with QUPER = Quality Performance reference model

QUPER model views: Benefit, Cost, Roadmap



QUPER example steps

- **Step 1 - Description**
  - *Quality indicator:* Time to play music [seconds]
  - *Quality type:* Performance
  - *Definition:* Measured from player invoke button pressed until music is played using 2 GB memory stick type X with 100 tracks with average duration of 3 min
- **Step 2 - Current reference products**
  - *Competitor Product X:* 4 seconds
  - *Competitor Product Y:* 2 seconds
  - *Own Product Z (Qref):* 3 seconds
- **Step 3 – Current market expectations**
  - *Utility breakpoint:* 5 seconds
  - *Differentiation breakpoint:* 1.5 seconds
  - *Saturation breakpoint:* 0.2 seconds
- **Step 4 – Estimate the closest cost barrier (CB1)**
  - *Q1:* 2 seconds
  - *C1:* 4 weeks
- **Step 5 – Estimate the second cost barrier (CB2)**
  - *Q2:* 1 second
  - *C2:* 24 weeks
- **Step 6 – Candidate targets**
  - *Min target:* 2 seconds – This target is possible without a new architecture, but needs some software optimization.
  - *Max target:* 1 second – If we create a new architecture, this target (which is better than differentiation) will be easy to reach. Users might require this level of quality within 2 years.

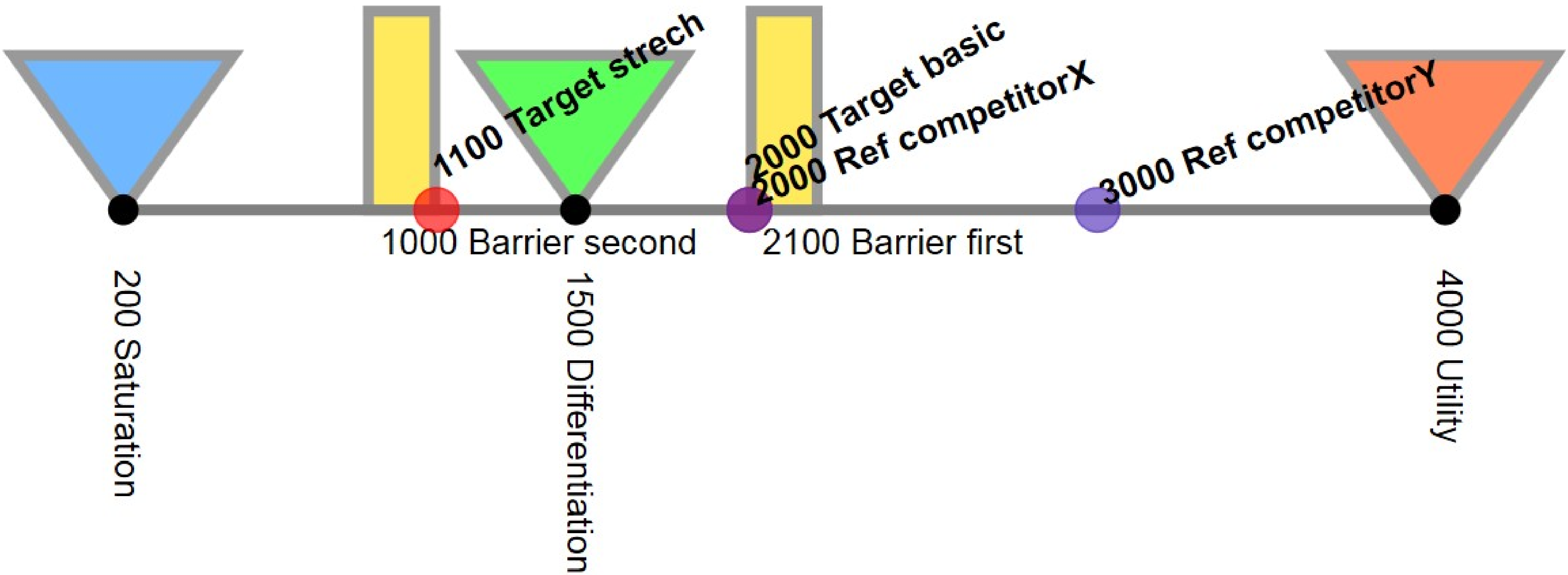
reqT QUPER example

```
val m = Model(
  Quality("mtts") has (
    Gist("Mean time to startup"),
    Spec("Measured in milliseconds using Test startup"),
    Breakpoint("utility") has Value(4000),
    Breakpoint("differentiation") has Value(1500),
    Breakpoint("saturation") has Value(200),
    Target("basic") has (
        Value(2000), Comment("Probably possible with existing architecture.")),
    Target("strech") has (
        Value(1100), Comment("Probably needs new architecture.")),
    Barrier("first") has (Min(1900), Max(2100)),
    Barrier("second") has Value(1000),
    Product("competitorX") has Value(2000),
    Product("competitorY") has Value(3000)),
  Test("startup") verifies Quality("mtts"),
  Test("startup") has (
    Spec("Calculate average time in milliseconds of the startup time over 10 
executions from start button is pressed to logon screen is shown.")))
```

Targets represent (candidate) requirements.
The other stuff is there to define what we mean with the targets.

[Example modified from "Setting quality targets for coming releases with QUPER: an industrial case study",
R. Berntsson Svensson, Y. Sprockel, B. Regnell, S. Brinkkemper, *Requirements Engineering*, DOI: 10.1007/s00766-011-0125-0]

Quper export to svg with reqT




```
reqT.exporter.toQuperSpec(m).toSvgDoc.save("q.svg")
reqT.desktopOpen("q.svg")
```

Discussion QR

- What quality features of a word processor do you appreciate?

Fig 6.1 Quality factors

McCall	ISO 9126
US Airforce 1980	**Functionality**
Operation:	Accuracy
Integrity	Security
	Interoperability
Correctness !!	**Suitability !!**
	Compliance !!
	Reliability
Reliability	Maturity
	Fault tolerance !!
	Recoverability !!
Usability	**Usability**
Efficiency	**Efficiency**
Revision:	
Maintainability	**Maintainability**
Testability	Testability
Flexibility	Changeability
	Analysability !!
Transition:	**Stability !!**
	Portability
Portability	Adaptability
Interoperability	**Installability !!**
Reusability !!	**Conformance !!**
	Replaceability !!

Use as check lists

Fig 6.2 Quality grid

Quality factors for Hotel system	Critical	Impor-tant	As usual	Unim-portant	Ignore
Operation					
Integrity/security			X		
Correctness			X		
Reliability/availab.		1			
Usability		2			
Efficiency			X		
Revision					
Maintainability			X		
Testability			X		
Flexibility			X		
Transition					
Portability					X
Interoperability	3			4	
Reusability					X
Installability		5			

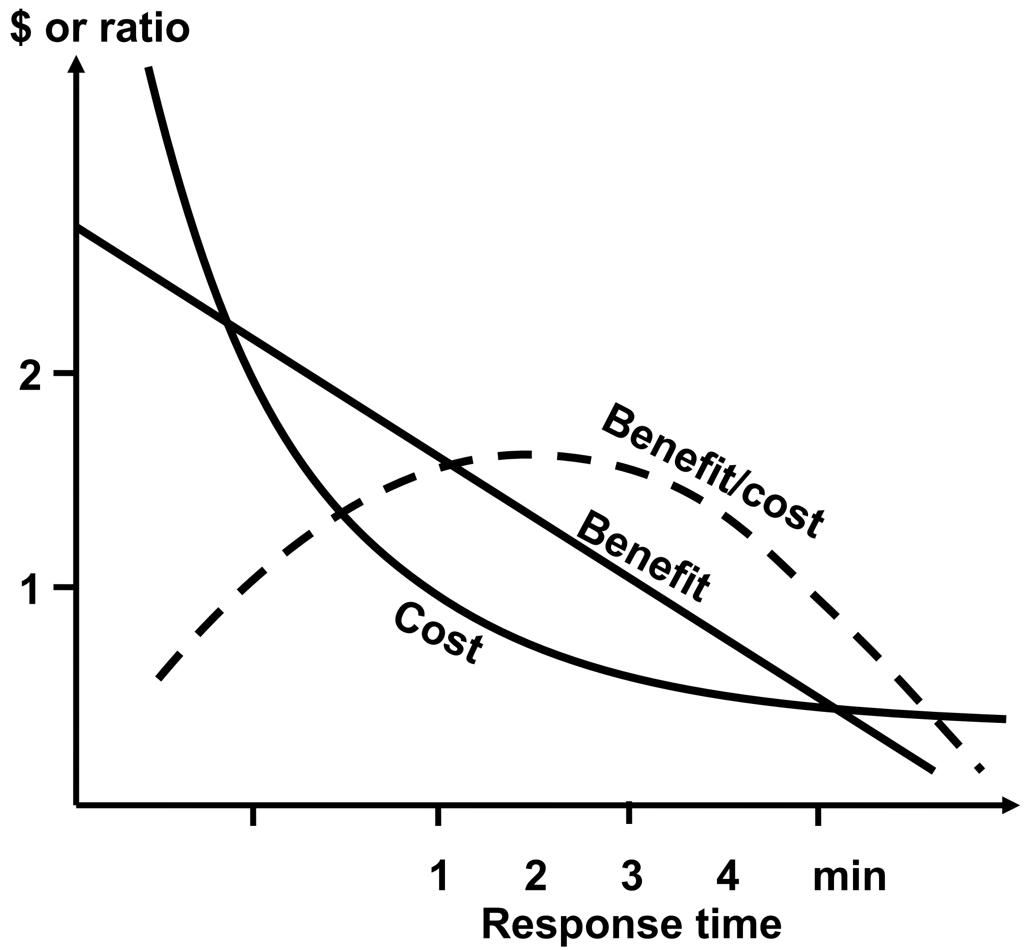
Concerns:

1. Hard to run the hotel if system is down. Checking in guests is impossible since room status is not visible.
2. We aim at small hotels too. They have less qualified staff.
3. Customers have many kinds of account systems. They prioritize smooth integration with what they have.
4. Integration with spreadsheet etc. unimportant. Built-in statistics suffice.
5. Must be much easier than present system. Staff in small hotels should ideally do it themselves.

Fig 6.3A Open metric and open target

Physical limits	R1: Product shall detect speed violation and take photo within 0.5 seconds.
Best available is 4 minutes?	R2: Product shall compute a room occupation forecast within 2 minutes.
Nobody strives for 2 minutes	R3: Product shall compute a room occupation forecast within 4 minutes.
Open target but how important?	R4: Product shall compute a room occupation forecast within ___ minutes.
Open target + expectations	R5: Product shall compute a room occupation forecast within ___ minutes. (Customer expects one minute.)
Supplier uses another approach?	R6: Forecast shall be computed with exponential trend smoothing and seasonal adjustments.
Open metric	R7: The supplier shall specify the forecast accuracy for hotels similar to ours.

Fig 6.3C Cost/benefit of response time



From: Soren Lauesen: Software Requirements

Fig 6.4 Capacity and accuracy requirements

Capacity requirements:

R1: The product shall use < 16 MB of memory even if more is available.

R2: Number of simultaneous users < 2000

R3: Database volume:
#guests < 10,000 growing 20% per year
#rooms < 1,000

R4: Guest screen shall be able to show at least 200 rooms booked/occupied per day, e.g. for a company event with a single "customer".

Accuracy requirements:

R5: The name field shall have 150 chars.

R6: Bookings shall be possible at least two years ahead.

R7: Sensor data shall be stored with 14 bit accuracy, expanding to 18 bits in two years.

R8: The product shall correctly recognize spoken letters and digits with factory background noise ___ % of the time. Tape B contains a sample recorded in the factory.

Fig 6.5A Performance requirements

Performance requirements:

R1: Product shall be able to process 100 payment transactions per second in peak load.

R2: Product shall be able to process one alarm in 1 second, 1000 alarms in 5 seconds.

R3: In standard work load, CPU usage shall be less than 50% leaving 50% for background jobs.

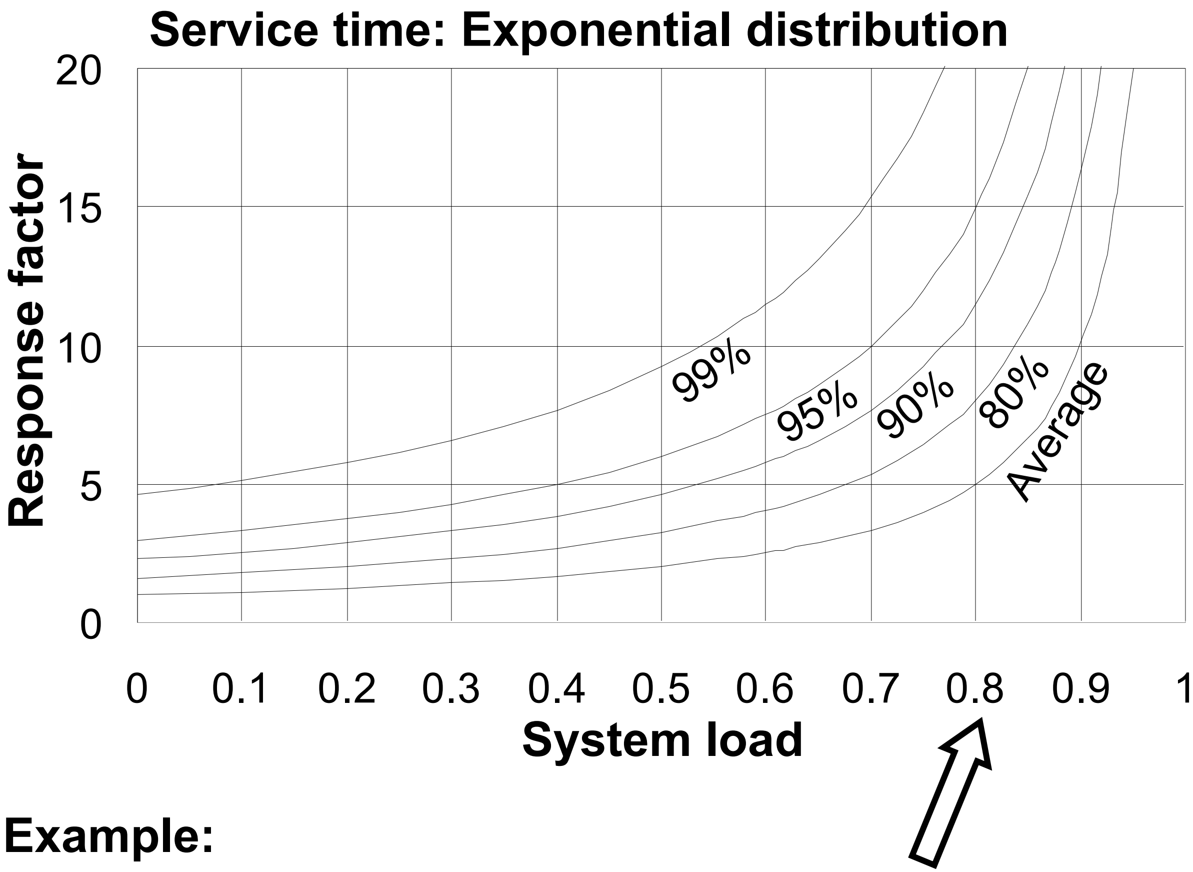
R4: Scrolling one page up or down in a 200 page document shall take at most 1 s. Searching for a specific keyword shall take at most 5 s.

R5: When moving to the next field, typing must be possible within 0.2 s. When switching to the next screen, typing must be possible within 1.3 s. Showing simple report screens, less than 20 s. (Valid for 95% of the cases in standard load)

R6: A simple report shall take less than 20 s for 95% of the cases. None shall take above 80s. (UNREALISTIC)

Cover all product functions?

Fig 6.5B   Response times, M/M/1



Example:
Service time: Time to process one request
Average service time: 8 s (exp. distr.)
Average interarrival time: 10 s (exp. distr.)
System load: 8/10 = 0.8

Average response time:
5 × service time = 40 s

90% responses within:
12 × service time = 96 s

Fig 6.6A   Usability

Usability requirements?

R1: System shall be easy to use??

R2: 4 out of 5 new users can book a guest in 5 minutes, check in in 10 minutes, . . .   *New user* means . . . Training . . .

Achieving usability

- Prototypes (mockups) before programming.
- Usability test the prototype.
- Redesign or revise the prototype.

Easier programming. High customer satisfaction.

Defect types

Program error: Not as intended by the programmer.

Missing functionality: Unsupported task or variant.

Usability problem: User cannot figure out . . .

Fig 6.6B Usability problems

Examples of usability problems

P1: User takes long time to start search. Doesn't notice "Use F10". Tries many other ways first.

P2: Believes task completed and result saved. Should have used *Update* before closing.

P3: Cannot figure out which discount code to give customer. Knows which field to use.

P4: Crazy to go through 6 screens to fill 10 fields.

Problem classification

Task failure: Task not completed - or believes it is completed.

Critical problem: Task failure or complaints that it is cumbersome.

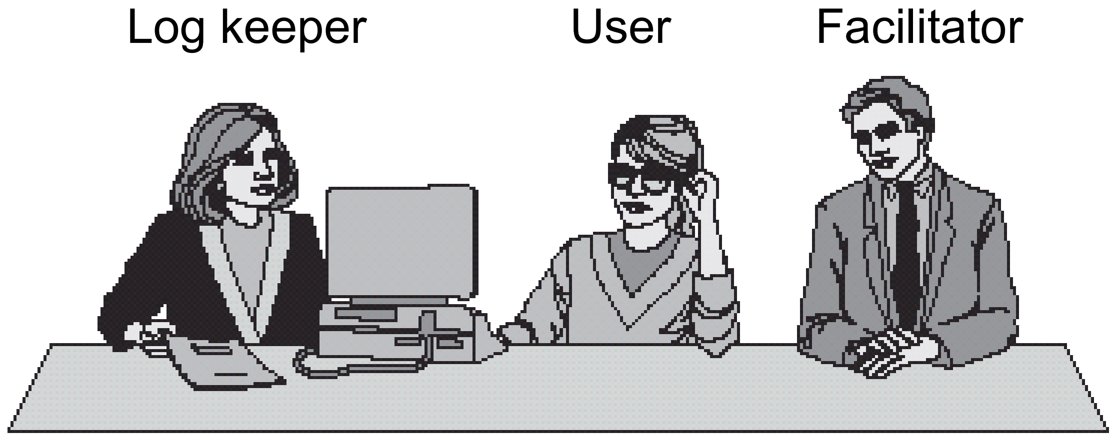
Medium problem: Finds out solution after lengthy attempts.

Minor problem: Finds out solution after short attempts

Fig 6.6C Usability test & heuristic evaluation

Usability test
Realistic introduction
Realistic tasks

Note problems
- Observe only or
- Think aloud & ask



Heuristic evaluation
Expert's predicted problems
- Inspection/Review

Usability test:
Cover all tasks?
Mockups find same problems
as test with final system?

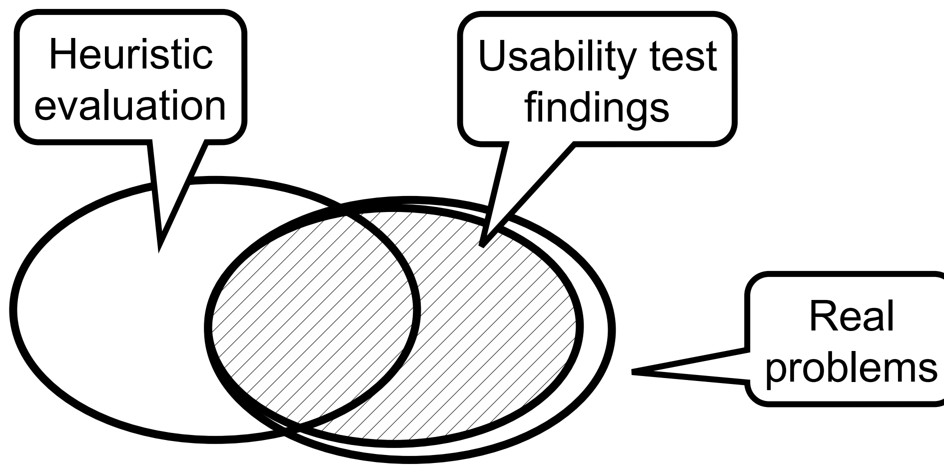


Fig 6.6D Defects & usability factors

Defect correction

Program errors	**Usability problems**
Expected	Surprising?
Inspection OK	Inspection low hit-rate
Detect in test stage	Detect in design stage
Mostly simple	Often redesign
Test equipment OK	Subjects hard to find

Usability

Fit for use = tasks covered — Functional requirements

\+

Ease of use =

- Ease of learning
- Task efficiency
- Ease of remembering
- Subjective satisfaction
- Understandability

Usability factors

<u>Fig 6.7(A) Usability requirements</u>

	Risk: Cust.	Risk: Suppl
Problem counts R1: At most 1 of 5 novices shall encounter critical problems during tasks Q and R. At most 5 medium problems on list.		
Task time R2: Novice users shall perform tasks Q and R in 15 minutes. Experienced users tasks Q, R, S in 2 minutes.		
Keystroke counts R3: Recording breakfast shall be possible with 5 keystrokes per guest. No mouse.		
Opinion poll R4: 80% of users shall find system easy to learn. 60% shall recommend system to others.		
Score for understanding R5: Show 5 users 10 common error mesages, e.g. *Amount too large.* Ask for the cause. 80% of the answers shall be correct.		

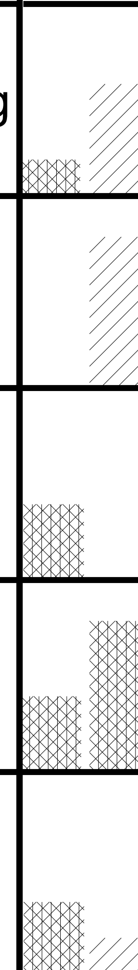
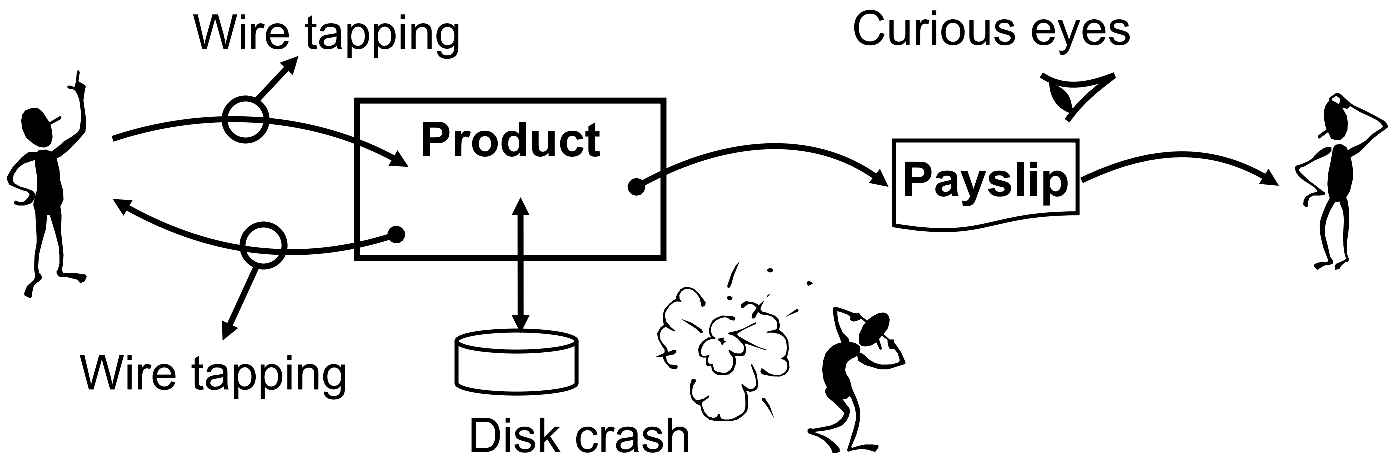


Fig 6.7(B) Usability requirements

	Risk Cust.	Risk Suppl
Design-level reqs R6: System shall use screen pictures in app. xx, buttons work as app. yy.		
Product-level reqs R7: For all code fields, user shall be able to select value from drop-down list.		
Guideline adherence R8: System shall follow style guide zz. Menus shall have at most three levels.		
Development process reqs R9: Three prototype versions shall be made and usability tested during design.		

Fig 6.8A Threats



Threats	Violate	Prevention, e.g.
Input, e.g.		
Mistake	Integrity	Logical checks
Illegal access	Authenticity	Signature
Wire tapping	Confidentiality	Encryption
Storing, e.g.		
Disk crash	Availability	RAID disks
Program error	Integrity	Test techniques
Virus deletes data	Availability	Firewall
Output, e.g.		
Transmission	Availability	Multiple lines
Fraud	Confidentiality	Auditing
Virus sends data	Authenticity	Encryption

Fig 6.9 Security requirements

R1: Safeguard against loss of database. Estimated losses to be < 1 per 50 years.

R2: Safeguard against disk crashes. Estimated losses to be < 1 per 100 years.

R3: Product shall use duplicated disks (RAID disks).

R4: Product shall safeguard against viruses that delete files. Remaining risk to be < ______.

R5: Product shall include firewalls for virus detection.

R6: Product shall follow good accounting practices. Supplier shall obtain certification.

R7: Product shall prevent users deleting invoices before transfer to the account system.

R8: The supplier shall as an option offer features for checking and reserving deposits made by credit cards.

R9: The supplier must enclose a risk assessment and suggest optional safeguards.

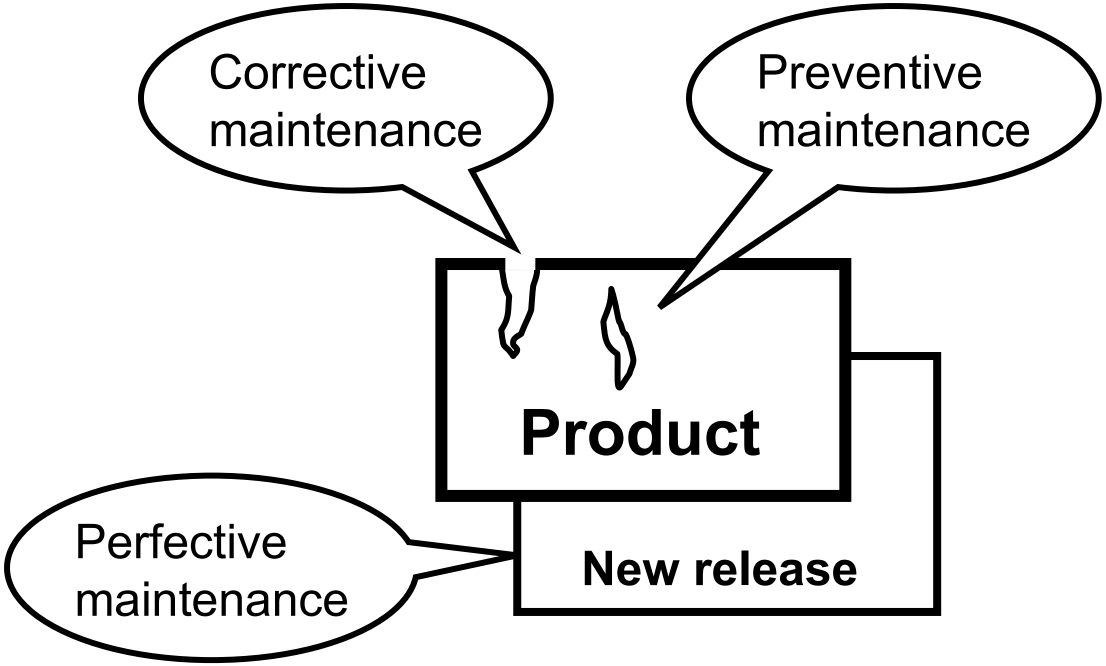
From: Soren Lauesen: Software Requirements

Examples: Capacity and Performance <=> Usability

```
Model(
  Quality("dbCapacity") has
    Spec("#guests < 10,000 growing 20% per year, #rooms < 1,000"),
  Quality("calendarAccuracy") has
    Spec("Bookings shall be possible at least two years ahead."),
  Quality("forecastPerformance") has
    Spec("Product shall compute a room occupation forecast within
         ___ minutes. (Customer expects one minute.)"),
  Quality("taskTimeUsability ") has
    Spec("Novice users shall perform tasks Q and R in 15 minutes.
         Experienced users tasks Q, R, S in 2 minutes."),
  Quality("taskTimeUsability") requires (Task("Q"), Task("R"),
         Task("S")),
  Quality("peakLoadPerformance") has
    Spec("Product shall be able to process 100 payment transactions
         per second in peak load."))
```

[Examples modified from Lauesen: "Software Requirements – Styles and Techniques"]

Fig 6.10 Maintainance



Maintenance cycle:

Report: Record and acknowledge.
Analyze: Error, change, usability, mistake? Cost/benefit?
Decide: Repair? reject? work-around? next release? train users?
Reply: Report decision to source.
Test: Test solution. Related defects?
Carry out: Install, transfer user data, inform.

Fig 6.11A Maintainability requirements

	Risk Cust.	Risk Suppl
Maintenance performance R1: Supplier's hotline shall analyze 95% of reports within 2 work hours. Urgent defects (no work around) shall be repaired within 30 work hours in 95% of the cases.		
R2: When reparing a defect, related non-repaired defects shall be less than 0.5 in average.		
R3: For a period of two years, supplier shall enhance the product at a cost of ___ per Function Point.		
Support features R4: Installation of a new version shall leave all database contents and personal settings unchanged.		
R5: Supplier shall station a qualified developer at the customer's site.		
R6: Supplier shall deposit code and full documentation of every release and correction at ____________.		

Fig 6.11B Maintainability requirements

	Risk: Cust.	Risk: Suppl
Development process requirements R7: Every program module must be assessed for maintainability according to procedure xx. 70% must obtain “highly maintainable” and none “poor”.		
R8: Development must use regression test allowing full re-testing in 12 hours.		
Program complexity requirements R9: The cyclomatic complexity of code may not exceed 7. No method in any object may exceed 200 lines of code.		
Product feature requirements R10: Product shall log all actions and provide remote diagnostic functions.		
R11: Product shall provide facilities for tracing any database field to places where it is used.		

From: Soren Lauesen: Software Requirements

Fig 6.3B Planguage version of target etc.

Forecast speed [Tag]: How quickly the system completes a forecast report [Gist]

Scale: average number of seconds from pushing button, to report appearing.

Meter: Measured 10 times by a stopwatch during busy hours in hotel reception.

Must: 8 minutes, because the competitive system does it this fast.

Plan: ____ (supplier, please specify).

Wish: 2 minutes.

Past: Done as batch job taking about an hour.

Overview of styles for specifying functional requirements (Swedish terminology)

Datakravstilar:
- ✓ Datamodell (=E/R-diagr.)
- ✓ Dataordlista
- ✓ Reguljära uttryck
- ✓ Virtuella fönster

Funktionella kravstilar:
- ✓ Kontextdiagram
- ✓ Händelse- & Funktionslistor
- ✓ Produktegenskapskrav
- ✓ Skärmbilder & Prototyper
- ✓ Uppgiftsbeskrivningar
- ✓ Egenskaper från uppgifter
- ✓ Uppgifter och stöd
- ✓ (Levande) Scenarier
- ✓ Högnivåuppgifter
- ✓ Användningsfall
- ✓ Uppgifter med data
- ✓ Dataflödesdiagram
- ✓ Standardkrav
- ✓ Krav på utvecklingsprocessen

Funktionella detaljer:
- Enkla och sammansatta funktioner
- Tabeller & Beslutstabeller
- Textuella processbeskrivningar
- ✓ Tillståndsdiagram
- Övergångsmatriser
- Aktivitetsdiagram
- ✓ Klassdiagram
- Samarbetsdiagram
- ✓ Sekvensdiagram

Speciella gränssnitt
- Rapporter
- Plattformskrav
- Produktintegration
- Tekniska gränssnitt

Special interfaces Summary

- **Platform requirements**
  - Requirements on what the product shall run on now and in the future
  - Dealing with existing and planned platforms
  - Can be very complex and technically detailed depending on the product and contracting situation

- **Technical interfaces**
  - Requirements on interactions with other systems
  - Many different ways to specify technical interfaces
  - Performance and capacity requirements can be very difficult to understand and validate
  - Prototype and test the communication early

Fig 5.3A   Who can integrate?

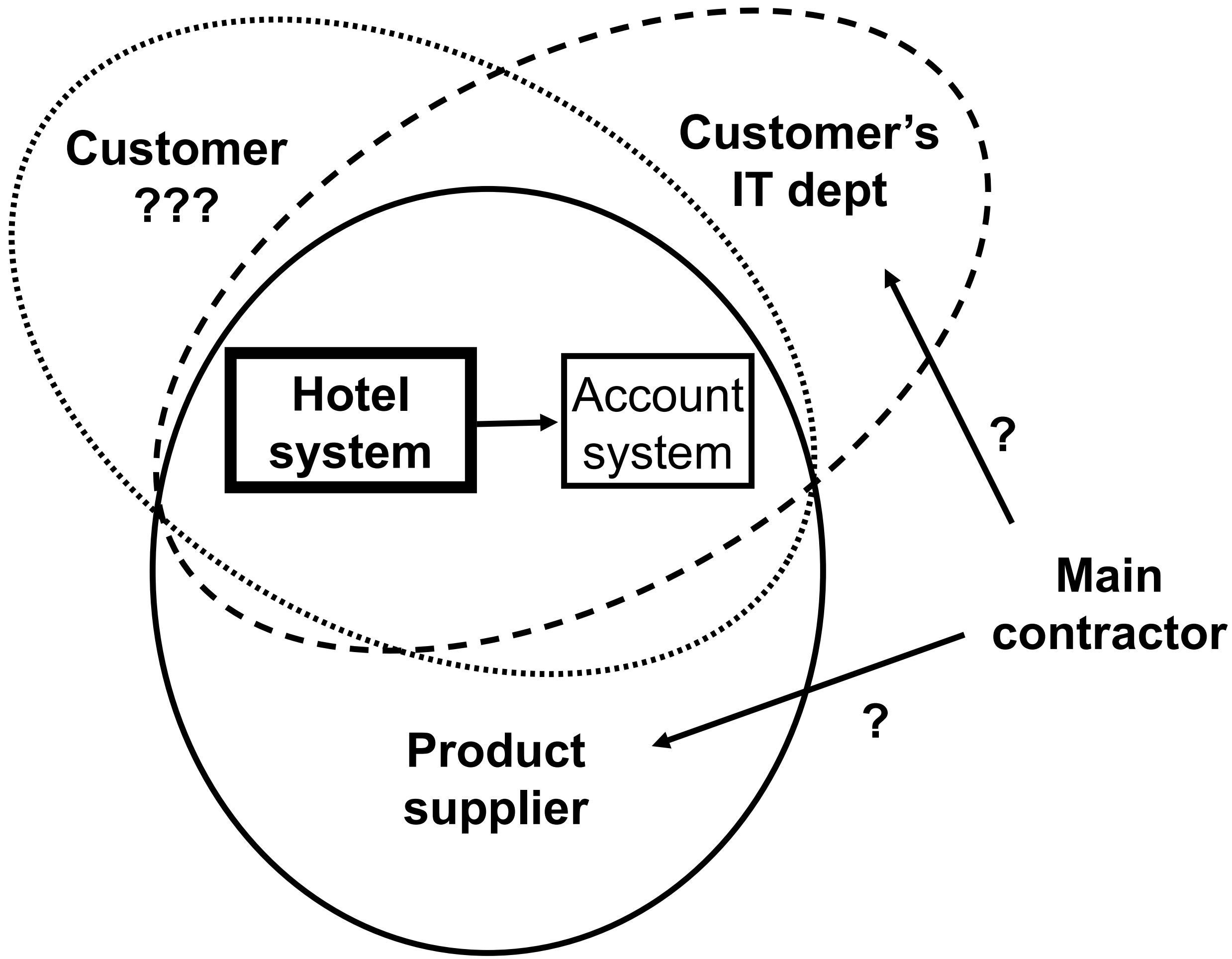
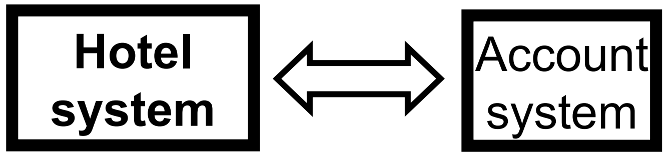


Fig 5.5 Technical interfaces



Communication channel

Physical channel:

File, TCP/IP, object calls . . .

Message formats:

Data descr, call params

Protocol:

State diagram, sequence diagram
formal data descr, SDL . . .

Semantics: about what?

E/R, tasks, activity diagrams

Verify early:
Functional prototypes

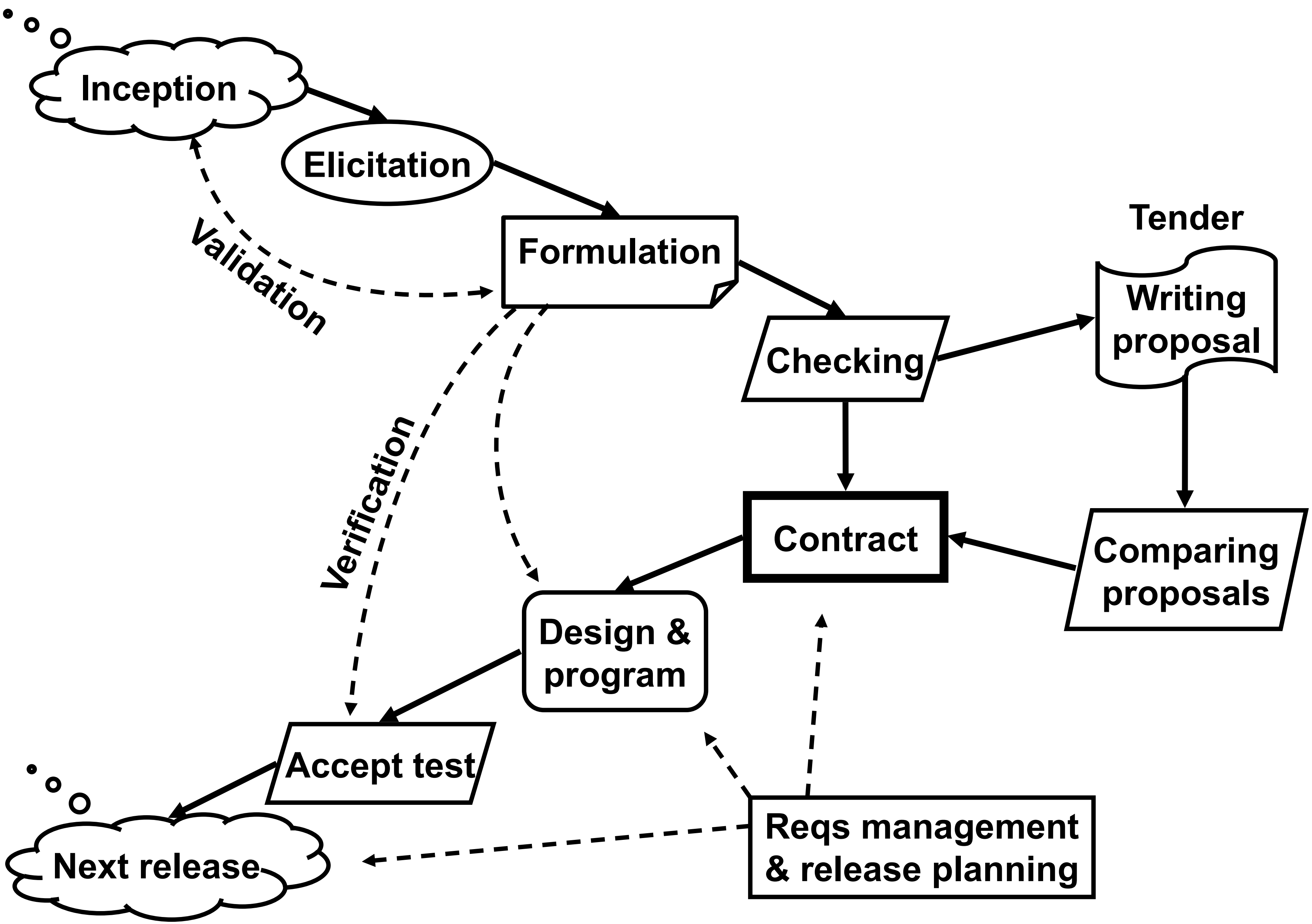
High risk requirements

```
Quality("performance") has Spec("The response time
   shall be at most 0.5 seconds on average when
   moving from one screen to another. The response
   time shall never be above 2 seconds.")
```

- Suppler A: We didn't notice any problems. Our response time is of that magnitude.
- Supplier B: We don't care. We'll find a way out later.
- Suppler C: We state as an assumption that 95% of the cases will be sufficient.
- Supplier D: We fulfill the requirement although it will be expensive.
- Supplier E: We tell the customer what it would cost and why, and then offer a reasonable alternative. Eventually, we offer the full solution as an expensive option.

[Lauesen: 7.5, p. 310]

Fig 7. Requirements in product life cycle



Spårbarhet (Traceability)

Ändringar i intressenternas behov, önskemål och tekniska antaganden kan kräva radikal omvärdering av kravens relevans.

Ansvar för kravuppfyllelse behöver kopplas till systemkomponenter så att ansvarighet och påverkansanalys kan utrönas

Källa

Framåt till krav

Bakåt från krav

Requirements document

Req A1: Printing

Dependecy

Req F8: Color

Framåt från krav

Bakåt till krav

```
package samples.simple;

public class Hello1
{
   public void printHello()
   {
      System.out.println( "Hello!" );
   }
}
```

Design/
Kod/Test

Hur intressenterna medverkat till kraven är viktig information vid validering.

Kravuppfyllelse behöver verifieras och design utan krav ("gold-plating") behöver undvikas.

Different methods to detect defects (reading techniques)

Ad hoc

- To your best ability (no specific guidelines)

Checklist

- A list of questions or check items direct the review

Perspective-based reading

- Different reviewers inspect from different perspectives and their findings are combined:
  e.g. user, designer, tester – perspectives,
  or from the perspective of different tasks/use cases

N-fold inspection

- N independent groups run inspection process in parallel

Different kinds of checks

- Content of spec
- Structure of spec
- Consistency of spec

Fig 9.2A   Contents check

Does the spec contain:

- Customer, sponsor, background
- Business goals + evidence of tracing

- Data requirements
  (database, i/o formats, comm. state, initialize)

- System boundaries & interfaces
- Domain-level reqts (events & tasks)
- Product-level reqts (events & features)
- Design-level reqts (prototype or comm. protocol)
- Specification of non-trivial functions
- Stress cases & special events & task failures

- Quality reqts (performance, usability, security . . .)

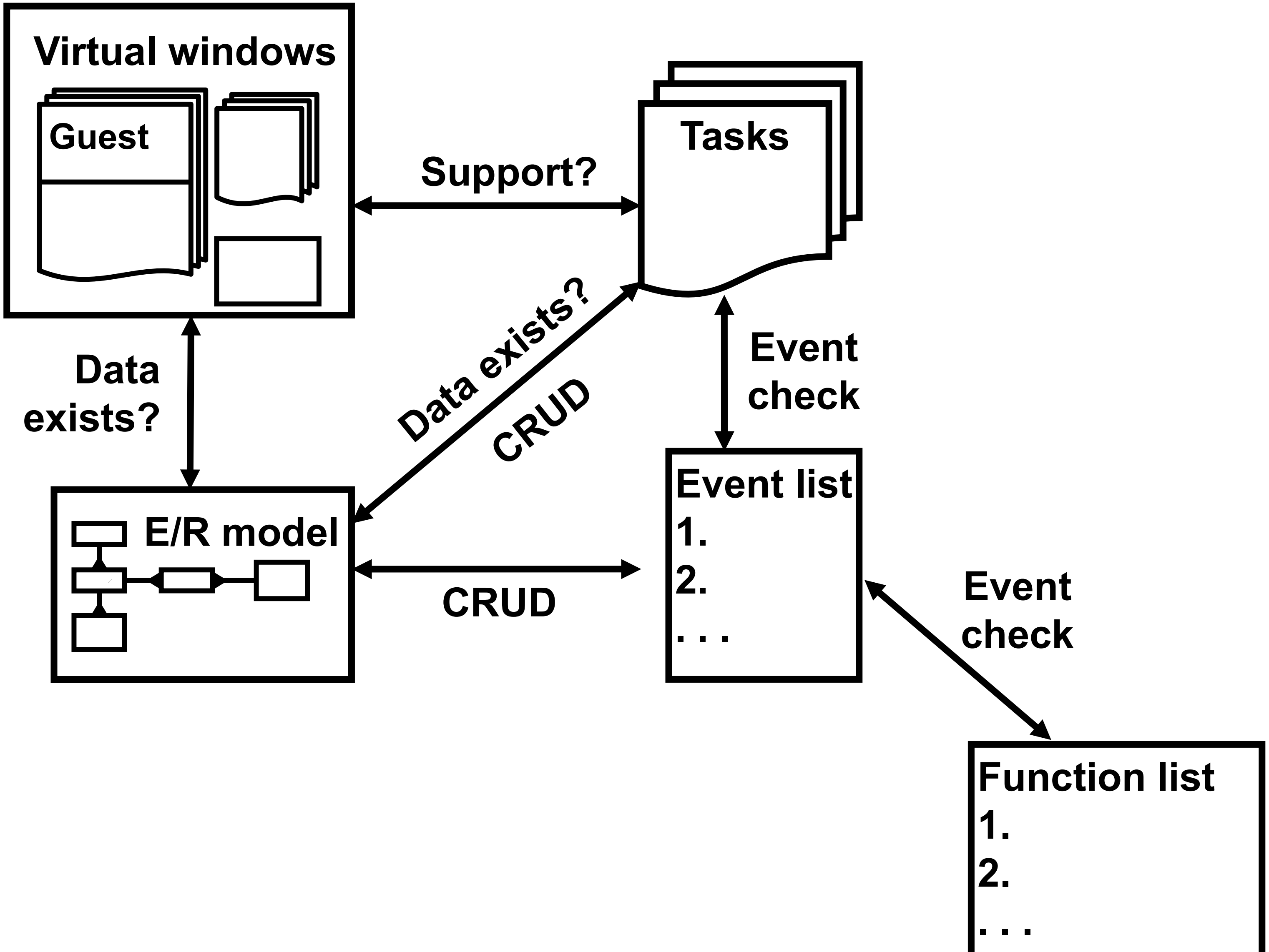
- Other deliverables (documentation, training . . .)
- Glossary (definition of domain terms . . .)

Fig 9.2B   Structure check

Does the spec contain:

- ~~Number or~~ Id for each requirement
- Verifiable requirements
- Purpose of each requirement
- Examples of ways to meet requirement
- Plain-text explanation of diagrams, etc.
- Importance and stability for each requirement
- Cross refs rather than duplicate information
- Index
- An electronic version

Fig 9.2C Consistency checks



From: Soren Lauesen: Software Requirements

Fig 9.2D CRUD+O matrix

Create, Read, Update, Delete + Overview

Task \ Entity	Guest	Stay	Room	RoomState	Service	ServiceType
Book	C U O	C	O	U O		
CheckinBooked	RU	U O	O	U O		
CheckinNonbkd	C U O	C	O	U O		
Checkout	U	U O	R	U		
ChangeRoom	R	R	O	U O		
RecordService			O		C	R
PriceChange			C UDO			C UDO
Missing?	D	D		C?UD?	UD	

SLUT+Ö
Skapa
Läsa
Uppdatera
Ta bort
Översikt

From: Soren Lauesen: Software Requirements

Fig 9.3 Checks against surroundings

Reviews

Review:
Developers and customer review all parts.

Goal-means analysis:
Goals and critical issues covered?
Requirements justified?

Risk assessment:
Customer assesses his risk.
Developers assess their risk.
High-risk areas improved.

Tests

Simulation and walk-through
Follow task descriptions. Correct?
Supported?

Prototype test (experiment with prototypes):
Requirements meaningful and realistic?
Prototype used as requirement?

Pilot test (install and operate parts of system):
Cost/benefit?
Requirements meaningful and realistic?

Fig 9.4(A) Check list

Project:	Noise Source Location, NSL vers. X	**Date, who:** 99-03-15, JPV
Contents check	**Observations - found & missing**	**Problem?**
Customer & sponsor	Missing, OK	
. . .		
Data: Database contents	Class model as intermediate work product	
. . .		
Initial data & states	Missing	Seems innocent, but caused many problems particularly when screen windows were opened.
Functional reqs: Limits & interfaces		
Product-level events and functions	Mostly as features	
. . .		
Special cases: Stress cases		
Power failure, HW failure, config.	Missing	**Problem.** Front-end caused many problems

From: Soren Lauesen: Software Requirements

Project:	Noise Source Location, NSL vers. X	**Date, who:** 99-03-15, JPV
Contents check (2)	**Observations - found & missing**	**Problem?**
Quality reqs: Performance	Missing, also in parts not shown here.	**Problem.** Response time became important.
Capacity, accuracy	Missing, also in parts not shown here.	**Problem.** Data volume, etc. became important.
Usability	Missing	Would have been useful
Interoperability	Missing	External dataformats, robot role, etc. caused problems
. . .		
Other deliverables: Documentation	Missing	Unimportant. Company standards exist.
. . .		

Structure check	**Observations - found & missing**	**Problem?**
ID for each req.	OK	
Purpose of each requirement	Good. Domain described.	

Consistency checks	**Observations - found & missing**	**Problem?**
CRUD check: Create, read, update, delete all data?	Have been made	

Tests	**Observations - found & missing**	**Problem?**
Prototype test	Not done, nor during development.	**Should have been done**. Caused many problems later.

Fig 9.1 Quality criteria for a specification

Classic: A good requirement spec is:

Correct
Each requirement reflects a need.
Complete
All necessary requirements included.
Unambiguous
All parties agree on meaning.
Consistent
All parts match, e.g. E/R and event list.
Ranked for importance and stability
Priority and expected changes per requirement.
Modifiable
Easy to change, maintaining consistency.
Verifiable
Possible to see whether requirement is met.
Traceable
To goals/purposes, to design/code.

Additional:

Traceable from goals to requirements.
Understandable by customer and developer.

Korrekt
Fullständig
Otvetydig
Motsägelsefri
Rankad
Modifierbar
Verifierbar
Spårbar bakåt/framåt

Begriplig

Designoberoende
Motiverad
Koncis
Välorganiserad
...

Paper [AGRE]

Agile Requirements Engineering Practices:
An Empirical Study

by Balasubramaniam Ramesh and Lan Cao

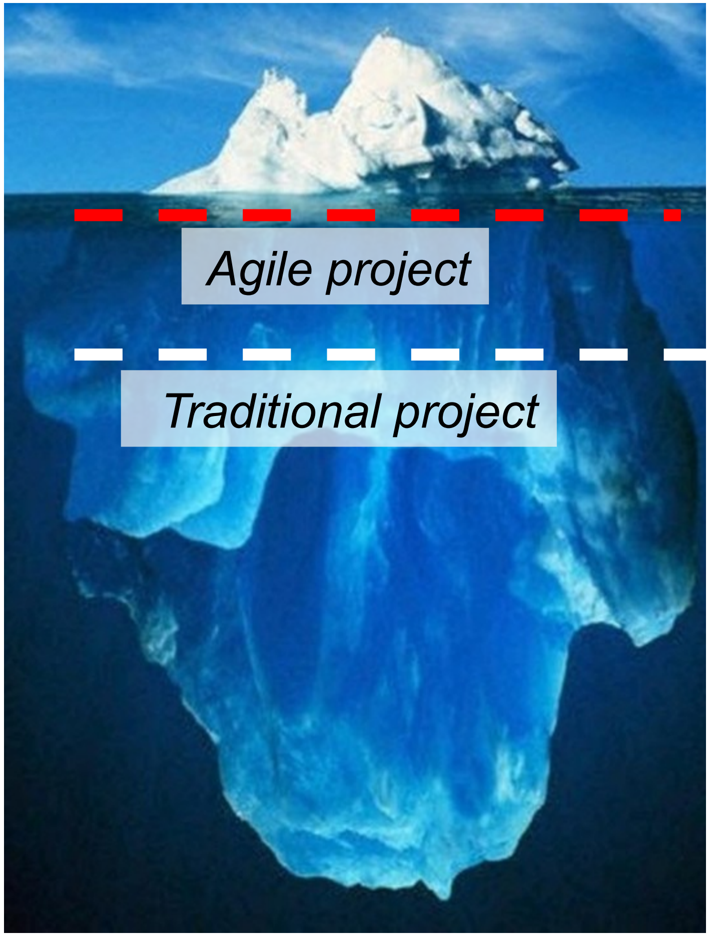
In: IEEE Software, pp. 60-67, January/February 2008

RE in Agile Projects [AGRE]

Practices

- *Iterative RE*: Gradual detailing
- Work order
  - *Extreme prioritization*: Just-in-time
  - *Constant planning*
- Integrated RE:
  Dev roles more involved in RE
  - *Face-to-face communication*
  - *Reviews & tests*
  - *Prototyping*
  - *Test-driven development*

Level of detail at dev start



Agile RE practices in 16 companies

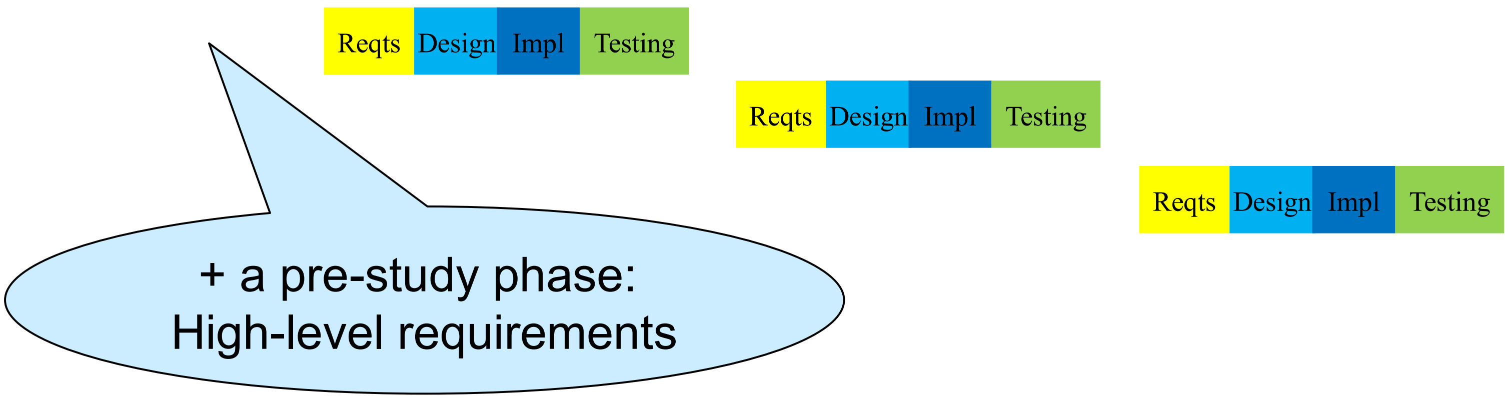
Adoption level	Practice: Face-to-face communication	Iterative RE	Extreme prioritization	Constant planning	Prototyping	Test-driven development	Reviews & tests
High	8	9	10	8	8	5	11
Medium	8	5	6	6	3	1	4
Low	0	2	0	2	0	0	1
None	0	0	0	0	5	10	0

Organization pseudonym	Industry and products
Enco	Energy and communications. Offers forecasting tools.
HealthCo	Healthcare and utilities. Offers an online service to help customers select health insurance and utility services.
Venture	Across industries. Helps brick-and-mortar companies develop a Web presence.
Entertain	Film and television industry. Offers high-tech indexing and search tools online.
HuCap	Administration. Carries out human-resource administration for other companies online.
TravelAssist	Transport and tourist industry. Offers online services.
ManageRisk	Across several industries. Offers insurance online.
Transport	Transportation and logistics industry. Offers services online.

Transport	Transportation and logistics industry. Offers services online.
ServeIT	Consulting and services. We studied the part of the firm that offers consulting services for business-to-business communication.
HealthInfo	Healthcare information systems. Offers information systems solutions to hospitals, physicians' offices, and home healthcare providers.
SecurityInfo	Security software. Offers software for Internet security.
AgileConsult	Software consulting. Offers consulting services on agile software development.
EbizCo	Packaged software development. Offers e-business connections and transactions.
FinCo	Online financial-transaction support. Offers online payments.
NetCo	Network software consulting. Offers services on developing network systems and architectures.
BankSoft	Banking information systems. Offers software that handles financial transactions.

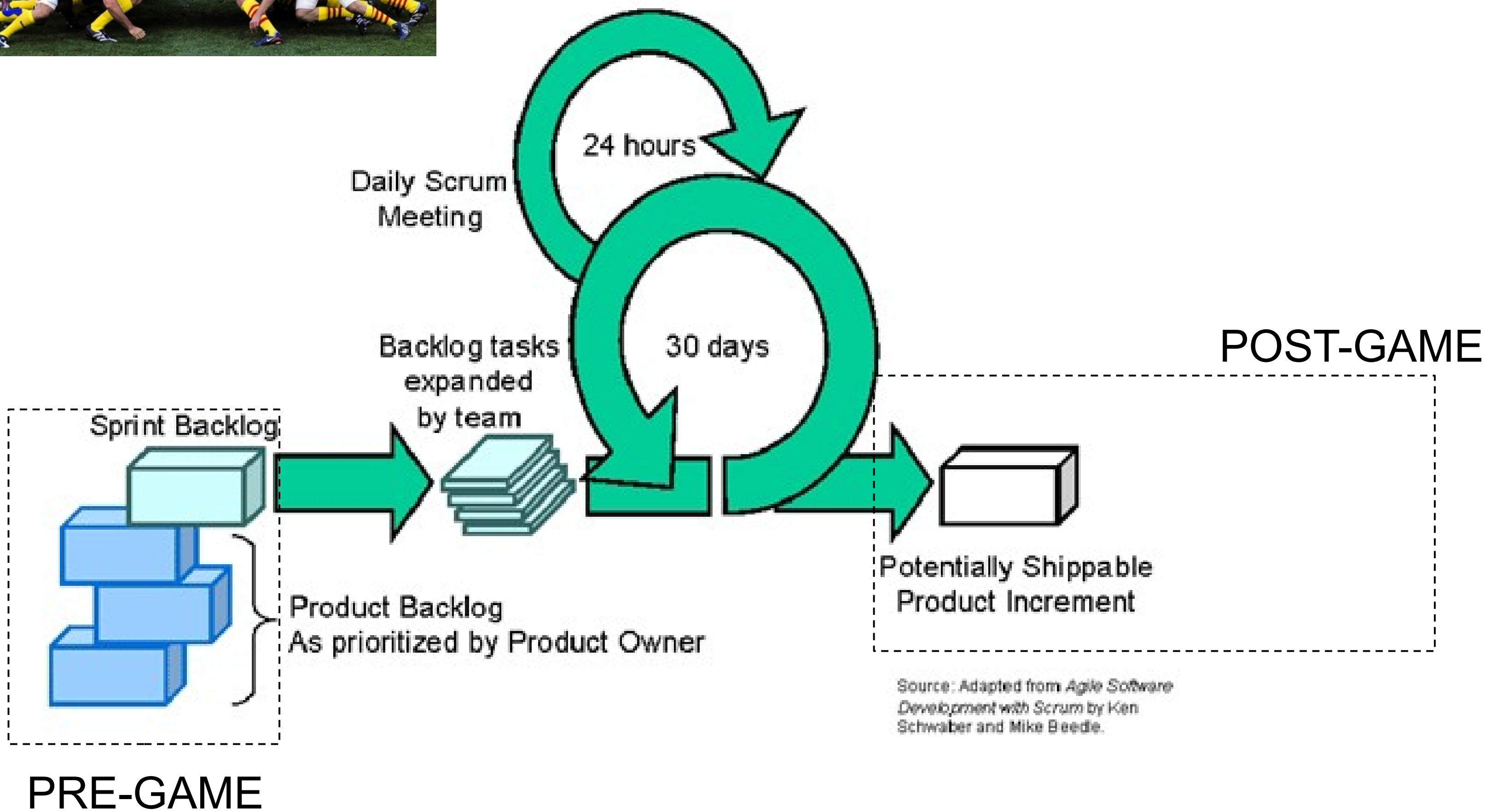
Traditional Development Process

Agile Development Process – Integrated RE



- Same activities, different sizing and timing
  - → Different principles and management approach
  - → Different people detailing requirements
  - → Different documentation formats

Scrum sprints - Time boxed iterations



- Requirements INTEGRATED in backlog, test cases, design docs etc

Continuous Feedback & Transparency

Business, Management and Development roles involved in

- Sprint planning meeting
- Daily stand-up meetings
- End-of-sprint demo
- Sprint retrospective meetings

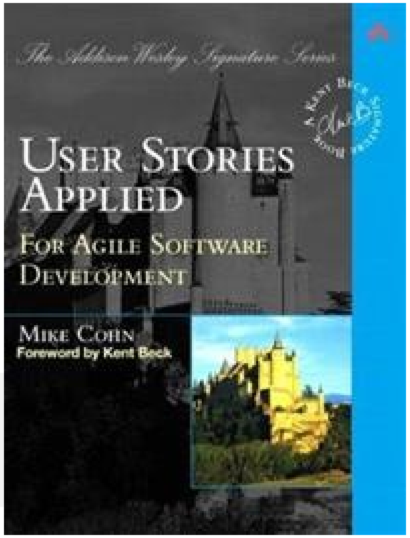
User story & Acceptance Criteria

User story:

As a passenger, I can cancel a flight reservation

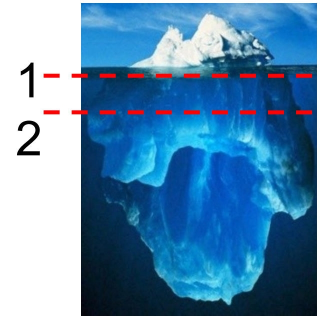
Acceptance criteria / test cases

- Verify that a premium member can cancel the same day without a fee
- Verify that a non-premium member is charged 10% for a same-day cancellation
- Verify that an email confirmation is sent
- Verify that the hotel is notified of any cancellation



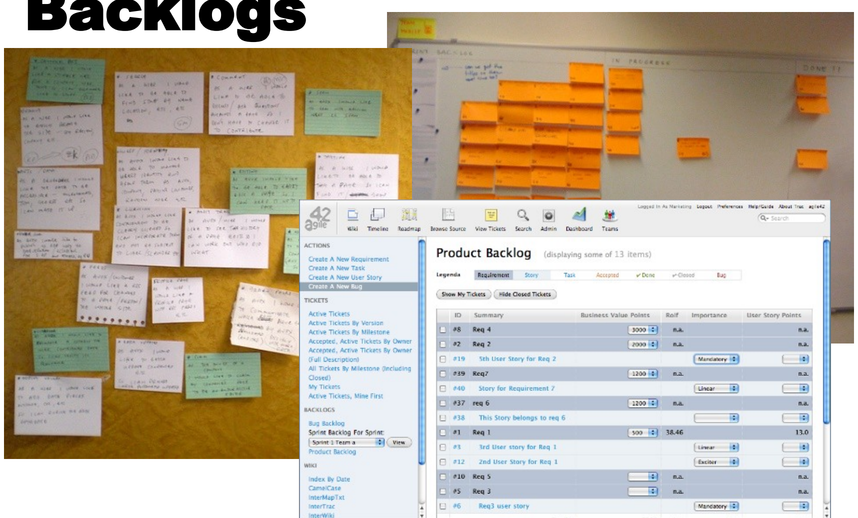
Cohn, Mike. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

Specification of user stories



1. Product Owner/Customer **defines** & **prioritizes** Epics/User stories in **product backlog**
2. Team **defines** details for each user story in **sprint backlog**
    1. Tasks
    2. Acceptance criteria & test cases

Backlogs



Story cards

Test Cases as Requirements in Agile practice

Benefits	Challenges
Elicitation and validation	
EB1 Cross-functional communication	EC1 Good Customer-Developer relationship
EB2 Align goals & perspectives between roles	EC2 Active customer involvement
EB3 Address barrier of specifying solutions	EC3 Sufficient technical and RE competence
EB4 Creativity supported by high-level of requirements	EC4 Complex requirements, e.g. quality requirements
Verification	
VB1 Supports regression testing	VC1 Varying (biased) results for manual tests
VB2 Increased requirements quality	VC2 Ensuring correct requirements info to test
VB3 Test coverage / RET alignment	VC3 Quality requirements
Tracing	
TB1 Implicit Requirements - test case tracing	TC1 Tool integration
Managing changes	
MB1 Communication of changes	MC1 Locating impacted requirements
MB2 Requirement are kept updated	MC2 Missing requirement context
MB3 Maintaining RET alignment	MC3 Multiple products in one product line
MB4 Detecting impact of changes	
Customer agreement/contractual	
CB1 Facilitate resolving conflicting views	CC1 Use-case related structuring
CB2 Support certification of compliance	

Table 7 in ATCR.pdf (optional paper in zip not included in exam)]

Bjarnason, Unterkalmsteiner, Borg, & Engström (2016). *A multi-case study of agile requirements engineering and the use of test cases as requirements.* Information and Software Technology, 77, 61-79.

Face-to-face communication

Direct communication between customer and development

- Techniques
  - User Stories == high-level requirements spec
  - Complemented by other artifacts, e.g. "backlog"
- Prerequisites
  - Active involvement of (knowledgeable) customers

Customers can steer project

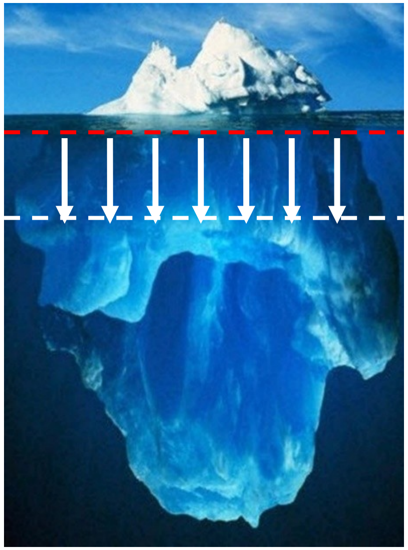
Avoids time-consuming documentation

Risk of inadequate requirements

On-site customer rep is challenging

Handling more than one customer

Relies on trust rather than agreed requirements



Iterative RE

Requirements **emerge during development** based on **initial high-level requirements**

- Techniques
  - Requirements analysis and detailing for each development cycle
  - Requirements intertwined with design

Good customer relationship

Clearer and understandable requirements due to direct customer interaction

Accurate cost and scheduling of project

Neglect of quality requirements

Lack of documentation beyond dev team

Extreme Prioritization & Constant Planning

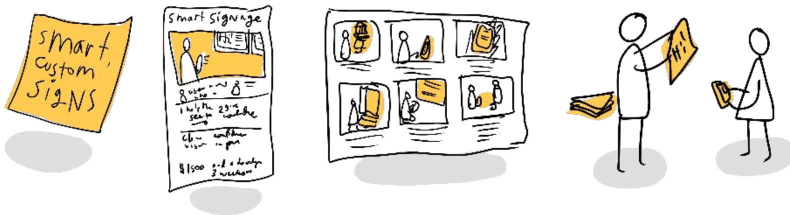
Aim to deliver **most valuable features first**

Responsive to changes in customer demands

- Techniques
  - Work on most valuable features first
  - Continuously revise prioritisation & planning (for each iteration)
  - Constant feedback from customer

Customer provides business prio
Re-prioritization supported by dev process
Early validation minimizes need & cost for major changes

Other criteria suffer, e.g. quality
Instability in dev work
Inadequate architecture and increased costs
Refactoring requires time and experience



Prototyping & Reviews & Acc Test

Communicate through prototypes and frequent review meetings

Involves customers, developers and testers

Requirements **validation** and **refinement** through feedback

- Techniques
  - End-of-sprint sign-off meeting

Efficient validation

Assess project status

Trust: Customer, Mgmt

Early problem identification

Risks with evolving prototypes in production

Unrealistic expections regarding leadtime

Weak formal validation, consistency checks

Dev of acc tests require access to customers

Test-Driven Development

Developers **create test before writing new code**

Tests specify expected behaviour of code

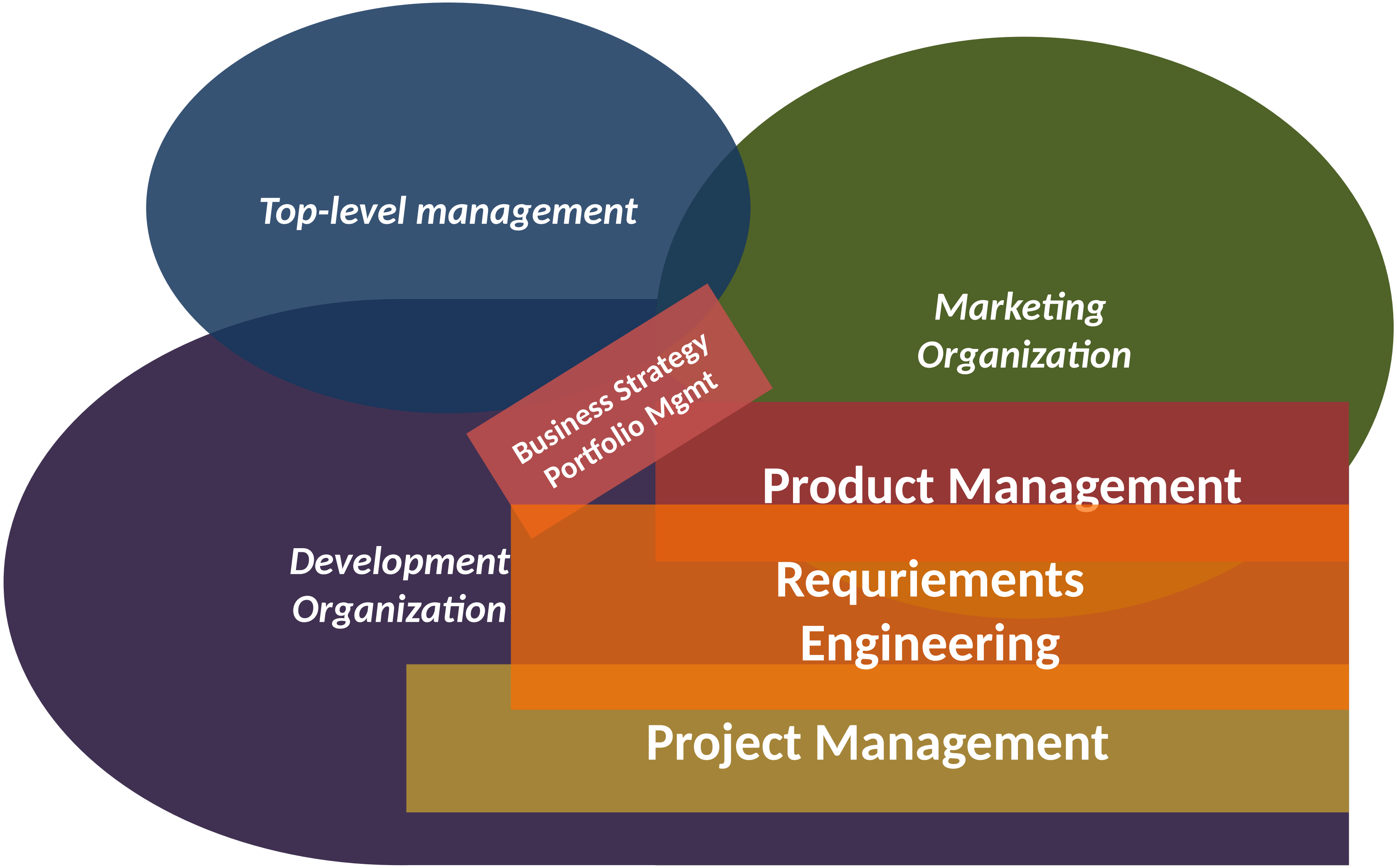
Tests capture complete requirements

Traces to production code facility reqts changes

Requires competence in testing, requirements understanding and customer collaboration

Most organizations fail to implement this practice

RE vs. Product & Project Mgmt



Decisions outcomes in MDRE

		Decision	
		Selected	*Rejected*
Requirements Quality	*alfa*	*A* Correct selection ratio	*B* Incorrect selection ratio
	beta	*C* Incorrect selection ratio	*D* Correct selection ratio

Product Quality: $Q_p = A/(A+C)$

Decision Quality: $Q_d = (A+D)/(A+B+C+D)$

[MDRE]

Finding the golden grains despite uncertain cost-value estimates

Figure 13.1 (a) Cost-Value Diagram with alfa-requirements (filled) and beta-requirements (empty).

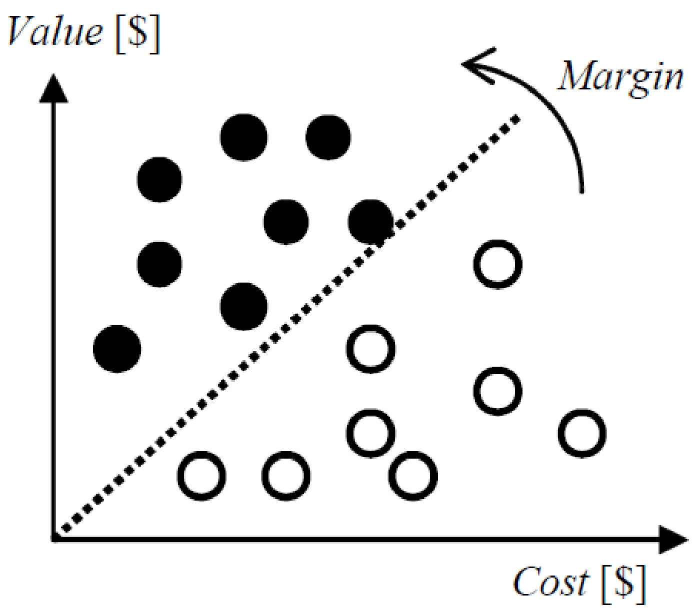
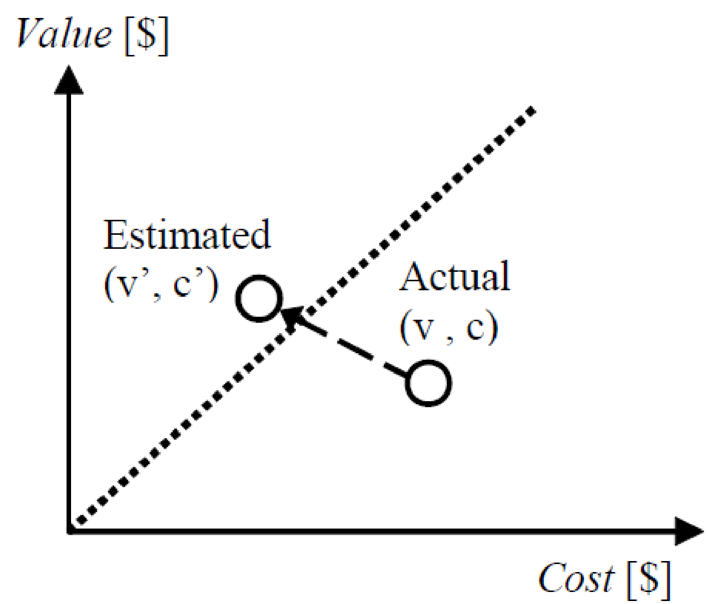


Figure 13.1 (b) Estimated values are differing from actual values causing wrong selection decision.



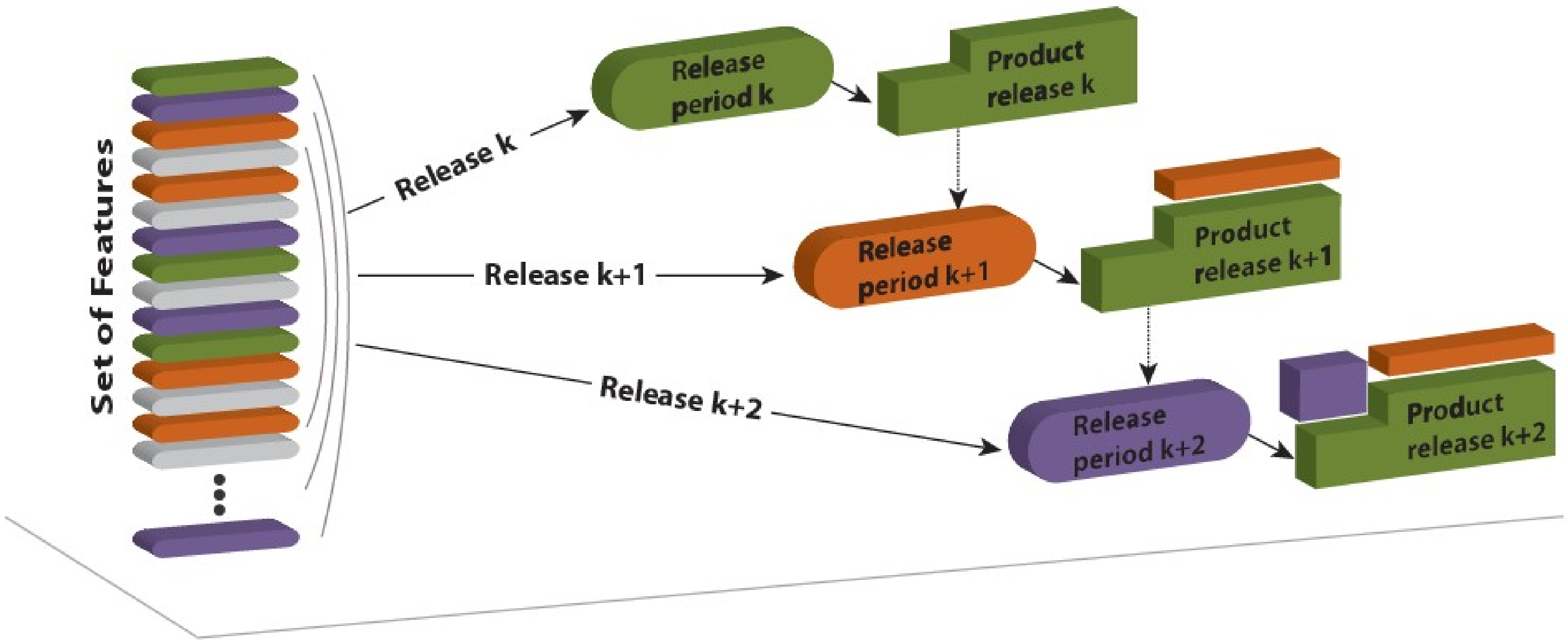
[MDRE]

Release Planning

Paper [RP] in compendium

- The art and science of software release planning
- Ruhe, G., & Saliu, M. O.
- IEEE software, 22(6), 47-53. 2005

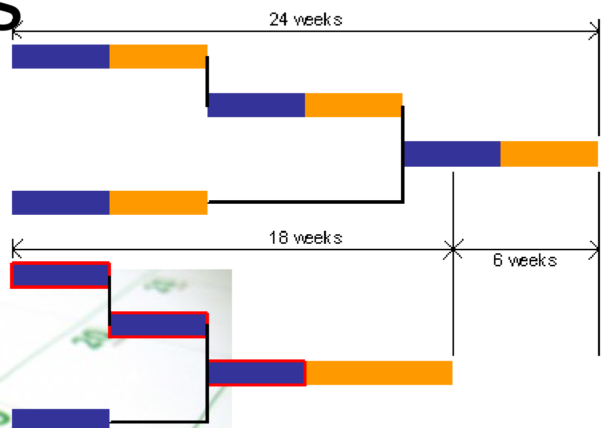
What is Release Planning?



[RP]

Release Planning involves...

- ...prioritization + scheduling under various constraints, e.g., resource and precedence constraints



[RP]

Example planning parameters

- Requirements priorities (from prioritization)
- Available resources
- Delivery time
- Requirements dependencies
  - Precedence, Coupling, Excludes
- System architecture
- Dependencies to the code base

[RP]

What is a good release plan?

- A good release plan should
  - Provide maximum business value by
    - offering the best possible blend of features
    - in the right sequence of releases
  - satisfy the most important stakeholders involved
  - be feasible with available resources, and
  - take dependencies among features into account

[RP]

Simplistic Release Planning

- Informal process
- Unclear rationale behind decisions
- No systematic management of dependencies
- Simplistic greedy allocation is no good
- A zillion possibilities already with 20 features and 3 releases:

  $4^{20} > 1.000.000.000.000 = 10^{12}$ possibilities

[RP]


1.000.000.000.000
RELEASE PLANS
imgflip.com

Why greedy allocation is bad

https://gist.github.com/bjornregnell/80897de5b109f36c1b7ae29f43e4aa7b

```scala
val m = Model(
  Feature("a") has (Benefit(90), Cost(100)),
  Feature("b") has (Benefit(85), Cost(90)),
  Feature("c") has (Benefit(80), Cost(25)),
  Feature("d") has (Benefit(75), Cost(23)),
  Feature("e") has (Benefit(70), Cost(22)),
  Feature("f") has (Benefit(65), Cost(20)),
  Feature("g") has (Benefit(60), Cost(10)),
  Feature("h") has (Benefit(55), Cost(30)),
  Feature("i") has (Benefit(50), Cost(30)),
  Feature("j") has (Benefit(45), Cost(30)),
  Release("r1") has Capacity(100),
  Release("r2") has Capacity(90))
```

```scala
def plan(input: Model,
      pickNext: (Model,Release)=>Option[Feature]): Model = {
  var result = input
  releases(input).foreach { r =>
    var next = pickNext(result, r)
    while (next.isDefined) {
      result = allocate(result, next.get, r)
      next = pickNext(result, r)
    }
  }
  result
}

plan(m, random)
plan(m, greedy)
```

```scala
def features(m: Model): Vector[Feature] = m.tip.collect{case f: Feature => f}
def releases(m: Model): Vector[Release] = m.tip.collect{case r: Release => r}
def allocate(m: Model, f: Feature, r: Release): Model = m + (r has f)
def isAllocated(m: Model, f: Feature): Boolean = releases(m).exists(r => (m/r).contains(f))
def allocatedCost(m: Model, r: Release): Int = (m/r).entities.collect{case f => m/f/Cost}.sum
def isRoom(m: Model, f: Feature, r: Release): Boolean = m/r/Capacity >= allocatedCost(m,r) + m/f/Cost
def featuresInGreedyOrder(m: Model): Vector[Feature] = features(m).sortBy(f => m/f/Benefit).reverse

def random(m: Model, r: Release): Option[Feature] = scala.util.Random.shuffle(features(m)).
  filter(f => !isAllocated(m,f) && isRoom(m,f,r)).headOption

def greedy(m: Model, r: Release): Option[Feature] =
  featuresInGreedyOrder(m).find(f => !isAllocated(m,f) && isRoom(m,f,r))
```

Optimal vs. Greedy

```
val optimal = Model(
  Feature("a") has (Benefit(90), Cost(100)),
  Feature("b") has (Benefit(85), Cost(90)),
  Feature("c") has (Benefit(80), Cost(25)),
  Feature("d") has (Benefit(75), Cost(23)),
  Feature("e") has (Benefit(70), Cost(22)),
  Feature("f") has (Benefit(65), Cost(20)),
  Feature("g") has (Benefit(60), Cost(10)),
  Feature("h") has (Benefit(55), Cost(30)),
  Feature("i") has (Benefit(50), Cost(30)),
  Feature("j") has (Benefit(45), Cost(30)),
  Release("r1") has (Capacity(100), Feature("c"), Feature("d"), Feature("e"), Feature("f"), Feature("g")),
  Release("r2") has (Capacity(90), Feature("h"), Feature("i"), Feature("j")))
```

```
def sumAllocatedBenefit(m: Model) =
  releases(m).map(r => (m/r).collect{case f: Feature => m/f/Benefit}.sum).sum

val beneftitOptimal = sumAllocatedBenefit(optimal)
val benefitGreedy   = sumAllocatedBenefit(plan(m,greedy))
val ratio = benefitGreedy.toDouble / beneftitOptimal
```