Software Engineering

3de BAC Informatica (Computer Science)
[Academic year 2023-2024]

© Prof. Serge Demeyer

(This slide deck is significantly revised compared to previous editions.
Extra - changed - new info is marked with variants of **New** **Revised**)

Universiteit Antwerpen

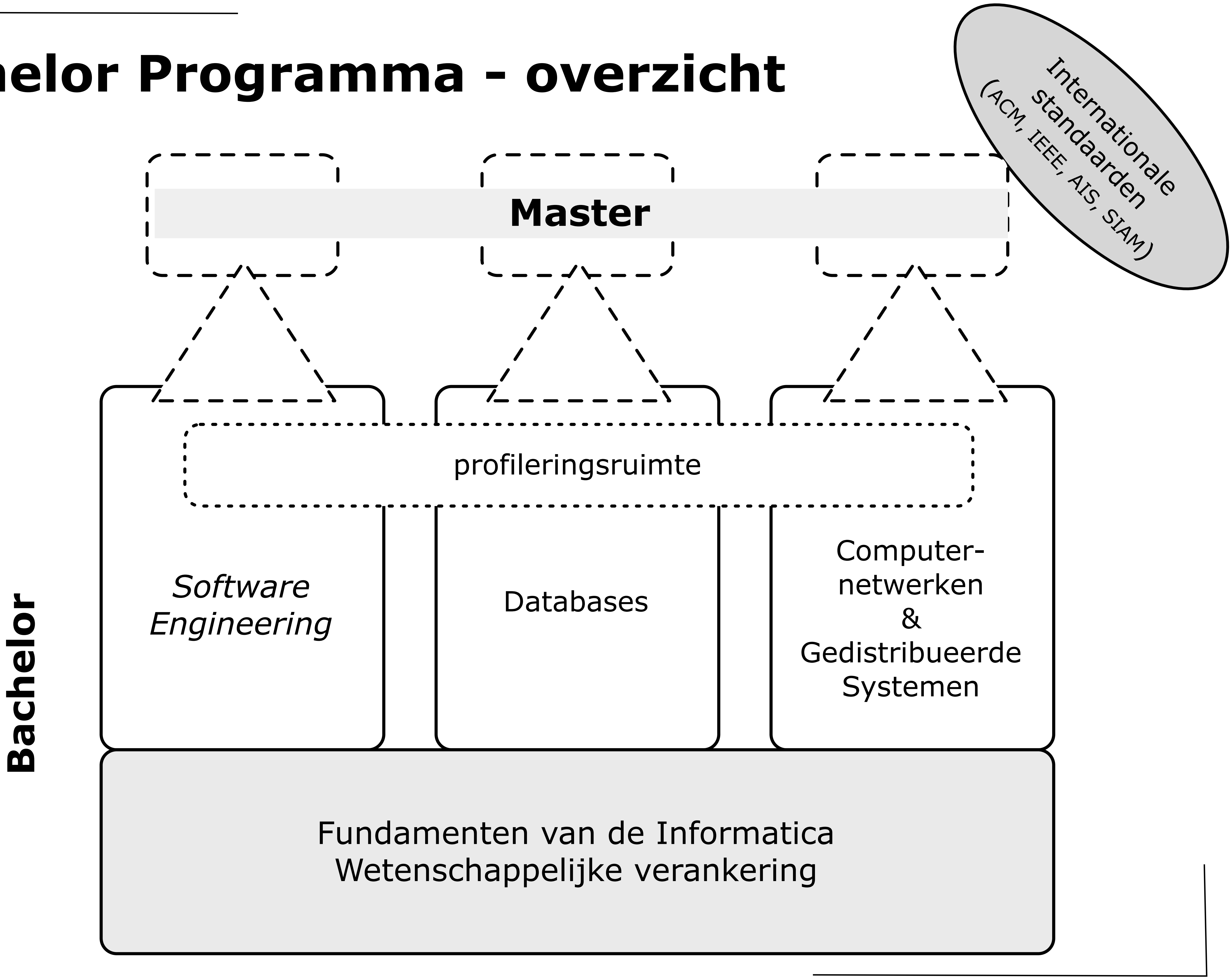
Opleiding Informatica — Doel en Ambitie

Doelstelling: Het doel van de opleiding Informatica aan de Universiteit Antwerpen is bekwame en wetenschappelijk gevormde informatici af te leveren. Na hun opleiding

- (a) zijn zij in staat zich de nieuwste technologische ontwikkelingen in de diverse deelgebieden eigen te maken,
- (b) kunnen zij deze waar nodig ook exploiteren binnen hun beroepscontext, en
- (c) kunnen zij zelf een originele bijdrage leveren tot de verdere evolutie van de informatica.

Ambitie: De onderwijscommissie Informatica van de Universiteit Antwerpen wil een unieke opleiding aanbieden gebaseerd op de aanbevelingen van gerenommeerde wetenschappelijke organisaties zoals ACM, IEEE en SIAM. Ze wenst dat haar alumni voldoende diepgang en flexibiliteit bezitten om gegeerd te zijn op de arbeidsmarkt en academische posities te bekleden in universiteiten en onderzoeksinstellingen met wereldfaam. Daartoe hanteert zij een transparant en activerend leerproces, conform de visie van de Universiteit Antwerpen rond “studentgecentreerd onderwijs”.

Bachelor Programma - overzicht



Internationale standaarden (ACM, IEEE, AIS, SIAM)

Master

profileringsruimte

Software Engineering

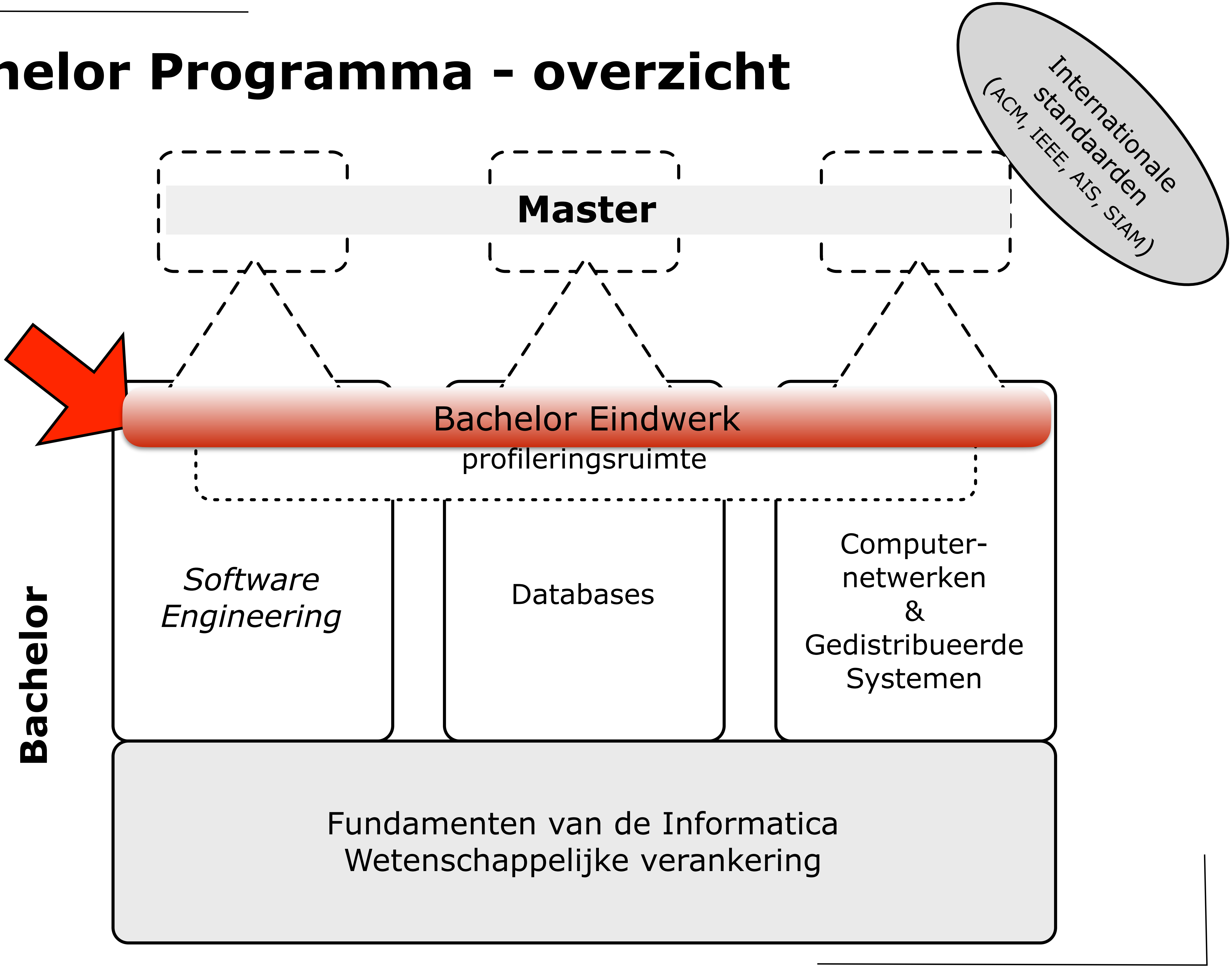
Databases

Computernetwerken & Gedistribueerde Systemen

Bachelor

Fundamenten van de Informatica
Wetenschappelijke verankering

Bachelor Programma - overzicht



Internationale standaarden (ACM, IEEE, AIS, SIAM)

Master

Bachelor Eindwerk

profileringsruimte

Software Engineering

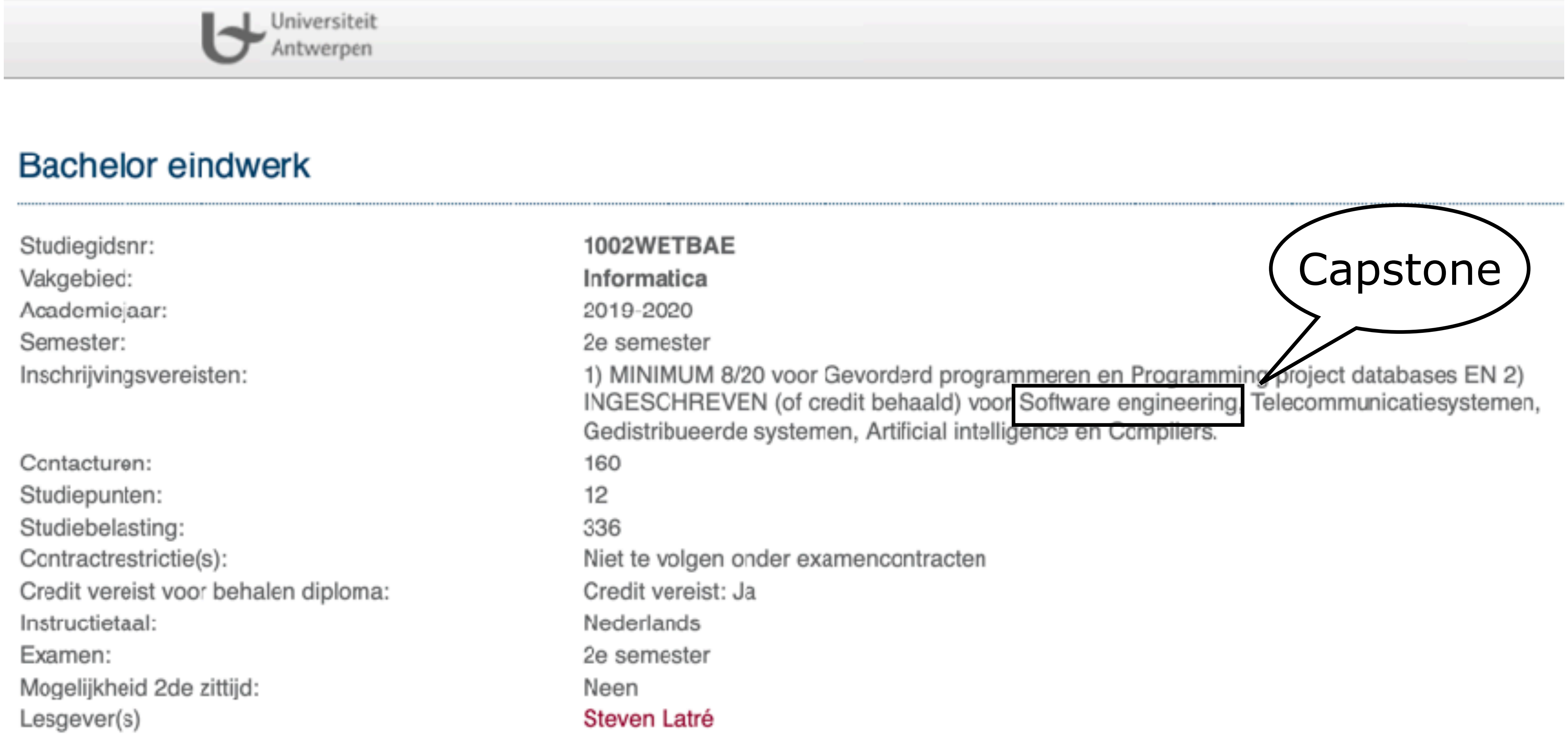
Databases

Computer-netwerken & Gedistribueerde Systemen

Fundamenten van de Informatica
Wetenschappelijke verankering

Bachelor

Bachelor Eindwerk



Universiteit
Antwerpen

Bachelor eindwerk

Studiegidsnr:	**1002WETBAE**
Vakgebied:	**Informatica**
Academiejaar:	2019-2020
Semester:	2e semester
Inschrijvingsvereisten:	1) MINIMUM 8/20 voor Gevorderd programmeren en Programming project databases EN 2) INGESCHREVEN (of credit behaald) voor Software engineering, Telecommunicatiesystemen, Gedistribueerde systemen, Artificial intelligence en Compilers.
Contacturen:	160
Studiepunten:	12
Studiebelasting:	336
Contractrestrictie(s):	Niet te volgen onder examencontracten
Credit vereist voor behalen diploma:	Credit vereist: Ja
Instructietaal:	Nederlands
Examen:	2e semester
Mogelijkheid 2de zittijd:	Neen
Lesgever(s)	Steven Latré

Capstone

Kerncompetenties Software Engineering

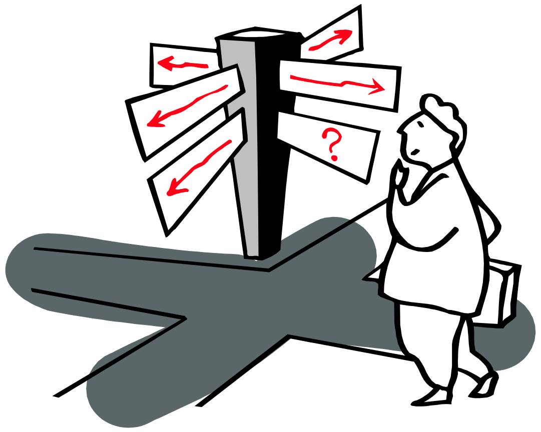
Bachelor = bekwame informaticus

Analyse en ontwerp voor kleinschalige software projecten	+
Implementatie van nieuwe softwaresystemen	+
Onderhoud van bestaande softwaresystemen	++
Implementatie en onderhoud van een databank	
Beheer van een lokaal netwerk	
Vakbekwaamheid	++
Maatschappij	+++
Communicatievaardigheden	+

Academische Bachelor = wetenschappelijke vorming

Wiskundige basis	
Formeel denken en abstraherend vermogen	+
Levenslang leren	++
Wetenschappelijke aanpak	+
Wetenschappelijke basis	
Autonoom en creatief functioneren	+

HOOFDSTUK 0 – Praktische Zaken



- Doel

  + Professionele Informaticus

  + Plaats in het Curriculum

    - Kerncompetenties

  + Beoordelingscriteria

  + Examen

  + Voorbeeldvragen

- Literatuur
- Inhoudstafel

Doel

- Programmaboekje
  + "Het doel bestaat erin om de student een brede basis te verschaffen in het bouwen van softwaresystemen die te complex zijn om door één persoon gerealiseerd te worden."

  + Dus
    - brede basis
      > véél technieken
    - complexe systemen
      > schaalbare technieken

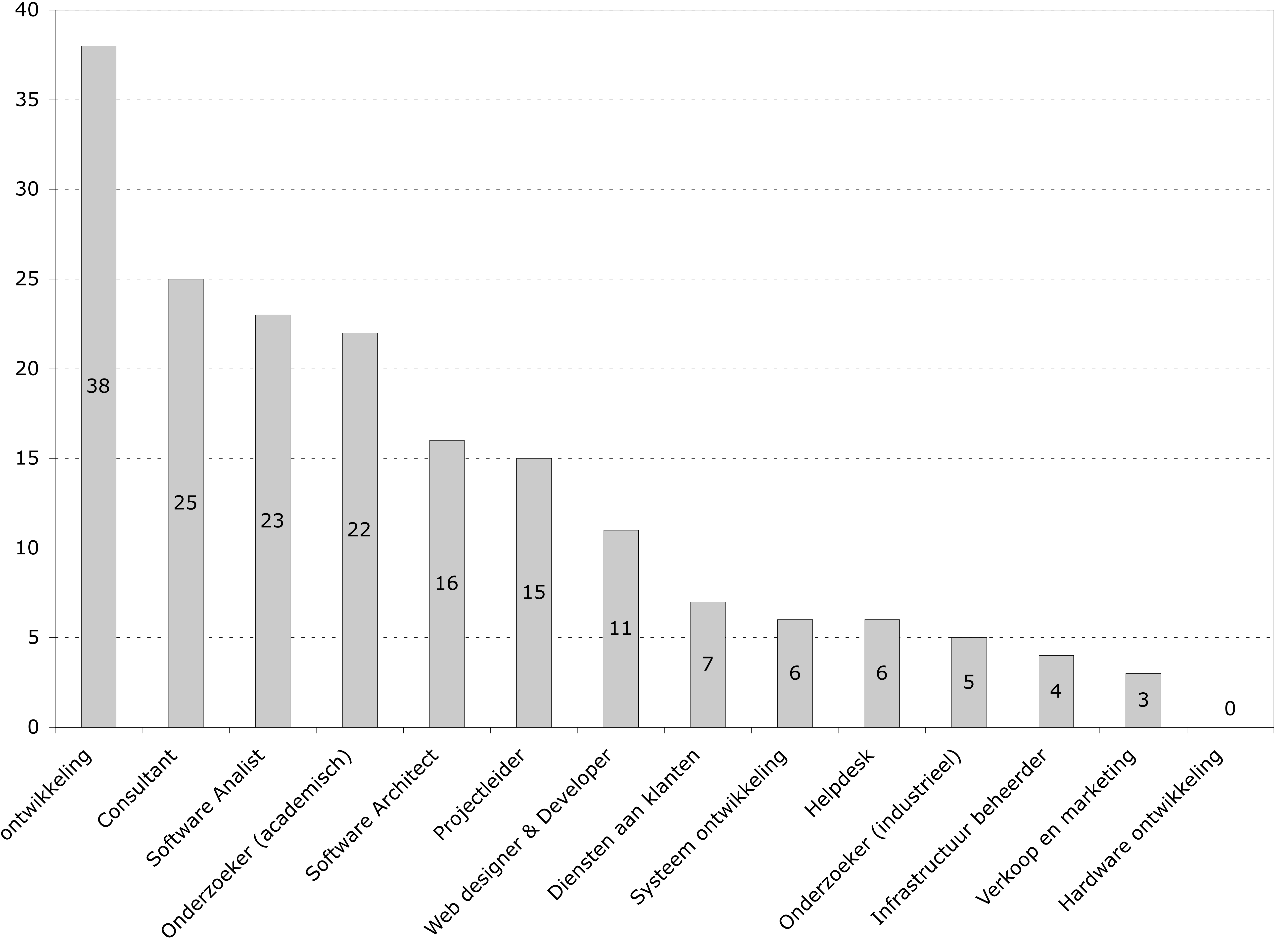
Professionele Informaticus

- diverse domeinen
  + traditionele “data processing”
    (banken, verzekeringen)
  + spitstechnologie
    (cloud, AI, cyber-physical)
  + telecommunicatie
    (netwerkbeheer, e-commerce)
  + ...
  + onderzoek

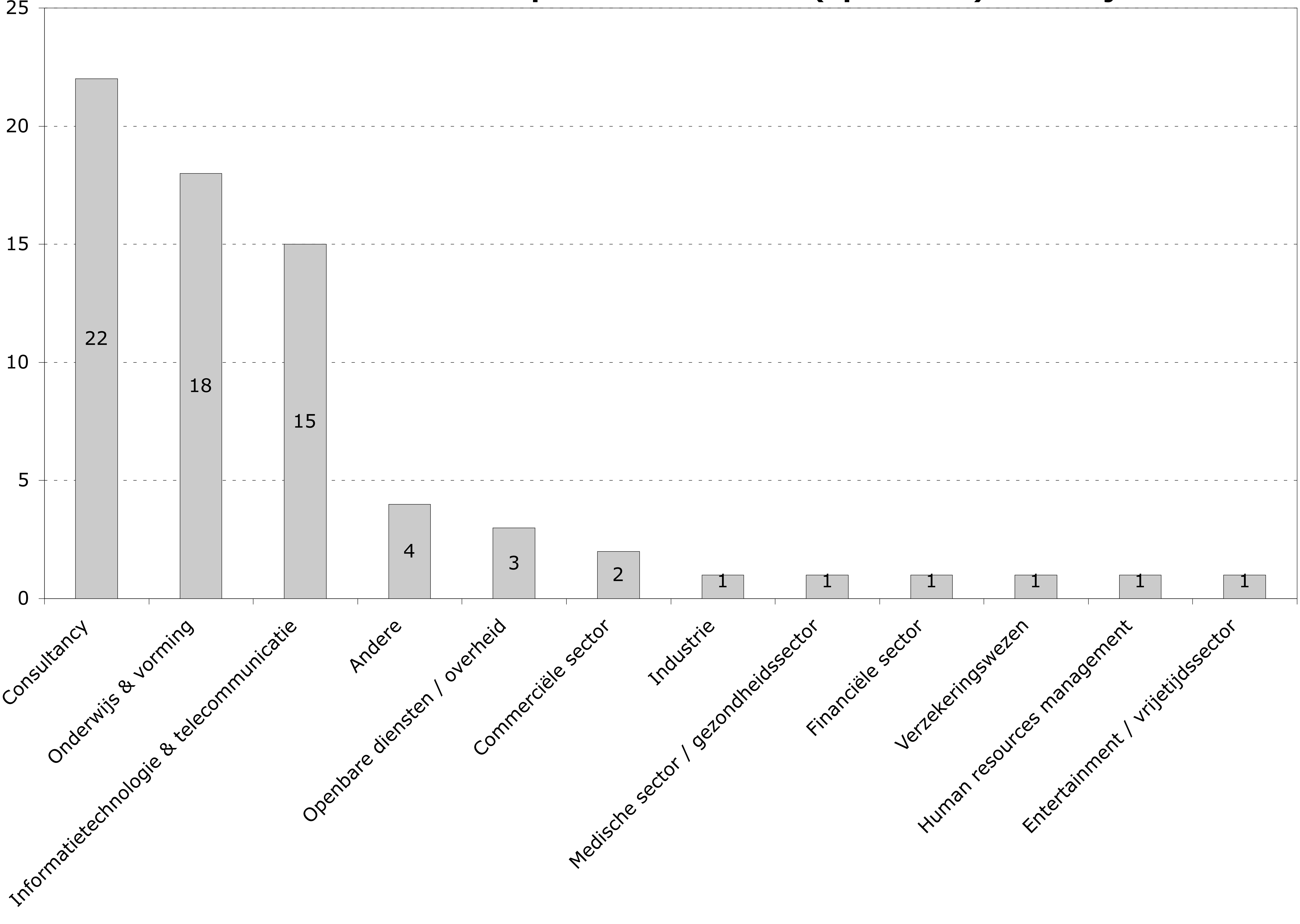
- zij verwachten
  + technische virtuozen
    - diverse specialiteiten
  + groepsspelers
    - sociale vaardigheden

Public Domain

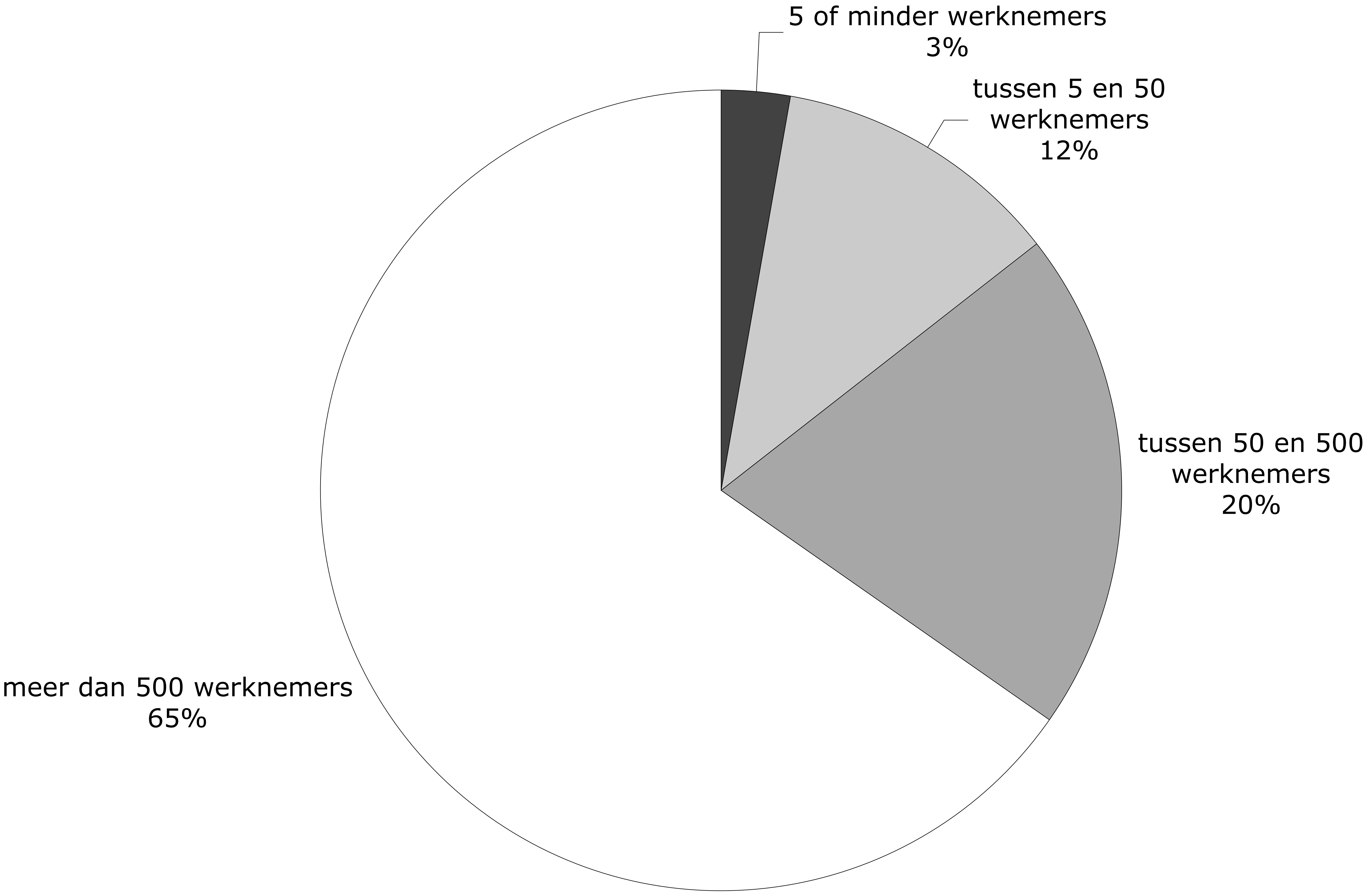
Enquête onder alumni (april 2008) — JobProfiel



Enquête onder alumni (april 2008) — Bedrijfssector



Enquête onder alumni (april 2008) — Bedrijfsgrootte



Job Profiles (Software Engineering)

Niew

Test Lead

Software Architect

SCRUM Master

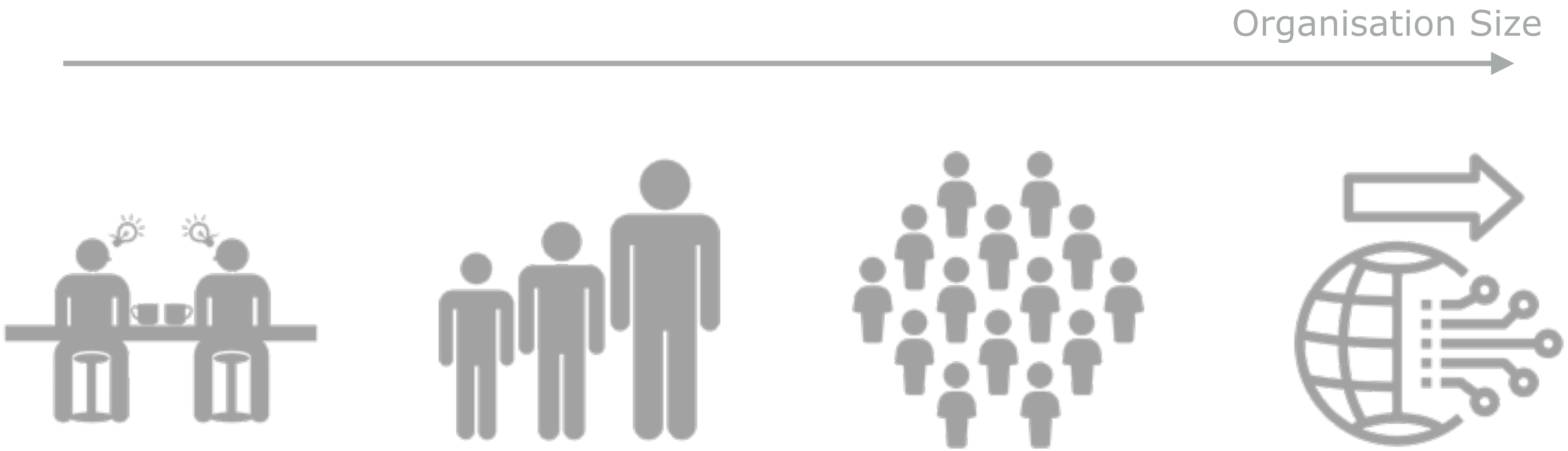
Project Manager

QA Engineer

Enterprise Architect

Team Lead

Chief Technology Officer (CTO)



Criteria - Selectie

- Accuraatheid
  + Een professionele software engineer werkt in groep en moet dus op een accurate manier kunnen communiceren met zijn collega's en eindgebruikers.
    * Juist gebruik van terminologie
    * Parate kennis definities

- Toepasbare kennis
  + Een professionele software engineer moet in staat zijn gekende technieken toe te passen in een variërende context.
    * "Know-how"
    * Oefeningen

SELECTIE

- Je *moet* dit kunnen demonstreren tijdens het examen om te slagen!

Criteria - Diversificatie

- Inzicht
  + Een professionele software engineer moet technische keuzes kunnen verantwoorden
    * "Know when"
    * Afwegingen maken

DIVERSIFICATIE:

- De mate waarin je dit kunt demonstreren tijdens het examen laat je toe je te onderscheiden van je collega's.

Tussentijdse Opdrachten

Herzien

- (Bijna) wekelijks
- Oefeningen: toepassing van de theorie van de week.
  - ⇒ Rapporteren via mondelinge presentatie (om de 2 à 3 weken)
- In groep: 3 personen.
- Kwalitatieve feedback ⇒ evolutie is belangrijk.

Evaluatie

- *Quotering volgens gewicht*
  - *30% - Mijlpaal 1 [Requirements + Architecture + Project Management]*
  - *50% - Mijlpaal 2 [Testing + Formal Specifications + State Charts]*
  - *20% - Mijlpaal 3 [Refactoring + Quality Control]*

Een resultaat > 12 op 20 is noodzakelijk om te slagen voor dit vak!

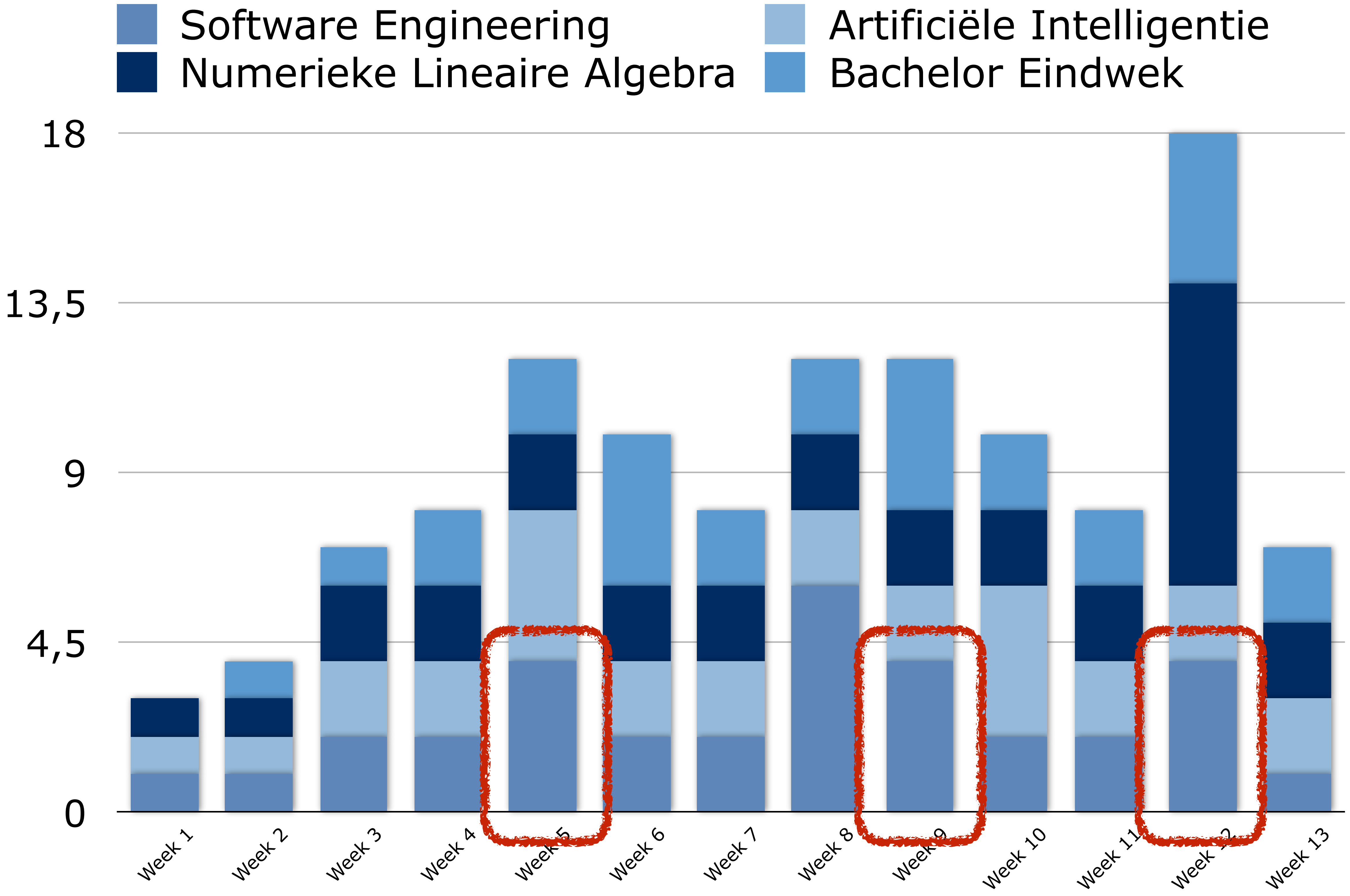
Ideale Semester Planning

Herzien

week 1	[T] Introduction	-- geen oefeningen
week 2	[T] Requirements	[P] Requirements
week 3	[T] Software Architecture	[P] Software Architecture
week 4	[T] Project Management	[P] Project Management
week 5	[T] Design by Contract	-- EVALUATIE
week 6	[T] Testing	[P] Testing
week 7	[T] Formal Specifications	[P] Formal Specifications
week 8	[T] Domain Models	[P] State Charts
week 9	[T] Software Quality	-- EVALUATIE
week 10	[T] Software Metrics	[P] Refactoring
week 11	[T] Refactoring	[P] Software Quality
week 12	(-- reserve)	-- EVALUATIE
week 13	[T] Conclusion	-- geen oefeningen

https://ansymore.uantwerpen.be/software-engineering-3e-bac/tijdsschema

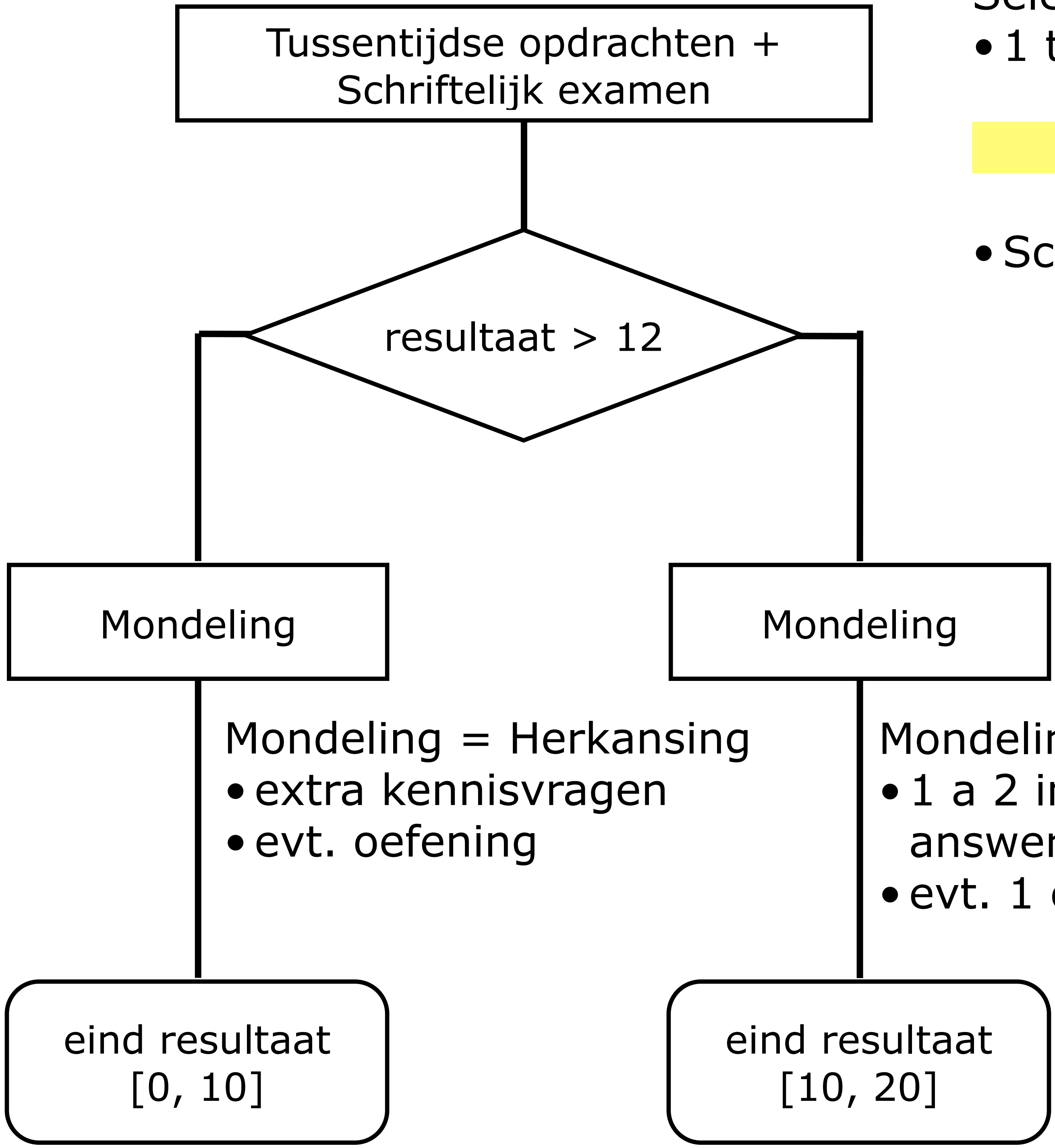
Uitstellen?


Software Engineering
Artificiële Intelligentie
Numerieke Lineaire Algebra
Bachelor Eindwek
18
13,5
9
4,5
0
Week 1
Week 2
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Week 11
Week 12
Week 13

Examen

Selectie (minimumnorm)

- 1 tussentijdse opdracht per hoofdstuk
  - Tijdens het jaar op te leveren
  - *Presentatie + Kwalitatieve feedback*
  - Resultaat > 12
- Schriftelijk examen
  - 1 kennis vraag per hoofdstuk
    - (cfr. "You should know the answer to these questions")
  - beperkte oefeningen
  - Resultaat > 12



Mondeling = Herkansing

- extra kennisvragen
- evt. oefening

Mondeling = Diversificatie

- 1 a 2 inzichtsvragen (cfr. "Can you answer the following questions")
- evt. 1 creatieve vraag

Voorbeeldvragen (schriftelijk)

Enkele voorbeeldvragen

1. Geef 2 redenen waarom het waterval model onrealistisch is.
2. “Het systeem is voor 93% correct” is een geldige uitspraak.
   Ja / Neen
   Waarom?
3. Bij het overschrijven van een methode in een subclasse ...
   (a) moet de preconditie gelijk blijven
   (b) mag de preconditie zwakker worden
   (c) mag de preconditie sterker worden
   Waarom?

Modelantwoorden

1.
   gebruikers kunnen behoeften nooit volledig specificeren
   een werkende versie is veel te laat beschikbaar
2. ~~Ja~~ / Neen
   Correctheid is een absolute eigenschap.
3. (b)
   Een subklasse moet minstens hetzelfde contract vervullen.

Voorbeeldvragen (mondeling)

Een voorbeeld v. e. inzichtsvraag

+ Leveren CRC-kaarten het best mogelijke ontwerp? Argumenteer?

+ Modelantwoord
  - er is geen eenduidig criterium om te meten wat het “beste” ontwerp is
  - de techniek is heel vrij: elke stap kan verscheidene goede antwoorden bieden
  - veel hangt af van de groepsdynamiek

Een voorbeeld v.e. creatieve vraag

+ Je baas heeft op de radio gehoord van het “log4j security exploit” en vraagt een veiligheidsplan. Wat zul je allemaal in dat veiligheidsplan opnemen ?

+ Modelantwoord
+ De context is niet voldoende duidelijk en je moet zelf vragen stellen om die helder te krijgen.
  - Wat voor soort systeem is het? Hoe hangt het aan het internet?
  - Wat voor soorten risico’s loopt het systeem? Hoeveel risico is je baas bereid te lopen? Hoeveel is hij bereid te investeren?

Criteria (ii)

- Levenslang leren
  + Een professionele software engineer zal zijn leven lang de technische evoluties op de voet moeten volgen.

      * Vele referenties naar boeken, artikels, world-wide web
      * “Engels” als voertaal voor de transparanten

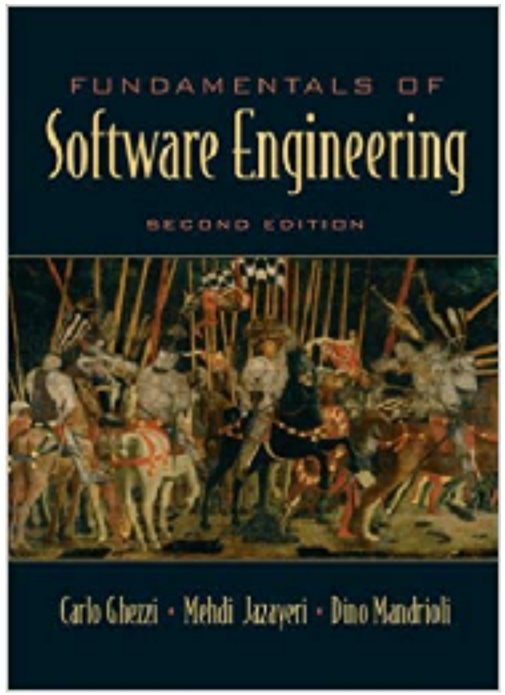
ACHTERGRONDINFORMATIE:
- Je wordt verondersteld zelf selectief met diverse bronnen om te gaan.

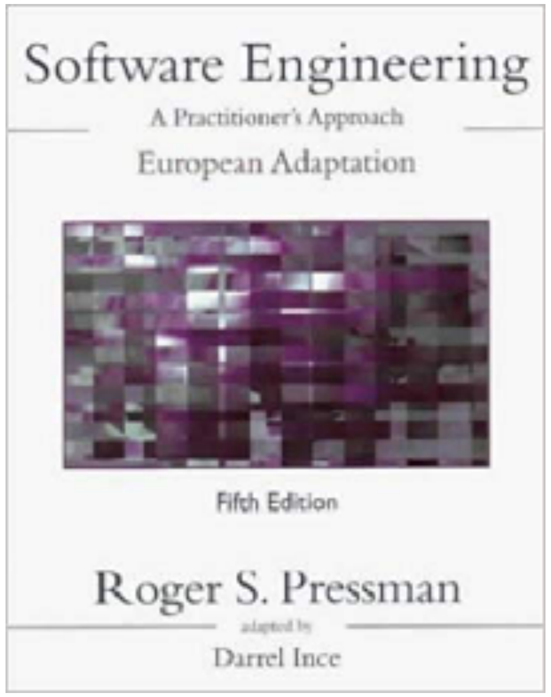
Literatuur

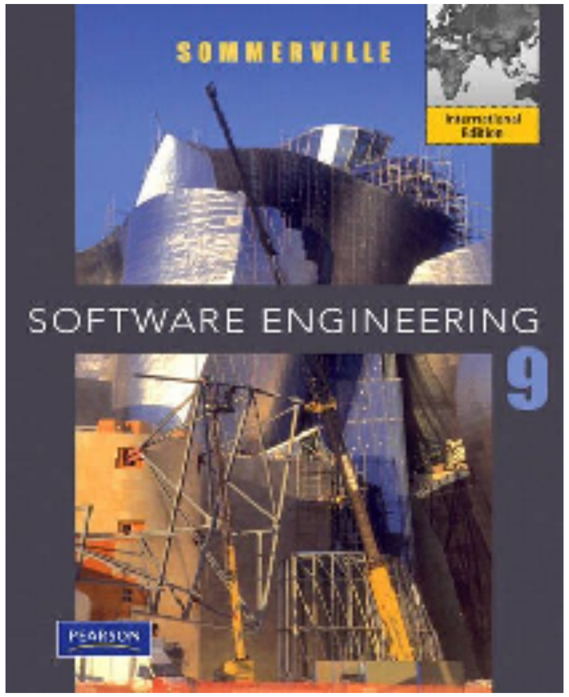
Aanbevolen is 1 van onderstaande boeken aandachtig door te nemen (INZICHT)

- + [Ghez02] Fundamentals of software engineering (Second edition), C. Ghezzi, M. Jazayeri, D. Mandroli, Prentice Hall, 2002.
  - > Tijdloos door de nadruk op onderliggende principes, maar daardoor moeilijk.
- + [Pres00] Software Engineering — A Practitioner's Approach, R. Pressman (Fifth Edition), Mc-Graw Hill, 2000. (2014 = Eighth edition)
  - > Zeer praktisch en zeer diep, maar anderzijds weinig selectief en volumineus (dus duur).
- + [Somm05] Software Engineering (Ninth Edition), I. Sommerville, Addison-Wesley, 2011. (2018 = Tenth edition)
  - > Zeer populair, zeer breed en makkelijk leesbaar, maar mist af en toe wat diepgang.

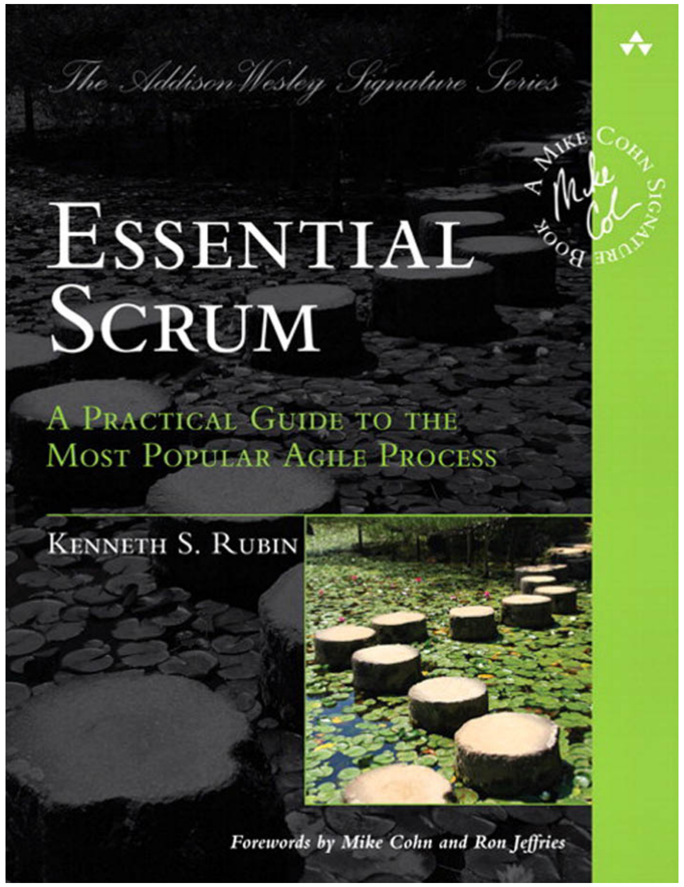
Andere literatuur wordt per hoofdstuk vermeld, incl. referenties op het web.





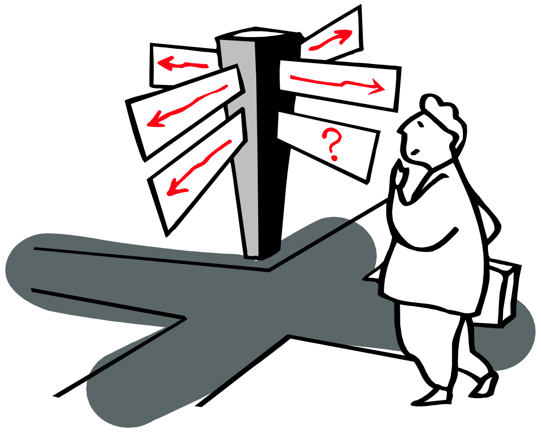


Literature (Agile)


The Addison-Wesley Signature Series
Essential Scrum
A Practical Guide to the Most Popular Agile Process
Kenneth S. Rubin
A Mike Cohn Signature Book
Forewords by Mike Cohn and Ron Jeffries

- [Rubi13] Essential Scrum: A Practical Guide to the most popular agile process. Kenneth S.Rubin. Addison-Wesley, 2013

CHAPTER 1 – Introduction



- Software Engineering
  - + Why & What
  - + Product & Process
    - ⇒ *Correctness & Traceability*
- Software Process
  - + Activities
  - + Iterative & Incremental Development
    - ⇒ *Risk*
  - + Sample Processes
    - - Unified Process
    - - Spiral model
    - - Agile Development
      - > Agile Manifesto, XP
    - ⇒ Scrum
- Software Product
  - + UML

CAPSTONE PROJECT

- ("Bachelor Eindwerk")
  - + assess the risk to your project
  - + apply Scrum process

Literature

- Other
  - + [Brue00] Object-Oriented Software Engineering, B. Bruegge, A. Dutoit, Prentice Hall, 2000.
    - One of the first software engineering textbooks with a specific object-oriented perspective
  - + [Gold95] Succeeding with Objects: Decision Frameworks for Project Management, A. Goldberg and K. Rubin, Addison-Wesley, 1995.
    - Explains how to define your own project management strategy

- Papers
  - + [Armo00] Phillip G. Armour, “The Five Orders of Ignorance”, COMMUNICATIONS OF THE ACM October 2000/Vol. 43, No. 10
    - A very good explanation of the “known knowns”; “unknown knowns” and “unknown unknowns” phenomenon
  - + [Larm2003] Craig Larman and Victor R. Basili, ”Iterative and Incremental Development: A Brief History", IEEE Computer, June 2003
    - An overview of how we improved upon the naive waterfall

Why Software Engineering?

A naive view on software development

Specification
Final Program

- But...
  + Where did the specification come from?
  + How do you know the specification corresponds to the user’s needs?
  + How did you decide how to structure your program?
  + How do you know the program actually meets the specification?
  + How do you know your program will always work correctly?
  + What do you do if the users’ needs change?
  + How do you divide tasks if you have more than a one-person team?

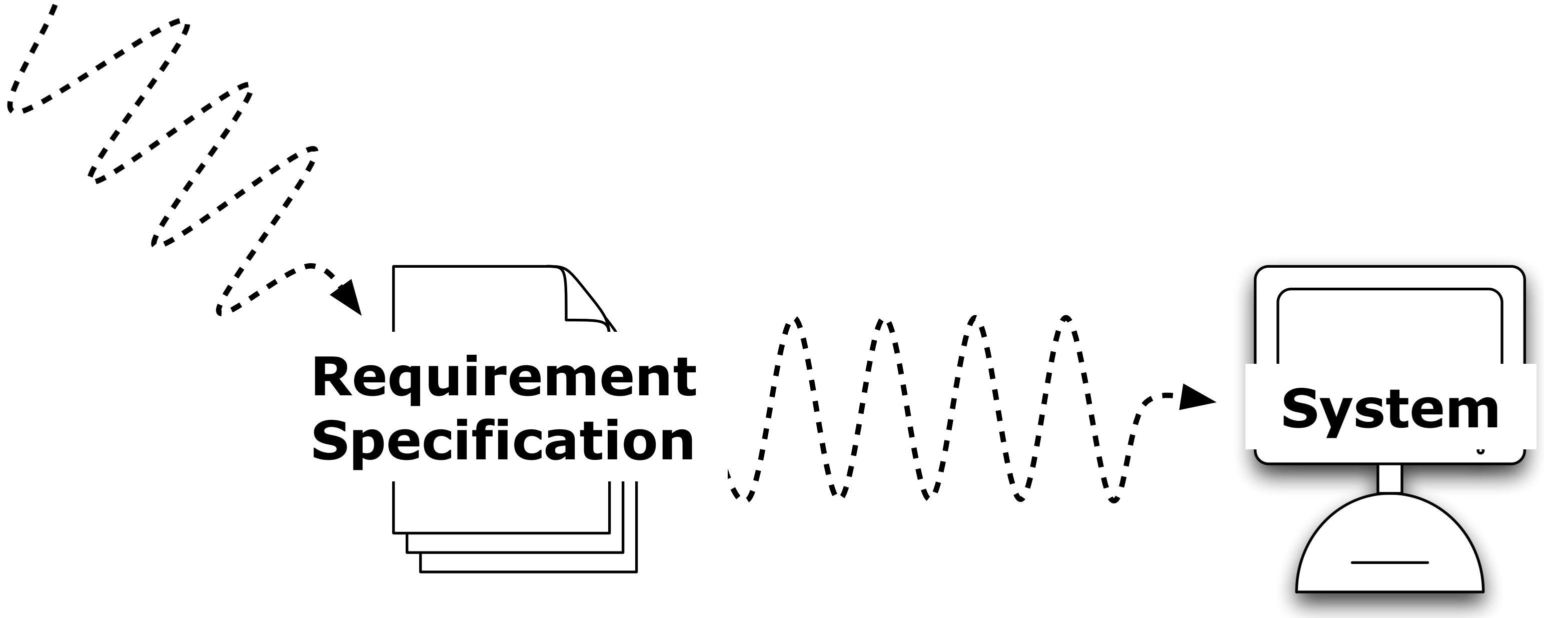
What is Software Engineering?

- Some Definitions and Issues
  + “state of the art of developing quality software on time and within budget” [Brue00]
    - Trade-off between perfection and physical constraints
      > Software engineering has to deal with real-world issues
    - State of the art!
      > “best practice” is a moving target ⇒ life-long learning

  + “multi-person construction of multi-version programs” [Parn75]
    - Team-work
      > Scale issue + Communication Issue
    - Successful software systems must evolve or perish
      > Change is the norm, not the exception

  + “software engineering is different from other engineering disciplines” [Somm05]
    - Not constrained by physical laws
      > limit = human mind
    - It is constrained by political forces
      > balancing stake-holders

Product and Process



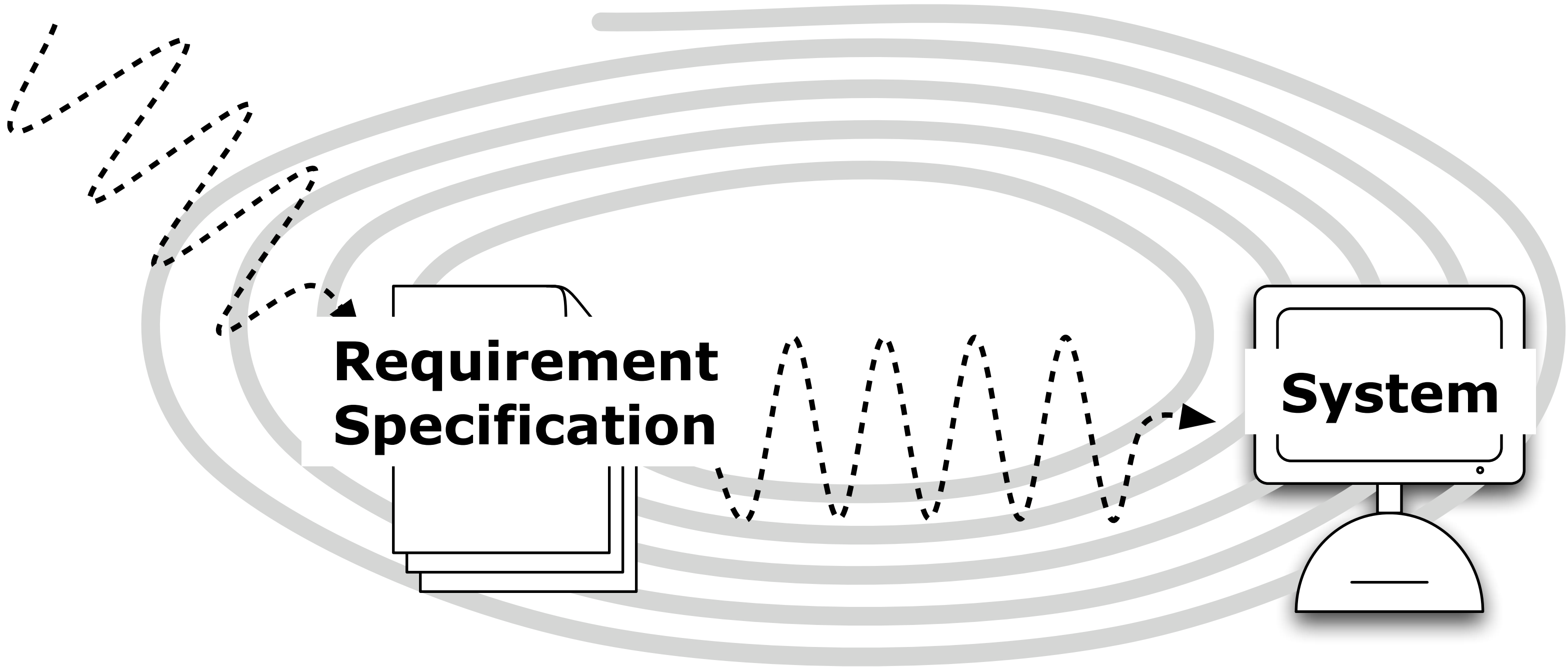
Product

- = What is delivered to the customer
- [Requirements Specification + System (+ all documentation, manuals, ...)]

Process

- = Collection of activities that leads to (a part of) a product
- [During process we apply techniques]

Evaluation Criteria



2 evaluation criteria to assess techniques applied during process

Correctness

- Are we building the right product? = VALIDATION
- Are we building the product right? = VERIFICATION

Traceability

- Can we deduce which product components will be affected by changes?

Traceability

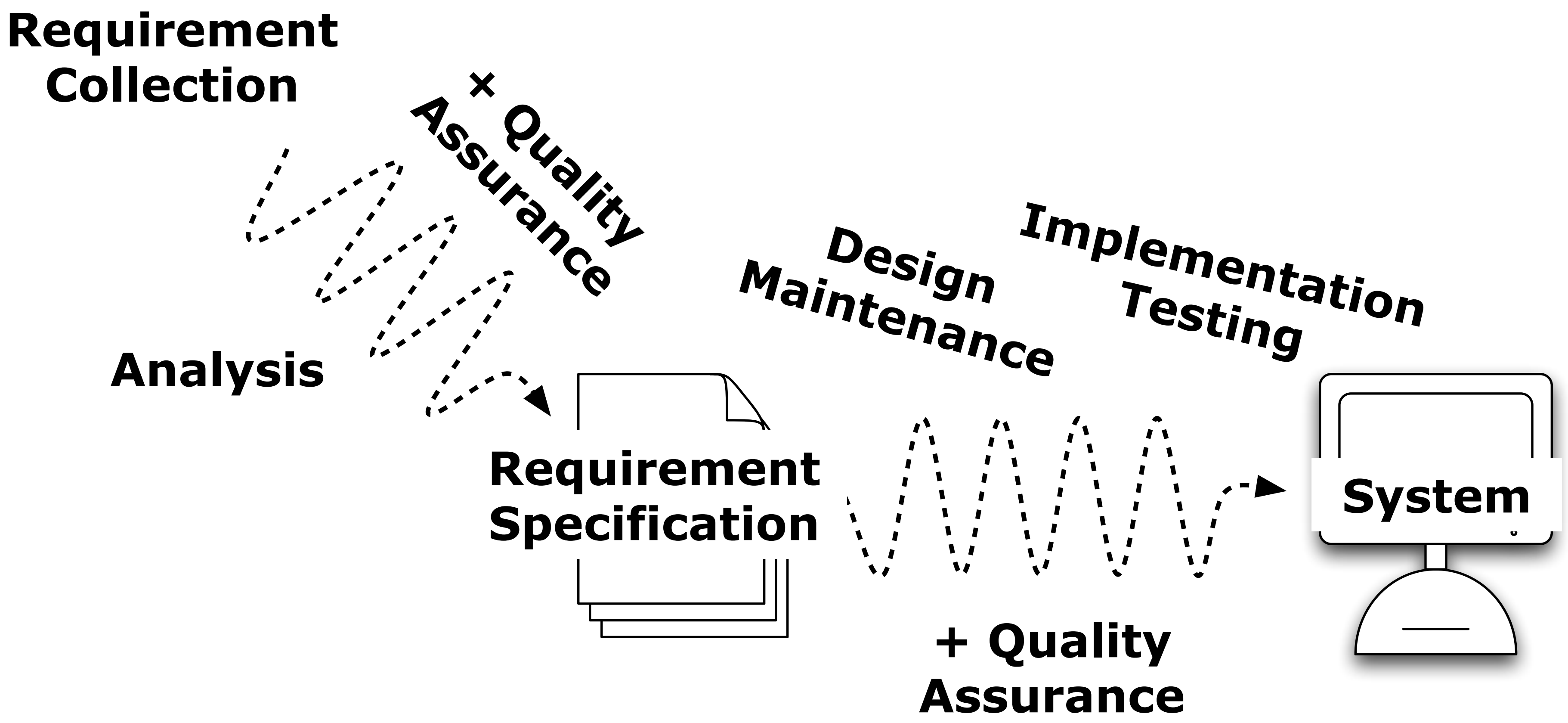
How to predict impact of changes?
Maintain relationship

- from component to requirement that caused its presence
- from requirement that must be changed when component is adapted

	Comp 1	Comp 2	…	…	…	…	…	Comp m
Req 1				x				
Req 2	x							x
…								
…		x						
…						x	x	
Req n								x

This table is *virtual*: it is much too large to maintain explicitly!
⇒ A good process should help you deducing this relationship.

Software Process Activities (i)



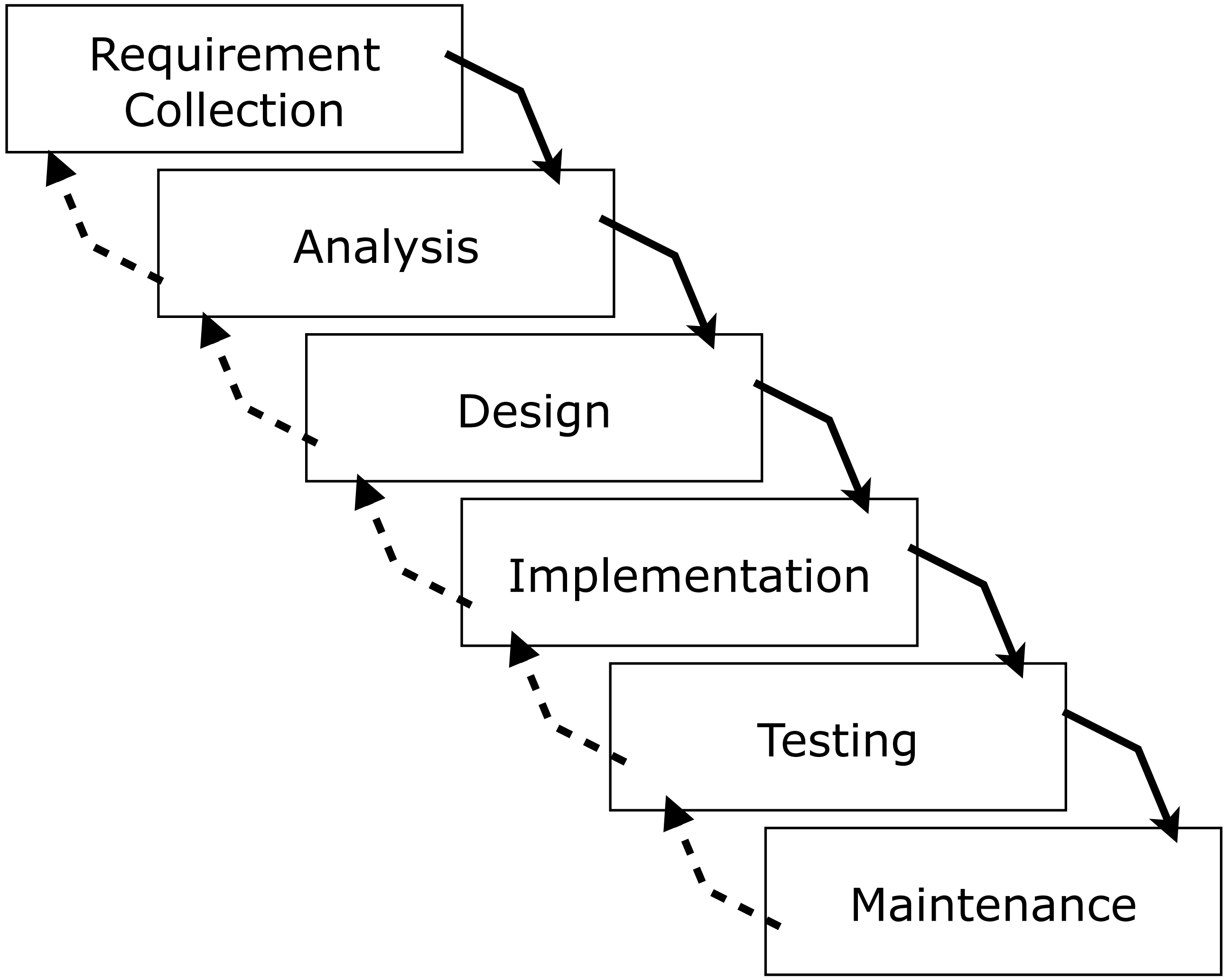
Software Process Activities (ii)

- Requirements Collection (a.k.a. Requirements Elicitation)
  + Establish customer's needs
- Analysis
  + Model and specify the requirements ("what")
- Design
  + Model and specify a solution ("how")
  + system design (architecture) + detailed design (object design, formal spec)
- Implementation
  + Construct a solution in software
- Testing
  + Verify the solution against the requirements
- Maintenance
  + Change a system after its been deployed
  + = Repair defects + adapt to new requirements
- Quality Assurance
  + Make sure all above goes well
  = Deliver quality, on time and within budget

VALIDATION

VERIFICATION

The Waterfall Software Lifecycle



The classical software life cycle models the software development as a step-by-step "waterfall" between the various development activities.

- going backward is possible but should be an exception (implies a mistake)

The waterfall model is popular for upper management, because

- *Visible*: it is easy to control project progress

> Very explicit in project bidding & contract negotiations!

The waterfall model is unrealistic for large projects, because

- *Complete*: a customer cannot state all requirements explicitly
- *Idealistic*: in real projects iteration occurs (but tools and organisation obstruct)
- *Time*: A working version of the system is only available late in the project
- *Change*: it is very difficult and costly to adapt to changes in the requirements

Iterative and Incremental Development

- A good process must mix two principles (see [Gold95], p. 94-96)

- Iterative Development
  + Controlled reworking of a system part to make improvements
    - We get things wrong before we get them right
      (Software development is a learning experience)

- Incremental Development
  + Make progress in small steps to get early tangible results
    - Always have a running version
      (Control your learning via concrete intermediate steps)

Knowns & Unknowns

[This is terminology used for planning military campaigns.]
Phillip G. Armour, "The Five Orders of Ignorance", COMMUNICATIONS OF THE ACM October 2000

Known knowns

- = the things you know you know
  You can safely make assumptions here during planning

Known unknowns

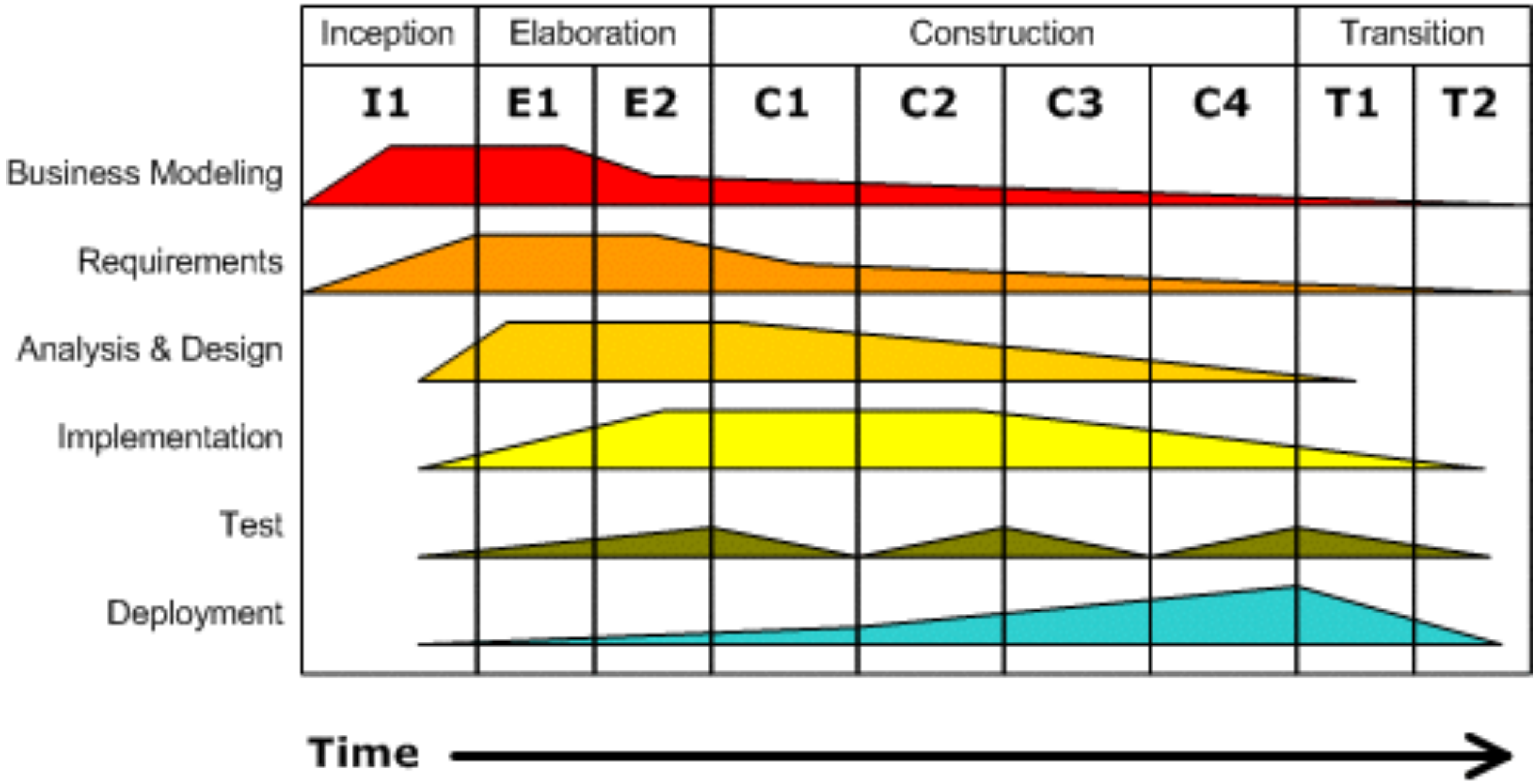
- = the things you know, you don't know
  You can prepare for these during planning

Unknown unknowns

- = the things you do not know, you don't know
  These you cannot prepare for during planning
  … the best you can do is being aware and spot opportunities
  + do a thorough *risk analysis*

- software projects (compared to other engineering projects) have lots of "unknown unknowns"
  + Not constrained by physical laws
  + Many stakeholders ⇒ strong political forces around project

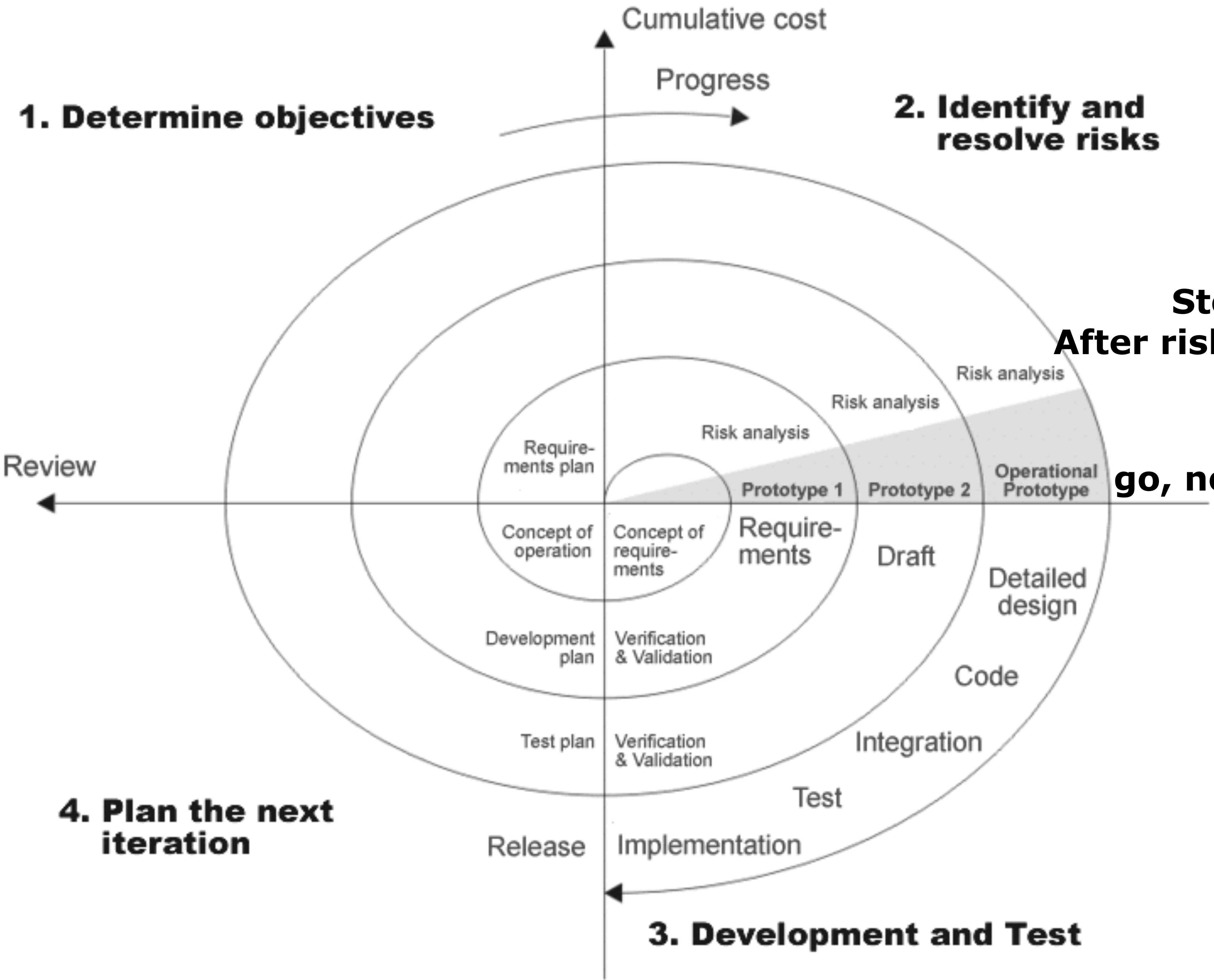
The Unified Process



DutchGuilder Wikipedia

How do you plan the number of iterations? How do you decide on completion?

Boehm’s Spiral Lifecycle



Stop?
After risk analysis

go, no-go decision

© Image adapted from Boehm, B. (1988) A Spiral Model of Software Development and Enhancement. IEEE Computer, 21 (5), 62-72.

Risk Analyis (a.k.a. Risk Management)

Risk Identification

> Identify risk factors via “risk item checklist” (see [Pres00])

- Project Risks: e.g., staffing risk
- Technical Risks: e.g. “leading edge” technology
- Business Risks: e.g., market risk (building a product that nobody wants)

Risk Projection (Risk Estimation)

- For each risk factor, estimate the likelihood and the impact
  - + 3 point likert scale:
    - low - medium - high
  - + 5 point likert scale
    - [impact] insignificant - minor - moderate - major - catastrophic
    - [likelihood] almost certain - likely - possible - unlikely - rare
- Prioritize the list

Risk Assessment

- For each “important” risk factor, take action to reduce risk
  - + important? Depending on your risk appetite
- … or terminate project
- Examples
  - + Staff does not have the right skills ⇒ Define training plan and hire extra staff
  - + “Leading edge” technology ⇒ Build a prototype to evaluate benefits/drawbacks
  - + Market risk ⇒ do a market study

Risk Projection (refined)

Risk = impact * likelihood

		impact		
		Low	Medium	High
likelihood	High	low	medium	high
	Medium	low	medium	medium
	Low	low	low	low

		impact				
		insignificant	minor	moderate	major	catastrophic
likelihood	almost certain	moderate	high	high	critical	critical
	likely	moderate	moderate	high	high	critical
	possible	low	moderate	high	high	critical
	unlikely	low	moderate	moderate	high	high
	rare	low	low	moderate	moderate	high

Risk Projection (continued)

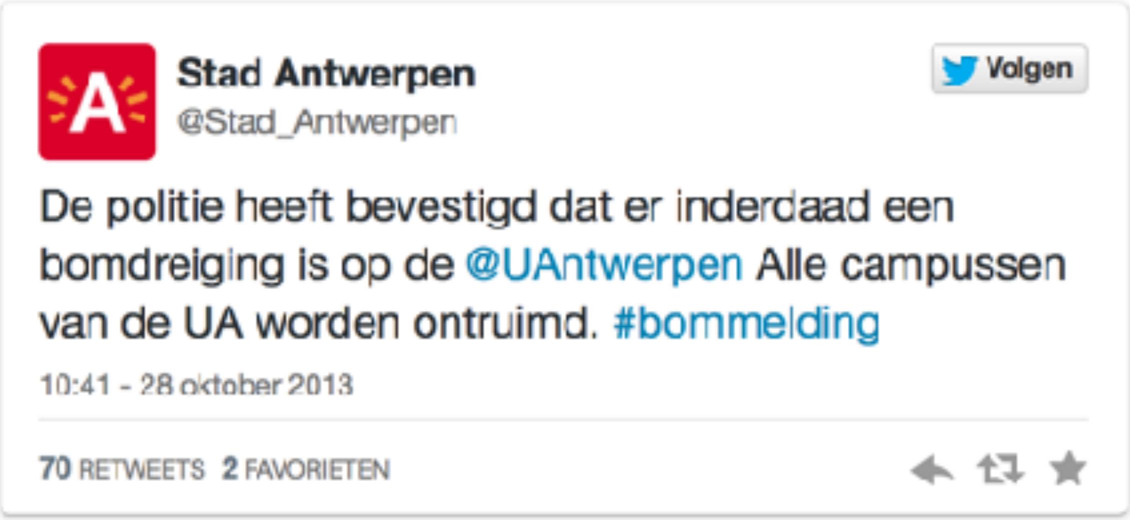
Sometimes a 3rd item is added to the equation

Risk = impact * likelihood * urgency

urgency = the time left before measures or responses would need to be implemented

less time available ⇒ risk becomes more critical

Risk Assessment (example)



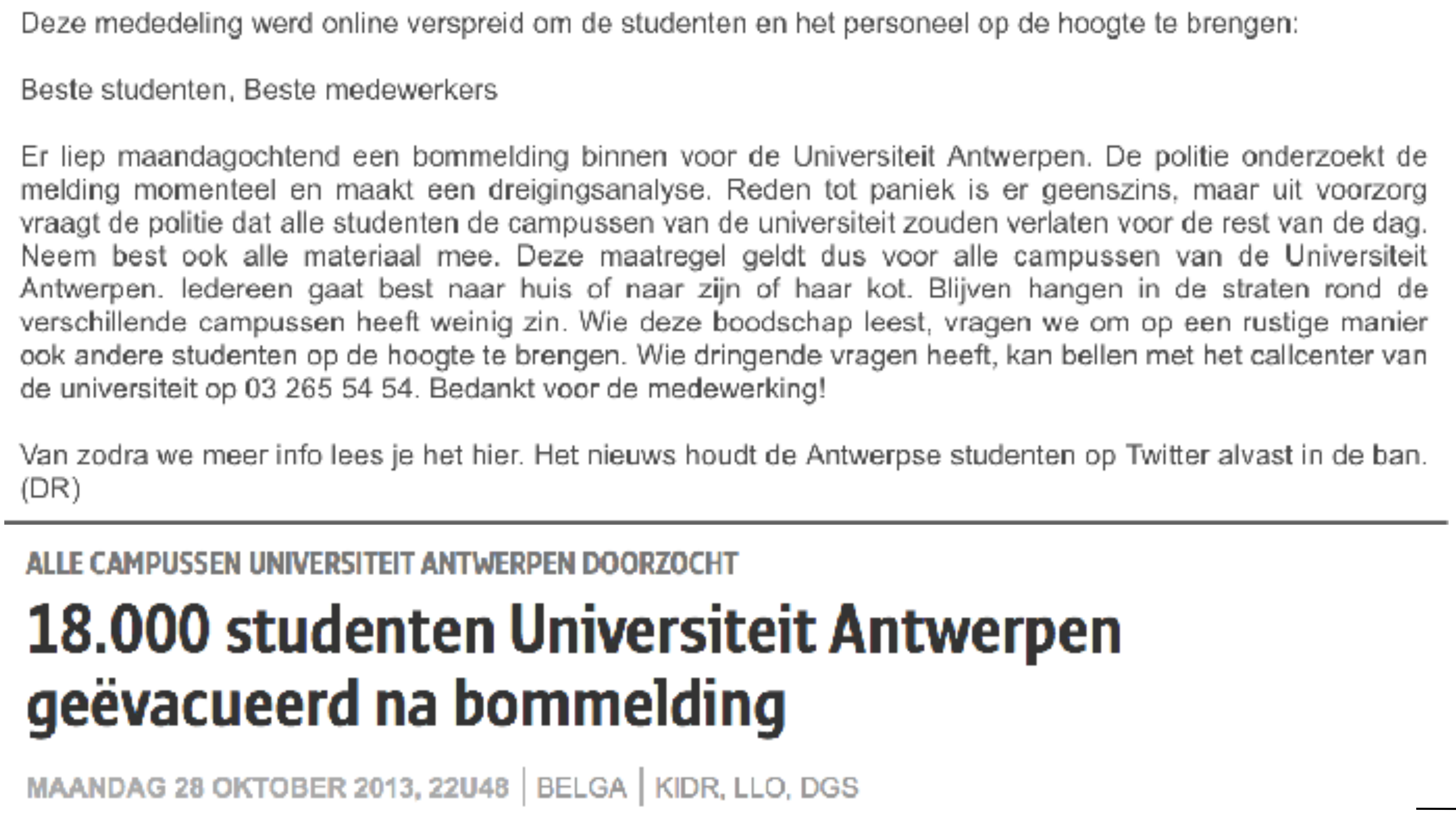
Stad Antwerpen
@Stad_Antwerpen

De politie heeft bevestigd dat er inderdaad een bomdreiging is op de @UAntwerpen Alle campussen van de UA worden ontruimd. #bommelding

10:41 - 28 oktober 2013

70 RETWEETS 2 FAVORIETEN

Risk?
- probability: extremely unlikely
  (however, 3 independent e-mails)
- urgency: extremely urgent
  (potential explosion within hours)
- impact … infinite
  (potential life loss of students)



Deze mededeling werd online verspreid om de studenten en het personeel op de hoogte te brengen:

Beste studenten, Beste medewerkers

Er liep maandagochtend een bommelding binnen voor de Universiteit Antwerpen. De politie onderzoekt de melding momenteel en maakt een dreigingsanalyse. Reden tot paniek is er geenszins, maar uit voorzorg vraagt de politie dat alle studenten de campussen van de universiteit zouden verlaten voor de rest van de dag. Neem best ook alle materiaal mee. Deze maatregel geldt dus voor alle campussen van de Universiteit Antwerpen. Iedereen gaat best naar huis of naar zijn of haar kot. Blijven hangen in de straten rond de verschillende campussen heeft weinig zin. Wie deze boodschap leest, vragen we om op een rustige manier ook andere studenten op de hoogte te brengen. Wie dringende vragen heeft, kan bellen met het callcenter van de universiteit op 03 265 54 54. Bedankt voor de medewerking!

Van zodra we meer info lees je het hier. Het nieuws houdt de Antwerpse studenten op Twitter alvast in de ban. (DR)

ALLE CAMPUSSEN UNIVERSITEIT ANTWERPEN DOORZOCHT

18.000 studenten Universiteit Antwerpen geëvacueerd na bommelding

MAANDAG 28 OKTOBER 2013, 22U48 | BELGA | KIDR, LLO, DGS

Risk Projection (duo exercise)

New

What is the risk that you will postpone the weekly software engineering assignments?

- If risk is medium, what mitigation actions will you take?
- If risk is high, what mitigation actions will you take?

		impact		
		Low	Medium	High
likelihood	High	low	medium	high
	Medium	low	medium	medium
	Low	low	low	low

Failure Mode and Effects Analysis (FMEA)

- A step-by-step approach for identifying all possible failures in a design, a manufacturing or assembly process, or a product or service.

  + *“Failure modes”*
    - means the ways, or modes, in which something might fail. Failures are any errors or defects, especially ones that affect the customer, and can be potential or actual.

  + *"Effects analysis”*
    - refers to studying the consequences of those failures.

FMECA: Failure Mode, Effect and Criticality Analyses

  + “*Criticality Analysis*”
    - used to chart the probability of failure modes against the severity of their consequences
    - mainly when systems are already in operation

Failure Mode and Effects Analysis (Example)

Potential Failure Mode	Potential Effects of Failures	Severity	Potential Causes of Failures	Current Process Control	Occurrence (± Likelihood)	Detection (± Urgency)	Critical (± Impact)	Risk Priority Number	Recommended Actions
Function: Dispense Fuel									
Does not dispense fuel	- Customer Dissatisfied - Discrepancy in bookkeeping	8	- Out of fuel - Machine jams - Power failure	- Out of fuel alert - Machine jam alert - none					
Dispense too much fuel	- Company loses money - Discrepancy in bookkeeping	8	- Sensor defect - Leakage	- none - pressure sensor					
Takes too long to dispense fuel	- Customer annoyed	3	- Power outage - Pump disrupted	- none - none					

Failure Mode and Effects Analysis (exercise)

New

Potential Failure Mode	Potential Effects of Failures	Severity	Potential Causes of Failures	Current Process Control	Occurrence (± Likelihood)	Detection (± Urgency)	Critical (± Impact)	Risk Priority Number	Recommended Actions
Function: Dispense Fuel									
Does not dispense fuel	- Customer Dissatisfied - Discrepancy in bookkeeping	8	- Out of fuel - Machine jams - Power failure	- Out of fuel alert - Machine jam alert - none					

Assess the risk for the "Does not Dispense Fuel" function (low - medium - high)

- What mitigation actions do you recommend?

https://www.vrt.be/vrtnws/nl/2023/09/13/waarom-je-niet-even-in-je-auto-mag-gaan-zitten-tijdens-het-tanke/

Prototyping

A *prototype* is a software program developed to test, explore or validate a hypothesis, i.e. to reduce risks.

***** proof-of-concept**

An *exploratory prototype*, also known as a throwaway prototype, is intended to validate requirements or explore design choices.

- UI prototype — validate user requirements
- rapid prototype — validate functional requirements
- experimental prototype — validate technical feasibility

An *evolutionary prototype* is intended to evolve in steps into a finished product

- grow, don’t build [Broo87]: “grow” the system redesigning and refactoring along the way
- combines incremental and iterative development

***** First do it, then do it right, then do it fast.**

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

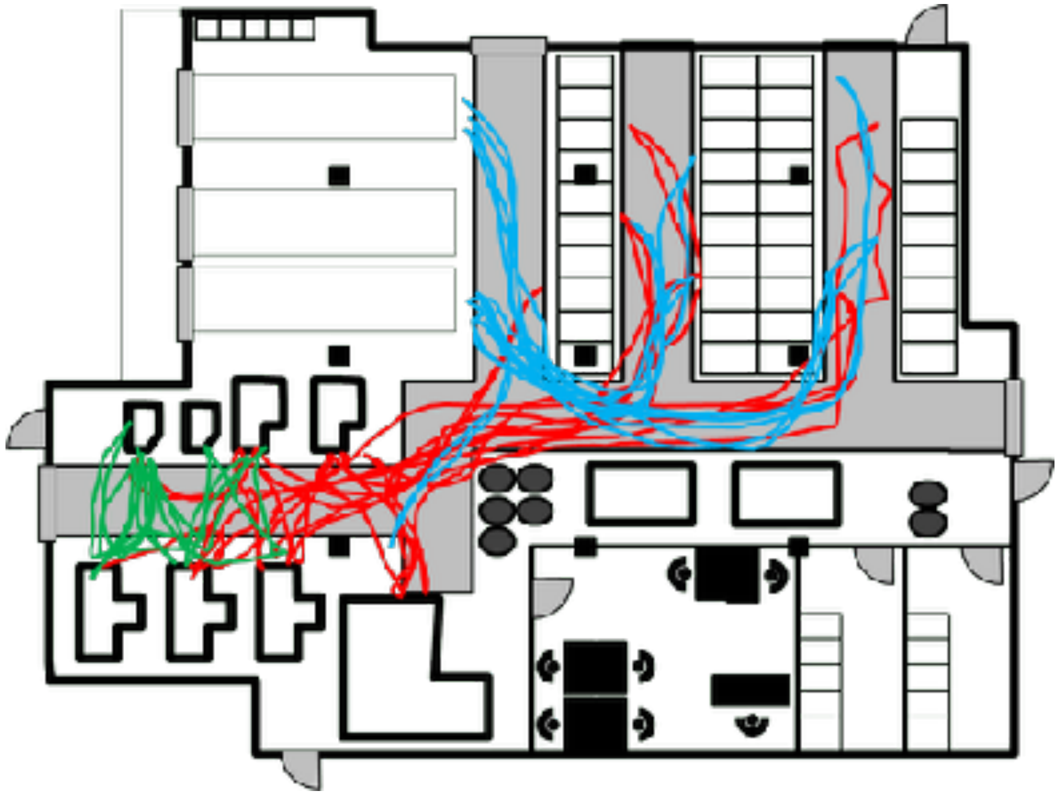
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

http://agilemanifesto.org/

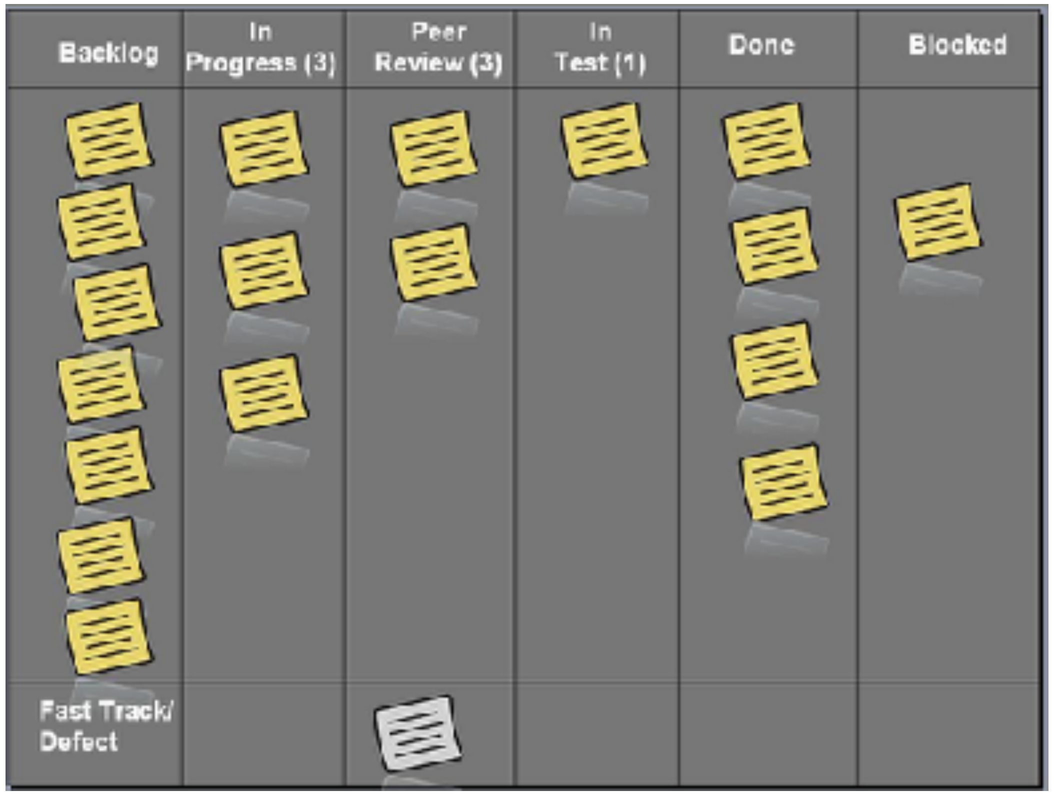
Lean Manufacturing

Eliminate Waste
(e.g. Spaghetti Diagrams)



© Christoph Roser on AllAboutLean.com

Smooth Flow
(e.g. KanBan Boards)



🄯 By Dr Ian Mitchell - Own work

eXtreme Programming (XP)

- Fine scale feedback
  + Pair programming
  + Planning game
  + Test-driven development
  + Whole team

- Continuous process
  + Continuous integration
  + Refactoring or design improvement
  + Small releases

- Shared understanding
  + Coding standards
  + Collective code ownership
  + Simple design
  + System metaphor

- Programmer welfare
  + Sustainable pace

- Coding
  + The customer is always available
  + Code the Unit test first
  + Only one pair integrates code at a time
  + Leave Optimization till last
  + No Overtime

- Testing
  + All code must have Unit tests
  + All code must pass all Unit tests before it can be released.
  + When a Bug is found tests are created before the bug is addressed (a bug is not an error in logic, it is a test you forgot to write)
  + Acceptance tests are run often and the results are published

Agile or not? There is no single truth …

Heavyweight

Lightweight

www.taxonweb.be

Google™

facebook

Spotify®

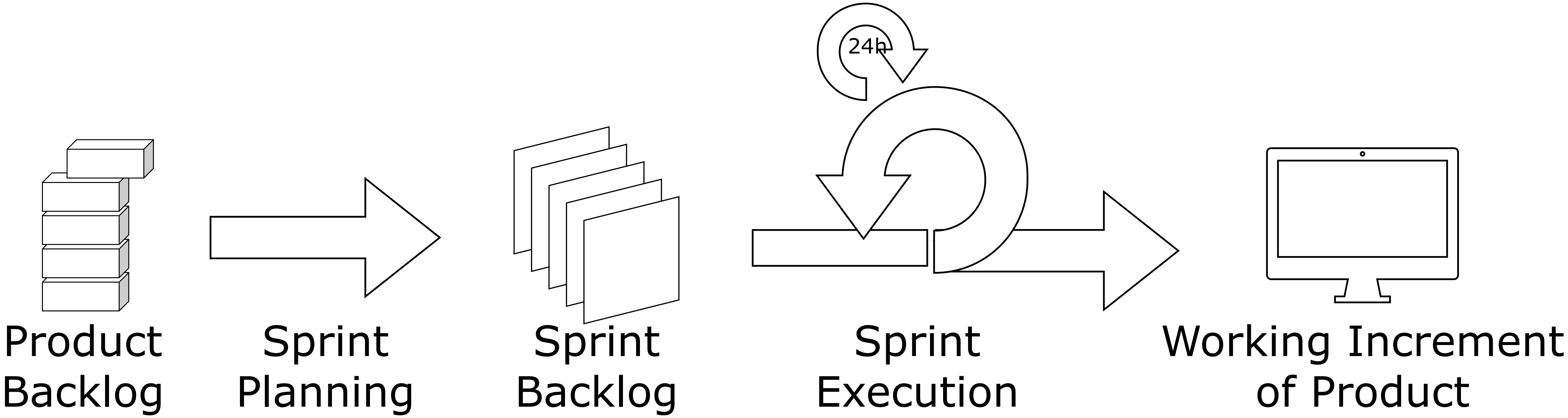
NETFLIX

Scrum — Sprints

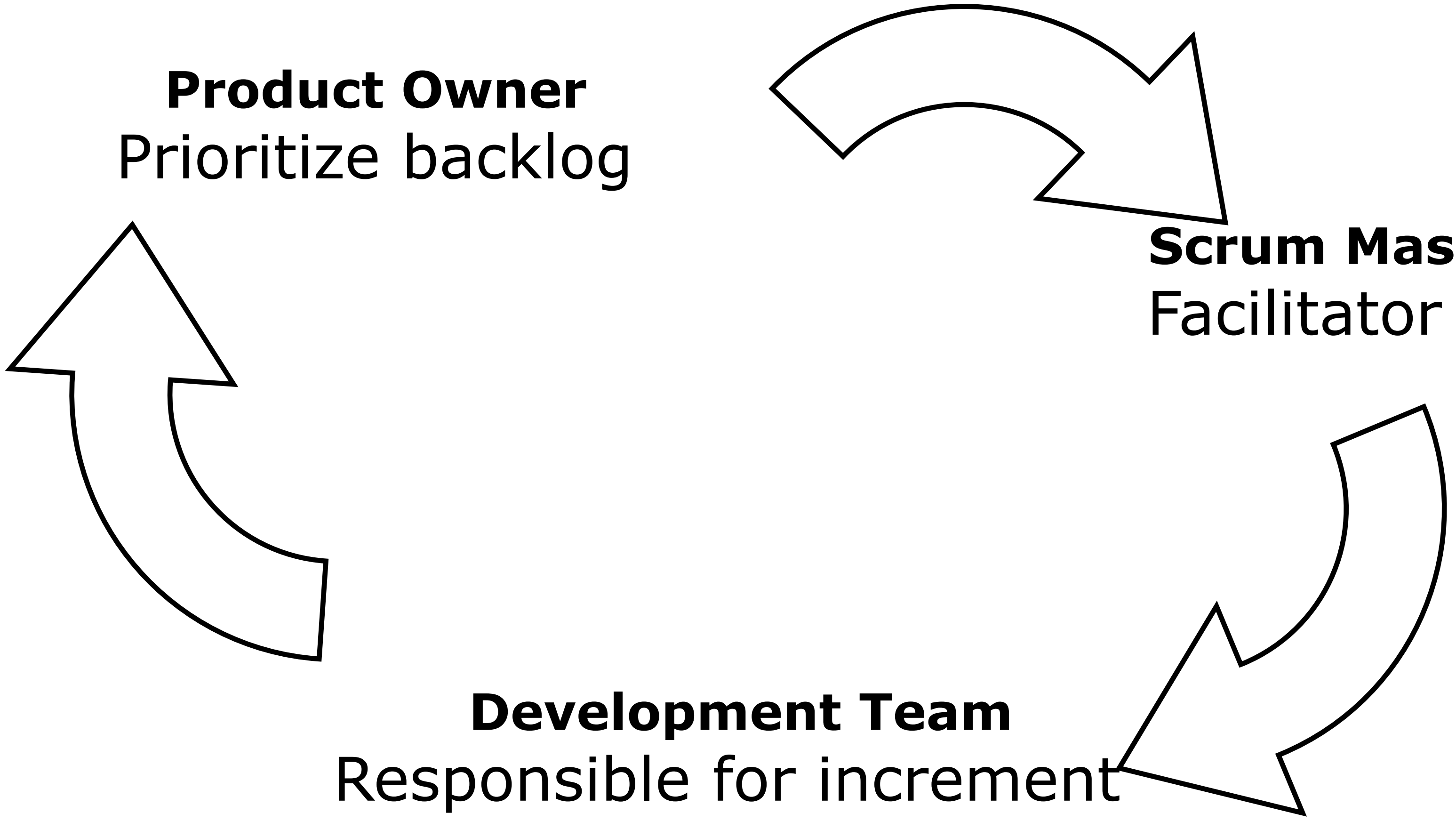
Sprint =

- 2-4 week period
- team creates a working (= potentially shippable) product increment
- features in increment are chosen from product backlog

Daily stand-up meeting!



Scrum — Roles



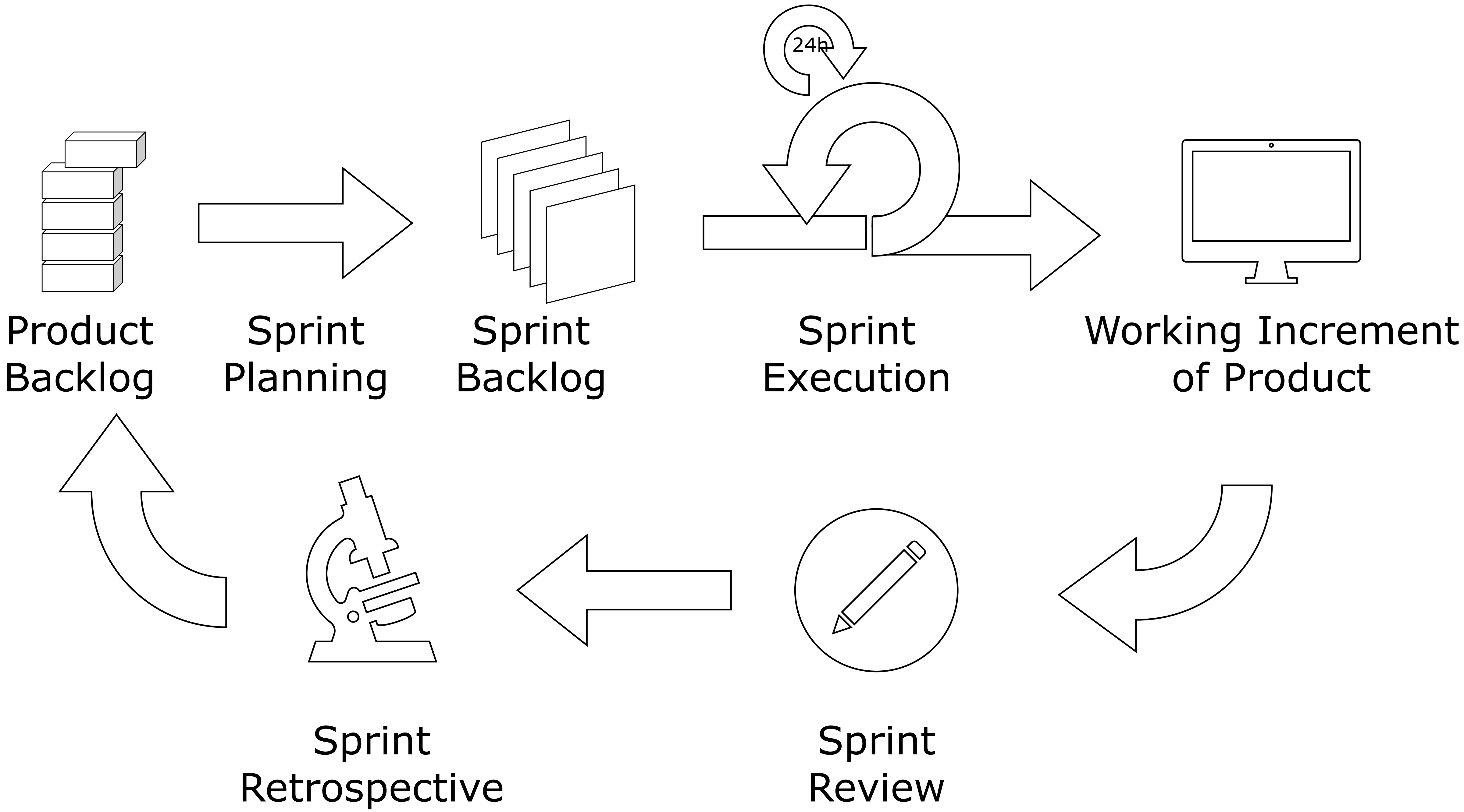
Product Owner
Prioritize backlog

Scrum Master
Facilitator

Development Team
Responsible for increment
to be added to the product

- 5-9 individuals
- self organizing

Scrum — Feedback Loop



Product Backlog

Sprint Planning

Sprint Backlog

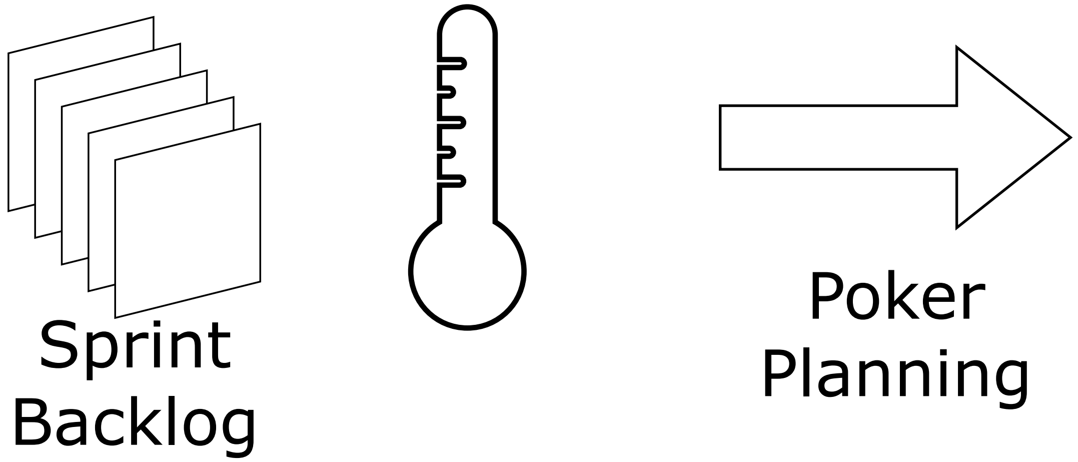
Sprint Execution

Working Increment of Product

Sprint Review

Sprint Retrospective

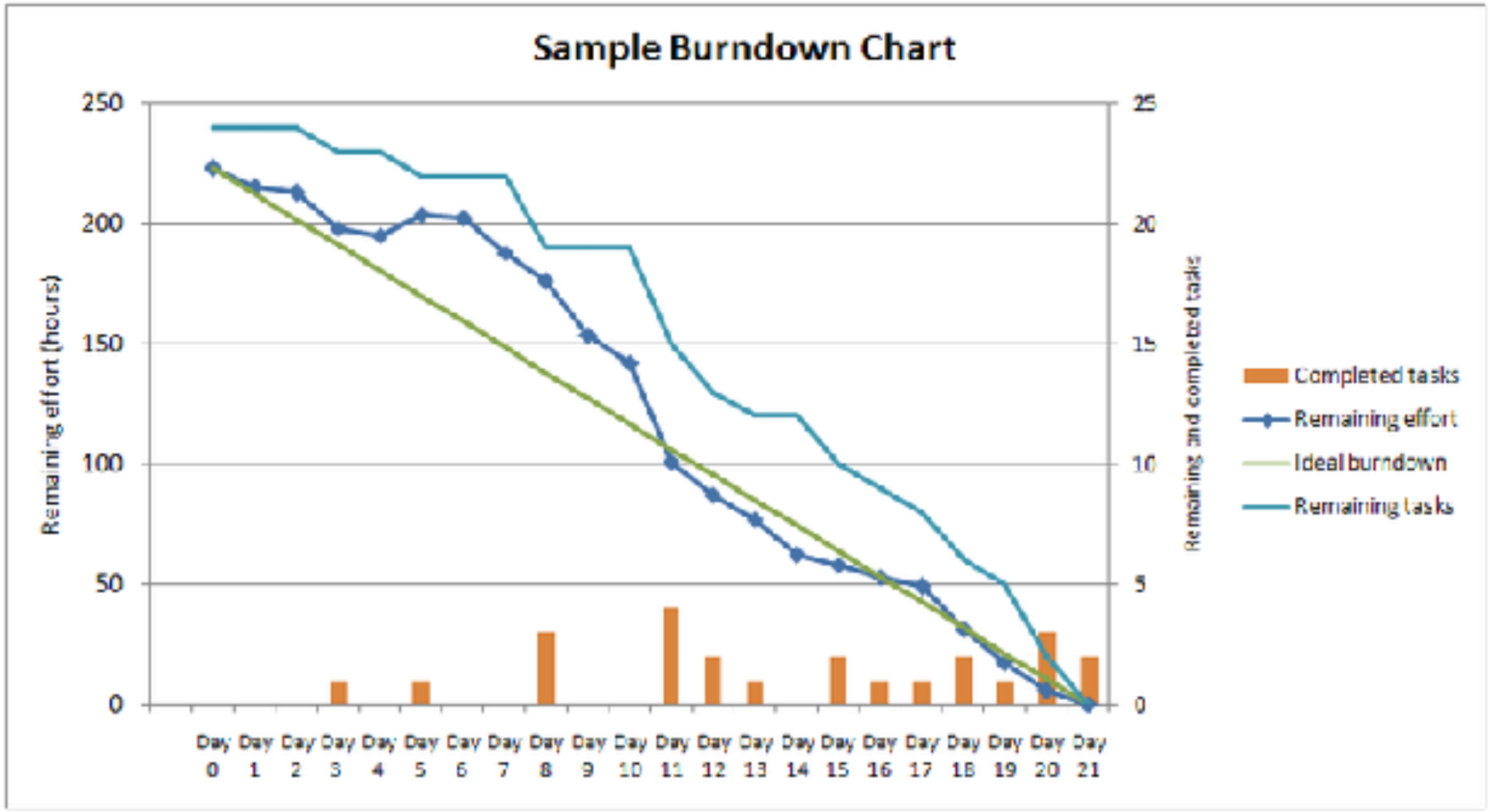
Scrum - Planning & Monitoring



Sprint Backlog

Poker Planning

Task	Points	Hours



Pablo Straub Wikipedia

UML - History

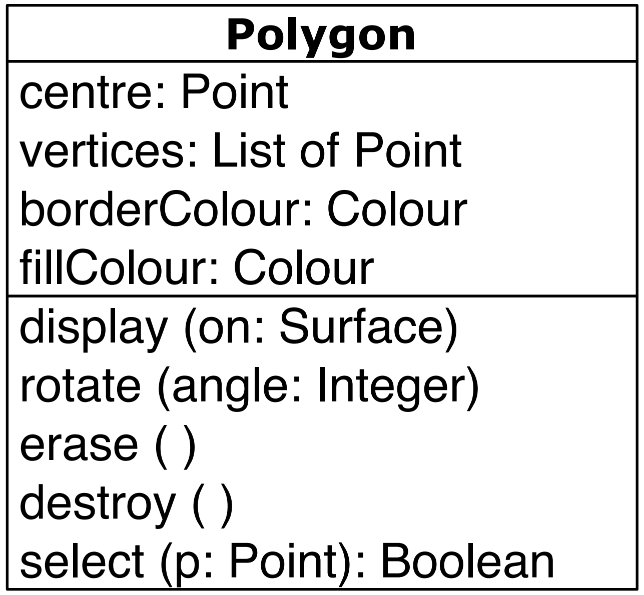
- First generation:
  + Adaptation of existing notations (ER diagrams, state diagrams...):
    - * Booch, OMT, Shlaer and Mellor,...
  + Specialized techniques:
    - * CRC cards; use-cases; design by contract

- Second generation:
  + Combination of “proven” ideas
    - * Fusion: Booch + OMT + CRC + formal methods

- Third generation:
  + Unified Modeling Language:
    - * uniform notation: Booch + OMT + Use Cases + Statecharts
    - * complete lifecycle support (the Unified Process)
    - * adaptable: you can extend the notation, choose your own process

Static UML - Classes (i)

Class name, attributes and operations:
(organized into compartments)

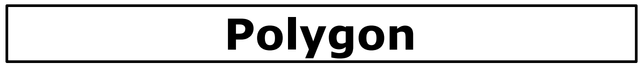


Polygon

centre: Point
vertices: List of Point
borderColour: Colour
fillColour: Colour

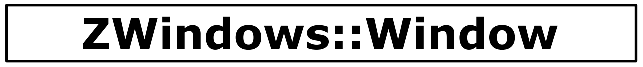
display (on: Surface)
rotate (angle: Integer)
erase ( )
destroy ( )
select (p: Point): Boolean

A collapsed class view.
(NB: attributes & operations not shown, so don't know whether empty or not!)



Polygon

Class with Package name:
(Optional, but useful for large systems !)



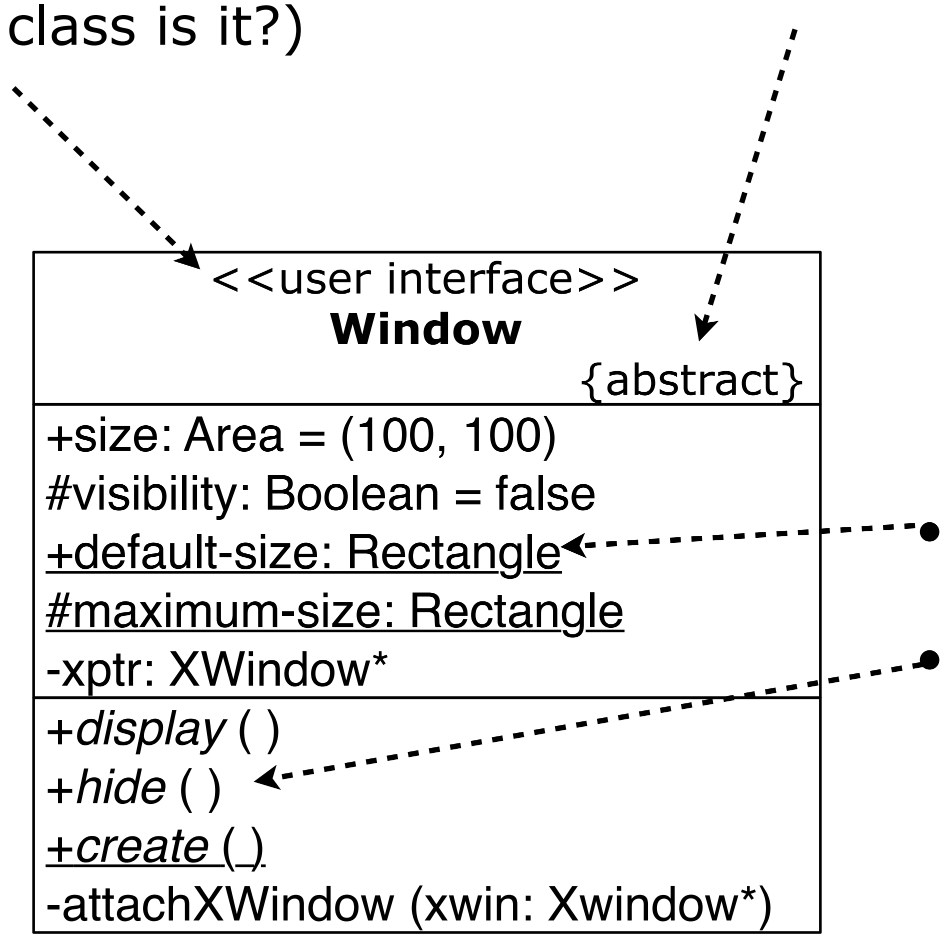
ZWindows::Window

Attributes and operations are also collectively called *features*.

Static UML - Classes (ii)

Stereotype
(what “kind” of class is it?)

User-defined properties
(e.g., abstract, readonly,
owner = “Pingu”)



<<user interface>>
Window
{abstract}

+size: Area = (100, 100)
#visibility: Boolean = false
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*

+*display* ( )
+*hide* ( )
+*create* ( )
-attachXWindow (xwin: Xwindow*)

+ = “public”
# = “protected”
- = “private”
(interpretation is open)

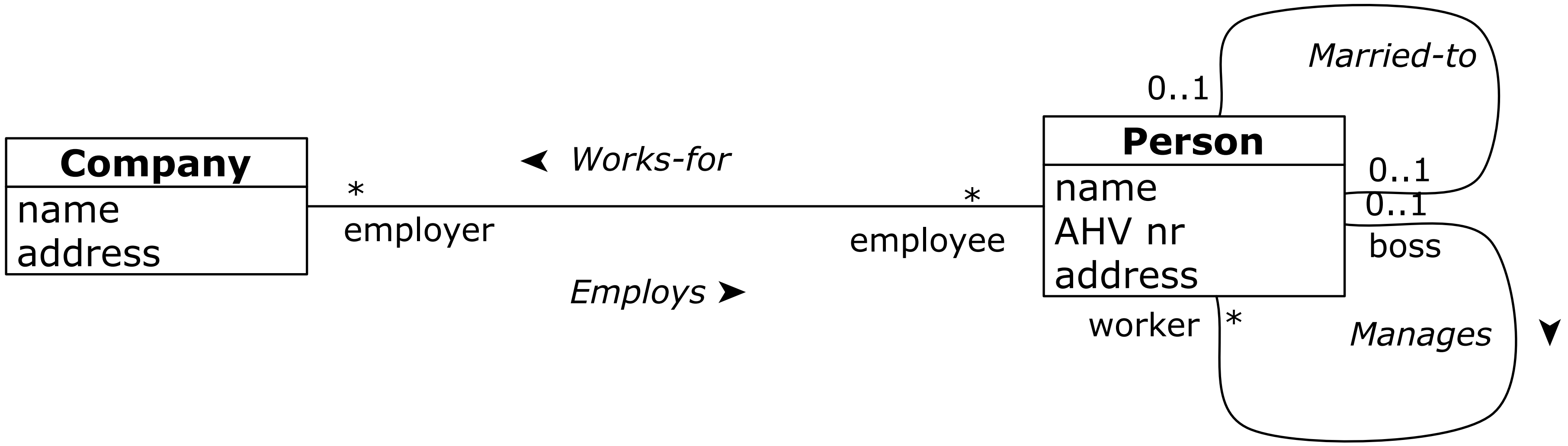
- underlined features have class scope
- *italic* features are abstract

- Attributes are specified as: name: type = initialValue { property string }
- Operations are specified as: name (param: type = defaultValue, ...) : resultType

Static UML - Associations

Associations

- denoted by a solid line.
- represents structural relationships between objects of different classes.



- optional *name* and *direction*
- (unique) *role names* and *multiplicities* at end-points (BEWARE POSITION)
- traverse using *navigation expressions*
  e.g., universityAntwerp.employee[name = “Demeyer”].wife

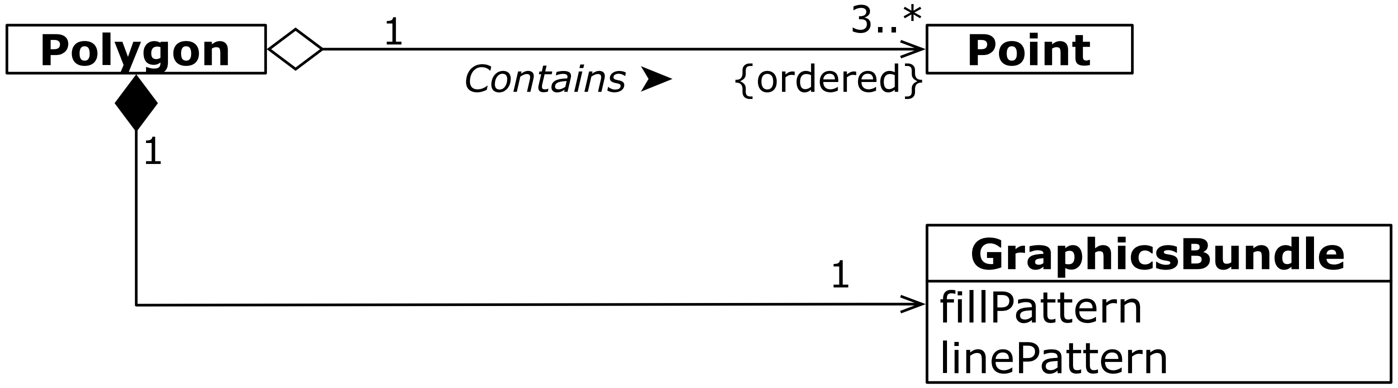
Static UML - Aggregation & Composition

Aggregation

- denoted by a hollow diamond
- whole-part relationship: part may exist without the whole
  (i.e. whole owns a reference to the part)

Composition

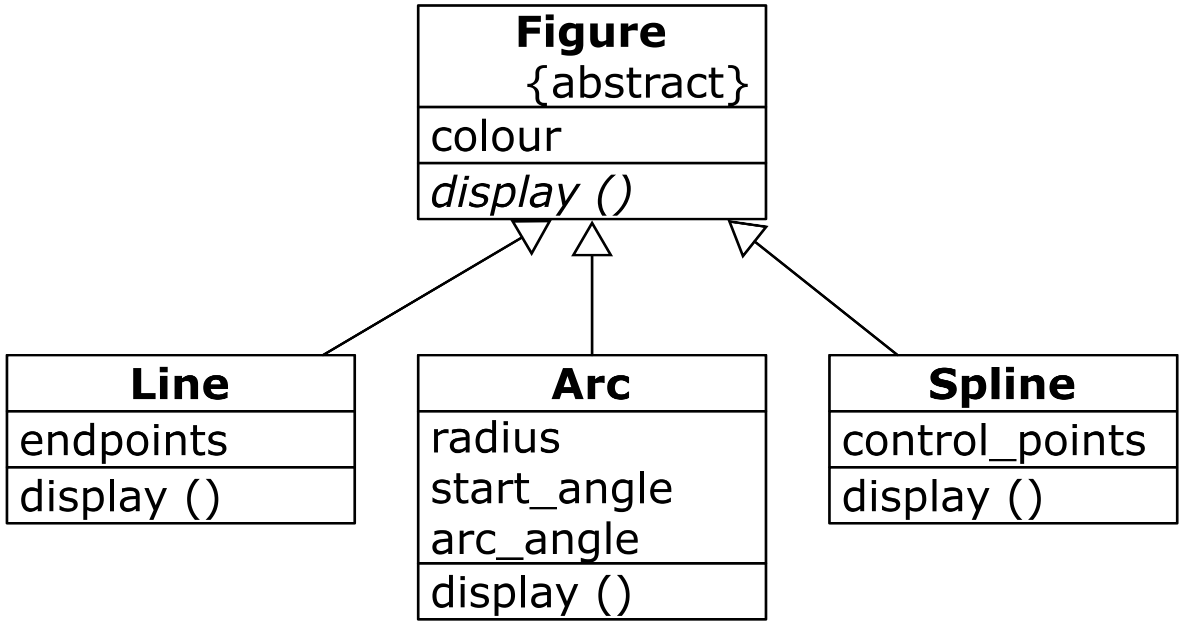
- denoted by a solid diamond
- whole-part relationship: part must always exist with the whole
  (i.e., whole owns the part)



Static UML - Generalization

Generalization

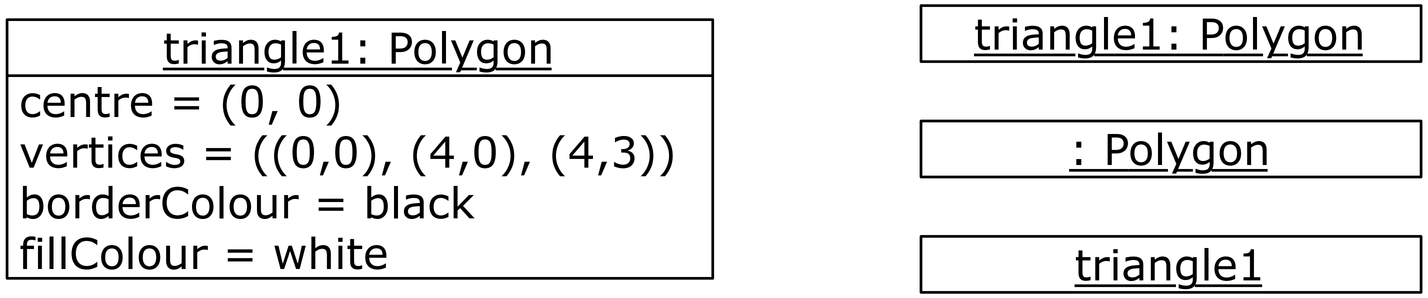
- denoted with a hollow arrow from the specific to the general
- represents inheritance, is-a relationships, code reuse relationship (philosophical debate: Square inherits from Rectangle or vice-versa)



Dynamic UML - Objects

Objects

- shown as rectangles with their name and type underlined in one compartment
- attribute values, optionally, in a second compartment
- the name of the object may be omitted (then colon must be kept with class name)
- the class of the object may be supressed (together with the colon) to represent an anonymous object



triangle1: Polygon

centre = (0, 0)
vertices = ((0,0), (4,0), (4,3))
borderColour = black
fillColour = white

triangle1: Polygon

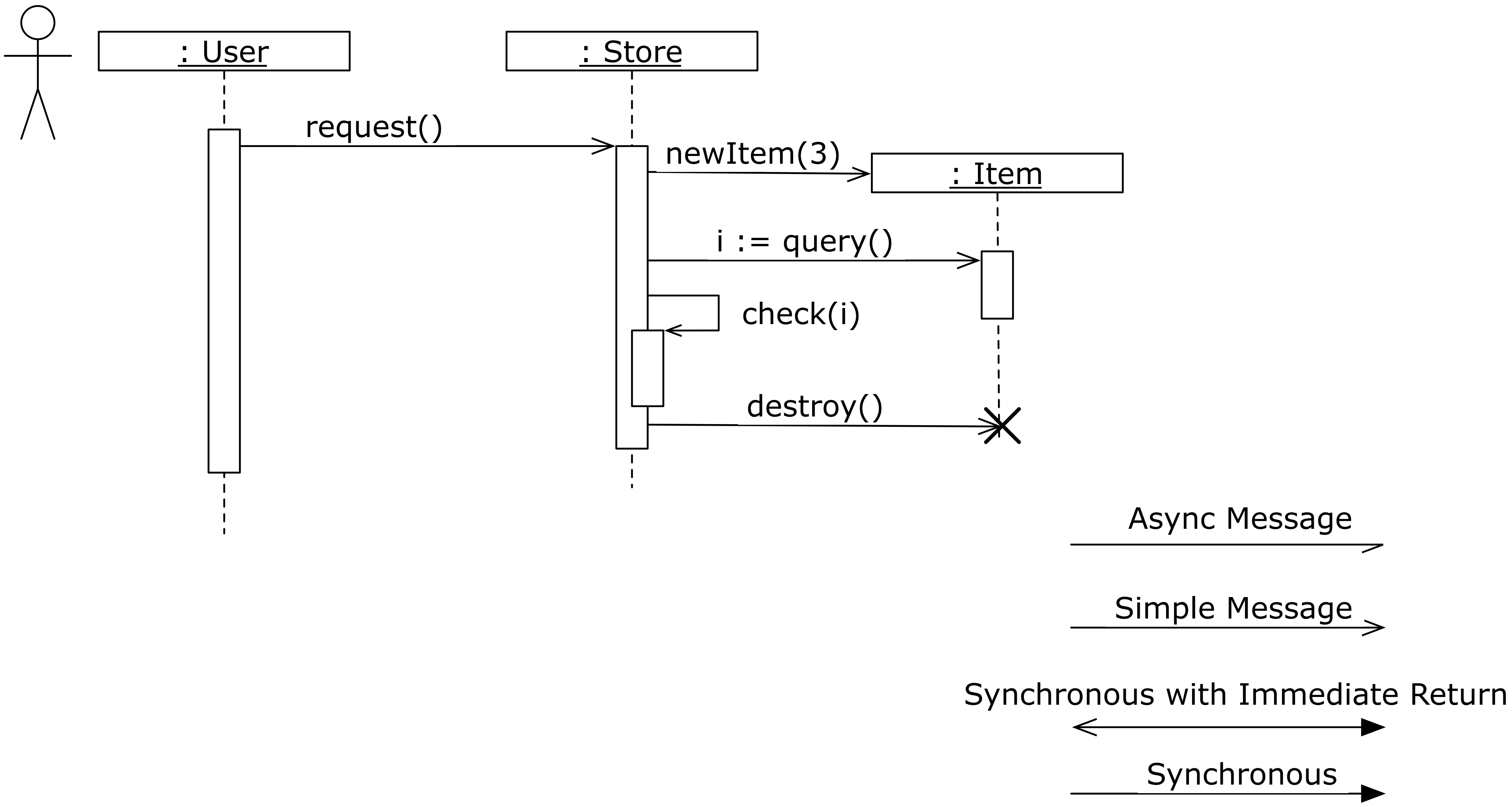
: Polygon

triangle1

Dynamic UML - Sequence Diagrams

Sequence Diagrams

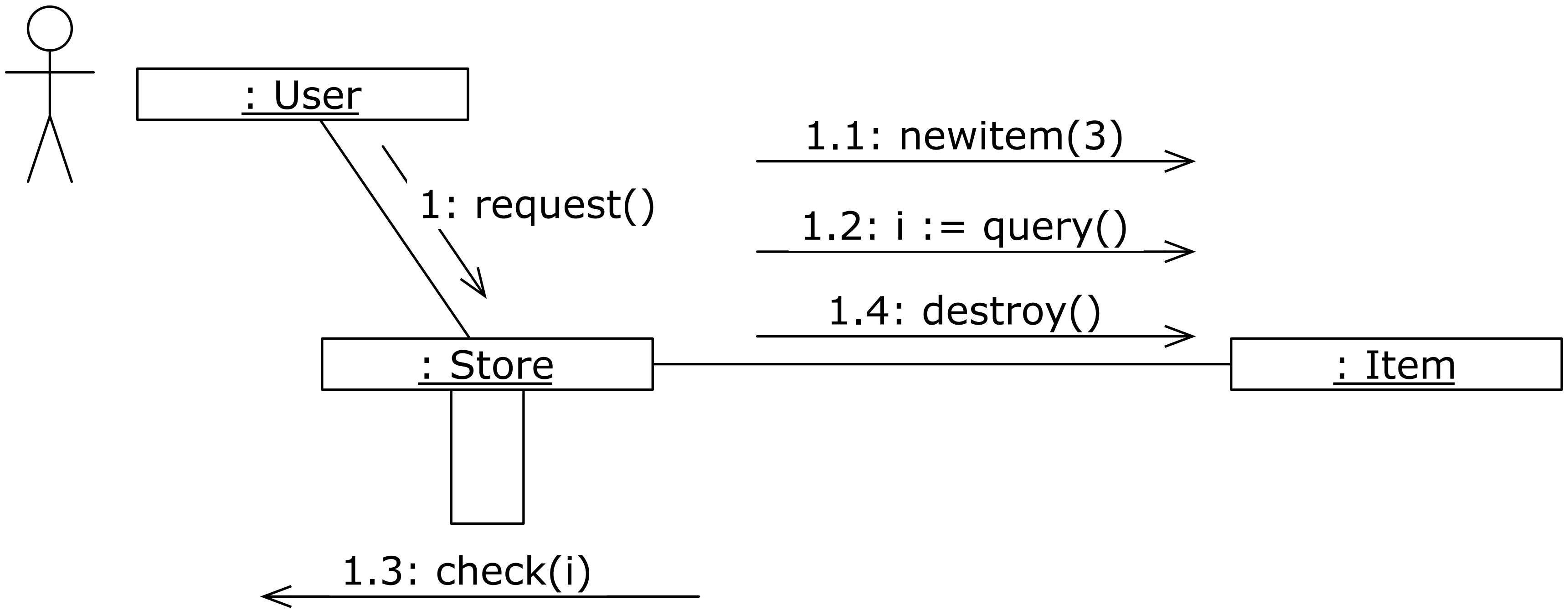
- Object at top, lifeline as dashed vertical line (time flows from top to bottom)
- Method execution as rectangle, message sends as arrow with message name
- Possibility to show concurrency via special arrowheads



Dynamic UML - Collaboration Diagrams

Collaboration Diagrams

- Objects with associations positioned freely in the diagram
- Messages with little arrows near to associations
- Message sequences follow from hierarchical numbering
- Expressibility is identical to sequence diagrams
  - + ⇒Freedom in lay-out but message sequence difficult to follow



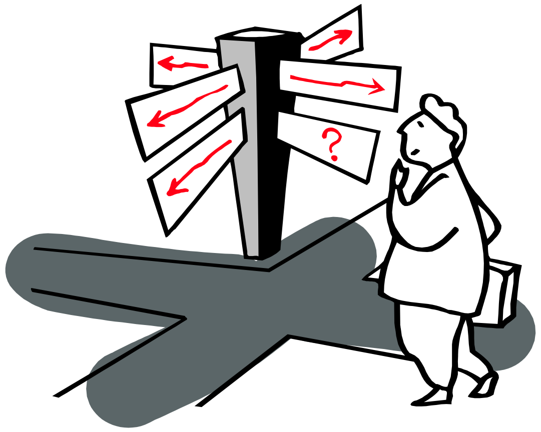
Summary (i)

- You should know the answers to these questions:
  - How does Software Engineering differ from programming?
  - Why is programming only a small part of the cost of a “real” software project ?
  - Give a definition for “traceability”.
  - What is the difference between analysis and design?
  - Explain verification and validation in simple terms.
  - Why is the “waterfall” model unrealistic? Why is it still used?
  - Can you explain the difference between iterative development and incremental development?
  - How do you decide to stop in the spiral model?
  - How do you identify risk? How do you asses a risk? Which risks require action?
  - What is Failure Mode and Effects Analysis (FMEA)?
  - List the 6 principles of extreme programming.
  - What is a “sprint” in the SCRUM process?
  - Give the three principal roles in a scrum team. Explain their main responsibilities.
  - Draw a UML class diagram modelling marriages in cultures with monogamy (1 wife marries 1 husband), polygamy (persons can be married with more than one other person), polyandry (1 woman can be married to more than one man) and polygyny (1 man can be married to more than one woman).
  - Draw a UML diagram that represents an object “o” which creates an account (balance initially zero), deposits 100$ and then checks whether the balance is correct.

Summary (ii)

- Can you answer the following questions?
  + What is your preferred definition of Software Engineering? Why?
  + Why do we choose “Correctness” & “Traceability” as evaluation criteria? Can you imagine some others?
  + Why is “Maintenance” a strange word for what is done during the activity?
  + Why is risk analysis necessary during incremental development?
  + How can you validate that an analysis model captures users’ real needs?
  + When does analysis stop and design start?
  + When can implementation start?
  + Can you compare the Unified Process and the Spiral Model?
  + Can you explain the values behind the Agile Manifesto?
  + Can you identify some synergies between the techniques used during extreme programming?
  + Can you explain how the different steps in the scrum process create a positive feedback loop?
  + How does scrum reduce risk?
  + Is it possible to apply Agile Principles with the Unified Process?
  + Did the UML succeed in becoming *the* Universal Modeling Language? Motivate your answer.

CHAPTER 2 – Requirements



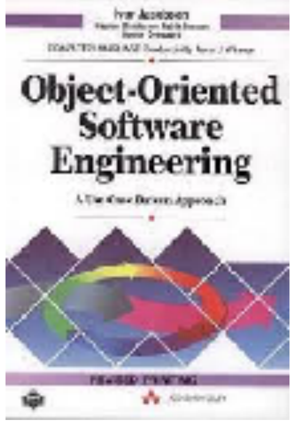
- Introduction
  + When, Why, Where, What
- Iteratively Developing Use Cases
  + Inception
    - Scope Definition + Risk Identification
    - Actors & Use cases + Project Plan
  + Elaboration
    - Primary & Secondary Scenarios
- Scrum: User Stories
    - Behaviour driven (template)
    - Conditions of Satisfaction
    - INVEST Criteria
      > Definition of Ready
    - Granularity: Epic / Features / Sprintable Stories
      > Product Roadmap
      > *Minimum Viable Product*
- Safety Critical
  + Misuse Cases
  + Safety Stories
- Conclusion
  + Use Cases / User Stories
    - Correctness & Traceability

Literature

Books

- [Ghez02], [Somm05], [Pres00]
  + Chapters on Specification/ (OO)Analysis/ Requirements Engineering

- Use Cases
  + [Schn98] Applying Use Cases - a Practical Guide, Geri Schneider, Jason, P. Winters, Addison-Wesley, 1998.
    - An easy to read an practical guide on how to iteratively develop a set of use cases and how to exploit it for project planning.
  + [Jaco92] Object-Oriented Software Engineering: A Use-Case Driven Approach, I. Jacobson et. al., Addison-Wesley, 1992.
    - The book that introduced use-cases



- User Stories
  + [Rubi13] Essential Scrum: A Practical Guide to the most popular agile process. Kenneth S.Rubin. Addison-Wesley, 2013.
    - The chapter on “Requirements and user stories”

Literature (Safety Critical)

Agile Requirements

- “User stories as lightweight requirements for agile clinical decision support development", Vaishnavi Kannan, et. al. In Journal of the American Medical Informatics Association, Volume 26, Issue 11, November 2019, Pages 1344–1354 https://doi.org/10.1093/jamia/ocz123
  + Mixing use cases with user stories in a medical (= safety-critical) context

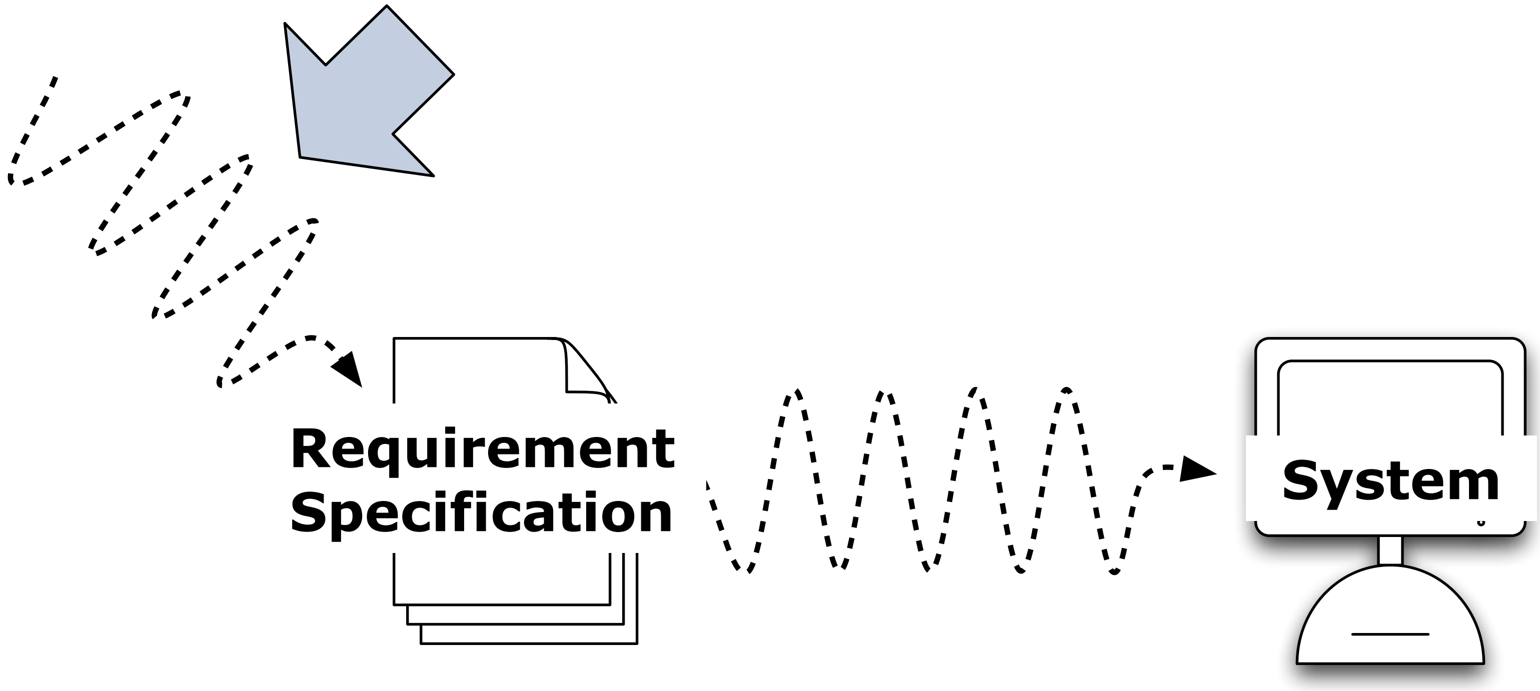
- *Making sense of MVP (Minimum Viable Product) – and why I prefer Earliest Testable/ Usable/Lovable*
  Posted on 2016-01-25 – 12:14 by Henrik Kniberg
  https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp
  + A plea for early feedback from actual users

Safety Critical

- Ian Alexander, "Misuse Cases: Use Cases with Hostile Intent" in IEEE Software, vol. 20, no. 01, pp. 58-66, 2003. doi: 10.1109/MS.2003.1159030
  + Explains a negative form of a use case; a negative scenario. Use- and misuse-case diagrams are valuable in threat and hazard analysis.

- Jane Cleland-Huang and Michael Vierhauser, "Discovering, Analyzing, and Managing Safety Stories in Agile Projects," 2018 IEEE 26th International Requirements Engineering Conference (RE), 2018, pp. 262-273, doi: 10.1109/RE.2018.00034.
  + Addresses the specific problems of discovering, analyzing, specifying, and managing safety requirements within the agile Scrum process.

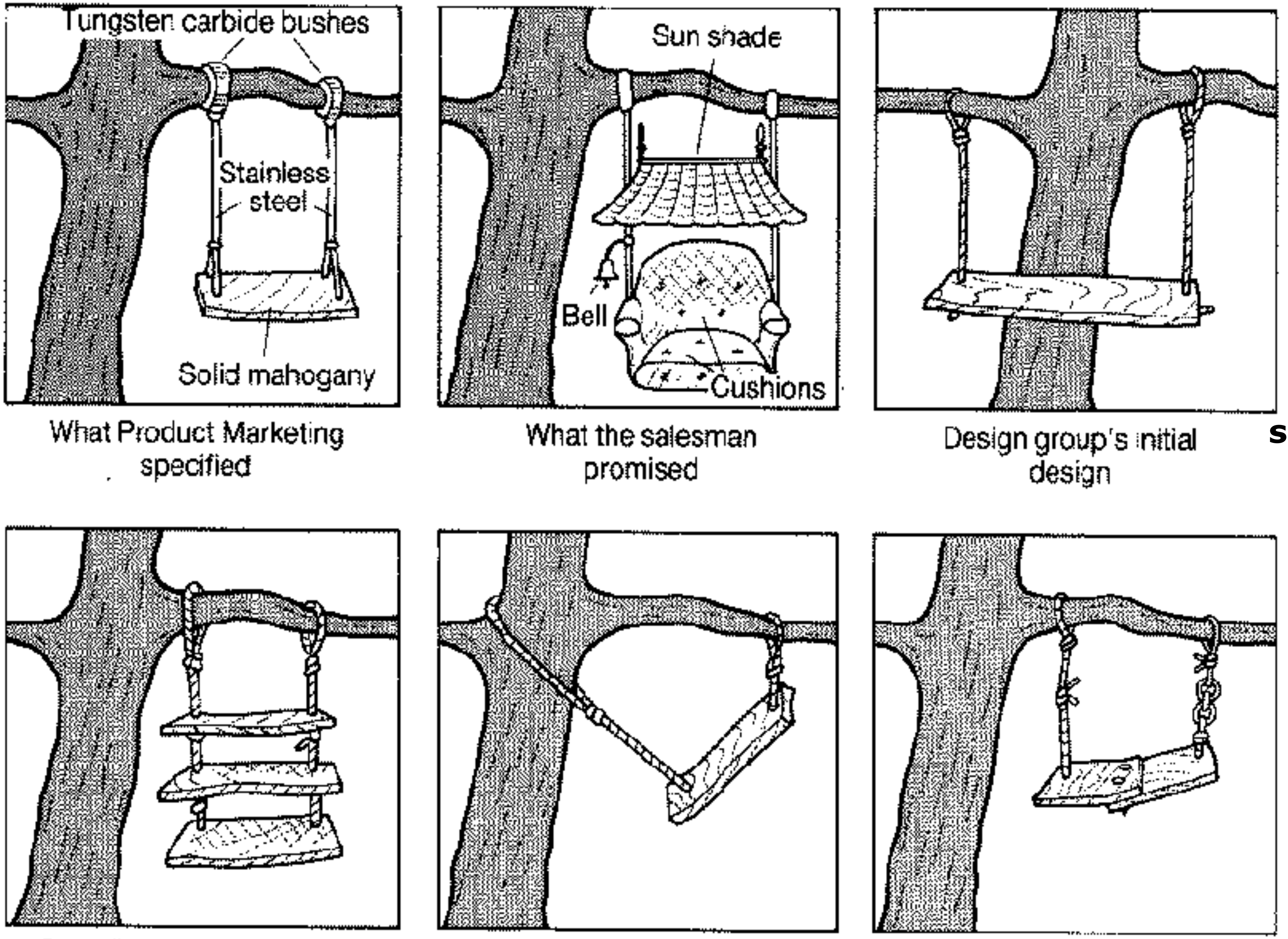
When Requirements?



A requirements specification must be

- *understandable*: so that we can decide what is in- and out scope
  + Do all parties agree?
- *precise*: so that parties agree what's inside and outside the system
  + Can you write an acceptance test for each requirement?
- *open*: so that developers have enough freedom to pick an optimal solution
  + Requirements specify the "what", not the "how ".

Why Requirements?



What Product Marketing specified

What the salesman promised

Design group's initial design

Corp. Product Architecture's modified design

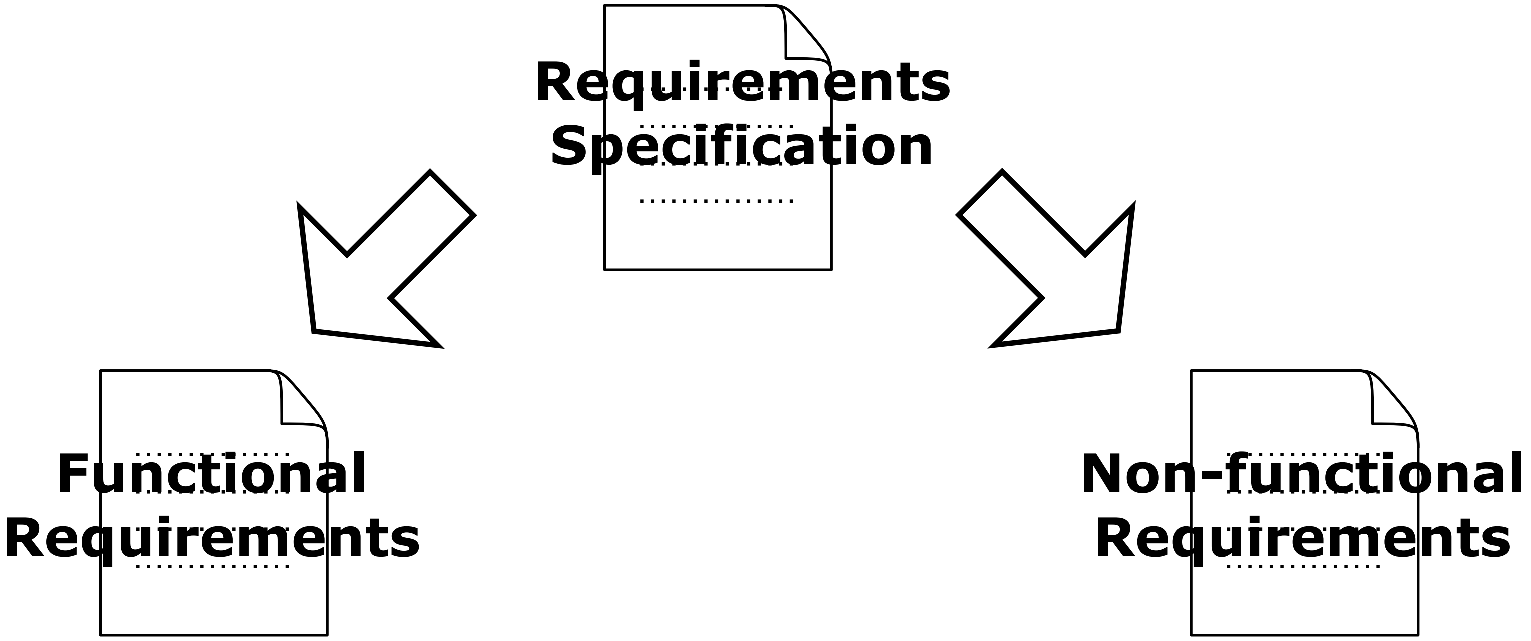
Pre-release version

General release version

Numerous stakeholders & Limited resources

© — source unknown

Where Requirements?

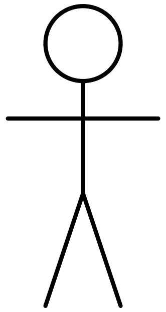


Requirements Specification

Functional Requirements

Non-functional Requirements

- functionality as demanded by the end users



- constraints placed on the global system or the development process.
- quality attributes, such as performance, user-friendliness, maintainability, …

What are Use Cases?

- Use Case
  - + A use case describes outwardly visible requirements of the system
  - + A use-case is a generic description of an entire transaction executed to achieve a *goal* (= the use case goal) and involving several *actors*.

- Actors
  - + Actors have responsibilities
  - + To carry out responsibilities, an actor sets goals
  - + Primary actor (= stakeholder) has unsatisfied goal and needs system assistance
  - + Secondary actor provides assistance to satisfy the goal

- Scenario
  - + Scenario = an instance of a use-case, showing a typical example of its execution
    - Use case = Primary “success” scenario and secondary “alternative” scenarios
    - Scenario shows how objects interact to achieve the use case goal = UML Sequence diagrams & Collaboration Diagrams

Kinds of Use Cases

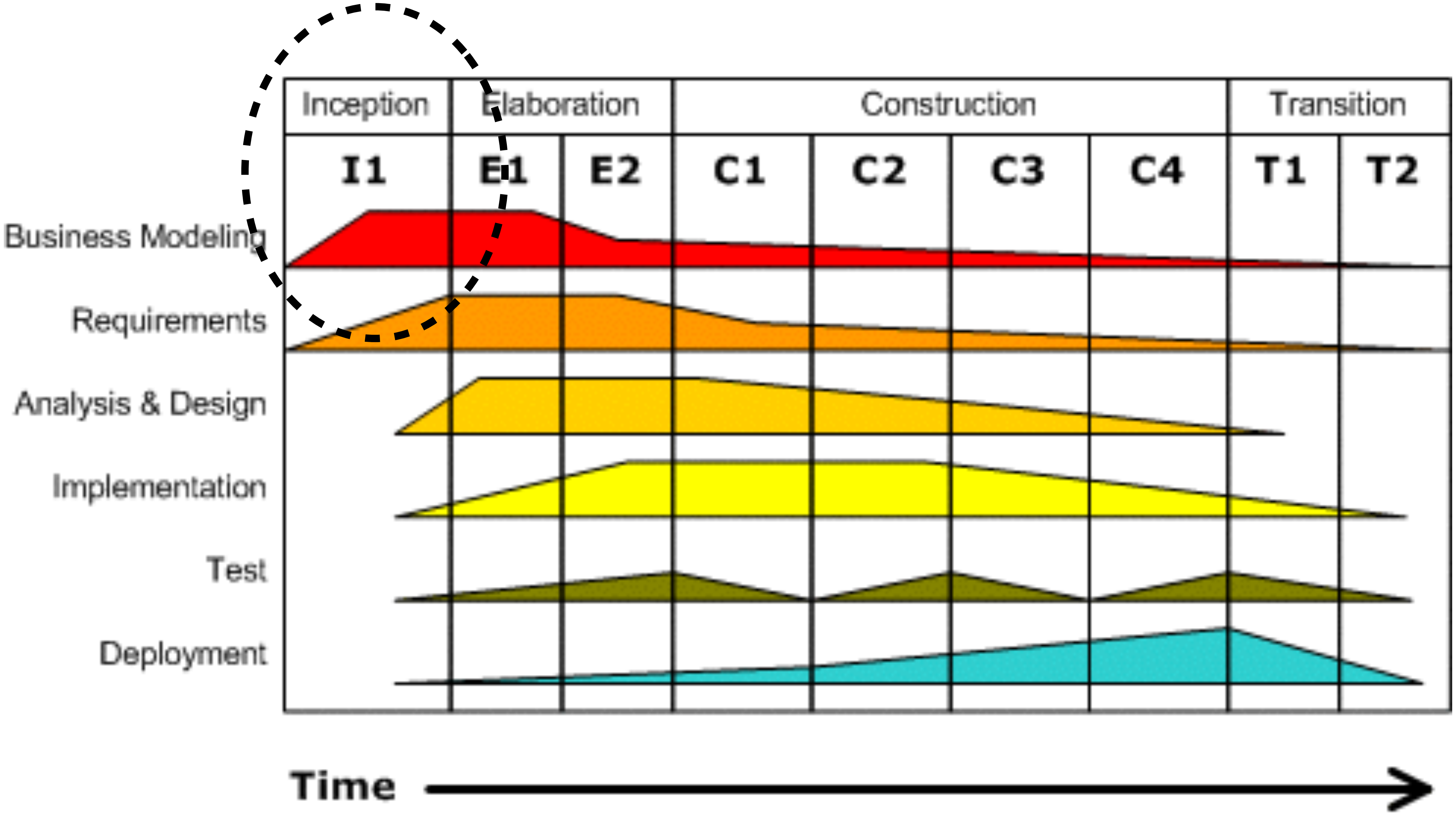
There is not a "one size fits all": use cases depend on your purpose

- Scope
  + Brain-storm mode vs. full-fledged detailed specification
- Intended Audience
  + end-user vs. system development team vs. internal documentation
- Granularity
  + summary vs. detailed; overall system function vs. specific feature
  + Brief Use Case — Casual Use Case — Fully Dressed Use case
- Black-Box vs. White-Box
  + with or without knowledge about (business) processes used to achieve goal

Depending on your purpose some kind of Use Case Template is selected

Unified Process: Inception

Use cases work well when starting an iterative/incremental process!
+ e.g. *inception* & elaboration phase in Unified Process



DutchGuilder Wikipedia

Inception: System Scope

+ During inception you must define the system's scope
  - used to decide what lies inside & outside the system

- Scope
  + should be *short*
    - (1 paragraph for small projects;
      1/2 a page for mid-size projects;
      2-3 pages for large projects)
      * long statements are not convincing

  + should be *written* down
    - later reference when prioritizing use cases

  + should have end-user *commitment* (*)
    - end-user involved in writing
      * formally approved by a project steering committee

(*) The difference between "involvement" and "commitment"? In a Ham and Egg Breakfast...the chicken is involved and the pig is committed!

System Scope: Example

- (Example from [Schn98])

  + "We are developing order-processing software for a mail-order company called National Widgets, which is a reseller of products purchased from various suppliers.
    - Twice a year the company publishes a catalogue of products, which is mailed to customers and other interested people.
    - Customers purchase products by submitting a list of products with payment to National Widgets. National Widgets fills the order and ships the products to the customer's address.
    - The order-processing software will track the order from the time its is received until the product is shipped.
    - National Widgets will provide quick service. They should be able to ship a customer's order by the fastest, most efficient means possible."

Evaluate the previous scope description

1. Summarise the purpose of the system in a single word

2. What quality criteria are important?

3. What is clearly outside scope?

4. (Technical) opportunities to improve?

Analyzing the Example

- The previous example of a system scope description is
  + short (1/2 a page)
    * quick assessment of what’s the system supposed to do
  + goal-oriented (track orders)
    * open for various solutions
  + includes criteria (*quick* service, track *all* of the ordering process, …)
    * will be used to evaluate whether we accomplished the goals
  + provides context
    - National Widgets is reseller ⇒ *external* suppliers & shipment
    * the system will not solve everything,
      some problems are out of scope

- … and very importantly
  + imperfect (*twice* a year? *on-line* catalogue?)
    - may be improved when understanding increases
    - … but goal and main criteria should not change once approved

Inception: Risk Factors

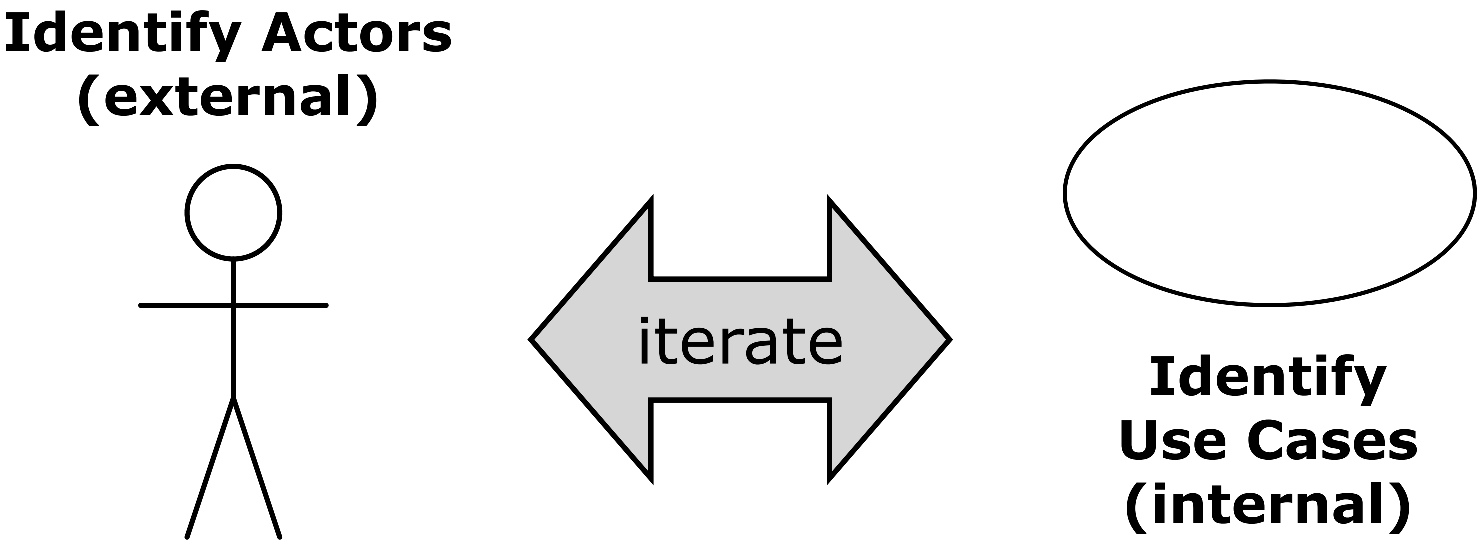
- During inception you must identify the project’s *risk factors*
  + you do not have control over the system’s context and it will change
  + projects never go according to plan
  > identify potential problems early (… including wild success)
- Example

Context	Risk Factors	Impact	Likely	Urgency
Competitors	Time to market (too late/too early)			
Market trends	More internet at home			
Potential disasters	Suppliers don’t deliver on time			
	System is down			
Expected users	Too many/few users			
Schedule	Project is delivered too early/too late	<< Risky Path (Project Management)		
Technology	Dependence on changing technology			
	Inexperienced team			
	Interface with legacy systems			

Inception: System Boundaries

During inception you must specify the system boundaries

- what functionality is *internal* to the system (= use cases)
- what functionality is *external* but necessary for internal functionality (= actors)
  - + ⇒ the distinction is often not as clear as you would like it
  - + ⇒ iterate: identifying actors + identify use cases



Identify Actors (external)
iterate
Identify Use Cases (internal)

At least one actor must benefit from the use case (i.e. sees the use case value). The corresponding stakeholder will argue to keep the use case in the requirements!

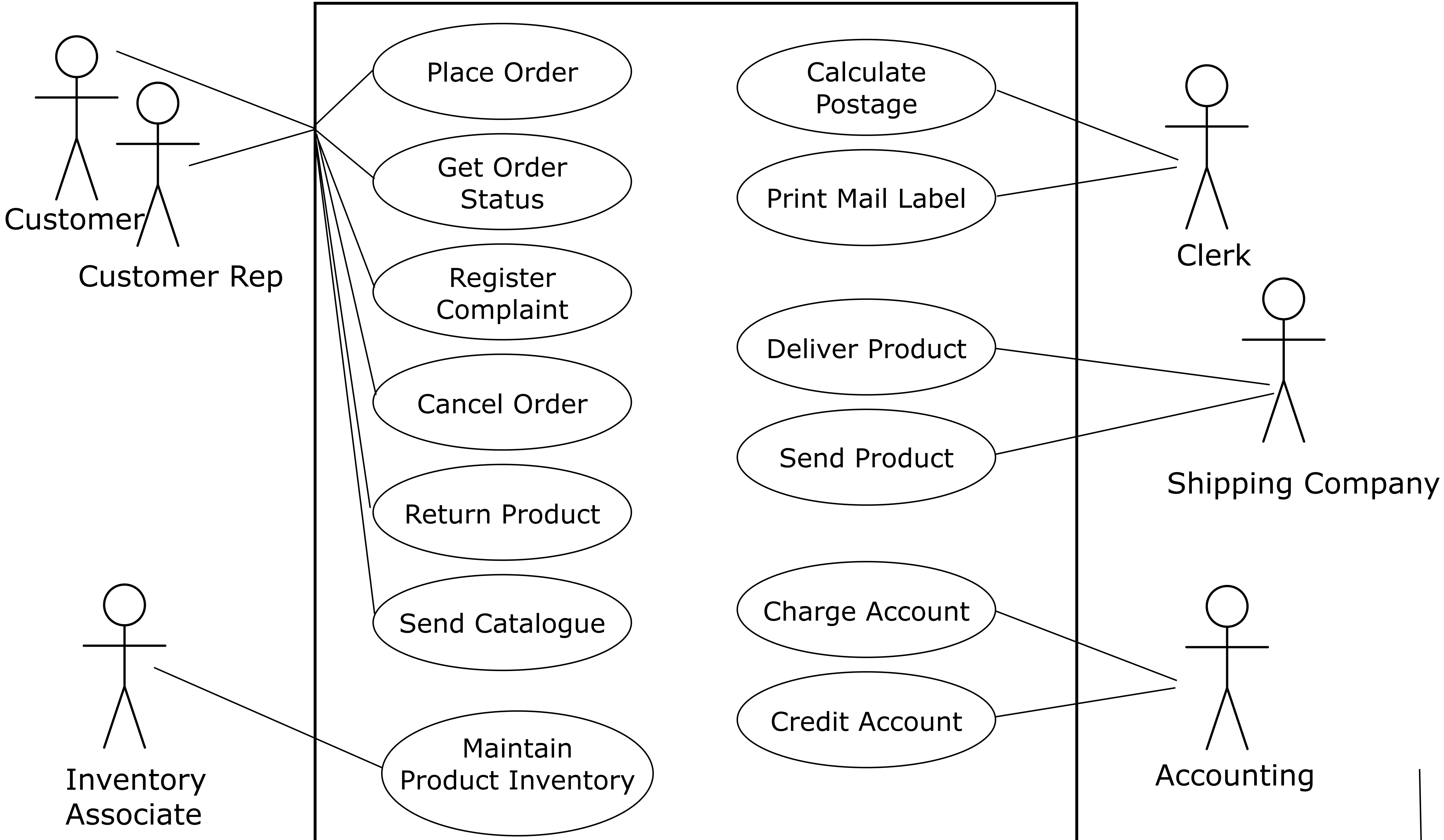
Identifying Actors & Use cases

- Actors
  - + Who uses the system?
  - + Who installs the system?
  - + Who starts up/shuts down the system?
  - + Who maintains the system?
  - + What other systems use this system?
  - + Who provides information to this system?
  - + Does anything happen automatically at a preset time?

- Use Cases
  - + What functions will the actor want from the system?
  - + What actors will create, read, update, or delete information stored inside the system?
  - + Does the system need to notify actors about changes in its internal state?
  - + Who gets information from this system?
  - + Are there any external events the system must know about?
  - + What actor informs the system about those events?

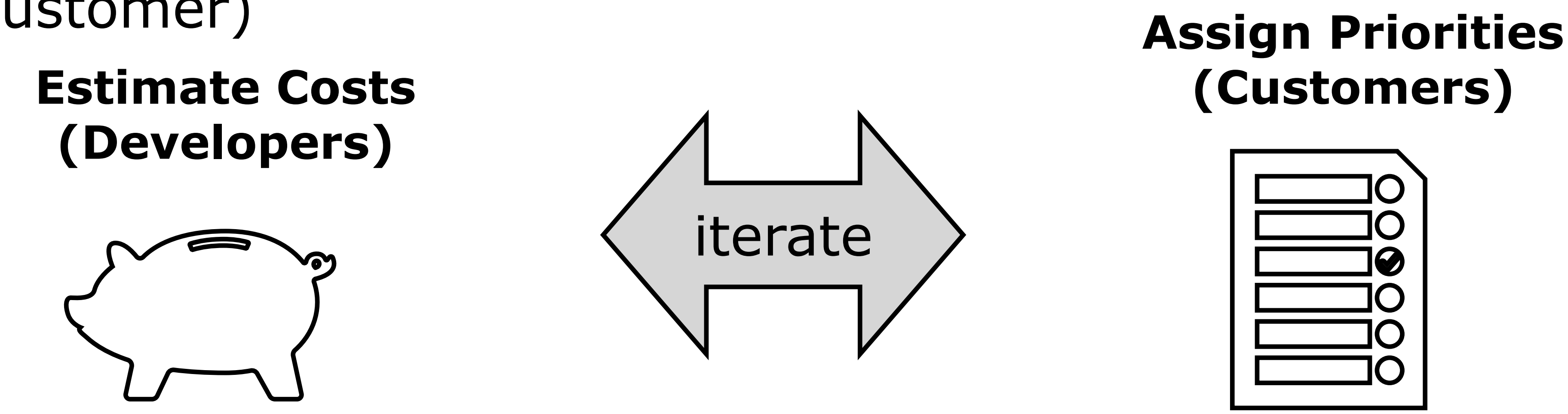
Above questions may help during the identification process.

Example: Actors & Use cases



Inception: Project Plan

- During inception you must specify the project plan
  = when to develop which use case
  + Includes intermediate milestones
  + based on Scope Definition & Risk Factors
  + may result in splitting/merging use cases
  + negotiate: estimate costs (=developer) + assign priorities (=customer)



Estimate Costs (Developers)

iterate

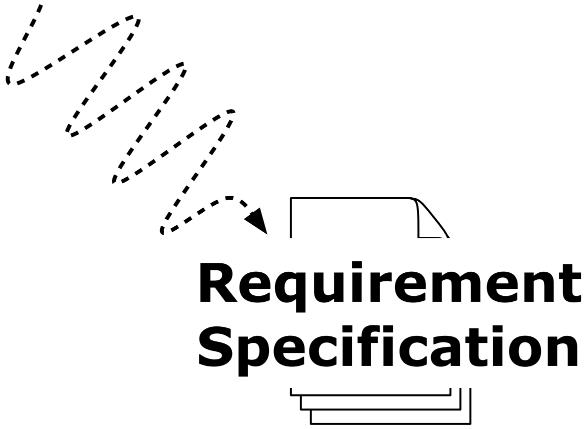
Assign Priorities (Customers)

- Good negotiations obey 2 strict rules
  + *Developers estimate cost*; customers do not interfere.
    > Schedule slips are the responsibility of development team.
  + *Customers assign priorities*; developers do not interfere.
    > Deciding where the money is spent is the customers responsibility.

Terminating the Inception Phase

After inception the requirements specification consists of

- Scope definition
  + Short description involving goals, criteria, context
- Risk Factors
  + Events that may cause problems during project
- Actors
  + Represent the various stakeholders in the project
- Use cases
  + Represent transactions; valuable for at least one actor
- Project Plan
  + For each use case
    - Cost estimate (assigned by development team)
    - Priority (assigned by customers)
  + Time plan including intermediate milestones

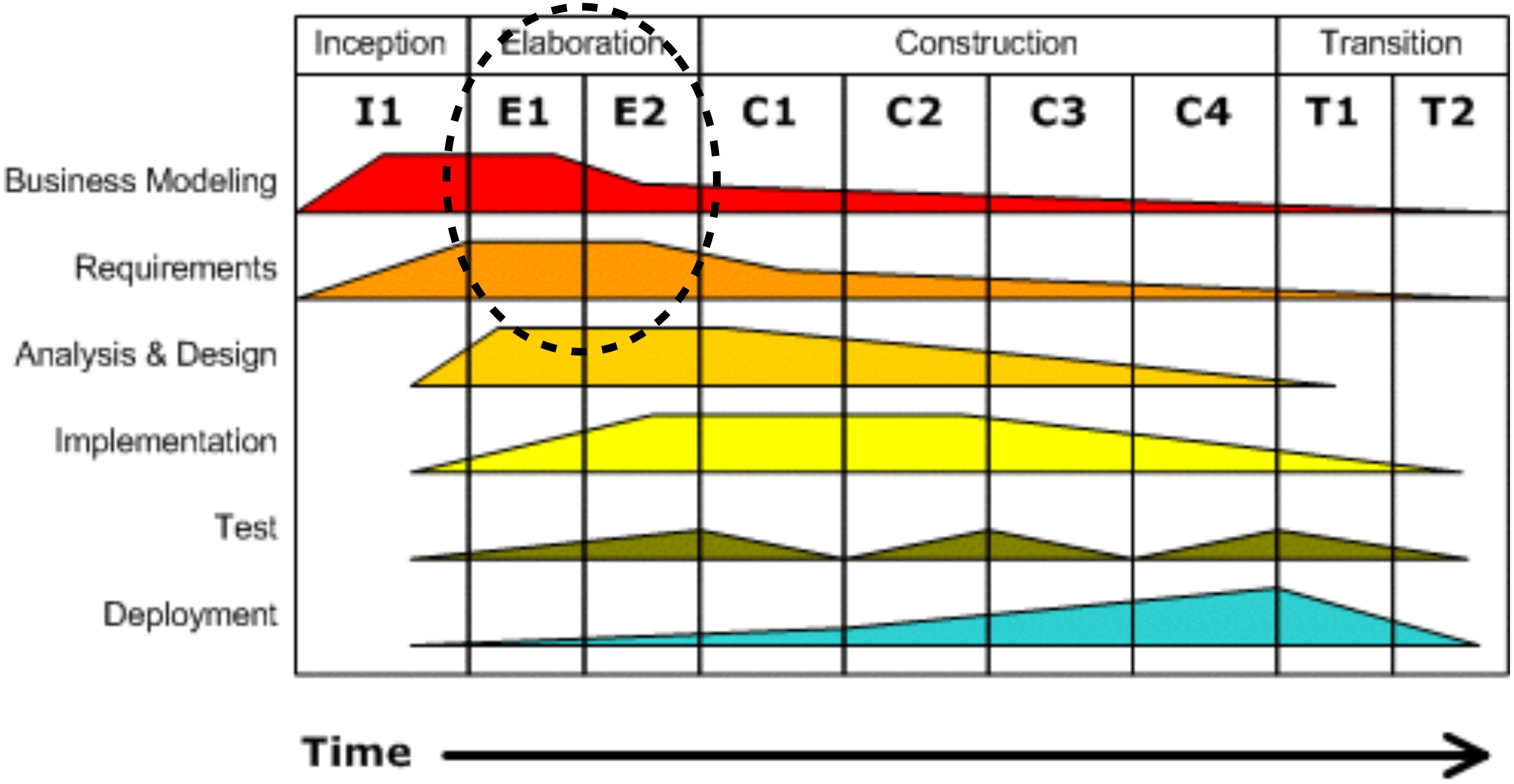


Formal approval by project steering committee

Unified Process: Elaboration

Use cases work well when starting an iterative/incremental process!

+ e.g. *inception* & elaboration phase in Unified Process



DutchGuilder Wikipedia

Elaboration: Primary & Secondary Scenarios

During elaboration you must refine the use cases via scenarios

- Scenario is one way to realize the use case
  - From the actors point of view!
- = List of steps to accomplish the use case goal

- Primary “success” scenario
  + = Happy day scenario
  + Scenario assuming everything goes right
  (i.e., all input is correct, no exceptional conditions, …)
- Secondary “alternative” scenarios
  + Scenario detailing what happens during special cases
  (i.e., error conditions, alternate paths, …)

Example: Place Order Scenario (1/2)

USE CASE 5	Place Order
Goal in Context	Customer issues request by phone to National Widgets; expects goods shipped and to be billed.
Scope & Level	Company, Summary
Preconditions	National Widgets has catalogue of goods
Success End Condition	Customer has goods, we have money for the goods.
Failed End Condition	We have not sent the goods, Customer has not spent the money.
Primary Actors	Customer, Customer Rep, Shipping Company
Secondary Actors	Accounting System, Shipping Company
Trigger	Purchase request comes in.

DESCRIPTION

Step	Action
1.	Customer calls in with a purchase request.
2.	Customer Rep captures customer info.
3.	WHILE Customer wants to order goods.
3.1.	Customer Rep gives Customer info on goods, prices, etc.

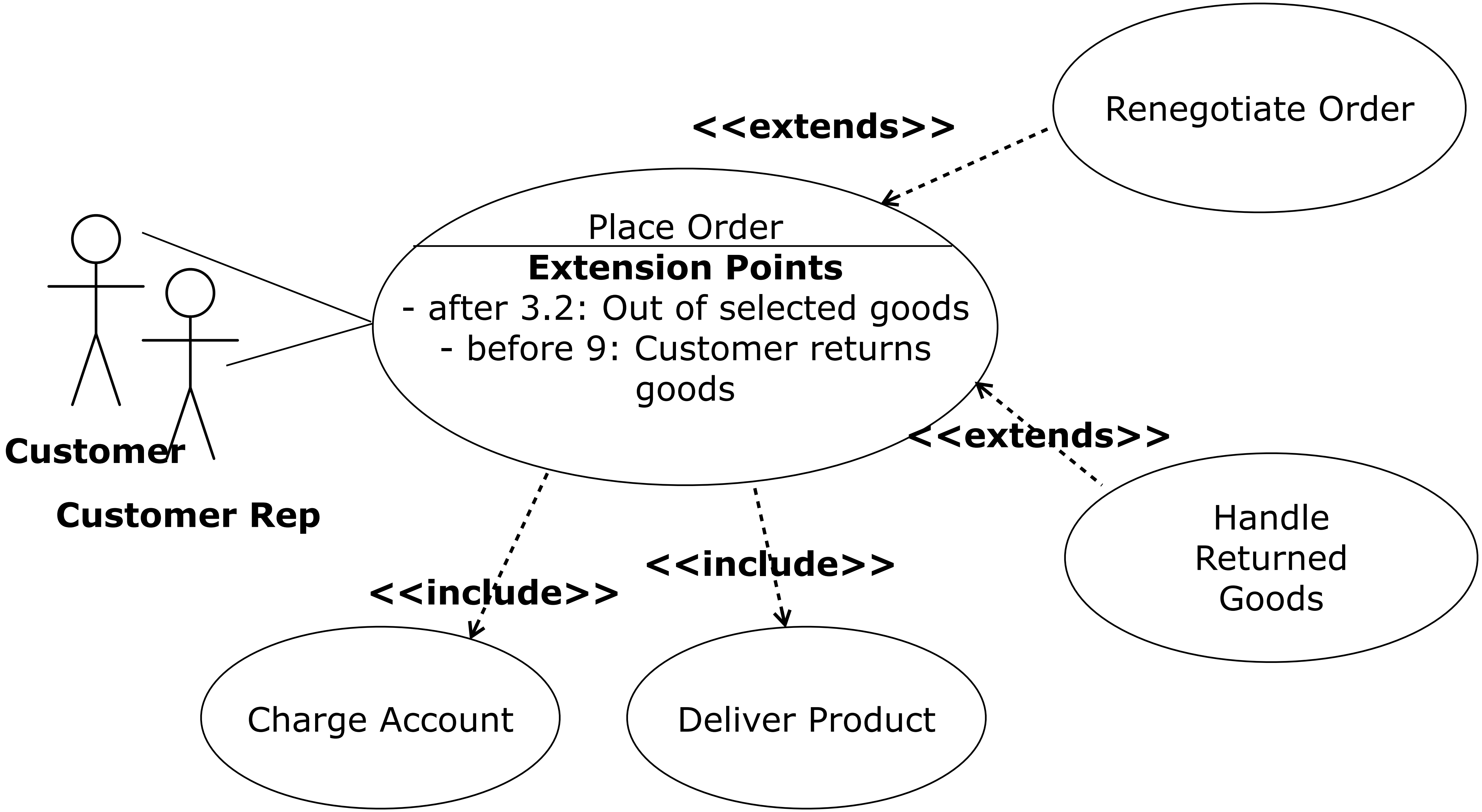
Example: Place Order Scenario (2/2)

3.2.	Customer selects good to add to order list.
4.	Customer approves order list.
5.	Customer supplies payment details.
6.	Customer Rep creates order.
7.	Customer Rep requests Accounting System to Charge Account.
8.	Customer Rep requests Shipping Company to Deliver Product.
9.	Customer pays goods.
Branch	SUBVARIATIONS
1.	Customer may use: (a) phone in, (b) fax in, (c) use web order form.
4.	Customer may pay via: (a) credit card; (b) cheque; (c) cash.
Branch	ALTERNATIVE PATHS
any	Customer may cancel transaction.
Branch	EXTENSIONS
After 3.2	Out of selected good: 3.2.a. Renegotiate Order (Use case 44).
Before 9	Customer returns goods: 9a. Handle returned goods (Use case 45).

Place Order Use Case Diagram

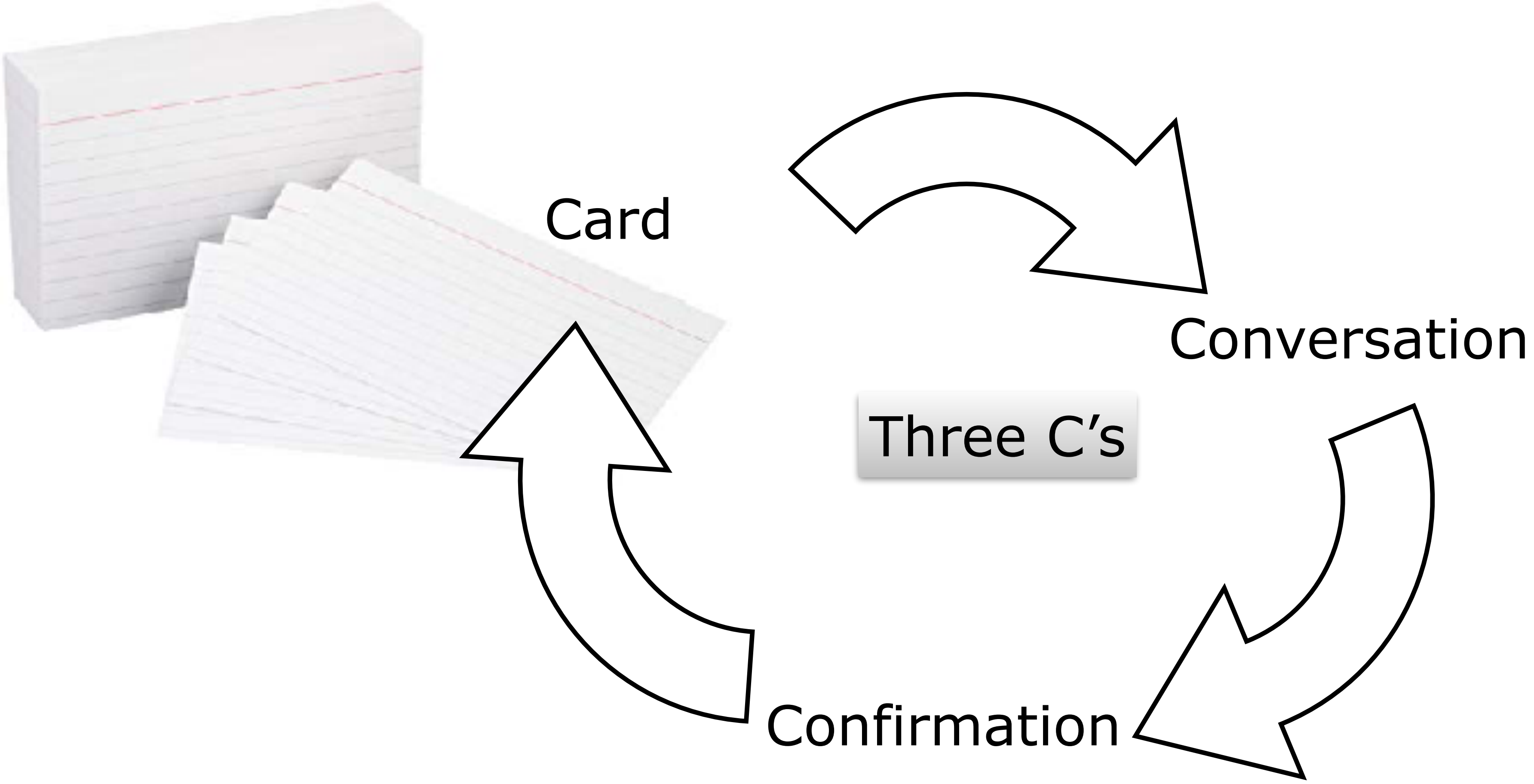
Stereotypes <<extends>> and <<include>> to specify use case relationships.

- Beware the direction of the arrows; it specifies change dependencies!



Scrum: User Stories

Agile Manifesto: We value customer collaboration over contract negotiation.



Behaviour Driven (User Stories)

Template

As a <user role>
I want to <goal>
so that <benefit>.

- …
- …
- …
-

Conditions of Satisfaction

Example

As a *clerk*
I want to *calculate stampage*
so that *goods get shipped fast*.

- Verify with nearby address
- Verify with overseas address
- Verify with parcels <= 1kg
- Verify with fragile parcel

As an *inventory associate*
I want to *minimise stock*
so that *we save warehouse costs*.
(See book of Müller for heuristics)

- References / explanations allowed
  + Conversation
- Coarse grained
  + break down in smaller chunks

INVEST Criteria

I	Stories should be *independent* of another and should not have dependencies on other stories
N	*Negotiable*: Too much detail on story limits conversation with the customer
V	Each story has to be of *value* to the customer
E	Stories should be small enough to *estimate*
S	Stories should be *small* enough to be completed in one iteration
T	*Testable*: Acceptance criteria should be available

What & Why
(not How)

Technical Stories?

CAPSTONE PROJECT

Independent vs. Interdependent

I	Stories should be *independent* of another and should not have dependencies on other stories

Desirable but ...
not as easy as it seems!

- Some user stories depend on one another for technical reasons.
  + e.g.: Storing items in a database before one can retrieve.
  > Work-around: dummy records
- Early stories take longer to implement
  + Creating the technical infrastructure
- Circular dependencies are a nightmare
  + Split the story in smaller chunks

Technical Stories

Technical stories express non-functional requirements in story form.

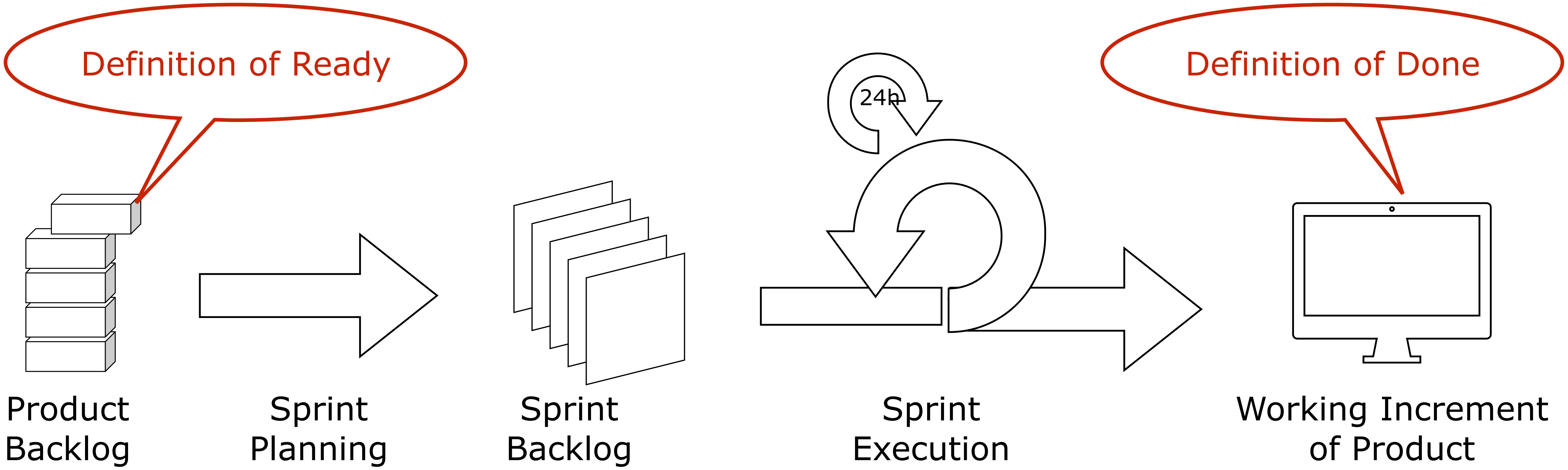
As a *developer*
I want to *migrate to Oracle version 3.2*
so that *we are not operating on a version that soon will retire*.

As a *security engineer*
I want to *ensure that HTTP, Radius SecureID, and LDAP authentication protocols are adhered to*
so that *we avoid backdoor attacks.*

Value?

- **Educate your product owners**
- **Calculate risk (Business case)**

Two sides of the same coin



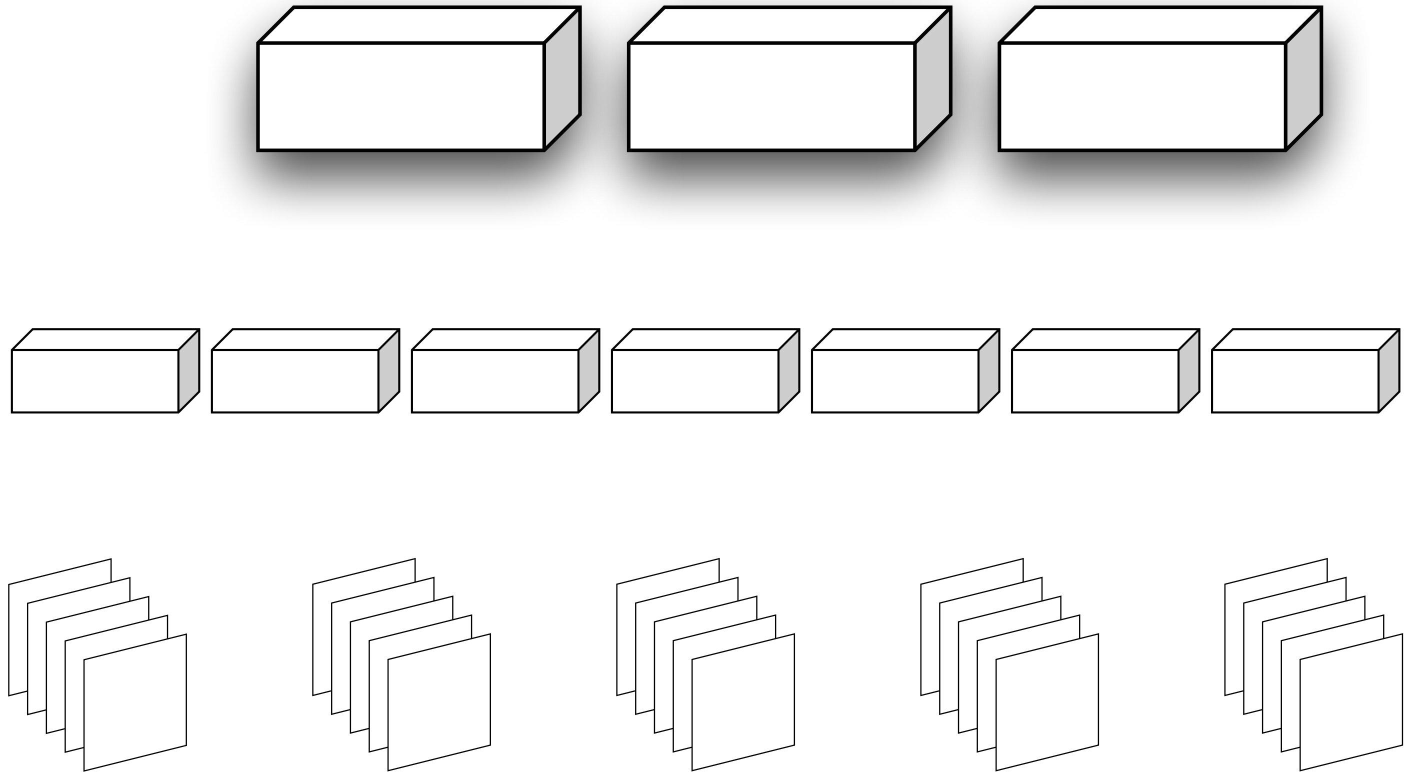
Definition of Ready

definition of ready = a checklist of the properties that must be satisfied before an item in the product back log can be moved into a sprint.

Example “Definition of Ready” checklist

- ✓ Business value is clearly articulated.
- ✓ Details are sufficiently understood by the development team so it can make an informed decision as to whether it can complete the Product Backlog Item.
- ✓ Dependencies are identified and no external dependencies would block the Product Backlog Item from being completed.
- ✓ Team is staffed appropriately to complete the Product Backlog Item.
- ✓ The Product Backlog Item is estimated and small enough to comfortably be completed in one sprint.
- ✓ Acceptance criteria are clear and testable.
- ✓ Performance criteria, if any, are defined and testable.
- ✓ Scrum team understands how to demonstrate the Product Backlog Item at the sprint review.

Product Backlog — Level of Detail

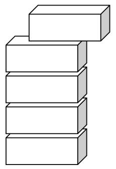
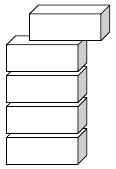
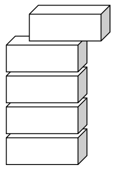


Epic	Months	Bigger than a release
Features	Weeks	Bigger than a sprint
Sprintable Stories	Days	Sprint Ready

CAPSTONE PROJECT

Product Roadmap (a.k.a. Release Roadmap)

Product Roadmap: communicates the incremental nature of how the product will be built and delivered over time, along with the important factors that drive each individual release.

	Q3—2020	Q4—2020	Q1—2021	Q2—2021
Market Map	Market Study	Advertising	Launch	
Market Events			CEBIT Fair	
Feature Map	…	…	…	
Release Plan				
Release Schedule	0.5	0.7	1.0	

Epic

New

Minimum Viable Product

A minimum viable product (MVP) is a version of a product with just enough features to be usable by early customers who can then provide feedback for future product development.

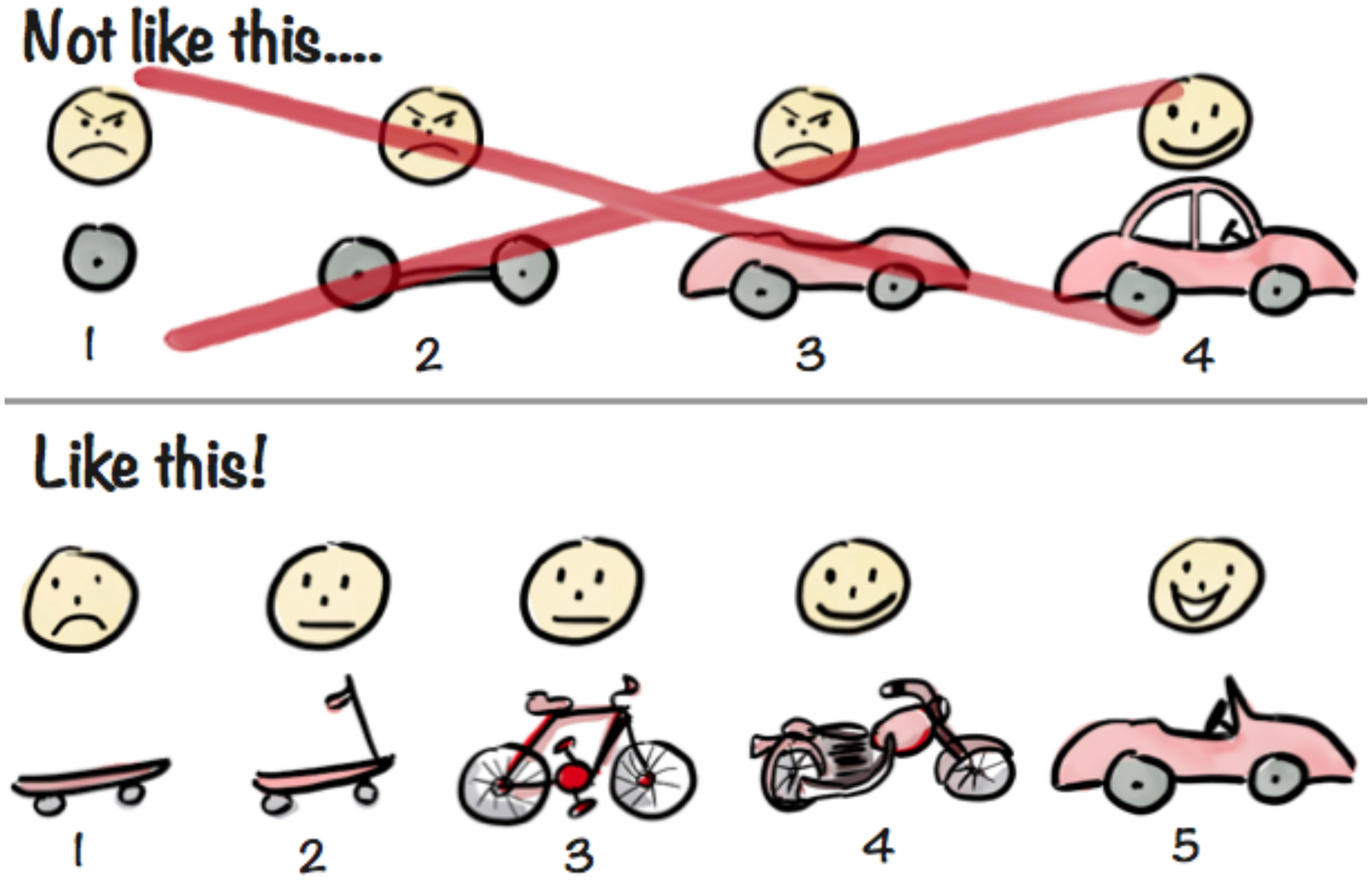
> First milestone for start-ups!

	Q3—2020	Q4—2020	Q1—2021	Q2—2021
Market Map	Market Study	Advertising	Launch	
Market Events			CEBIT Fair	
Feature Map	...	...	...	
Release Plan				
Release Schedule	0.5	0.7	1.0	

Minimum Viable Product: Iterations

New

Making sense of MVP (Minimum Viable Product) – and why I prefer Earliest Testable/Usable/Lovable
Posted on 2016-01-25 – 12:14 by Henrik Kniberg
https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp



Minimum Viable Product: Exercise

New

You an your neighbour are launching a start-up company with a mission to create a revolutionary new product.

- An on-line system for sharing course material in a post covid era (a.k.a. “Blackboard on steroids”)

Which iterations for your minimum viable product do you see?

Failure Mode and Effects Analysis (Example)

Potential Failure Mode	Potential Effects of Failures	Severity	Potential Causes of Failures	Current Process Control	Occurrence (± Likelihood)	Detection (± Urgency)	Critical (± Impact)	Risk Priority Number	Recommended Actions
Function: Dispense Fuel									
Does not dispense fuel	- Customer Dissatisfied - Discrepancy in bookkeeping	8	- Out of fuel - Machine jams - Power failure	- Out of fuel alert - Machine jam alert - none					
Dispense too much fuel	- Company loses money - Discrepancy in bookkeeping	8	- Sensor defect - Leakage	- none - pressure sensor					
Takes too long to dispense fuel	- Customer annoyed	3	- Power outage - Pump disrupted	- none - none					

Slide Adapted from Intro

We must be able to express *"can never happen"* scenarios

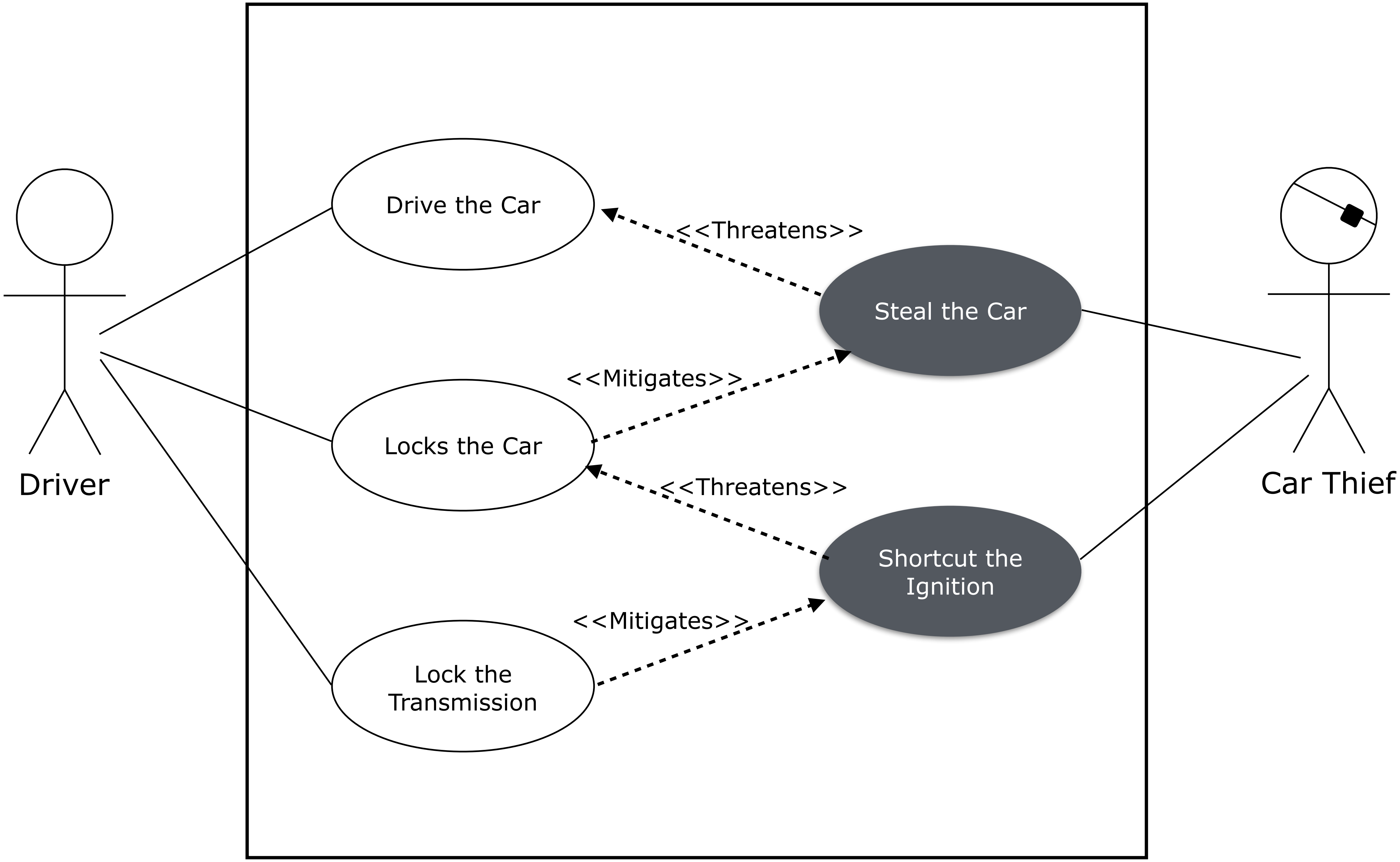
Misuse Cases

- = a use case from the point of view security of an actor hostile to the system under design.
  + Results from a Failure Mode and Effects Analysis (FMEA)

Adds extra items to a use case diagram

+ Misuse case (coloured black)
+ Negative actor (marked somehow)
+ "Threatens" relationship
  - Between misuse case and ordinary use case
  - ≈ Potential causes of failures (FMEA analysis)
+ "Mitigates" relationship
  - Between misuse case and ordinary use case
  - ≈ Current Process Control (FMEA analysis)

Misuse Case (Example)



© Adapted from Ian Alexander, "Misuse Cases: Use Cases with Hostile Intent"

Safety Stories

- = if satisfied, will prevent a hazard from occurring or reduce the impact of its occurrence

Extended template for Safety Stories (Easy Requirements Syntax — EARS)

- Ubiquitous: The <component name> shall <response>
- Event Driven: When <trigger> the <system name>
- State Driven: While <in a specific state>
  the <system name> shall <system response>
- State Option: Where <feature is included> the <system name> shall <system response>
- Unwanted Behavior: If <optional preconditions> <trigger>, then the <system name> shall <system response>

Safety Stories (Example)

Several variants of stories

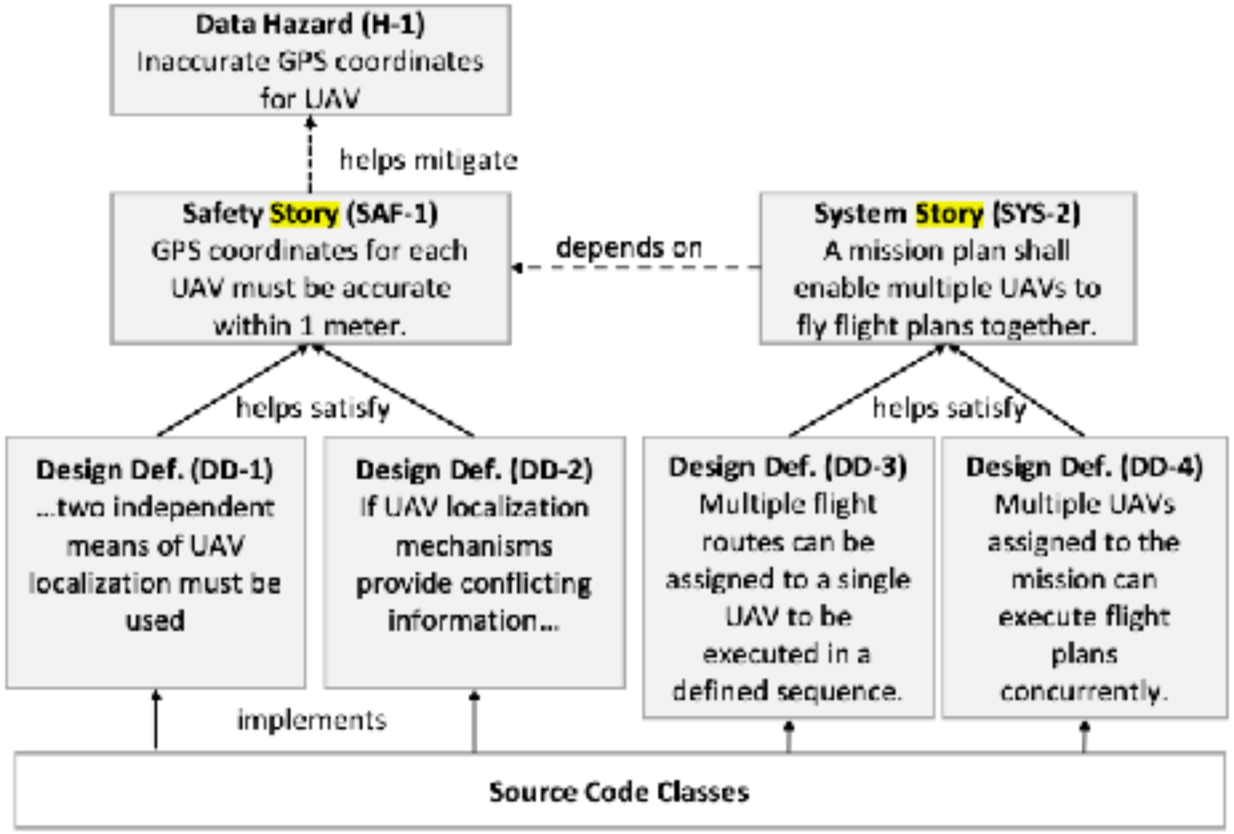
- System Story (SYS-1): A UAV (unmanned aerial vehicle) shall maintain a minimum separation distance from other UAVs at all times.

- Data Hazard (H-1): Inaccurate GPS (Global Positioning System) coordinates for UAV.
  + Failure Mode: GPS provides inaccurate readings.
  + Effect: Violation of minimum separation distance between two UAVs goes undetected, and UAVs collide in midair and then crash onto bystanders.
  + Level: Critical

- Safety Story (SAF-1): The GPS coordinates of each UAV must be accurate within one meter at all times.

- Design Definition (DD-1): When the Dronology system is deployed in an urban environment at least two independent means of UAV localization must be used.

Establish traceability links!



Data Hazard (H-1) Inaccurate GPS coordinates for UAV

helps mitigate

Safety Story (SAF-1) GPS coordinates for each UAV must be accurate within 1 meter.

depends on

System Story (SYS-2) A mission plan shall enable multiple UAVs to fly flight plans together.

helps satisfy

helps satisfy

Design Def. (DD-1) ...two independent means of UAV localization must be used

Design Def. (DD-2) If UAV localization mechanisms provide conflicting information...

Design Def. (DD-3) Multiple flight routes can be assigned to a single UAV to be executed in a defined sequence.

Design Def. (DD-4) Multiple UAVs assigned to the mission can execute flight plans concurrently.

implements

Source Code Classes

© Adapted from Jane Cleland-Huang et. al, "Discovering, Analyzing, and Managing Safety Stories in Agile Projects,"

Conclusion

Use cases / User Stories help you to specify good requirements because it is easier to make them … (a) understandable; (b) precise; and (c) open.

	Use Cases	User Stories
Understandable	Actors provide an end users perspective	<User Roles> provide an end users perspective
Precise	Scenarios are sufficiently detailed to test (path coverage)	“Conditions of Satisfaction“ ⇒ test scenarios
Open	Actors perspective emphasizes the what (and much less the how)	The INVEST criteria

But there is no guarantee, it still requires

- close interaction with various *stakeholders*
- iteration to improve earlier misconceptions
- … and lots of hard work.

Requirements & Correctness

- Are we building the system right?

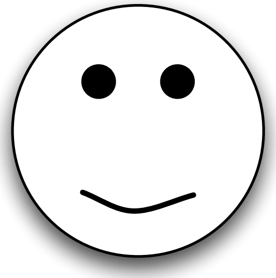
  + Good requirements will help to validate solution against requirements.

  - Testing
    Writing black box regression tests should be easy.

  + … however, step to system design (architecture)

    + detailed design (objects) is hard.

  - Use cases & User Stories tend to result in hard to maintain systems



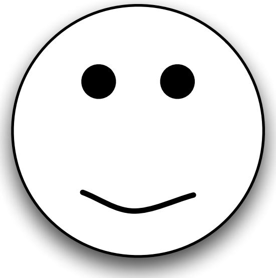
- Are we building the right system?

  + Good requirements should be easy to verify

  - Understandable & precise

  + … however

  - we may omit requirements
    - * Completeness is not guaranteed!
    - * Complementary Failure Mode and Affect Analysis (FMEA)
  - Focus on scenarios restricts evolving requirements
    - * Requirements should specify “what” not “how”



Requirements & Traceability

- Requirements ⇔ System 😐

  + Via proper naming conventions
    - … including names of regression tests

- Requirements ⇔ Project Plan 🙂

  + Use cases & User Stories form good milestones
    - Less so for misuse cases and safety stories
  + Estimating development effort is feasible
    - Balancing Numerous stakeholders
      against Limited resources

Use cases form a good base for negotiating the project plan.
User Stories (epics) form a good basis for negotiating the product roadmap.

Summary(i)

- You should know the answers to these questions
  + Why should the requirements specification be understandable, precise and open?
  + What’s the relationship between a use case and a scenario?
  + Can you give 3 criteria to evaluate a system scope description? Why do you select these 3?
  + Why should there be at least one actor who benefits from a use case?
  + Can you supply 3 questions that may help you identifying actors? And use cases?
  + What’s the difference between a primary scenario and a secondary scenario?
  + What’s the direction of the <<extends>> and <<includes>> dependencies?
  + What is the purpose of technical stories in scrum?
  + List and explain briefly the INVEST criteria for user stories.
  + Explain briefly the three levels of detail for Product Backlog Items (Epic, Features, Stories).
  + *What is a minimum viable product?*
  + Define a misuse case.
  + Define a safety story.

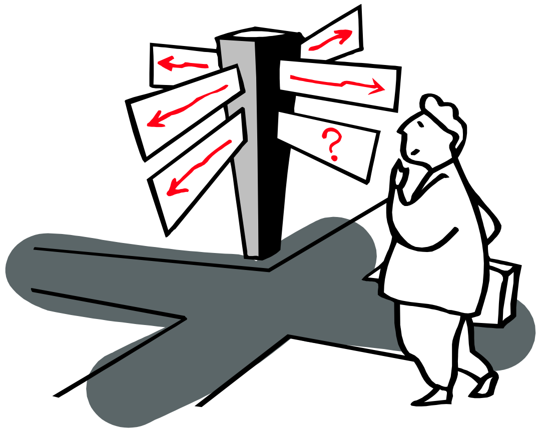
- You should be able to complete the following tasks
  + Write a requirements specification for your bachelor capstone project.

CAPSTONE PROJECT

Summary(ii)

- Can you answer the following questions?
  + Why do use cases fit well in an iterative/incremental development process?
  + Why do we distinguish between primary and secondary scenarios?
  + What would you think would be the main advantages and disadvantages of use cases?
  + How would you combine use-cases to calculate the risky path in a project plan?
  + Do use-cases work well with agile methods? Explain why or why not.
  + Can you explain the use of a product roadmap in scrum?
  + Choose the three most important items in your “Definition of Ready” checklist. Why are these most important to you?
  + Can you relate scrum user stories to some of the principles in the Agile Manifesto?
  + How would you turn an FMEA analysis into a misuse case diagram?
  + Elaborate on the relationship between an FMEA analysis and the variants of safety stories.

CHAPTER 3 – Software Architecture



- Introduction
  + When, Why and What?
  + Functional vs. Non-functional
  + Coupling and Cohesion
  + Patterns
- Macro architecture
  + Layered Architecture
  + Pipes and Filters
  + Blackboard Architecture
  + Model-View-Controller
- Micro Architecture
  + Observer
  + Abstract Factory
  + Adapter (a.k.a. Wrapper)
- Other Patterns
  + Security, …
  + *Microservices*
- Conclusion
  + Architecture in UML
  + Architecture Assessment
    - ATAM
  + Architecture in SCRUM
    - Spike
    - *Architecture Runway*
    - *GuardRails*
  + Correctness & Traceability

Literature (1/2)

Software Engineering Text Books

- [Somm05]: chapter “Architectural Design”
- [Pres00]: chapter “Architectural Design”

Books on Software Architecture

- [Shaw96] Software architecture: perspectives on an emerging discipline, Mary Shaw, David Garlan, Prentice-Hall, 1996.
  + The book introducing software architecture.
- [Bass03] Software architecture in practice (2nd edition), Len Bass, Paul Clements, Rick Kazman, Addison-Wesley, 2003.
  + A very deep and practical treatment of software architecture, *incl. ATAM*. (The book received an award.)

Articles

- [Kruc95] Philippe Kruchten "The 4+1 View Model of Architecture ", IEEE Software, November 1995 (Vol. 12, No. 6) pp. 42-50.
  + A paper that illustrates convincingly the need for various perspectives on the design of a system.

Literature (2/2)

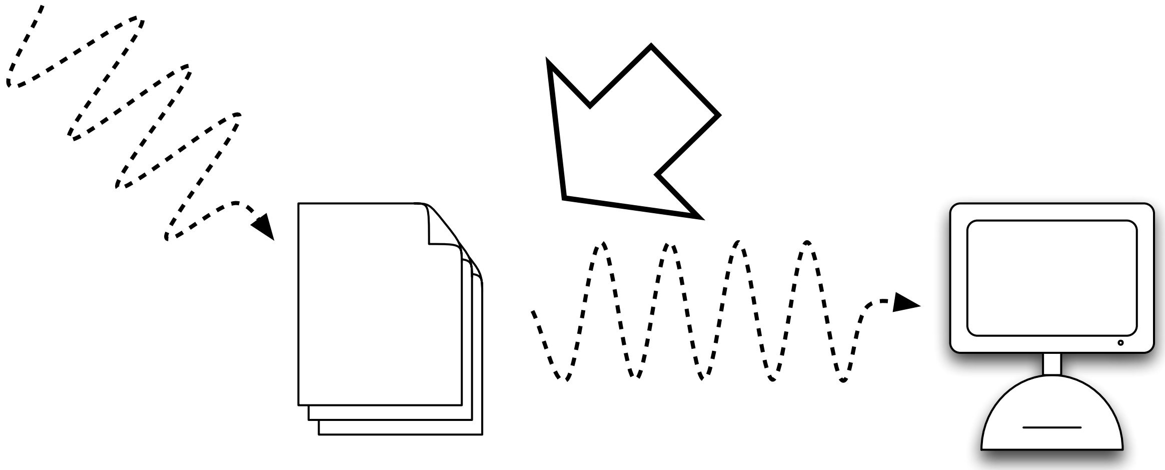
Pattern Language

- [Foot97] Big Ball of Mud, Brian Foote, Joseph Yoder; Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97)
  + http://www.laputan.org/mud/mud.html; most popular architecture.

Pattern Catalogues

- [Busc98] Pattern-Oriented Software Architecture: A System of Patterns, Frank Buschman, Regine Meunier, Hans Rohnert, Peter Somerlad, Michael Stal, Wiley and Sons, 1996.
  + Introduces architectural styles in pattern form. Also covers some design patterns and idioms.
    > At architecture (= "macro-architecture") level
- [Gamm95] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995.
  + The classic; commonly referred to as the "Gang of Four (GOF)"
    > At design (= "micro-architecture") level
- [Shum06] Security Patterns: Integrating Security and Systems Engineering, Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad, Wiley & Sons, 2006.

When Architecture?

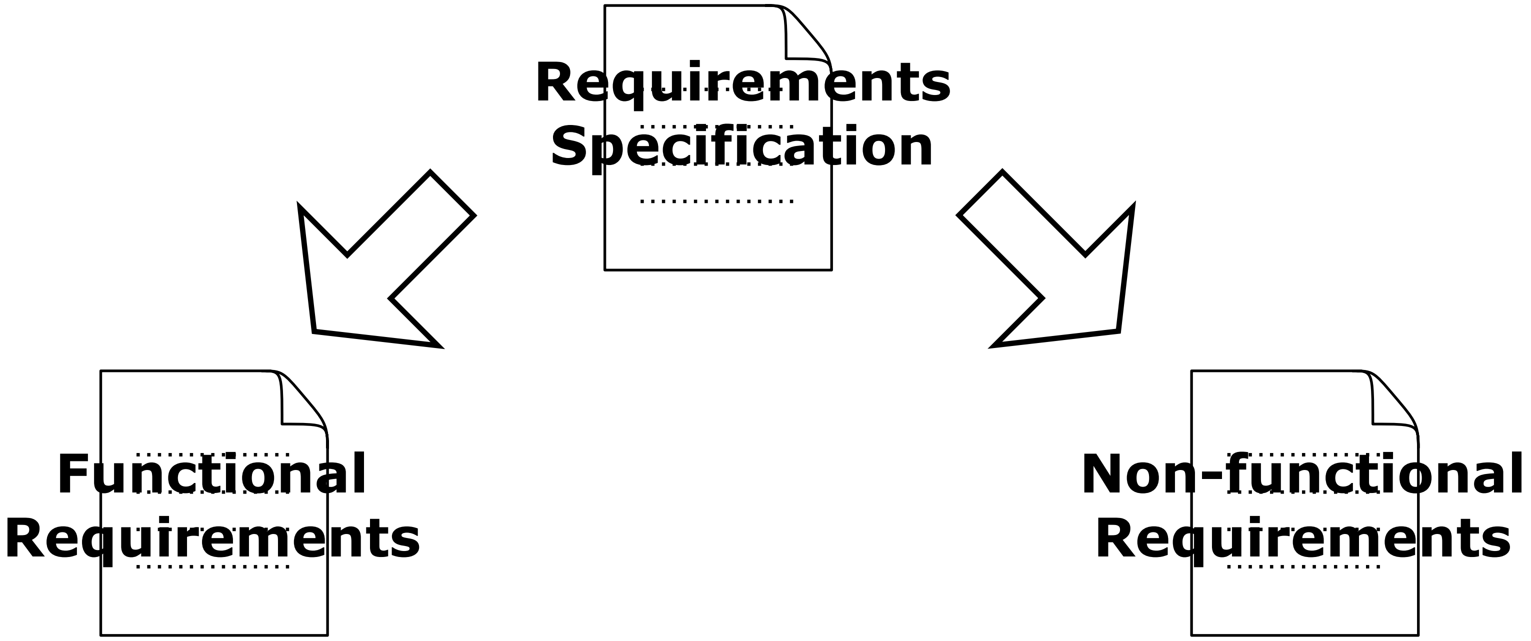


Designing a software system requires *course-grained decomposition*
⇒ organize work in the development team

Conway's law
Organizations which design systems are constrained to produce designs which are copies of the communications structure of these organizations. [Conw68]

- If you have 4 groups working on a compiler; you'll get a 4-pass compiler

Why Architecture



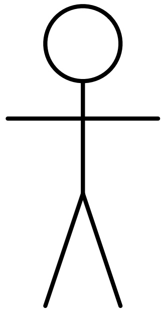
Requirements Specification

Functional Requirements

Non-functional Requirements

- functionality as demanded by the end users

- constraints placed on the global system or the development process.
- quality attributes, such as performance, user-friendliness, maintainability, …

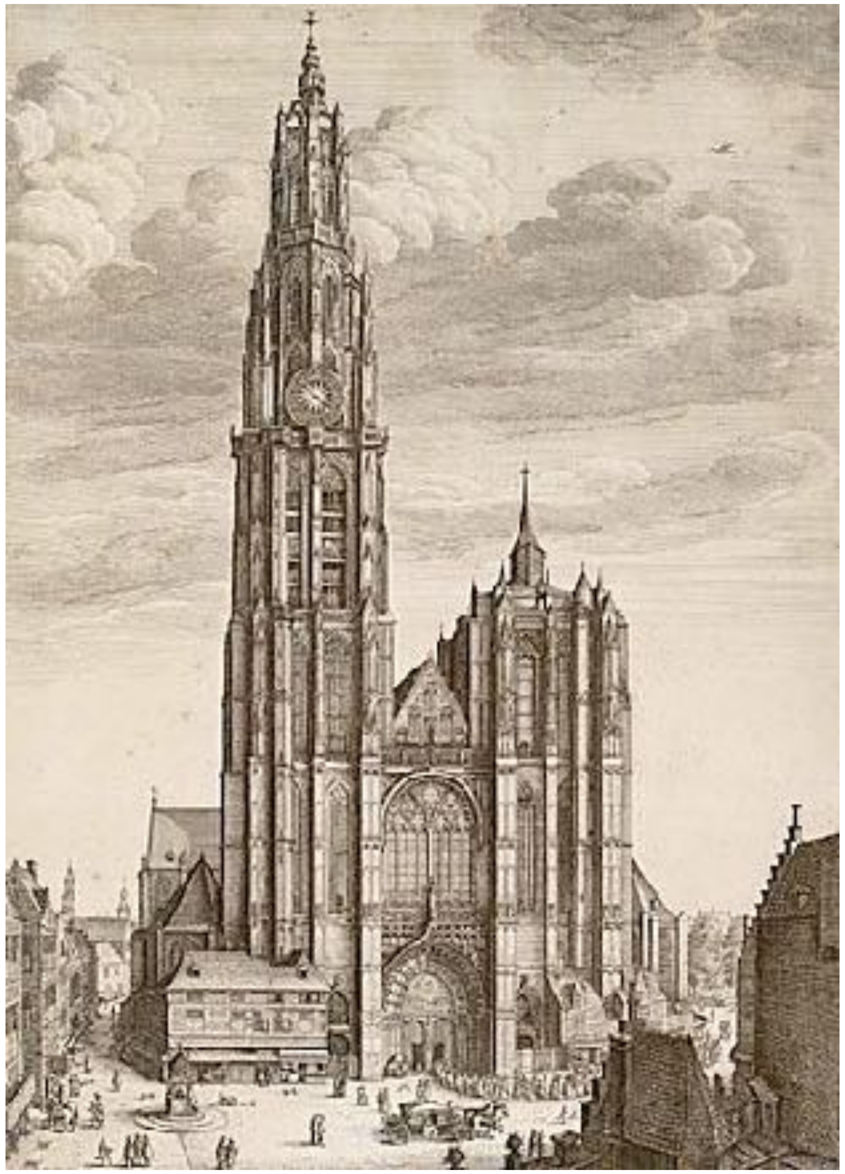


ARCHITECTURE

- map requirements onto system structure
  = map function onto form

SCALE ISSUE

Characteristics of a Gothic Cathedral



Public Domain

Architecture as a Metaphor

Public Domain

Parallels

- Architects are the technical interface between the customer and the contractor.
- A poor architectural design cannot be rescued by good construction technology.
- There are architectural styles or schools.
  + (e.g., “ghotic” in buildings; “client-server” in software)

Differences

- Buildings are tangible, software is intangible.
  > Software Architecture is often expressed via metaphors.
- Buildings are rather static, software is quite flexible.
  > The underlying architecture allows to anticipate changes.
- Building architecture is supposed to be aesthetic.
  > Buildings avoid to mix styles; in software heterogeneity is considered good.
- A building architect carries legal responsibilities.
  > Usually a building architect is not employed by the constructor.

What is Software Architecture?

Software Architecture

- A description of *components* and the *connectors* between them.
  - + Typically specified in different views to show the relevant functional and non-functional properties.

Component

- An encapsulated part of a software system with a designated *interface*.
  - + Components may be represented as modules (packages), classes, objects or a set of related functions. A component may also be a *subsystem*.

Subsystem

- A component that is a system in its own right, i.e. can operate independently

Connector (a.k.a. Relationships)

- A connection between components.
  - + There are *static* connectors that appear directly in source code (e.g., use or import keywords) and *dynamic* connectors that deal with temporal connections (e.g., method invocations).

View

- Represents a partial aspect of a software architecture that shows specific *functional and non-functional properties*.

Functional vs. Non-functional Properties

- See [Bush98]

Functional property

- Deals with a particular aspect of the system’s functionality. Usually in direct relationship with a particular use case or conceptual class.

Non-functional property

- Denotes a a constraint placed on the global system or the development process. Typically deals with quality attributes that cross-cut the whole system design and are quite intangible.
- Typical non-functional properties
  + Changeability; systems must evolve or perish
  + Interoperability; interaction with other systems
  + Efficiency; use of resources such as computing time, memory, ...
  + Reliability; system will continue to function even in unexpected situations
  + Testability; feasibility to verify that requirements are covered
  + Reusability; ability to reuse parts of software system or process for constructing other systems

Architecture is about tradeoffs

Coupling and Cohesion

Coupling

- Measure of strength for a connector (i.e., how strongly is a component connected with other components via this connector)

Cohesion

- Measure of how well the parts of a component belong together (i.e., how much does the functioning of one part rely on the functioning of the other parts)

> Coupling and cohesion are criteria that help us to evaluate architecture tradeoffs.
> Minimize coupling and maximize cohesion

However …

- The perfect trade-off corresponds to a component that does nothing!
- Coupling at one level becomes cohesion at the next.

> More qualitative trade-off analysis is necessary

Patterns

Pattern

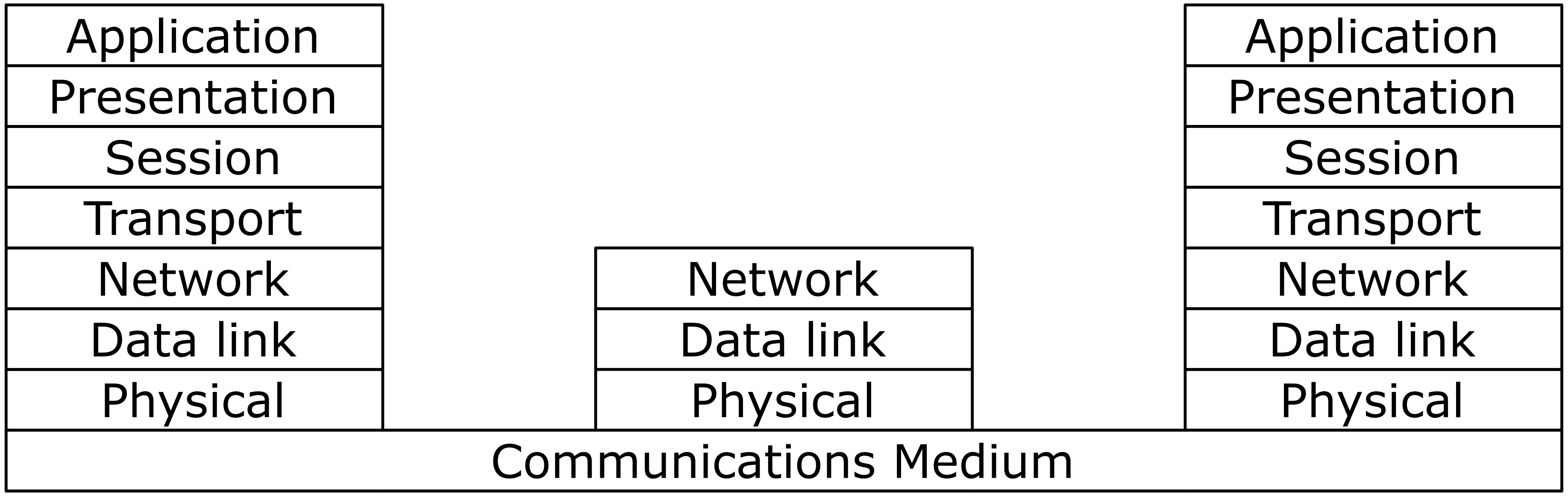
- The essence of a *solution* to a *recurring problem* in a particular context.
  - + Experts recall a similar solved problem and *customize* the solution.
  - + Patterns document *existing* experience.
  - + The context of a pattern states *when (and when not)* to apply the solution.
  - + A pattern lists the *tradeoffs* (a.k.a. forces) involved in applying the solution.

Pattern Form

- Patterns are usually written down following a semi-structured template.
  - + Patterns always have a *name*
  - + Patterns allow experts to have deep design discussions in a few words!

Layered Architecture in Networks

OSI Reference Model



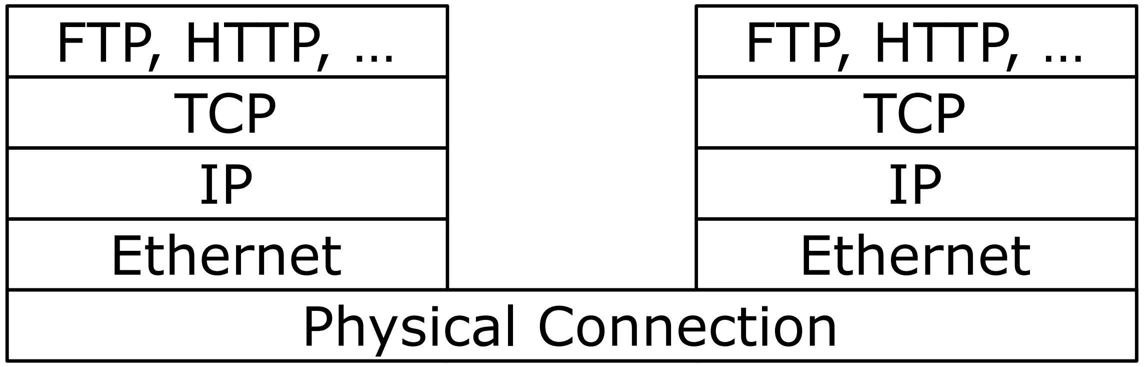
Application | Presentation | Session | Transport | Network | Data link | Physical

Network | Data link | Physical

Application | Presentation | Session | Transport | Network | Data link | Physical

Communications Medium

TCP/IP Stack



FTP, HTTP, … | TCP | IP | Ethernet

FTP, HTTP, … | TCP | IP | Ethernet

Physical Connection

3-Tiered Architecture

Application Layer

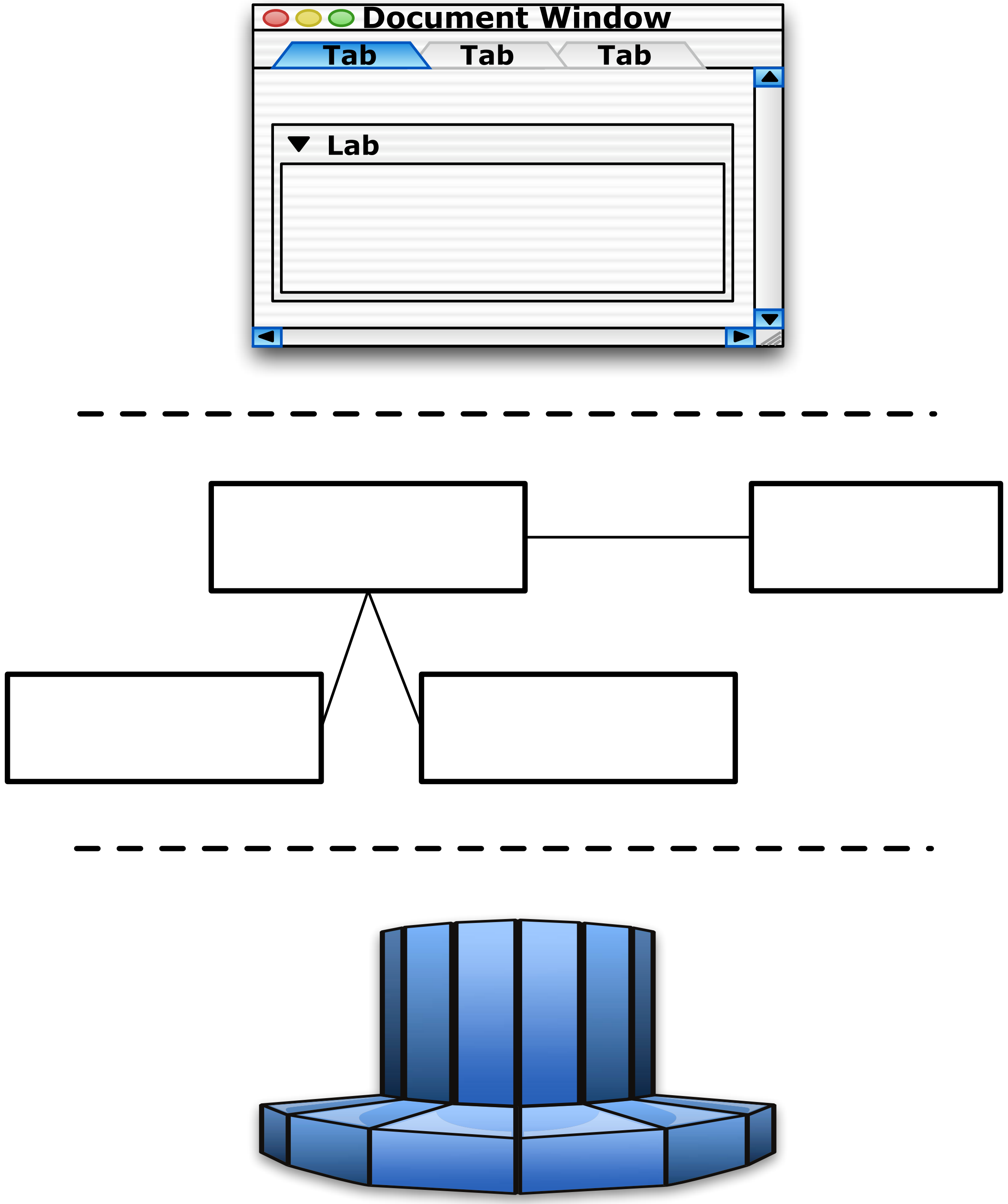
- Models the UI and application logic

Domain Layer

- Models the problem domain (usually a set of classes)

Database Layer

- Provides data according to a certain database paradigm (usually relational database)



Pattern: Layered Architecture

Context

- Requirements imply various levels of abstraction (low & high level)

Problem

- Need for portability and interoperability between abstraction levels

Solution

- Decompose system into layers;
  each layer encapsulates issues at same level
- Layer n provides services to layer n + 1
- Layer n can only access services at layer n - 1
  + Call-backs may be used to communicate back to higher layers
  + Relaxed variant allows access to all lower layers

Tradeoffs

- How stable and precise can you make the interfaces for the layers?
- How independent are the teams developing the different layers?
- How often do you exchange components in one layer?
- How much performance overhead can you afford when crossing layers?

Pipes and Filters Examples

UNIX shells

- `tar cf - .| gzip -cfbest| rsh hcoss dd`

data source =
current directory

pipe

filter =
compress

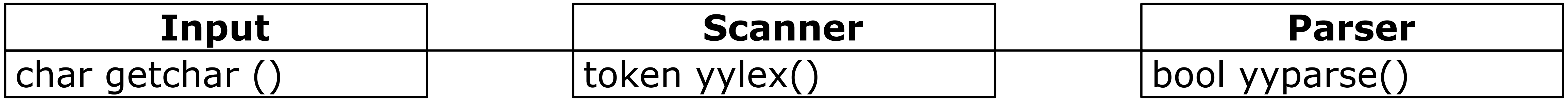
pipe

data sink =
remote host

Many CGI-scripts for WWW-forms

- data source is some filled in web-form
- filters are written via a number of scripting languages (perl, python)
- data sink is generated web page
  + Example: wiki-web pages (http://c2.com/cgi/wiki)

Scanners & Parsers in Compilers



Input
char getchar ()

Scanner
token yylex()

Parser
bool yyparse()

Pattern: Pipes and Filters

Context

- Processing data streams

Problem

- Flexibility (and parallelism) is required

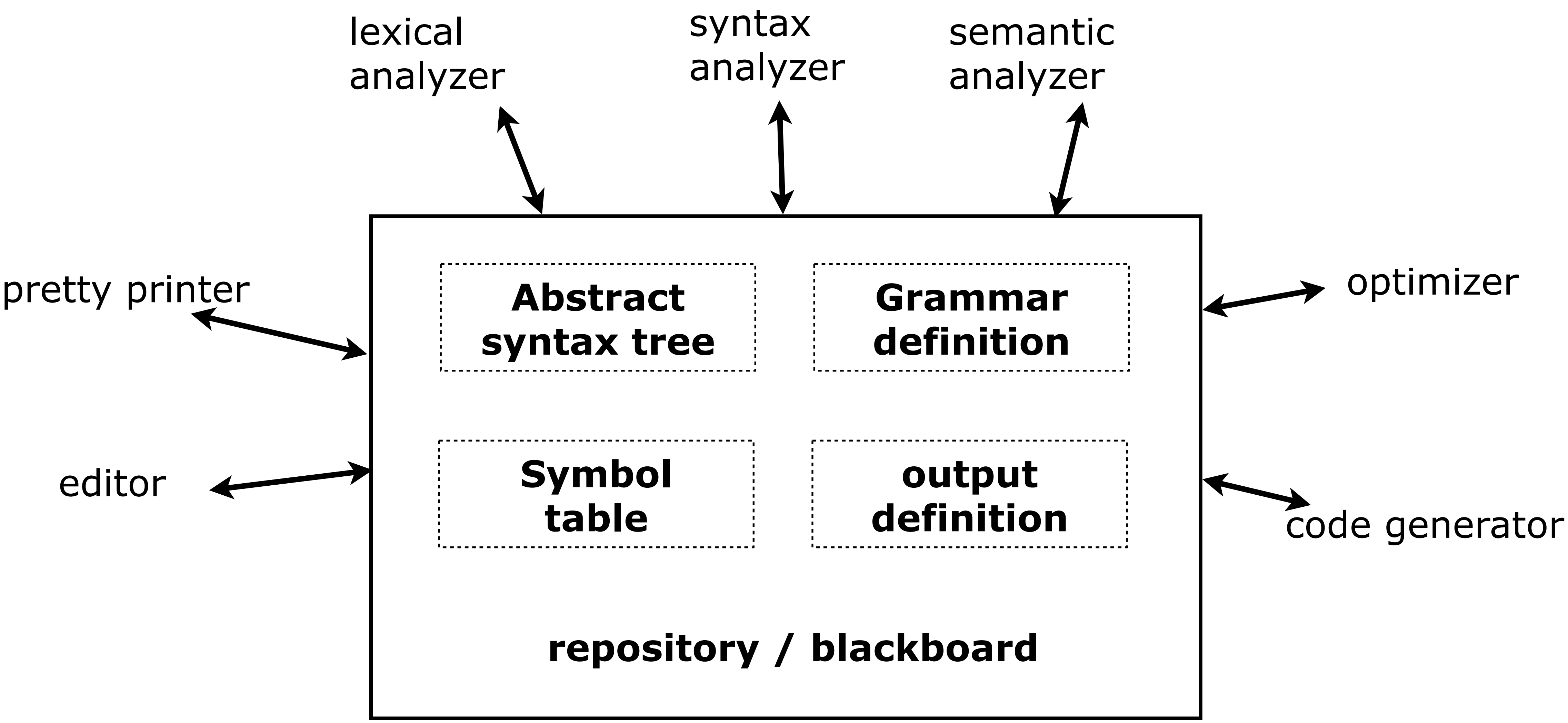
Solution

- Decompose system into filters, each with 1 input- and 1 output stream
- Connect output from one filter to input of another
  > Need a data source and data sink
- Variants
  + Push filter: filter triggers *next* one by pushing data on the output
  + Pull filter: filter triggers *previous* one by pulling data from the input

Tradeoffs

- How often do you change the data processing?
- How well can you decompose data processing into independent filters?
  + Sharing data other than in/out streams must be avoided
- How much overhead (task switching, data transformation) can you afford?
- How much error-handling is required?

Compilers as Blackboard Architecture



Pattern: Blackboard (a.k.a. Repository)

Context

- Open problem domain with various partial solutions

Problem

- Flexible integration of partial solutions

Solution

- Decompose system in 1 blackboard, several knowledge sources and 1 control
  + *Blackboard* is common data structure
  + *Knowledge sources* independently fill and modify the blackboard contents
  + *Control* monitors changes and launches next knowledge sources

Tradeoffs

- How well can you specify the common data structure?
- How many partial solutions exist? How will this evolve?
- How well can you compose an overall solution from the partial solutions?
- Can you afford partial solutions that do not contribute the current task?

Quizz

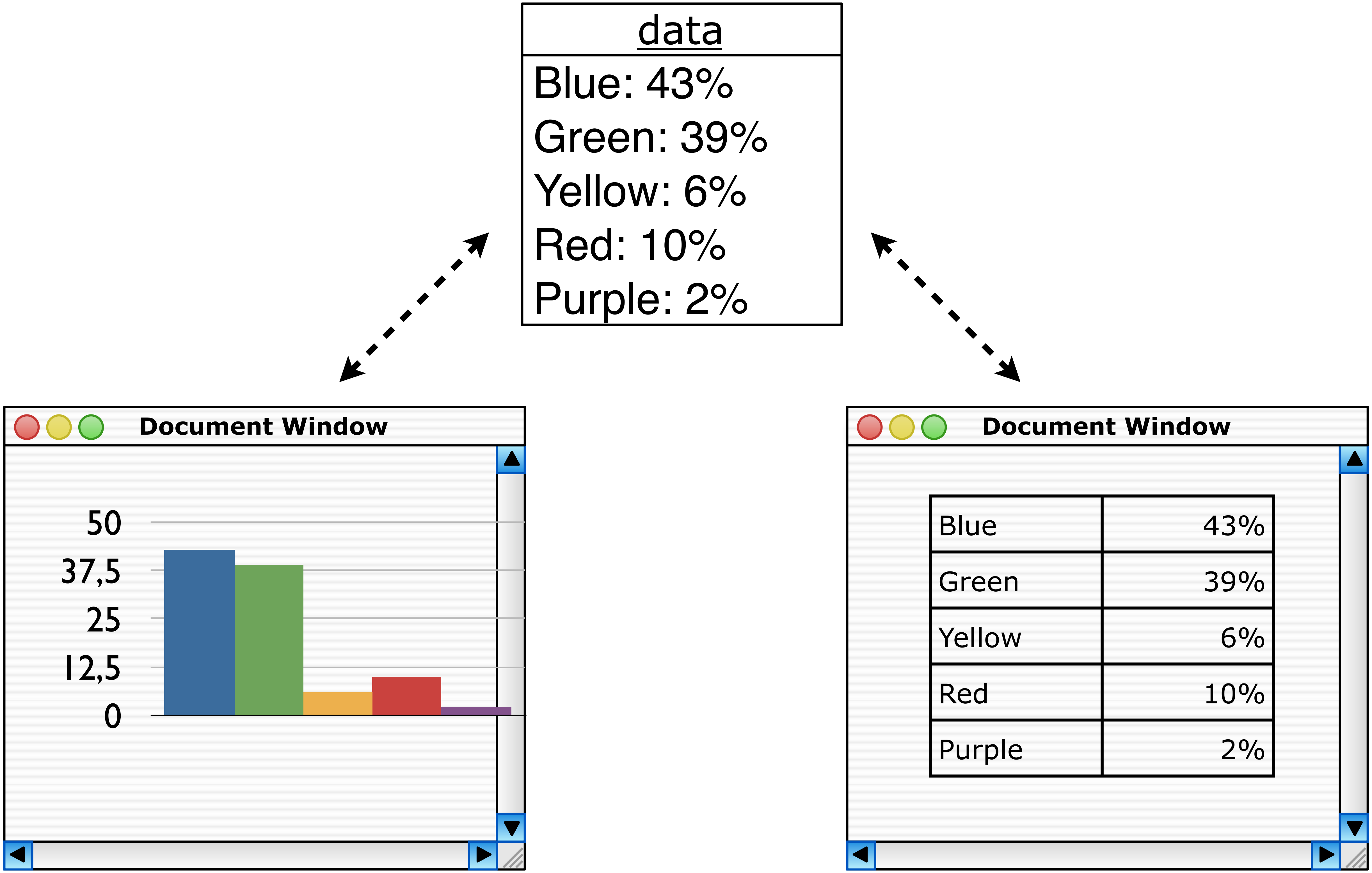
Why is a repository better suited for an integrated development environment than pipes and filters?

LLVM

CLION

VisualStudio

Interactive Applications



data

Blue: 43%

Green: 39%

Yellow: 6%

Red: 10%

Purple: 2%

Blue	43%
Green	39%
Yellow	6%
Red	10%
Purple	2%

Pattern: Model-View-Controller

Context

- Interactive application where multiple widgets act on same data

Problem

- Ensure consistency between the various widgets

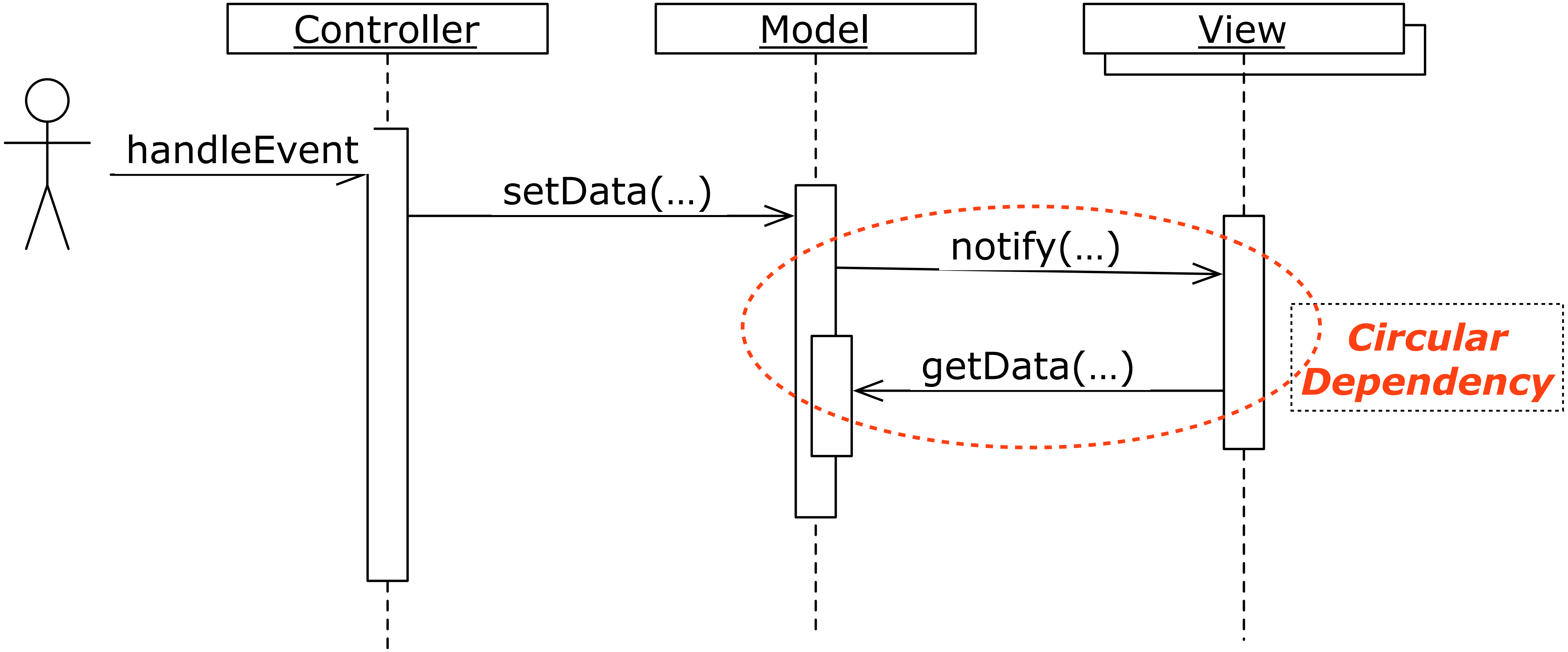
Solution

- Decompose system in a model, and several view-controller pairs
- Model: provides functional core (data)
  + registers dependent views/controllers
  + notifies dependent components about changes (send update)
- View: creates and initializes associated controller + displays information
  + responds to notification events (receive update)
- Controller: accepts user input events + translate events into requests to model and view
  + responds to notification events (receive update)

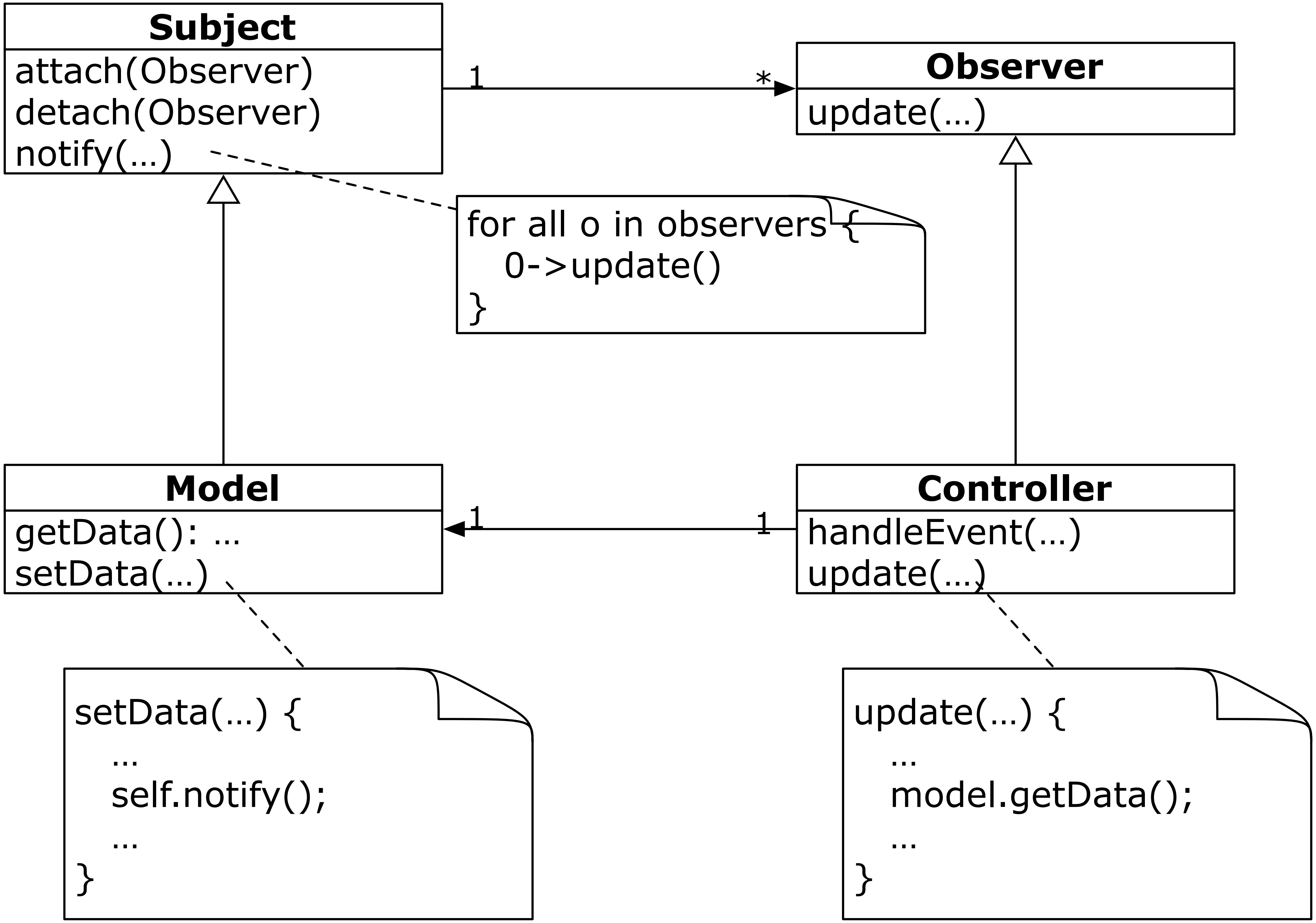
Tradeoffs

- How many widgets? How consistent? Should they be “plug able”?
- Increased complexity, especially without library of views/controllers
- Excessive number of updates if not carefully applied
- Close coupling between View-Controller;
  average coupling from View-Controller to Model

Problem: Circular Dependencies 1-N



Solution: Observer



```
for all o in observers {
   0->update()
}
```

```
setData(…) {
   …
   self.notify();
   …
}
```

```
update(…) {
   …
   model.getData();
   …
}
```

Pattern: Observer

Context

- Change propagation: when one class changes (the subject) others should adapt (the observers)

Problem

- Change propagation implies a circular dependency: (a) adapting requires the observers to access the subject; (b) changing requires the subject to notify the observers

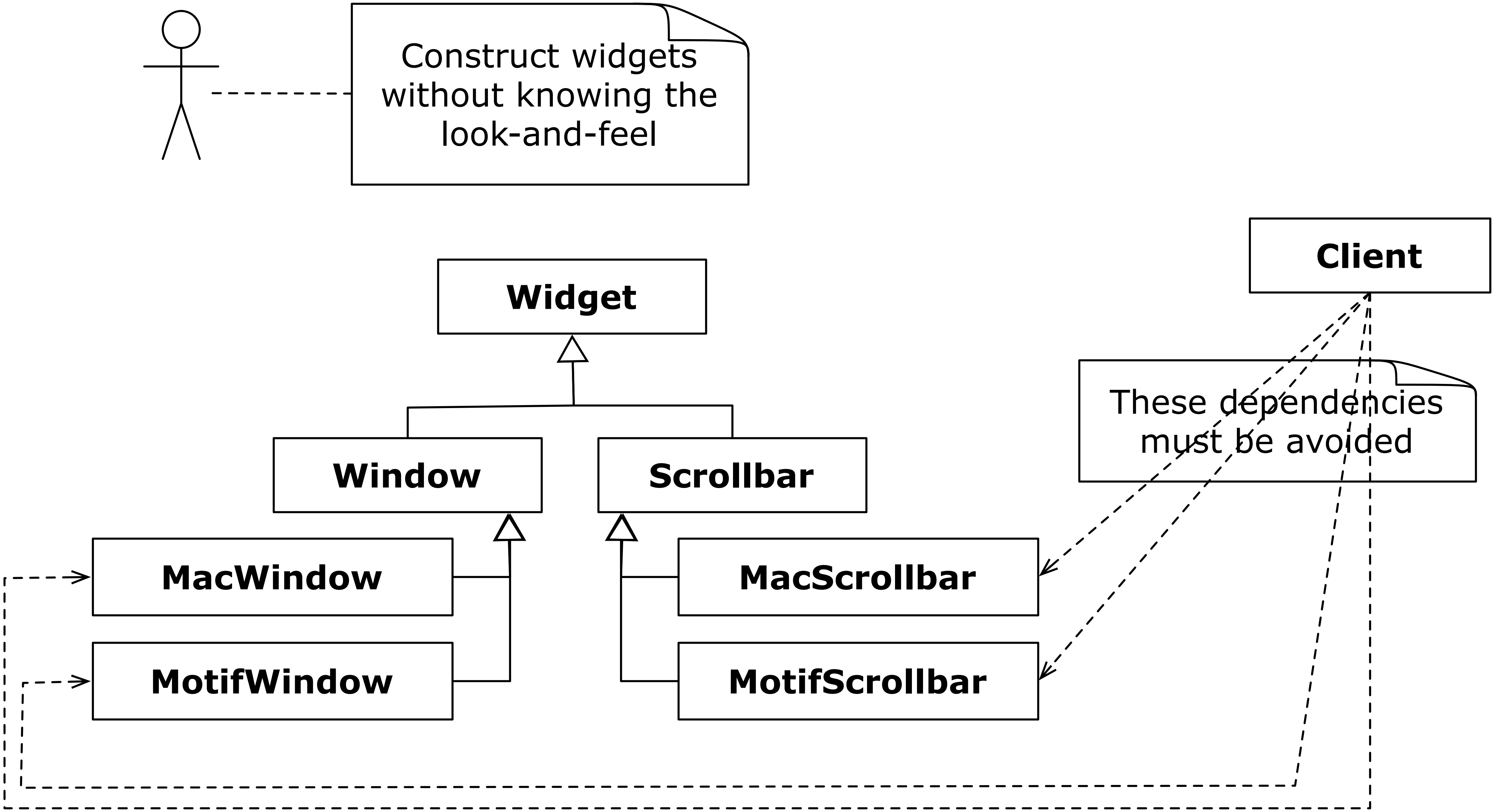
Solution

- Split the circular dependency; move one direction in new superclasses
- Force observers to register themselves on a subject before they will be notified
- Notification becomes anonymous and asymmetrical: subject notifies all observers

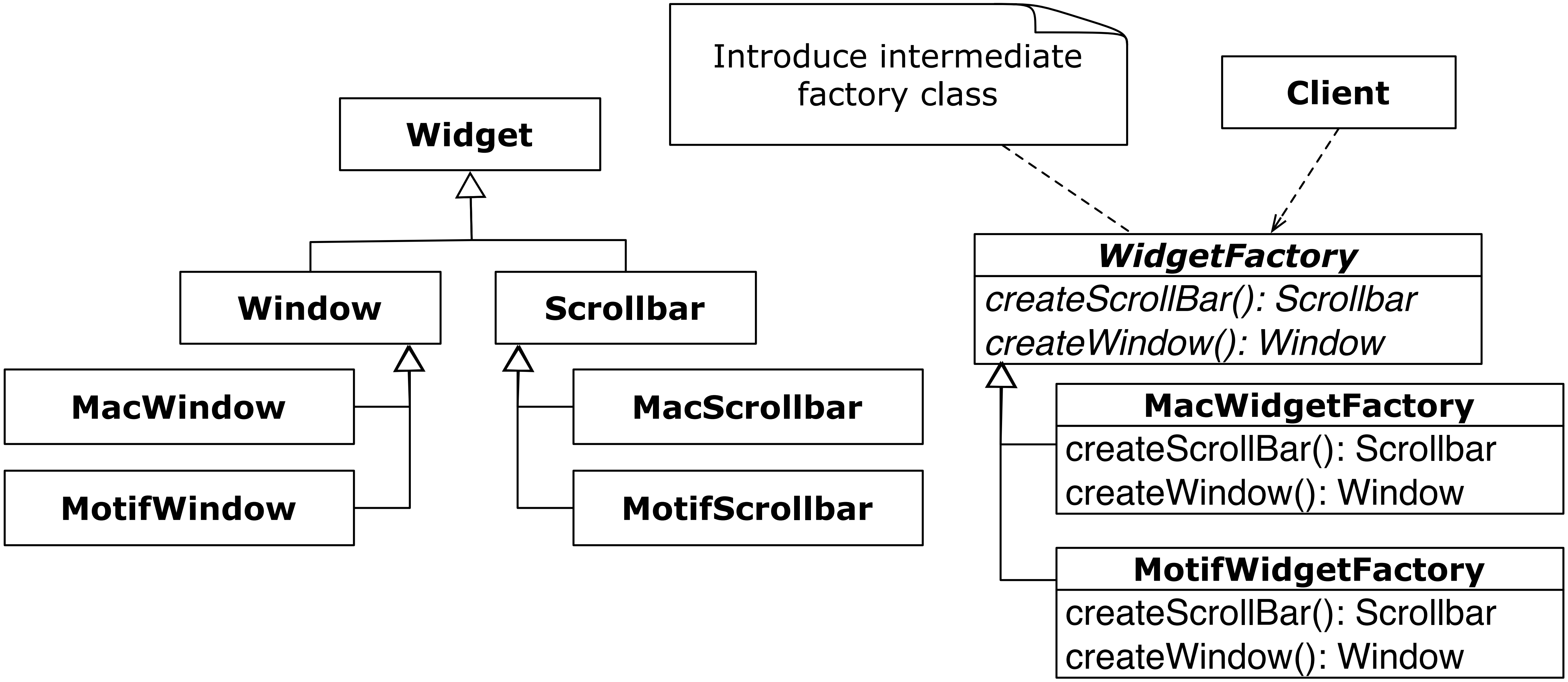
Tradeoffs

- Extra complexity: observers will receive more updates than necessary
  + extra program logic to filter the applicable notifications
- Restricts communication between subject and observer

Problem: Constructor Dependencies


Construct widgets without knowing the look-and-feel
Client
Widget
These dependencies must be avoided
Window
Scrollbar
MacWindow
MacScrollbar
MotifWindow
MotifScrollbar

Solution: Abstract Factory



Pattern: Abstract Factory

Context

- Class hierarchy with abstract roots representing a family of objects
  + concrete leaves representing particular configurations

Problem

- Invoking constructors implies tight coupling with concrete leaves instead of abstract roots

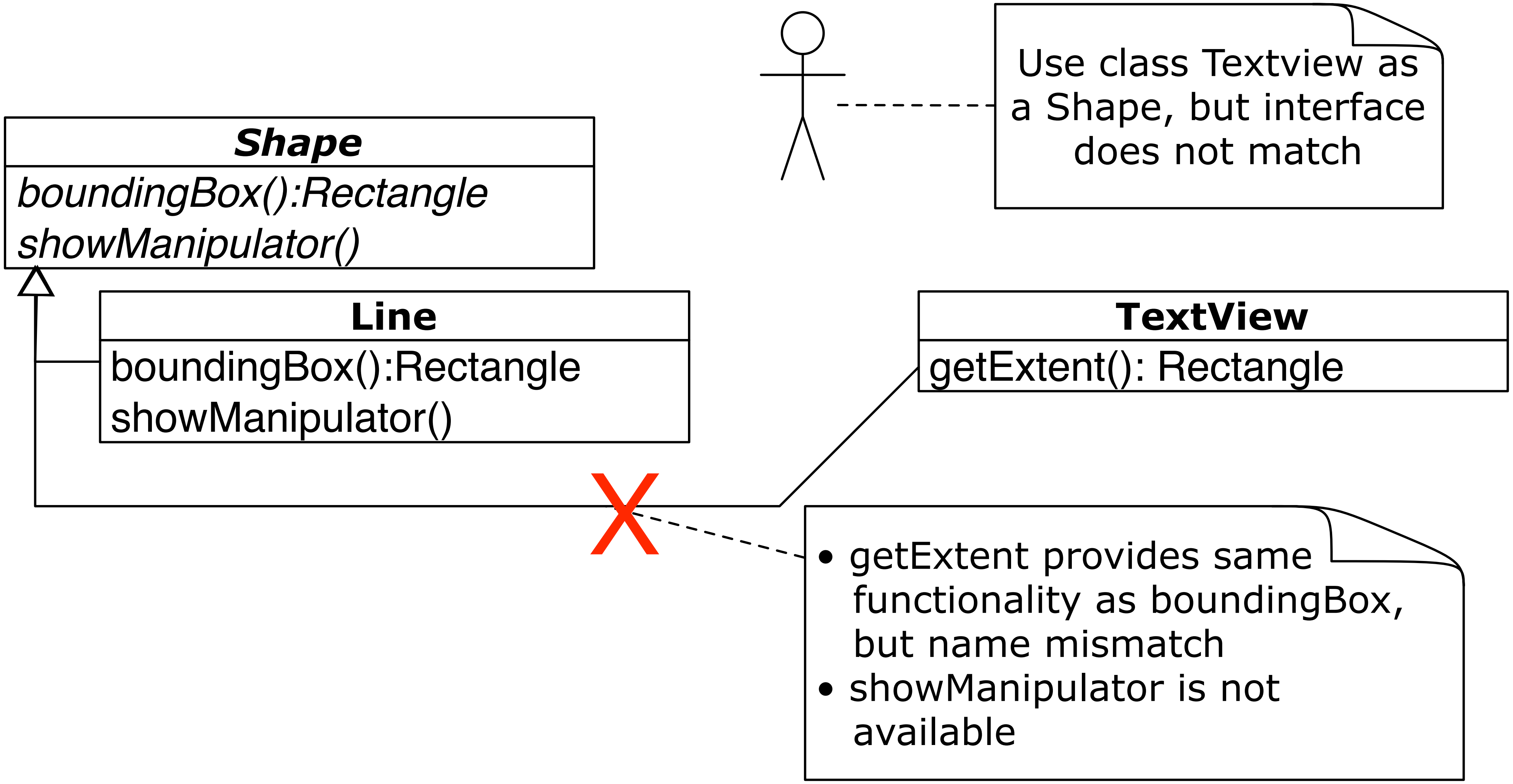
Solution

- Create an abstract factory class with operations for creating all abstract roots
- Create concrete factory classes for all possible configurations.

Tradeoffs

- How many members in the family? How many configurations?
- When do you switch configurations?
- How strict are the configurations?
- Can clients rely on the abstract interfaces?

Problem: Interface Mismatch

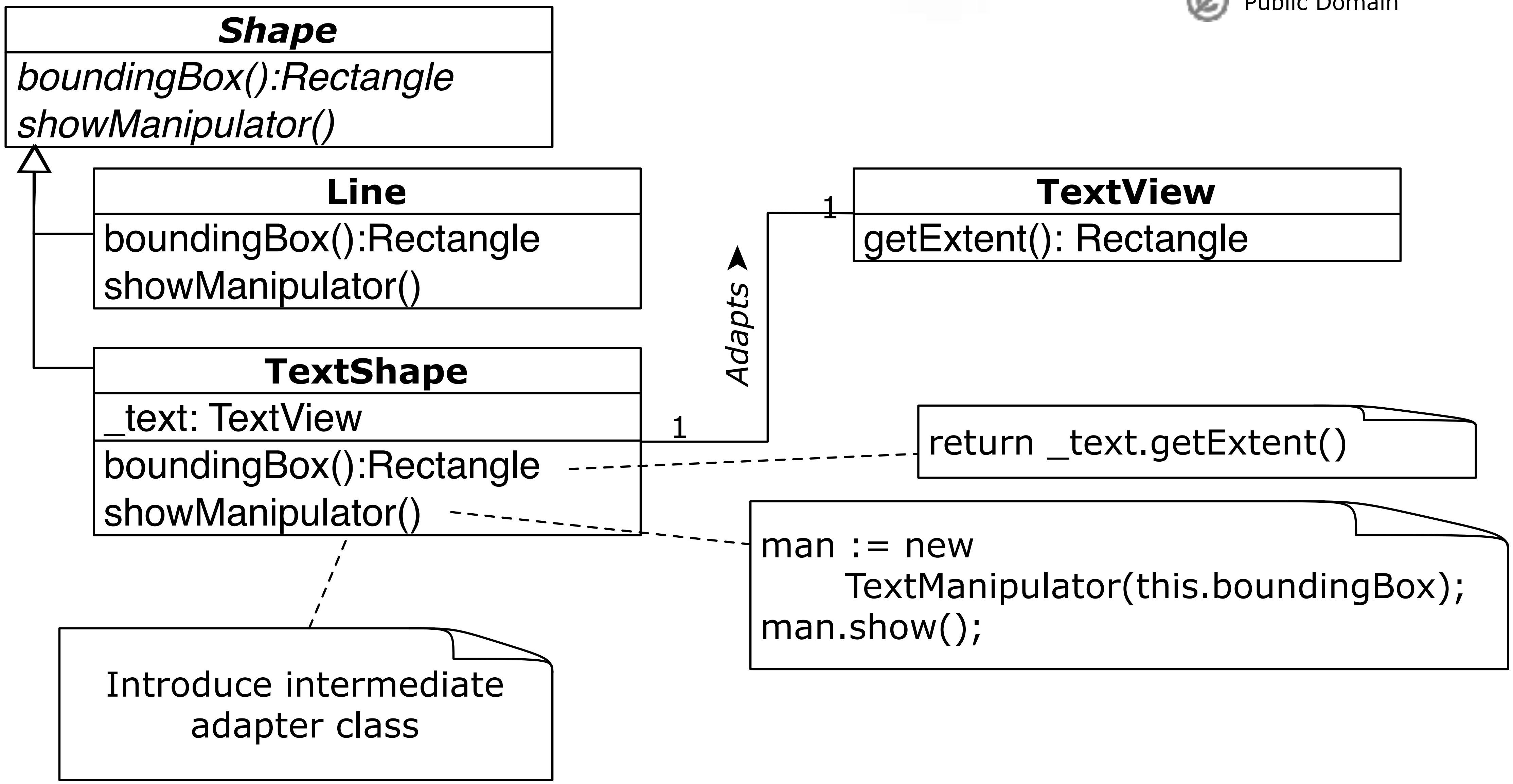


Use class Textview as a Shape, but interface does not match

- getExtent provides same functionality as boundingBox, but name mismatch
- showManipulator is not available

Solution: Adapter

Public Domain



Shape

boundingBox():Rectangle
showManipulator()

Line

boundingBox():Rectangle
showManipulator()

TextShape

_text: TextView

boundingBox():Rectangle
showManipulator()

TextView

getExtent(): Rectangle

```
return _text.getExtent()
```

```
man := new
     TextManipulator(this.boundingBox);
man.show();
```

Introduce intermediate adapter class

Pattern: Adapter (a.k.a.Wrapper)

Context

- Merge two separately developed class hierarchies

Problem

- Class provides (most of) needed functionality but interface does not match

Solution

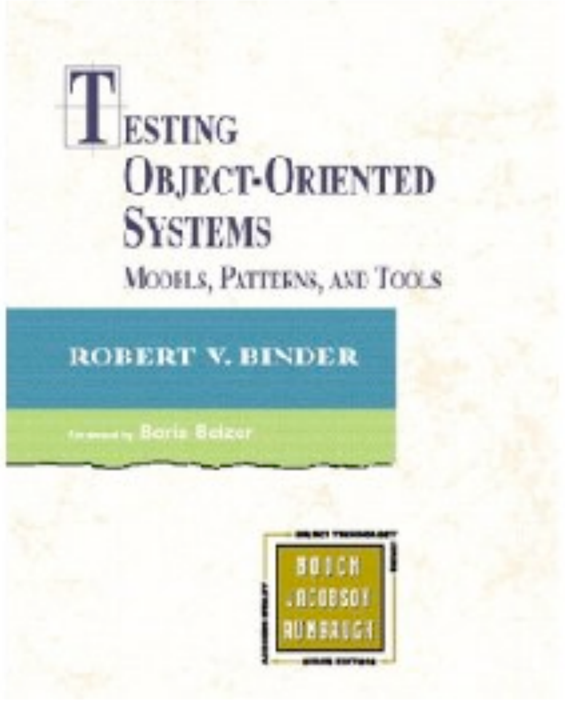
- Create an adapter class with one attribute of adaptee class
- Adapter class translates required interface into adaptee class

Tradeoffs

- How much adapting is required?
  - + For one class
  - + For the whole hierarchy
- How will the separately developed classes evolve?
- Does the merging work in one direction or in both directions?
- How much overhead in performance and maintenance can you afford?

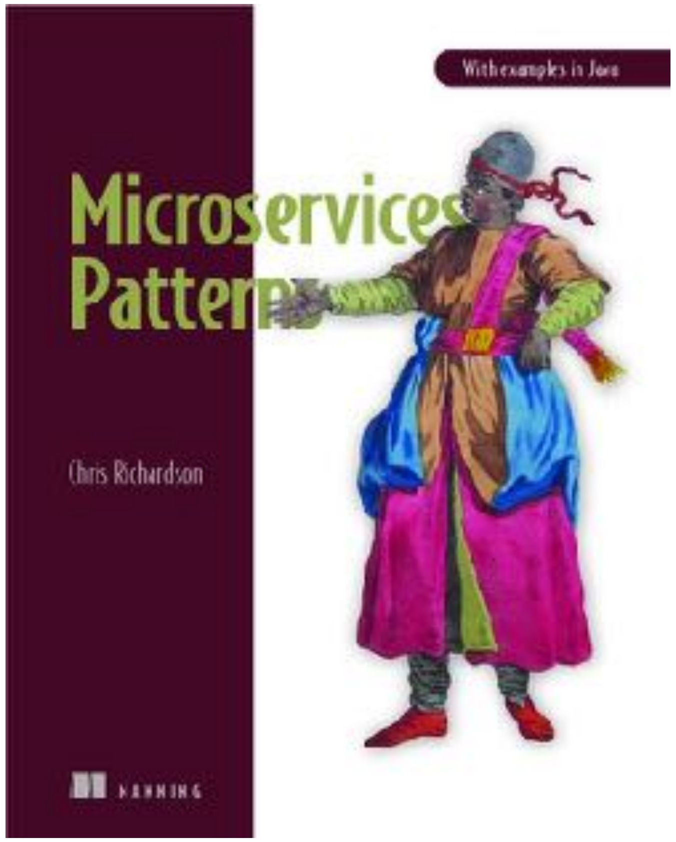
Other Pattern Catalogues

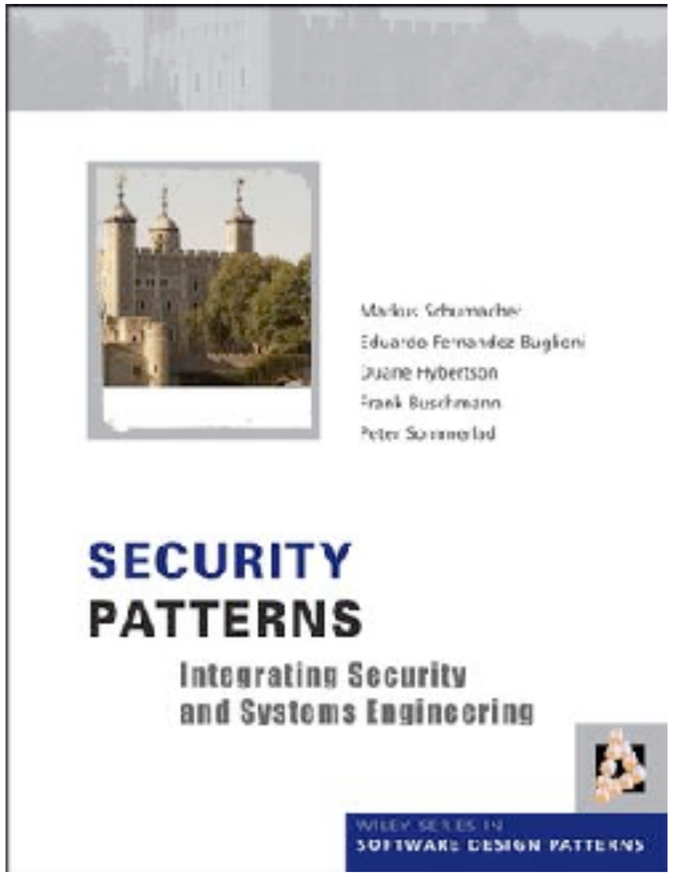
*** Revised ***
(MicroServices)



Testing

Microservices





Security

Reengineering

Security Pattern (sample): Single Access Point

Context

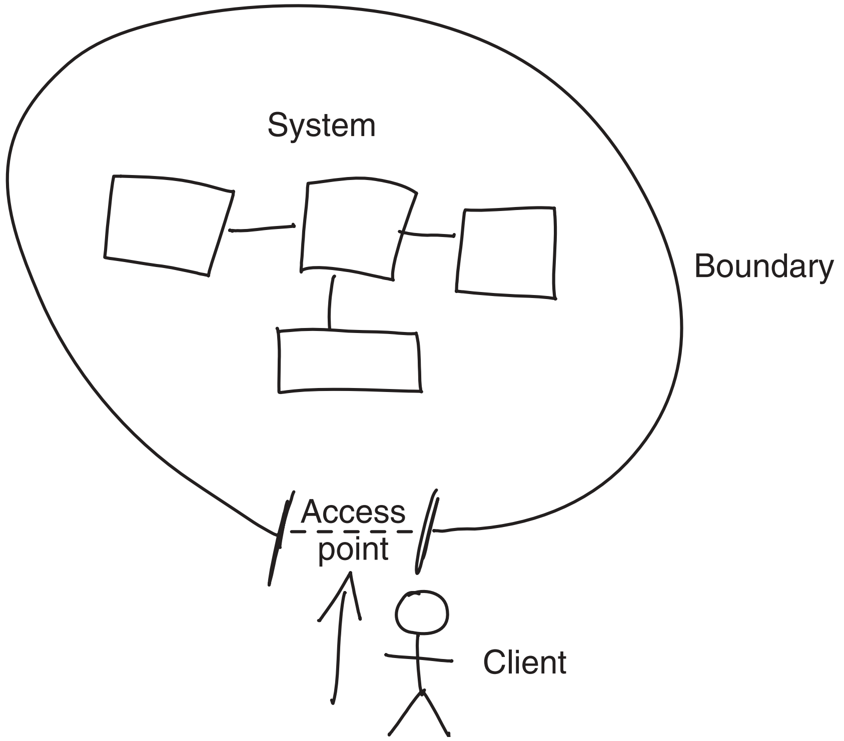
- Provide access to a system for external clients
- Ensure not misused or damaged by external clients

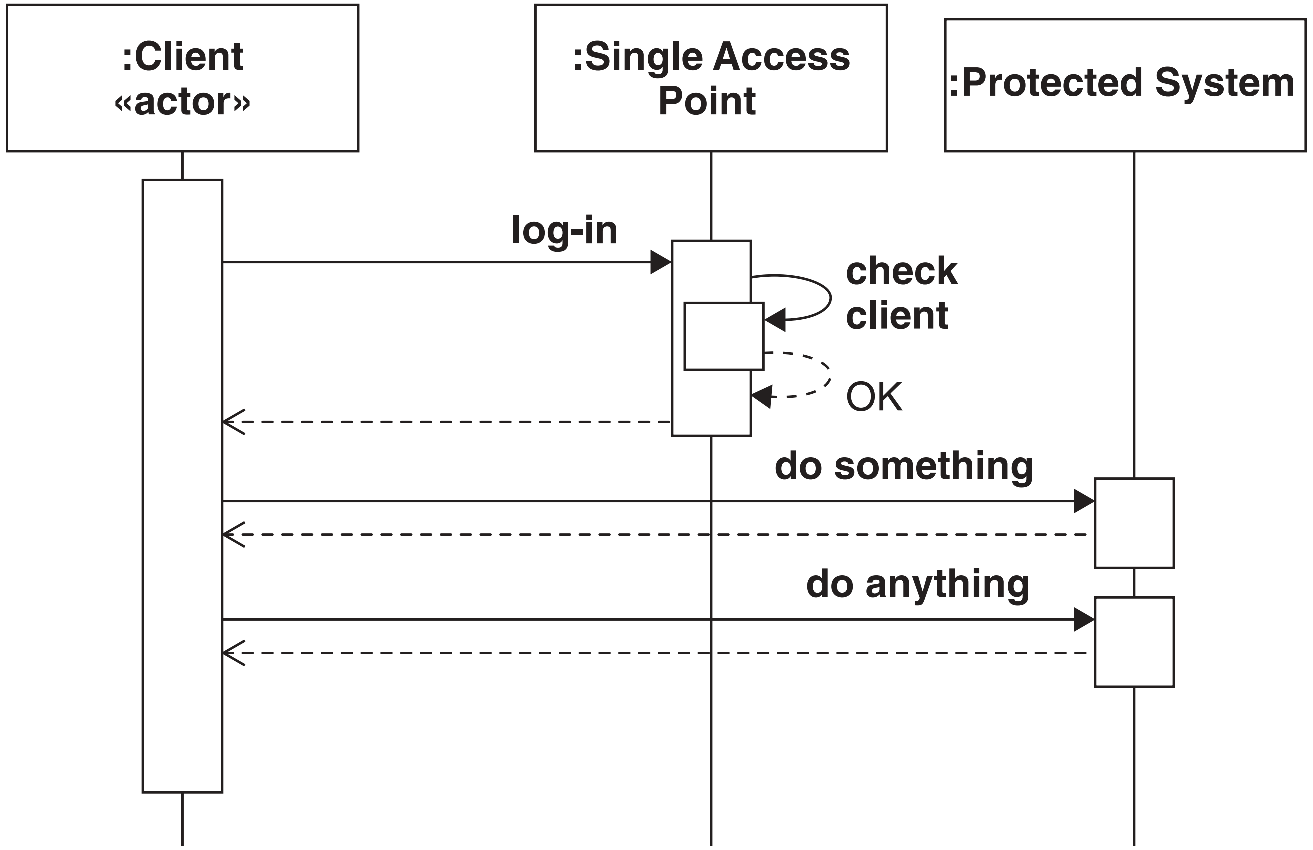
Problem

- External access ⇒ system's integrity in danger
- Complex inner structure ⇒ explosion of potential security breaches

Solution

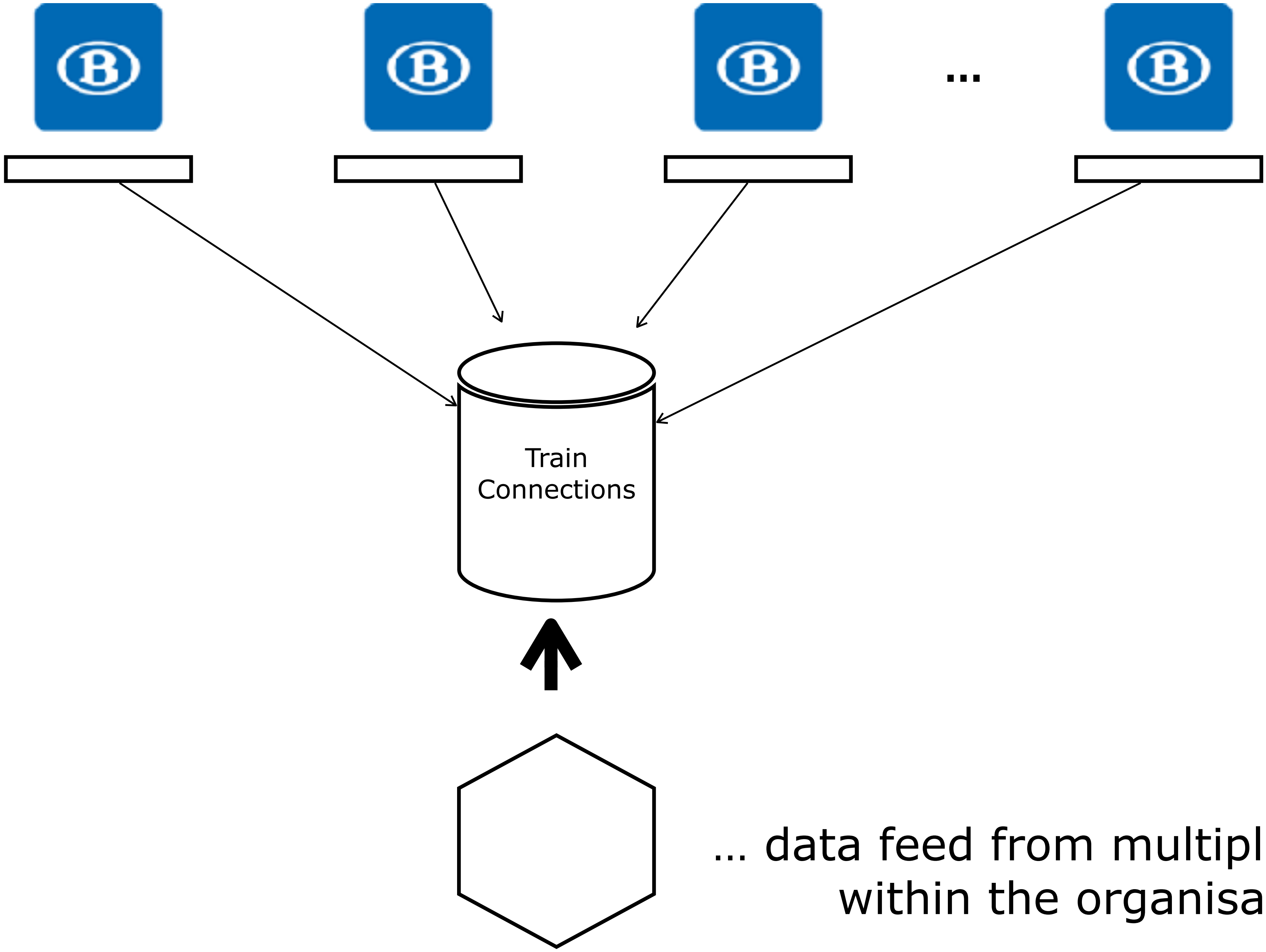
- Define single access point; check legitimacy





Context: (Train) App

Flexible demand with sometimes peak volume in transactions



… data feed from multiple sources within the organisation

Pattern: MicroServices

Context

- Distributed system (cloud) with multiple access points
  + Many read access - few write and update

Problem

- Elastic scaling of access points to deal with peak demand

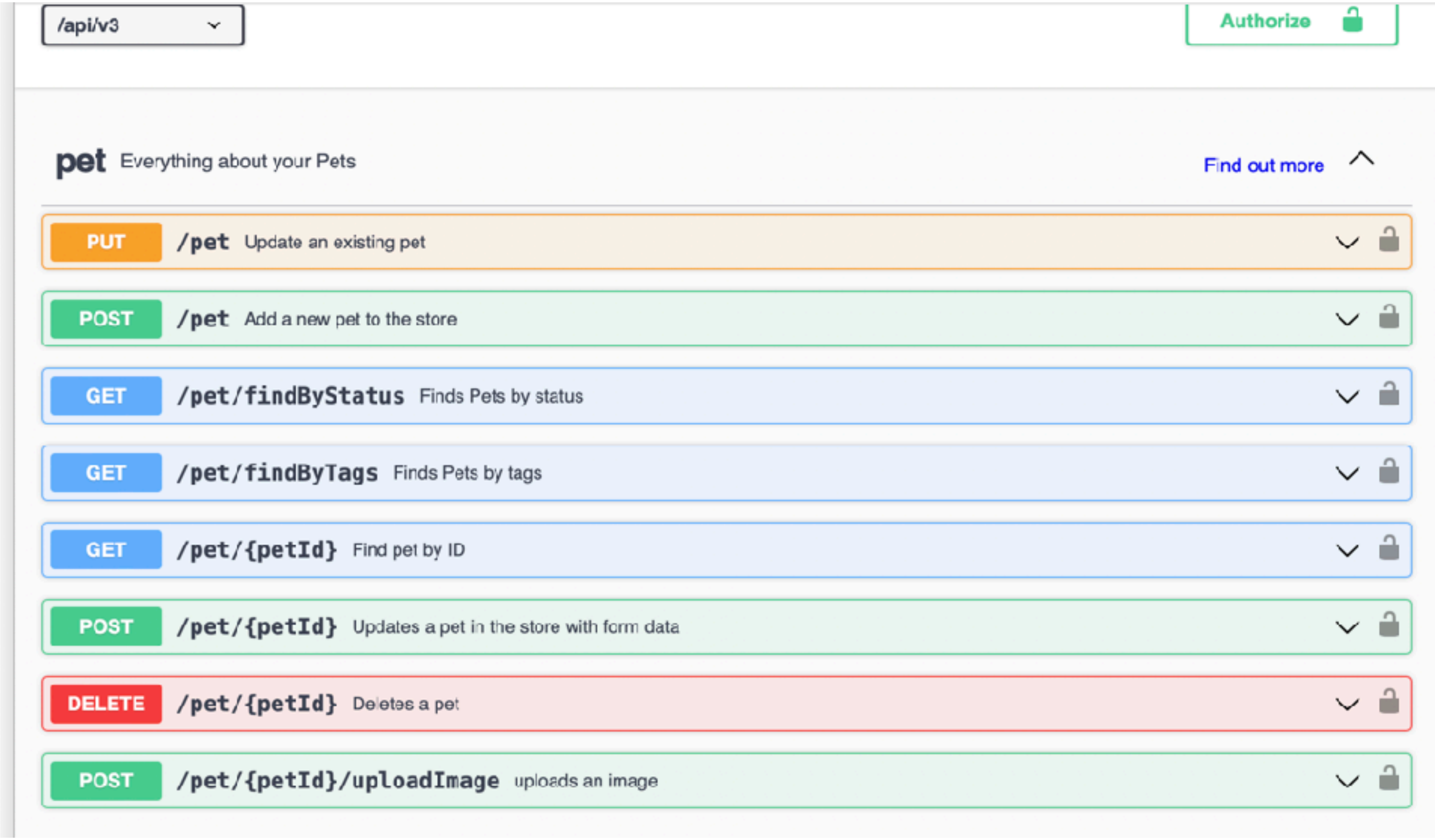
Solution

- *Microservices* structure an application as a collection of small, loosely coupled and independently deployable services.
  + Each of these services corresponds to a specific business functionality and can be developed, deployed and *scaled* independently.
- Each service is independent and communicates with others via well-defined APIs and protocols (REST-API)

Tradeoffs

- How much data sharing is needed?
  + Database per service (Database sharding — elastic split of database)
  + Event mechanism to notify updates
- How much communication needed?
  - Each service deployed by separate DevOps team.
  - Business transactions that span multiple services? (the Saga pattern)
- Resilience: what is a service is down?
  + Reroute calls to failing service (the Circuit breaker pattern)

MicroService Example - Pet Store (REST API)



pet Everything about your Pets

PUT `/pet` Update an existing pet

POST `/pet` Add a new pet to the store

GET `/pet/findByStatus` Finds Pets by status

GET `/pet/findByTags` Finds Pets by tags

GET `/pet/{petId}` Find pet by ID

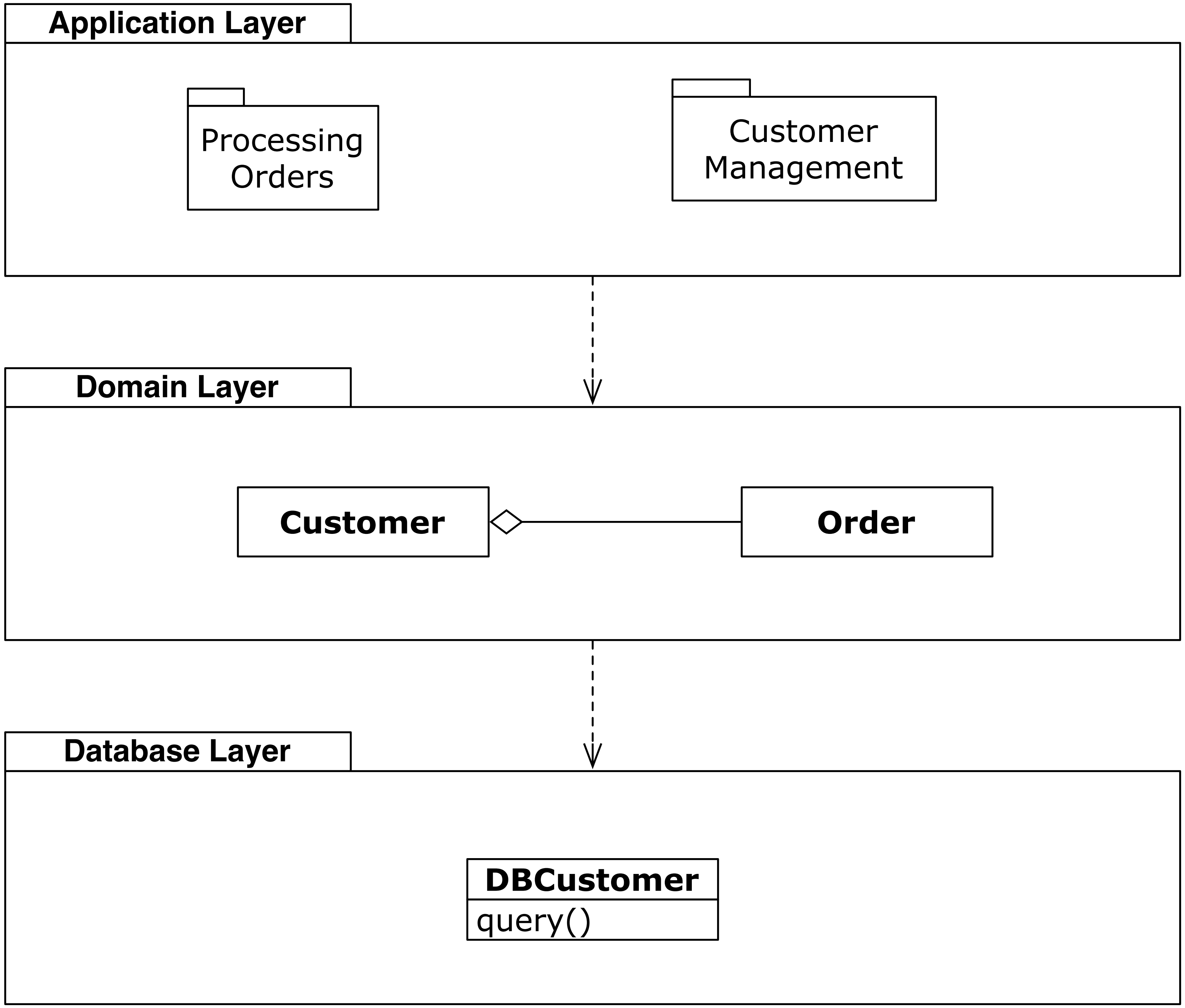
POST `/pet/{petId}` Updates a pet in the store with form data

DELETE `/pet/{petId}` Deletes a pet

POST `/pet/{petId}/uploadImage` uploads an image

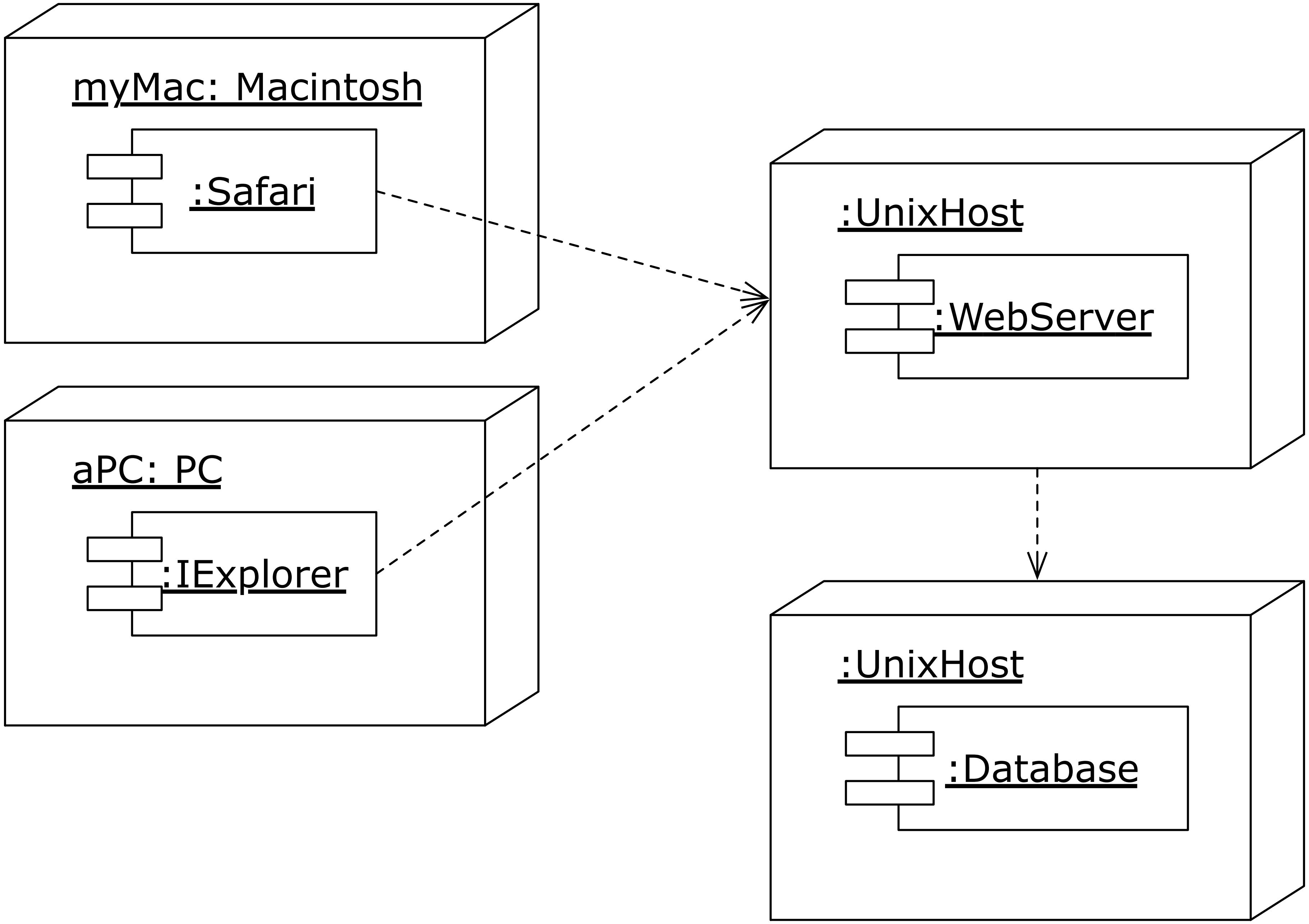
UML: Package Diagram

Decompose system in packages (containing any other UML element, incl. packages)



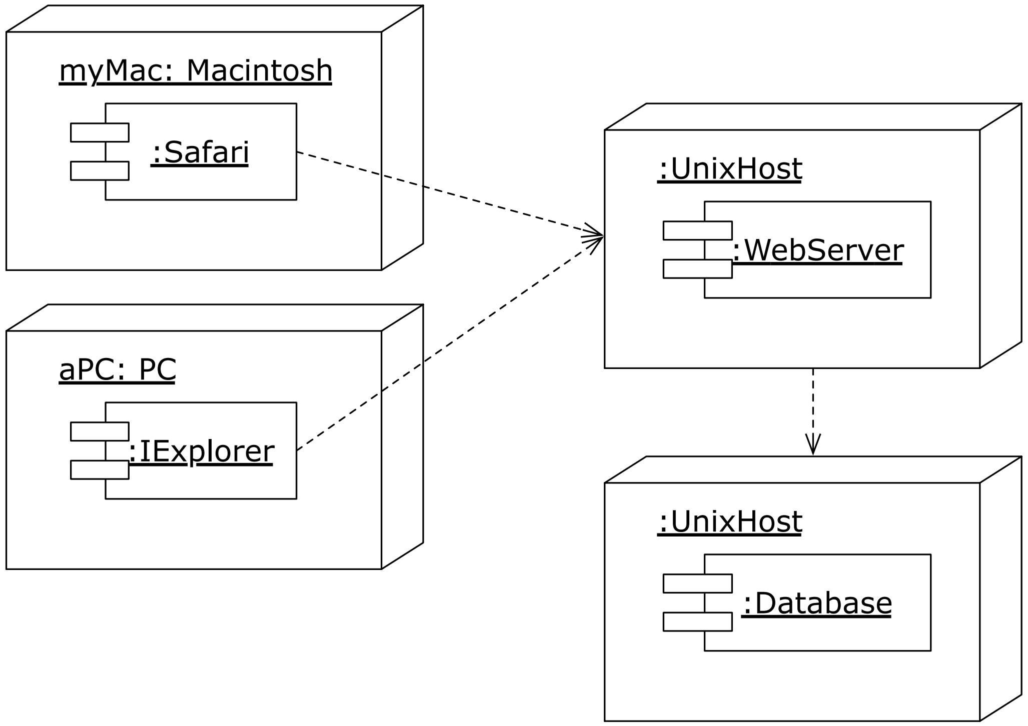
UML: Deployment Diagram

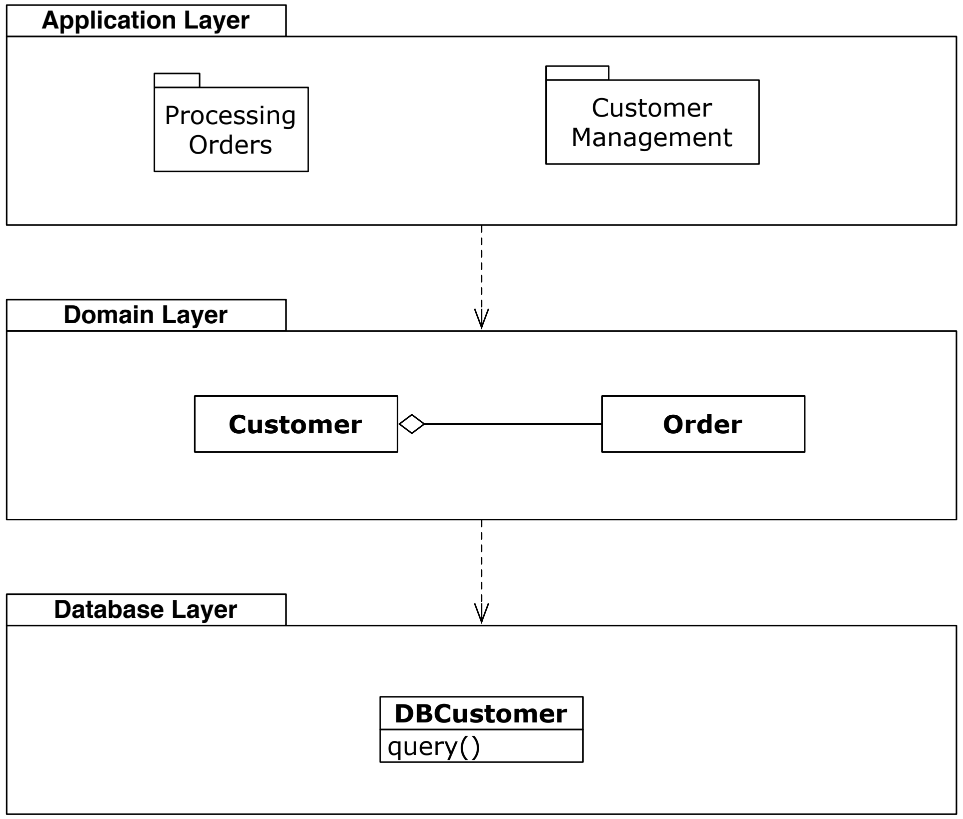
Shows physical lay-out of run-time components on hardware nodes.



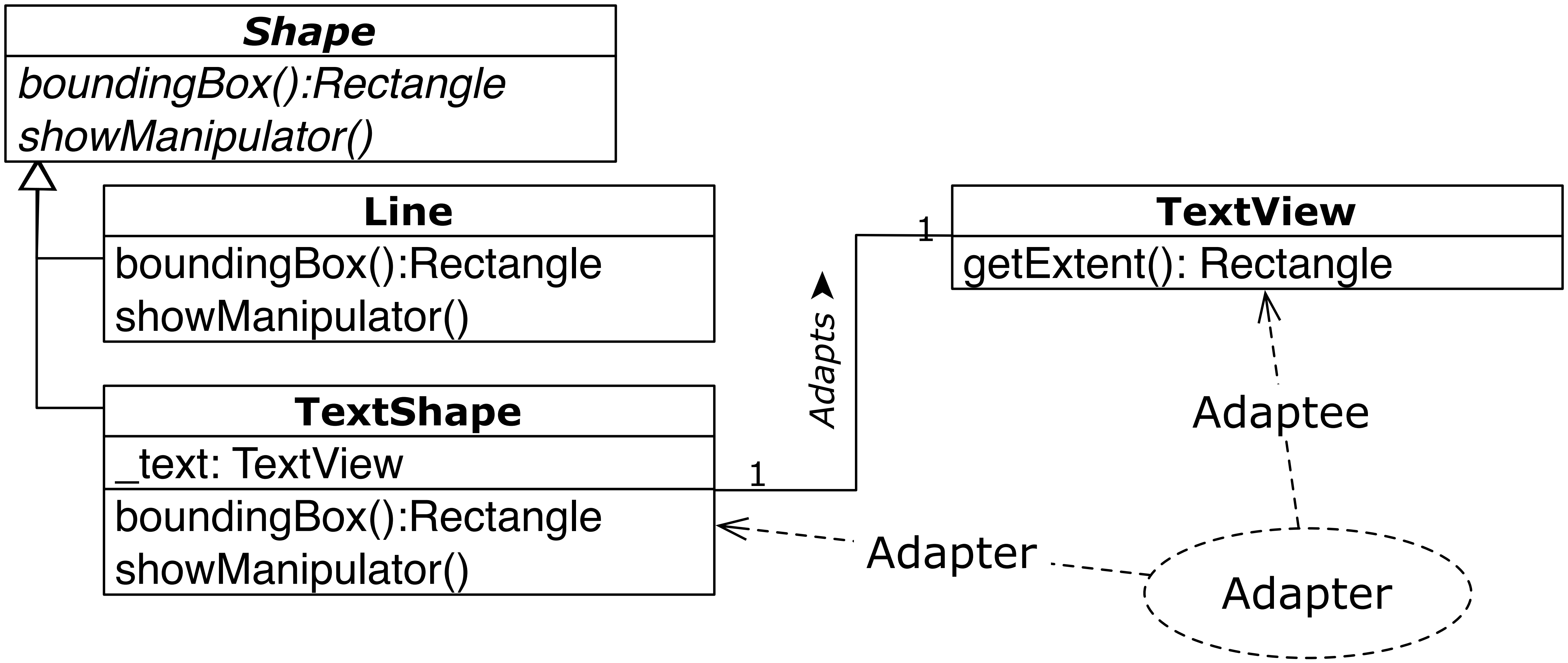
Deployment Diagram vs Package Diagram

- What’s the distinction between a package diagram and a deployment diagram?
- Which one would you use in the 4+1 architectural views?
  (logical view / development view / process view / physical view)





UML: Patterns



Architecture Assessment

Why?

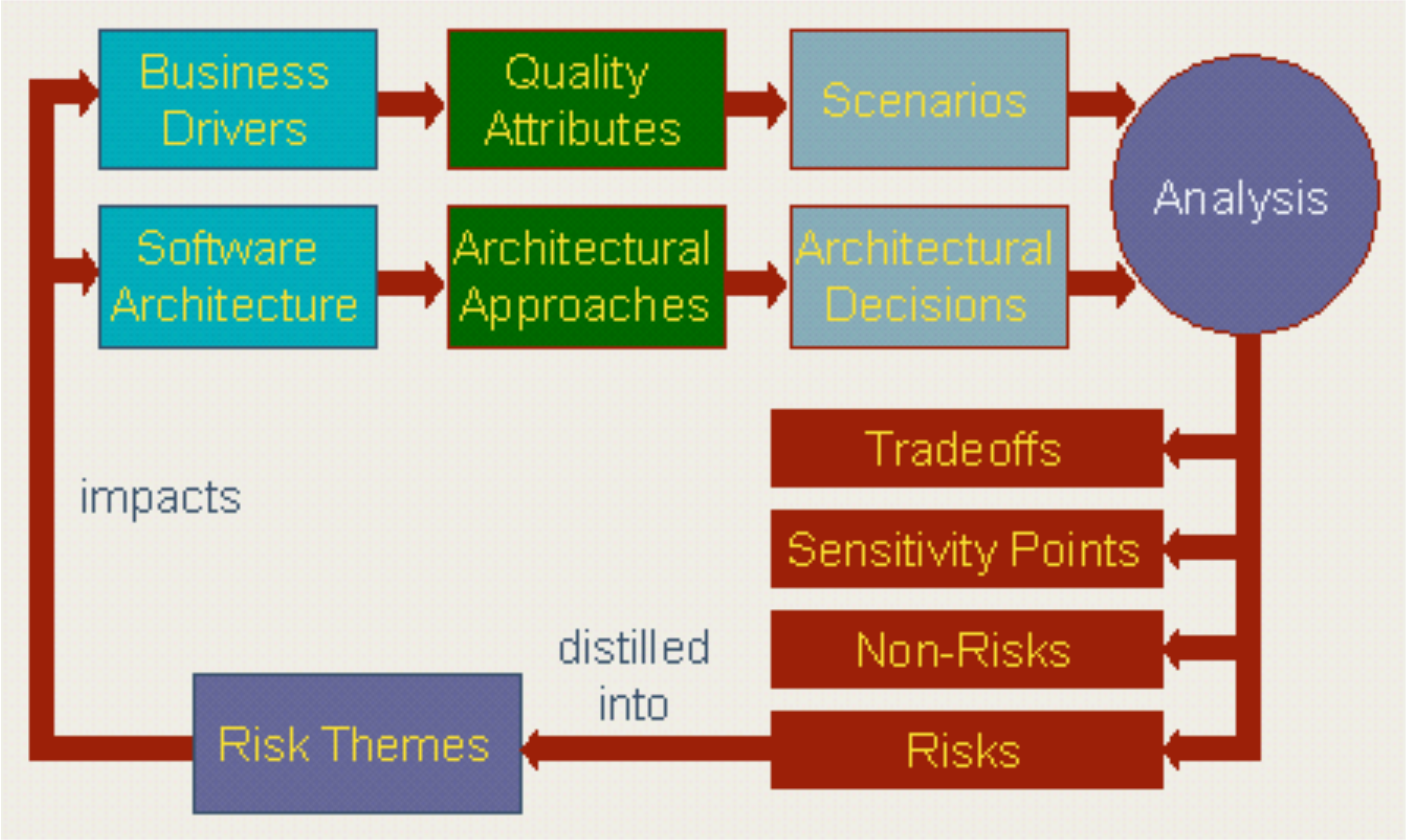
- The earlier you find a problem in a software project, the better.
  + Identify and assess risks!
- An unsuitable architecture is a recipe for disaster.
  + A poor architectural design cannot be rescued by good construction technology.
  + If you wait until the system is built, tackling architectural problems comes at a great cost

Architecture evaluation is a cheap way to avoid disaster.

- Organize review early in the process
  + An architecture evaluation doesn’t tell you “yes” or “no” or “6,75 out of 10”.
    > It tells you where the risks are.

Architecture Tradeoff Analysis Method(ATAM)

- originated from Software Engineering Institute (SEI) at Carnegie Mellon



Answers to two kind of questions:

- Is the architecture *suitable* for the system for which is was designed?
- Which of two or more competing architectures is the most *suitable* one for the system at hand?

ATAM Terminology

Risks are potentially problematic architectural decisions.	The rules for writing business logic modules in the second tier of your three-tier client-server style are not clearly articulated. This could result in replication of functionality, thereby compromising modifiability of the third tier.
Nonrisks are good decisions that rely on assumptions that are frequently implicit in the architecture.	Assuming message arrival rates of once per second, a processing time of less than 30 milliseconds, and the existence of one higher priority process, then a one-second soft deadline seems reasonable.
A **sensitivity point** is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response.	The average number of person-days of effort it takes to maintain the system might be sensitive to the degree of encapsulation of its communication protocols and file formats.
A **trade-off point** involves two (or more) *conflicting* sensitivity points.	If the processing of a confidential message has a hard real-time latency requirement then the level of encryption could be a trade-off point.

Architecture in scrum?

CAPSTONE PROJECT

Spike (a.k.a. Knowledge Acquisition Stories / Proof-of-concept)

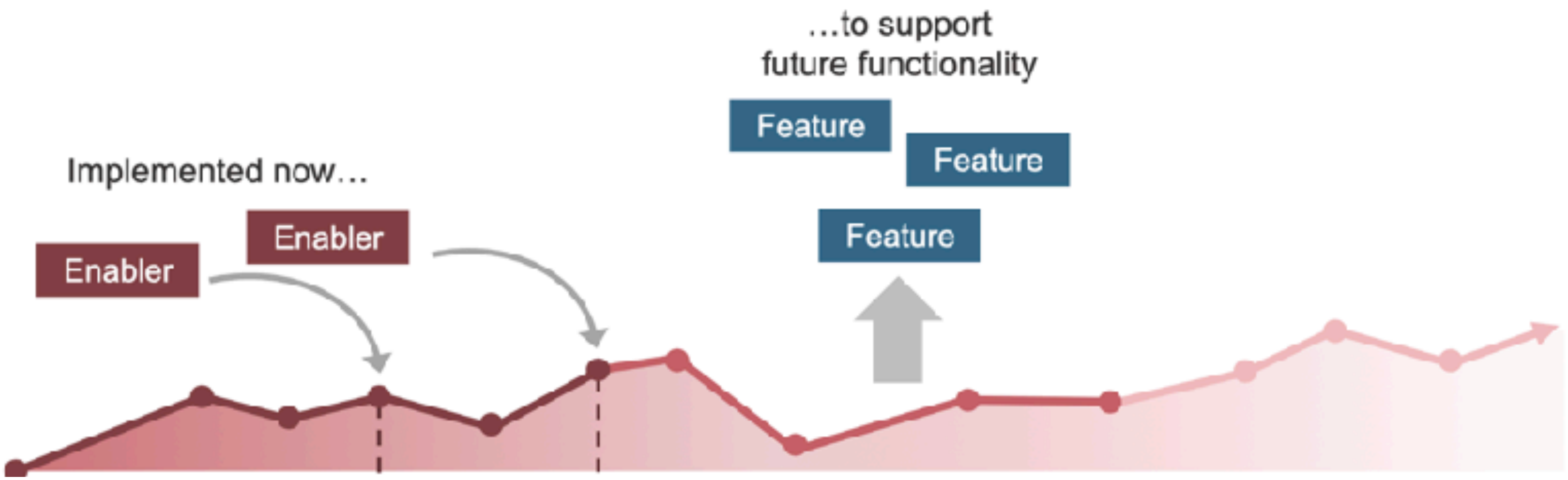
Spike	**Conditions of Satisfaction**
As a developer I want to prototype two alternatives for the … component so that I know ….	• Run Speed Tests • Run Load Tests • Run Security Tests • Write short memo comparing the results

Architecture Runway

*** New ***

While we must acknowledge emergence in design and system development, a little planning can avoid much waste. —James Coplien, Lean Architecture

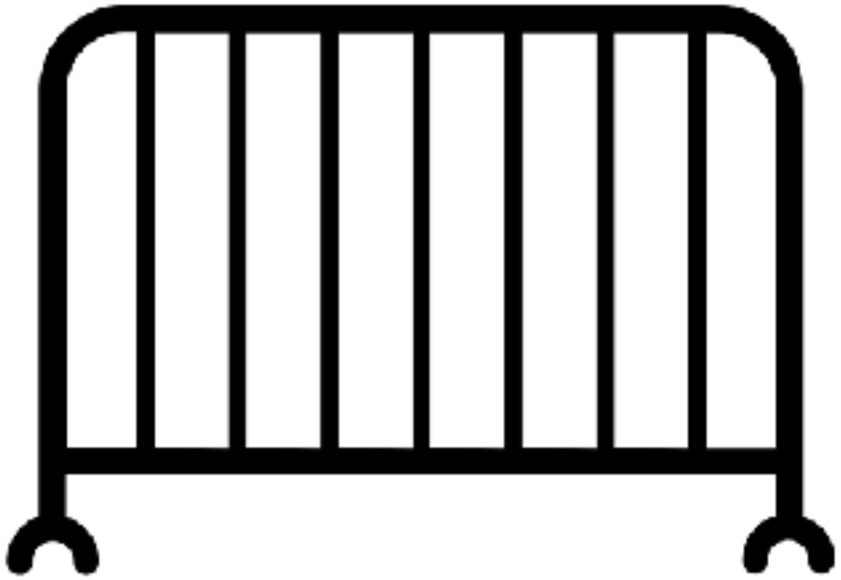
- Agile development avoids big design up-front
  - *emergent design*—defining and extending the architecture only as necessary to deliver the next increment of functionality.
  - *intentional architecture* — requires some centralized planning and cross-team coordination



© Scaled Agile, Inc.

GuardRails

*** New ***



- rules, standards and best practices related to the development pipeline
  + coding, building, testing, release, *design*, …

- Staying behind the guardrails = proceed without consulting other teams
- Moving outside = additional discussion or approval needed
  + Changing existing guardrails?
  + Adopting new guardrails?

Beware

Patterns

- Patterns define the essence of the solution
  > misinterpretation is common among people
- Patterns are “Expert” knowledge
  > “hammer looking for a nail” syndrome
- Patterns introduce complexity (more classes, methods, ...)
  > cost/benefit analysis

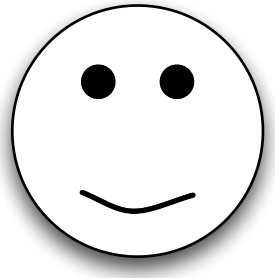
Architecture

- Architecture intends to tackle complexity
  > say less with more
- Architecture implies tradeoffs
  > a boxes and arrows diagram is not an architecture
  (at least consider coupling/cohesion)
- Architectural erosion
  > law of software entropy
  > “Big ball of mud” is most often applied in practice

Correctness & Traceability

Correctness

- Are we building the system right?
  - + Architecture deals with non functional requirements
    - Choosing the best architecture involves tradeoffs
  - + Architecture allows to scale up
    - Organize (testing) work in the team



- Are we building the right system?
  - + Indifferent

Traceability

- Requirements ⇔ System?
  - + Architecture implies extra abstraction level
  - + Software architecture is intangible
    - Traceability becomes more difficult

Summary (i)

You should know the answers to these questions

- What’s the role of a software architecture?
- What is a component? And what’s a connector?
- What is coupling? What is cohesion? What should a good design do with them?
- What is a pattern? Why is it useful for describing architecture?
- Can you name the components in a 3-tiered architecture? And what about the connectors?
- Why is a repository better suited for a compiler than pipes and filters?
- What’s the motivation to introduce an abstract factory?
- Can you give two reasons not to introduce an Adapter (Wrapper)?
- What problem does an abstract factory solve?
- List three tradeoffs for the Adapter pattern.
- How do you decide on two architectural alternatives in scrum?
- What’s the distinction between a package diagram and a deployment diagram?
- Define a sensitivity point and a tradeoff point from the ATAM terminology.

You should be able to complete the following tasks

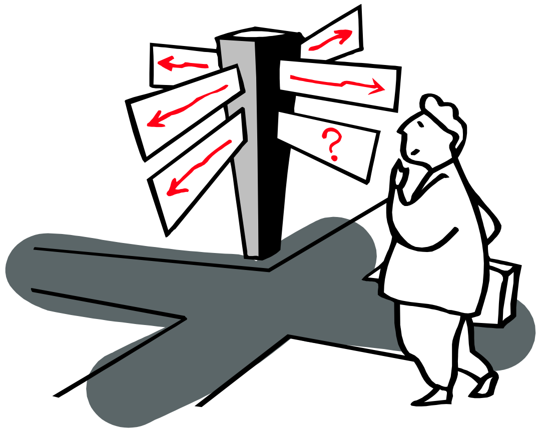
- Take each of the patterns and identify the components and connectors. Then assess the pattern in terms of coupling and cohesion. Compare this assessment with the tradeoffs.

Summary (ii)

Can you answer the following questions?

- What do architects mean when they say “architecture maps function onto form”? And what would the inverse “map form into function” mean?
- How does building architecture relate to software architecture? What’s the impact on the corresponding production processes?
- Why are pipes and filters often applied in CGI-scripts?
- Why do views and controllers always act in pairs?
- Explain the sentence “Restricts communication between subject and observer” in the Observer pattern
- Can you explain the difference between an architecture and a pattern?
- Explain the key steps of the ATAM method?
- *How can you balance emergent design with intentional architecture?*
- *What happens when your team goes outside the boundaries of the guardrail?*
- How would you organize an architecture assessment in your team?

CHAPTER 4 – Project Management



- Introduction
  + When, Why and What?
- Planning & Monitoring
  + PERT charts
  + Gantt charts
  + Uncertainty
    ⇒ Risk to the schedule
  + Dealing with delays
  + Monitoring: earned value analysis
    - Tasks completed, Time sheets
    - Slip Lines, Timelines
  + An afterthought: late projects … started late
- Organisation, Staffing, Directing
  + Belbin Roles
  + *Myers Briggs Type Inventory*
  + Team Structures
  + Directing Teams
- Scrum
  + Definition of Done
  + Scaling Scrum
- Conclusion
  + Correctness & Traceability

Literature

+ [Ghez02] In particular, “Management of Software Engineering”
+ [Pres00] In particular, “Software Project Planning” & “Project Scheduling and Tracking”
+ [Somm05] In particular, “Project Planning” & “Managing People”

- Other
  + [Hugh99] Software Project Management, B. Hughes and M. Cotterell, McGraw Hill, 1999.
    * Good practical examples on PERT, Gantt, Time-sheets, ...

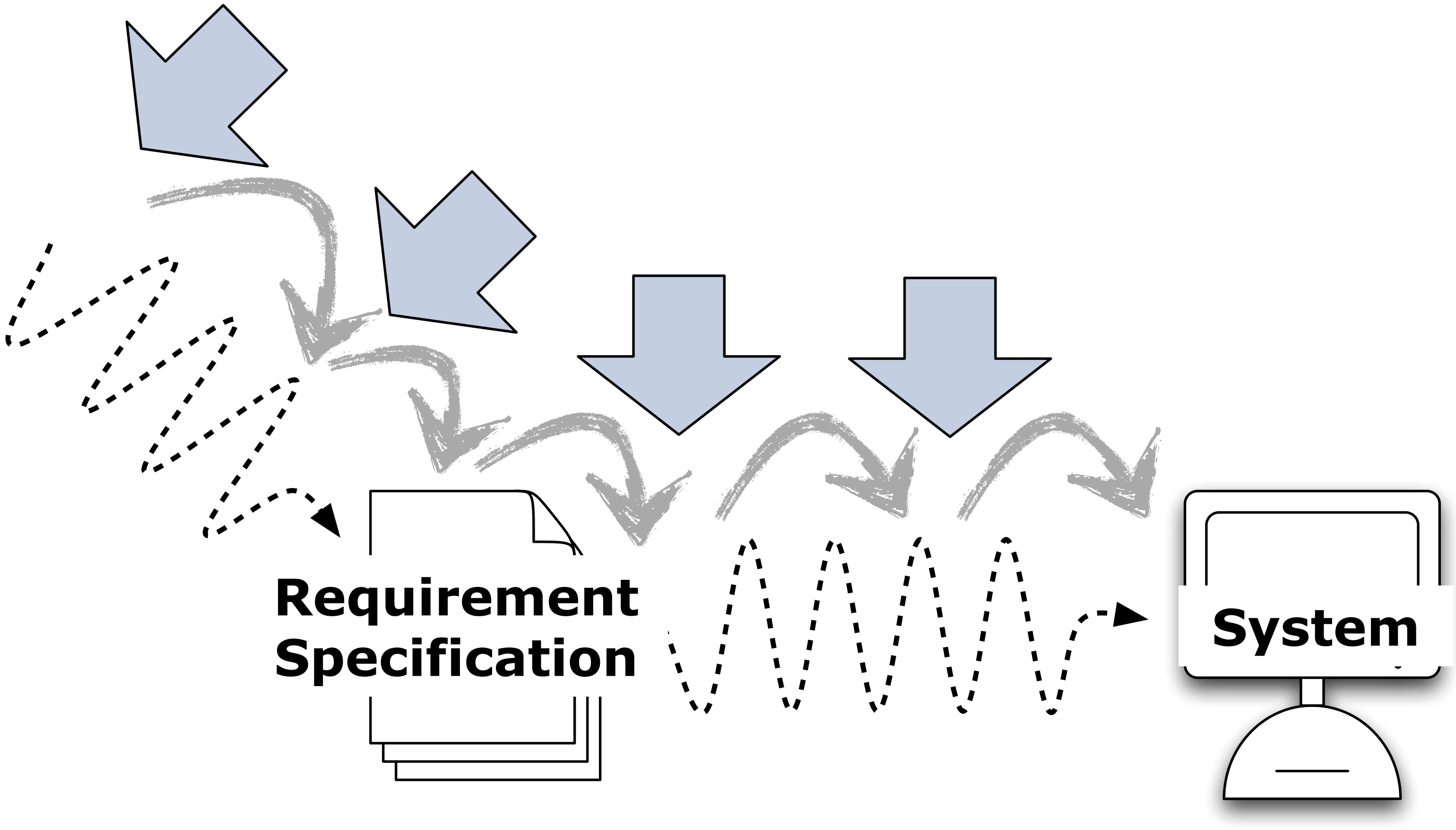
Literature - Papers

- [Henr99] Sallie M. Henry, K. Todd Stevens “Using Belbin's leadership role to improve team effectiveness: An empirical investigation.” ,Journal of Systems and Software, Volume 44, Issue 3, January 1999, Pages 241-250, ISSN 0164-1212.
  + Demonstrating that Belbin roles do make a difference in team efficiency, even for student projects

- [Dema11] Tom De Marco"All Late Projects Are the Same," IEEE Software, pp. 102-103, November/December, 2011
  + All projects that finish late have this one thing in common: they started late.

- [Yoge21] Yogeshwar Shastri, Rashina Hoda, Robert Amor “The role of the project manager in agile software development projects.” Journal of Systems and Software, Volume 173, 2021, 110871. https://doi.org/10.1016/j.jss.2020.110871.
  + Agile projects shouldn’t have project managers … or not?

When Project Management

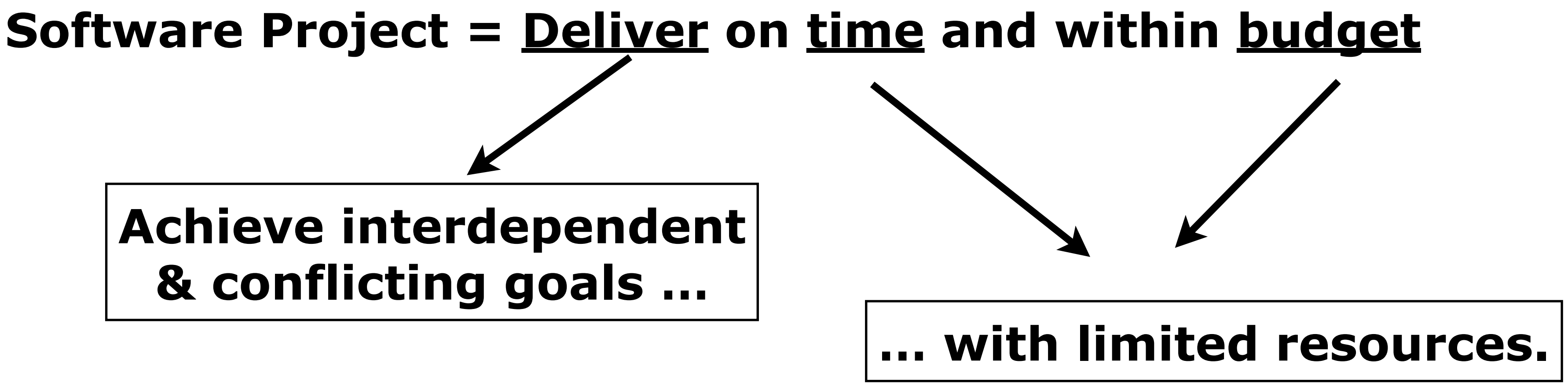


Ensure *smooth* process

Why Project Management?

Almost all software products are obtained via projects.
⇒ Every product is unique

(as opposed to manufactured products)



Software Project = <u>Deliver</u> on <u>time</u> and within <u>budget</u>

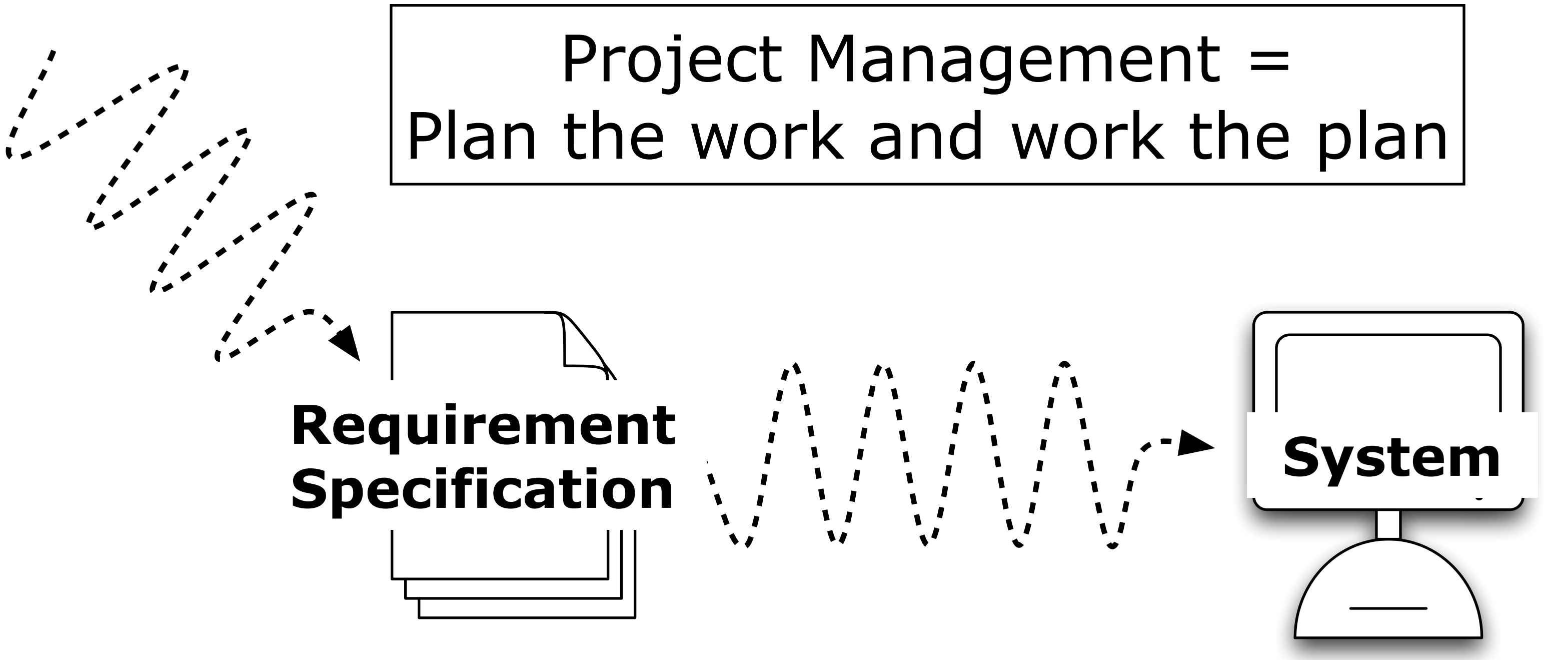
Achieve interdependent & conflicting goals …

… with limited resources.

Your project team is a resource!

What is Project Management?

Project Management =
Plan the work and work the plan



Management Functions

- Planning: Breakdown into tasks + Schedule resources.
- Organization: Who does what?
- Staffing: Recruiting and motivating personnel.
- Directing: Ensure team acts as a whole.
- Monitoring (Controlling): Detect plan deviations + take corrective actions.

Focus of this lecture is Planning & Monitoring.
(Other functions are best learned in real life.)

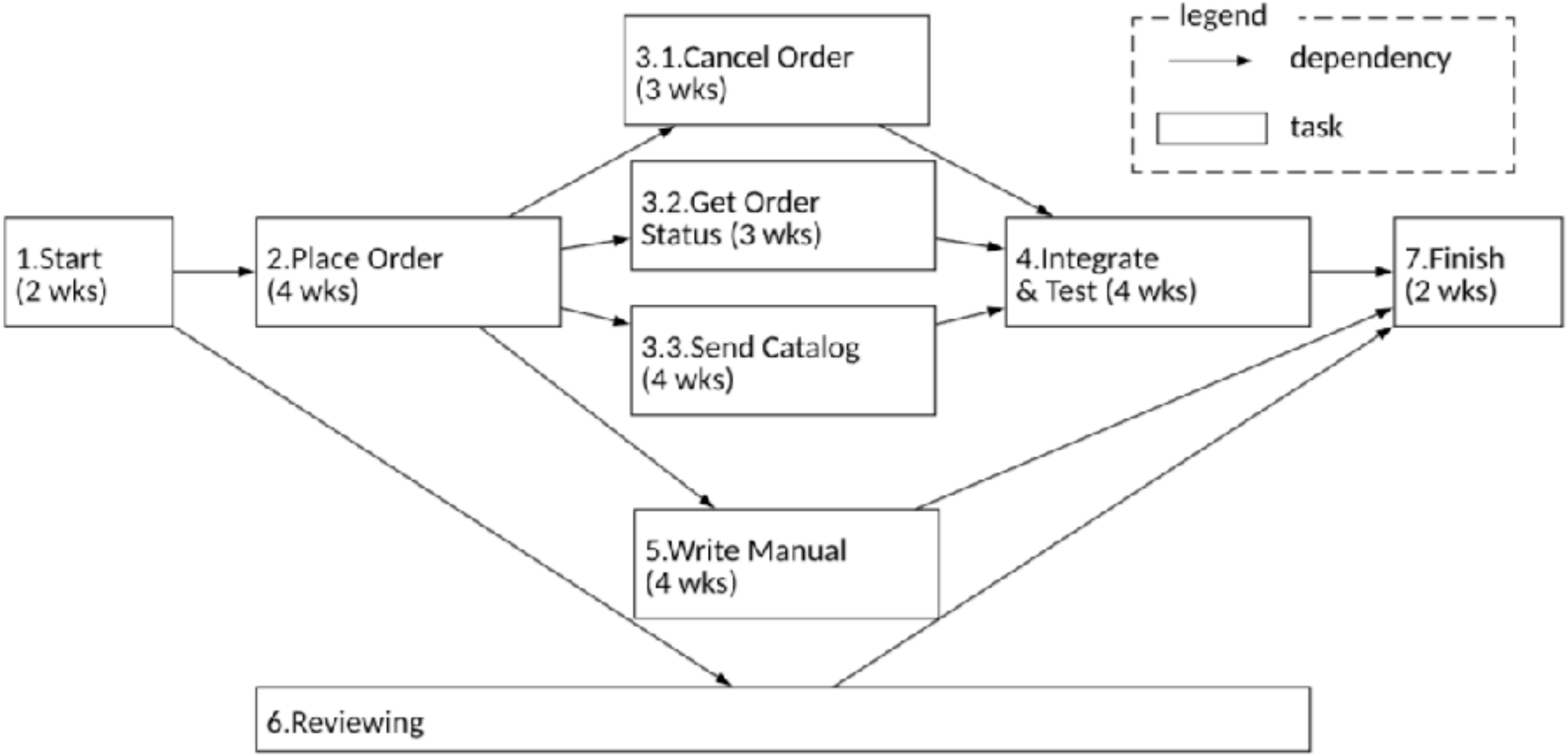
Tasks & Milestones

Good planning depends a lot on project manager's intuition and experience!

- Split project into tasks
  - Tasks into subtasks etc.
- For each task, estimate the task duration
  - Define tasks small enough for reliable estimation.
- Most tasks should end with a milestone.
  - Milestone = A verifiable goal that must be met after task completion
    > Verifiable? .... by the customer
  - Clear unambiguous milestones are a necessity!
    ("80% coding finished" is a meaningless statement)
  - Monitor progress via milestones
- Organize tasks concurrently to make optimal use of workforce
- Define dependencies between project tasks
  + Total time depends on longest (= critical) path in activity graph
  + Minimize task dependencies to avoid delays

Planning is iterative ⇒ monitor and revise schedules during the project!

PERT Chart: Task Dependencies



- 1 start node & 1 end node
- time flows from left to right
- node numbering preserves time dependencies
- no loops, no dangling nodes

Remember: small tasks & milestones verifiable by customer!

Finding the Critical Path

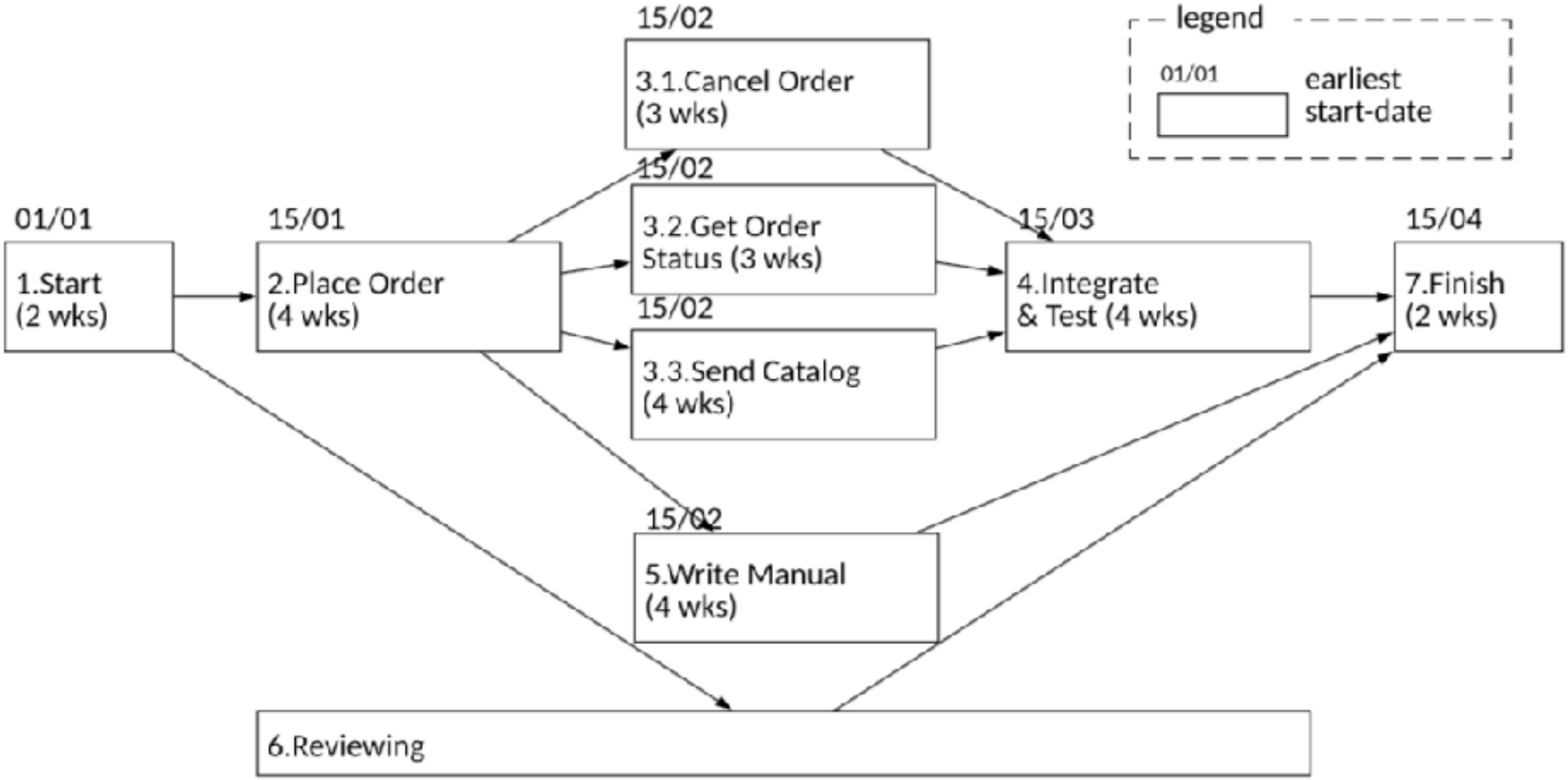
Revised
(Replaced node with task)

- Forward Pass: compute “earliest start-date” (ESD)
  > ESD (start-task) := start-date project

  + Breadth-first enumeration (use node numbering)
  + For each task: compute earliest start-date
    = Latest of all incoming paths
    > ESD (task) := latest of (
    > ESD (preceding task) + estimated task duration (preceding task))
- Backward Pass: compute “latest end-date” (LED)
  > LED (end-task) := ESD (end-task) + estimated task duration

  + Breadth-first enumeration (node numbering in reverse order)
  + For each task: compute latest end-date
    = Earliest of all outgoing paths
    > LED (task):= earliest of (
    > LED (subsequent task) - estimated task duration (subsequent task))
- Critical Path
  + = path where delay in one task will cause a delay for the whole project
  + path where for each task:
    > ESD(task) + estimated time (task)= LED(task)

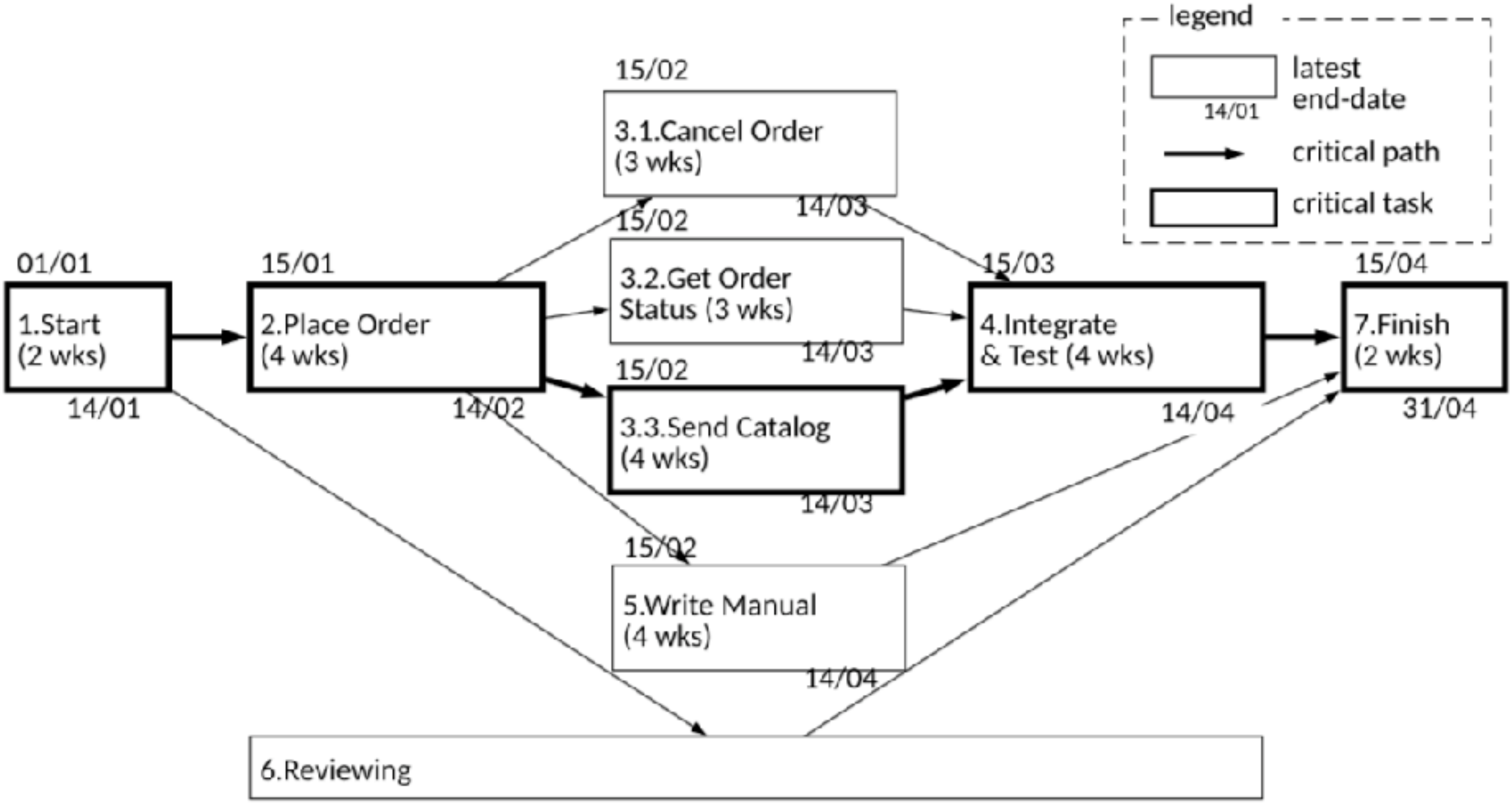
PERT Chart: Forward pass



This is a schedule with coarse grained granularity: 1 month is 4 weeks of 7 days (week 1 = 1-7; week 2 = 8-15; …)

ESD(1) := start-date project
ESD(2) := ESD(1) + time(1) := 01/01 + 2 weeks := 15/01
ESD(4) := latest (ESD(3.1) + 3 wks, ESD(3.2) + 2 wks, ESD(3.3) + 4 wks) := 15/03

PERT Chart: Backward pass + Critical path



- LED(7) := ESD(7) + time(7) := 15/04 + 2 wks := 31/04
- LED(6) := LED(7) - time(7) := 31/04 - 2 wks := 14/04
- LED(2) := earliest (LED(3.1) - 3 wks, LED(3.2) - 3 wks, LED(3.3) - 4 wks) := 14/02

When to use PERT Charts?

- Good for: Task interdependencies
  + shows tasks with estimated task duration
  + links task that depend on each other
  (depend = cannot start before other task is completed)
  + optimise task parallelism
  + monitor complex dependencies

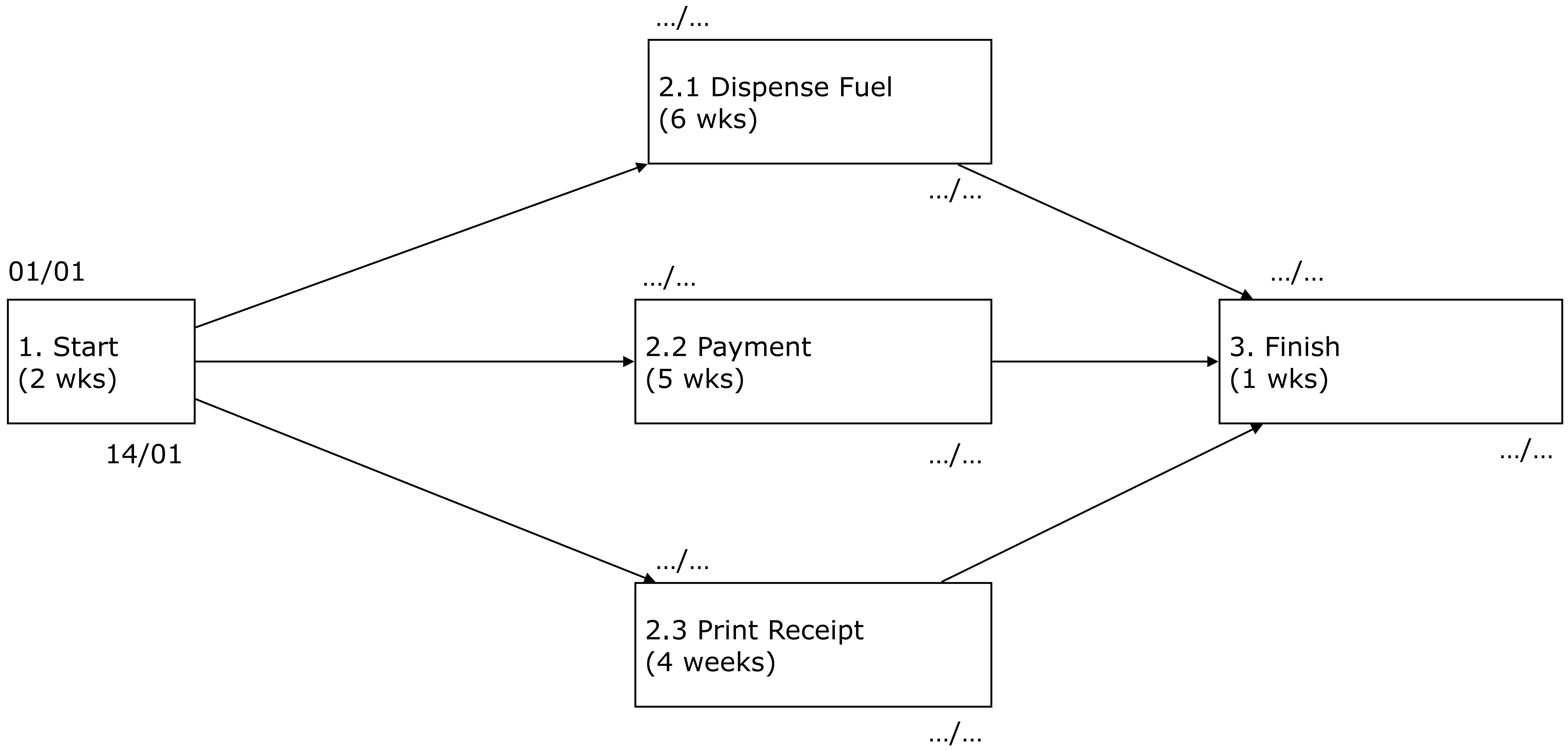
- Good for: Critical Path Analysis
  + calculate for each task: earliest start-date, latest finish-date
  (latest start-date, latest finish-date)
  + optimise resources allocated to critical path
  + monitor critical path

- Not for: Time management

(N.B.: PERT = Program Evaluation and Review Technique)

Critical Path (exercise)

Identify the critical path
- Forward pass: earliest start data
- Backward pass: latest end date



01/01

1. Start
(2 wks)

14/01

…/…

2.1 Dispense Fuel
(6 wks)

…/…

…/…

2.2 Payment
(5 wks)

…/…

…/…

2.3 Print Receipt
(4 weeks)

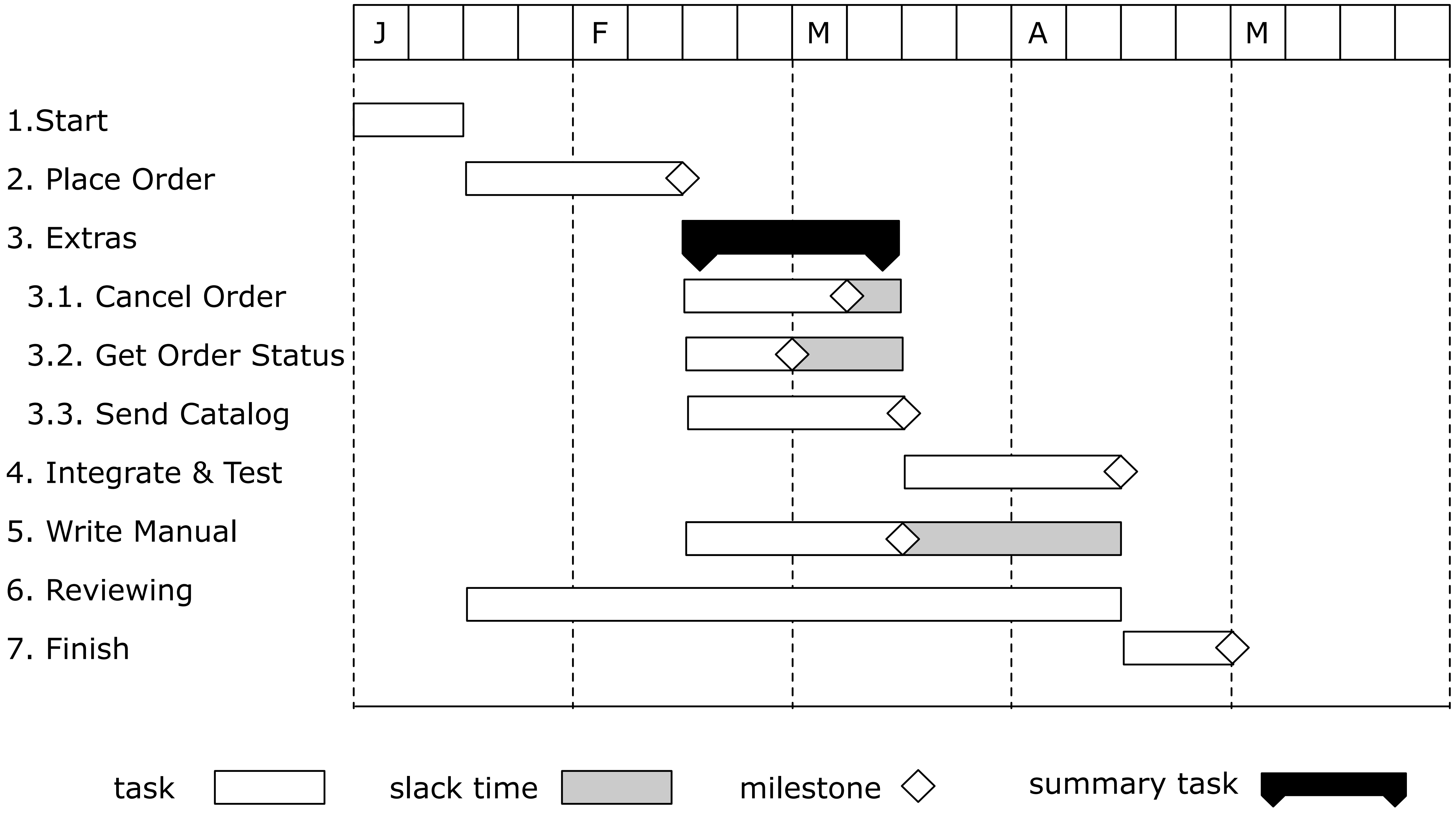
…/…

…/…

3. Finish
(1 wks)

…/…

Gantt Chart: Time Management



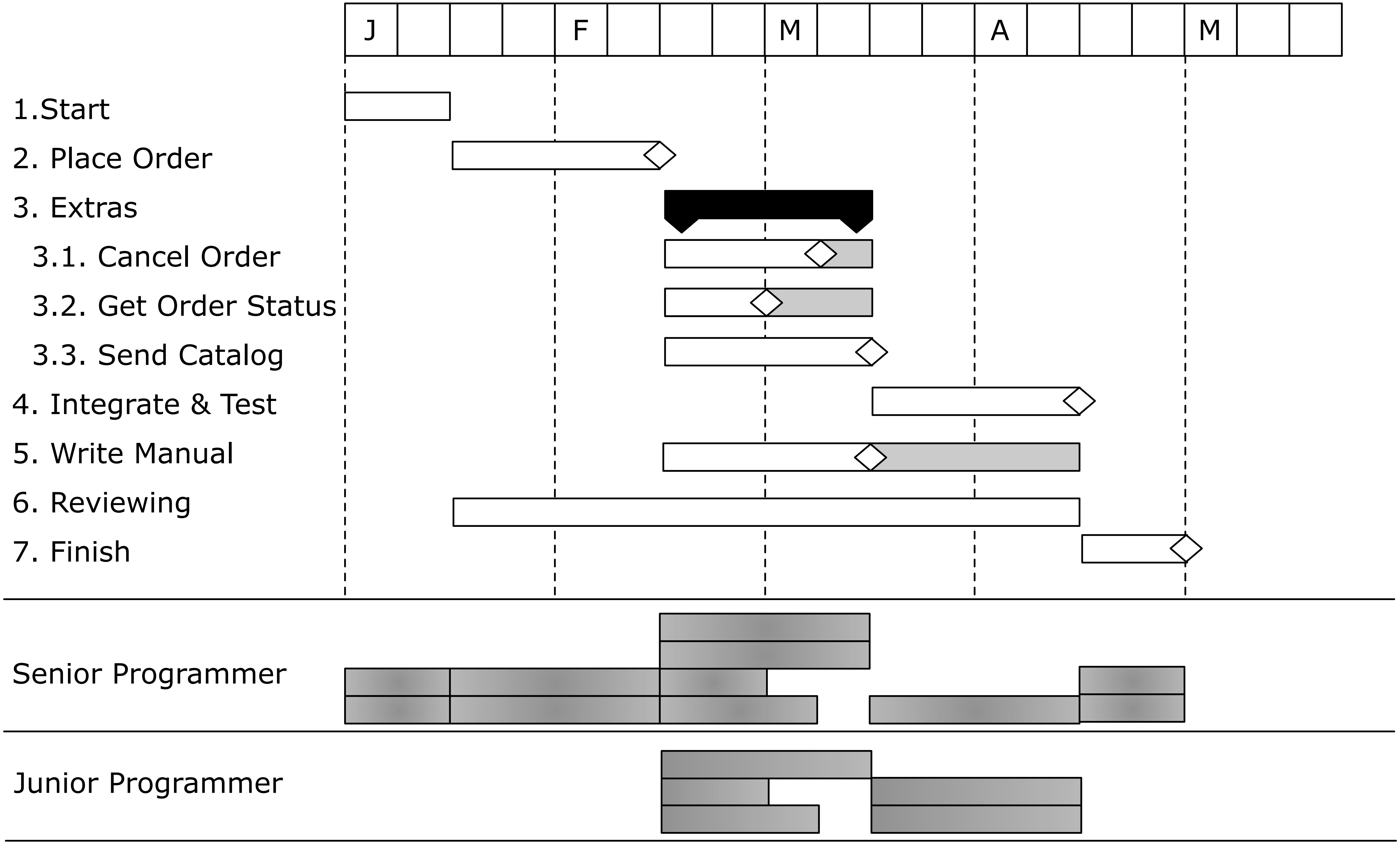
Resource Allocation

For each task, list the required resources.

- Mainly staff (incl. type of skills required)
- ... and special equipment

Activity	Resource	Time	Quantity	Notes
1	Senior Programmer	2 wks	2	Initially senior programmers only
2	Senior Programmer	4 wks	2	
3.1	Senior Programmer	3 wks	1	
	Junior Programmer	3 wks	1	Implementation: extra junior staff
3.2	Senior Programmer	2 wks	1	
	Junior Programmer	2 wks	1	
3.3	Senior Programmer	4 wks	1	
	Junior Programmer	4 wks	1	
4	Senior Programmer	4 wks	1	
	Junior Programmer	4 wks	2	
5	Senior Programmer	4 wks	1	
	Writer	4 wks	1	Manual
6	Quality Engineer	1 day/wk	1	Assistance from QA department
7	Senior Programmer	2 wks	2	

Gantt Chart: Resource Allocation

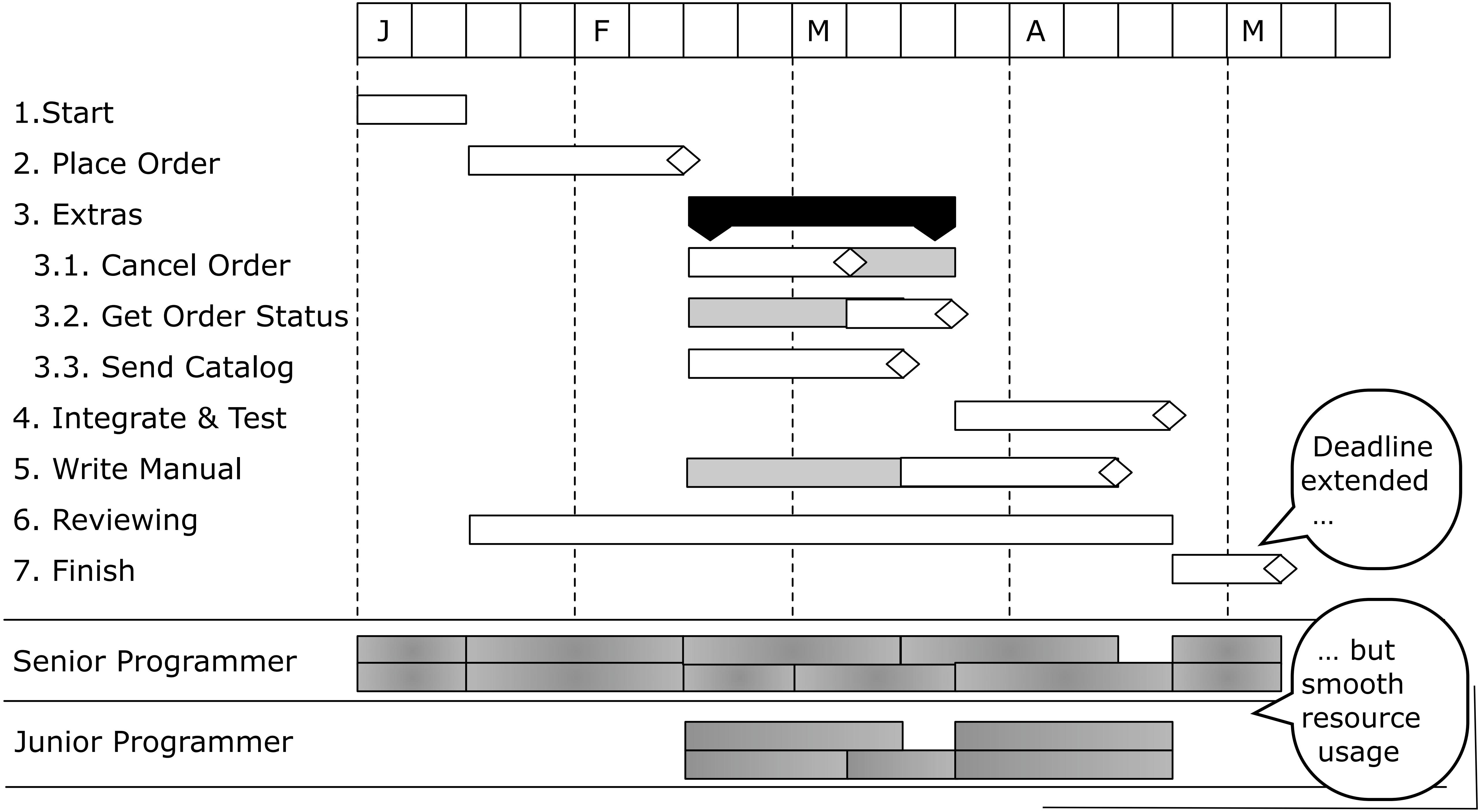


Scheduling tasks at earliest start dates typically gives uneven resource distribution!

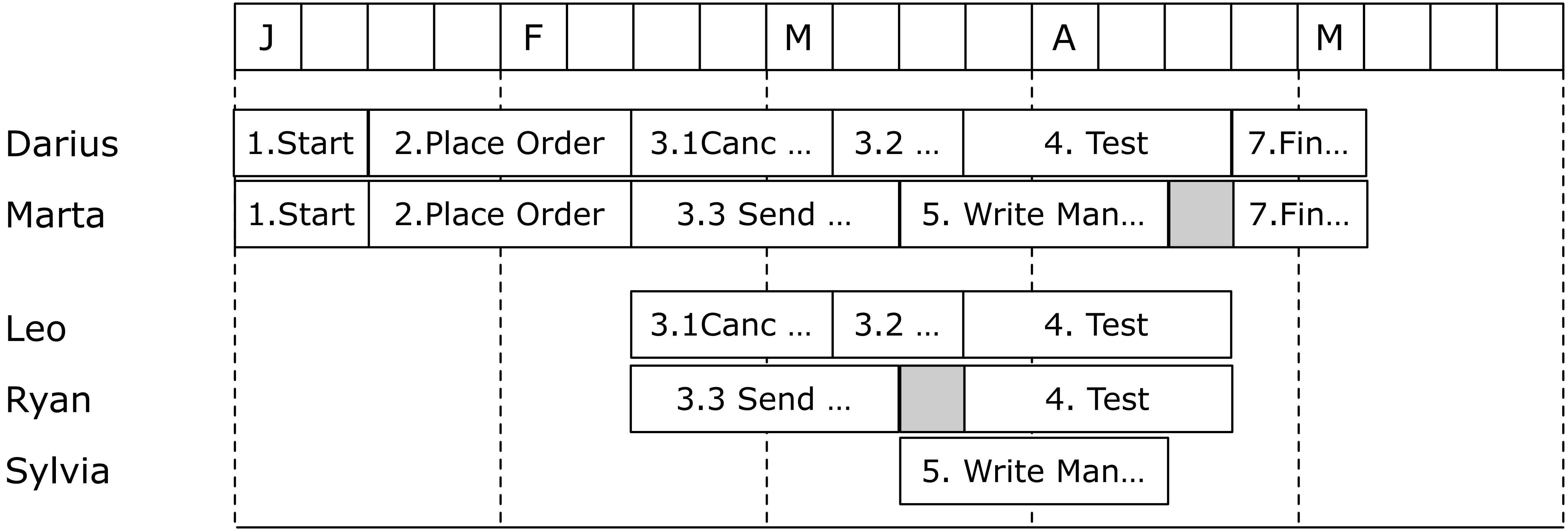
Gantt Chart: Optimized Resources

Shuffle tasks in time to optimise use of resourceS

- Distribute resources evenly (or with a smooth build-up and run-down)
- May require to extend termination date or to split tasks



Gantt Chart: Staff Allocation



(Overall tasks such as reviewing, reporting, ... are difficult to incorporate)

When to use Gantt Charts?

- Good for: Time management
  + shows tasks in time
  + optimise resources by managing “slack time”
  + monitor critical tasks (= without slack time)

- Good for: Resource and staff allocation
  + shows resource/staff occupation
  + optimize “free time” (= time without occupation)
  + monitor bottlenecks (= fully occupied resources / staff)

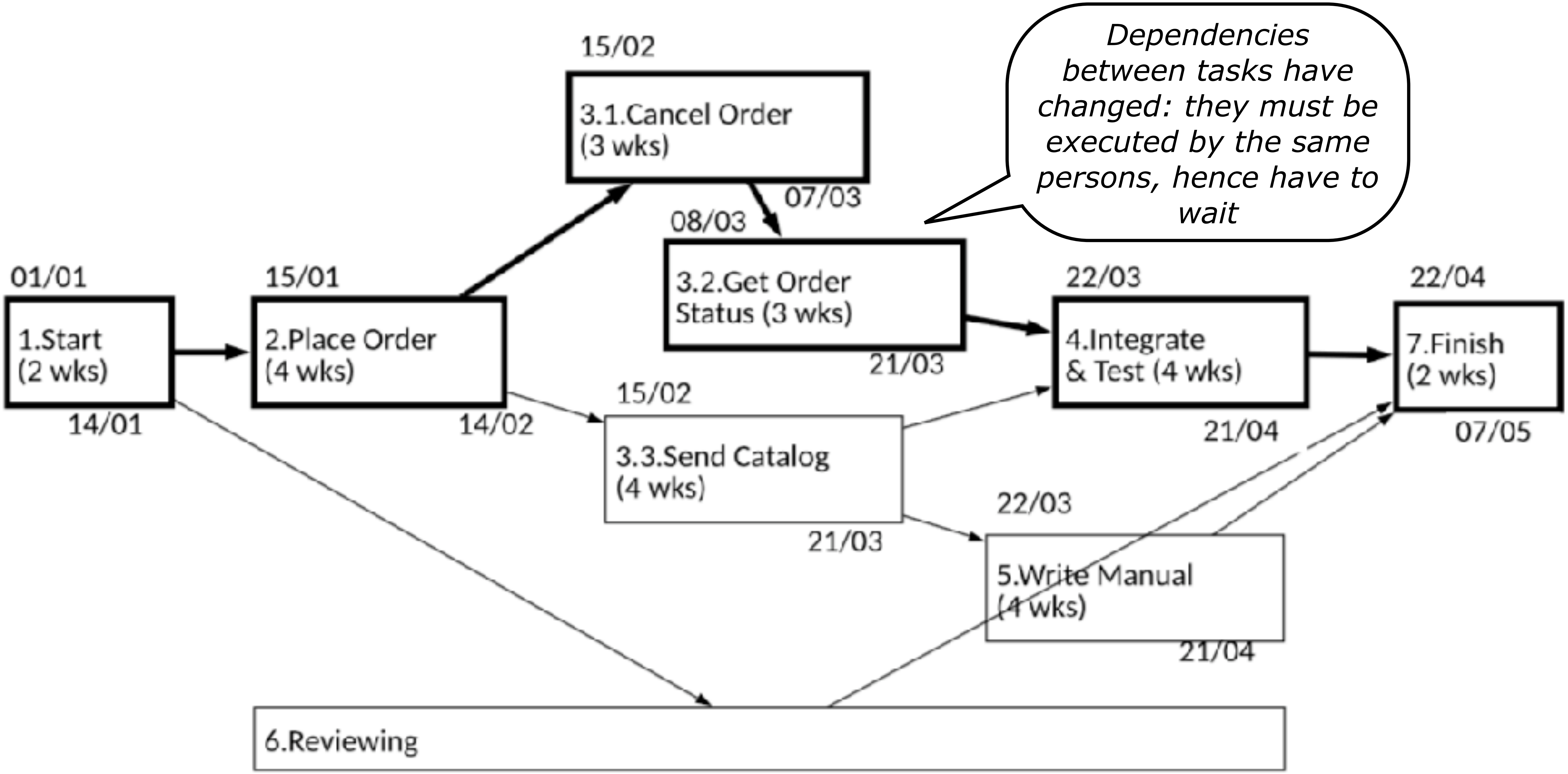
- Not for: Task Interdependencies

(N.B. Charts are developed by Henry Gantt; hence the name)

PERT Chart: Including Resources

Due to allocated resources, implicit dependencies are added...

- may give rise to different critical path
- may break “encapsulation” between groups of project tasks



Dependencies between tasks have changed: they must be executed by the same persons, hence have to wait

Uncertainty

- Planning under uncertainty
  + State clearly what you know and don’t know
  + State clearly what you will do to eliminate unknowns
  + Make sure that all early milestones can be met
    > However: tackle critical risks early

- Get commitment
  + from main parties involved, incl. management
  + The difference between “involvement” and “commitment”? In a Ham and Egg Breakfast...the chicken is involved and the pig is committed!

- Build confidence
  + within the team
  + with the customer
    > ... re-planning will not be considered harmful

A software project is like skiing down a black piste.
The ultimate goal is clear: getting down in one piece.
The way to reach the goal? ... One turn at a time. (See [Gold95])

Knowns & Unknowns

[This is terminology used for planning military campaigns.]
Phillip G. Armour, "The Five Orders of Ignorance", COMMUNICATIONS OF THE ACM October 2000

Slide Repeated from Intro

Known knowns

- = the things you know you know
  You can safely make assumptions here during planning

Known unknowns

- = the things you know, you don't know
  You can prepare for these during planning

Unknown unknowns

- = the things you do not know, you don't know
  These you cannot prepare for during planning
  … the best you can do is being aware and spot opportunities
  + do a thorough *risk analysis*

- software projects (compared to other engineering projects) have lots of "unknown unknowns"
  + Not constrained by physical laws
  + Many stakeholders ⇒ strong political forces around project

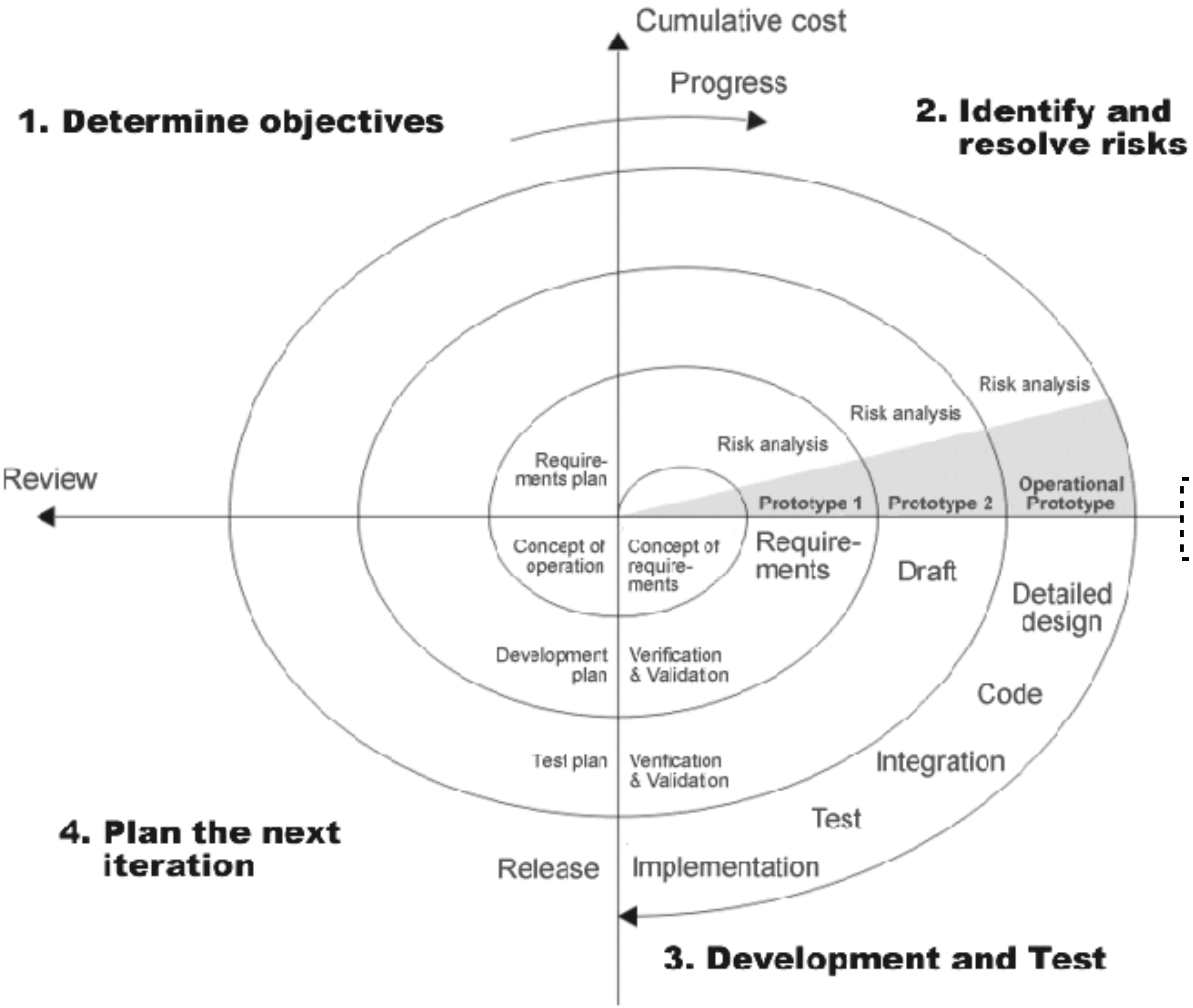
Inception: Risk Factors

Repeated from Requirements

- During inception you must identify the project’s *risk factors*
  + you do not have control over the system’s context and it will change
  + projects never go according to plan
    > identify potential problems early (… including wild success)
- Example

Context	Risk Factors	Impact	Likely	Urgency
Competitors	Time to market (too late/too early)			
Market trends	More internet at home			
Potential disasters	Suppliers don’t deliver on time			
	System is down			
Expected users	Too many/few users			
Schedule	Project is delivered too early/too late	<< Risky Path (Project Management)		
Technology	Dependence on changing technology			
	Inexperienced team			
	Interface with legacy systems			

Risk Analysis: Quantify Risks for Delays



Quantify Risk: Calculate the risk to the global schedule from the risk on the individual tasks

go, no-go decision

© Image adapted from Boehm, B. (1988) A Spiral Model of Software Development and Enhancement. IEEE Computer, 21 (5), 62-72.

Calculating Risky Path (1/2)

** Revised **
(Improved Formulas)

- (This calculation is an advanced but crucially important part of PERT)
- Estimate Task Time
  + For each task, estimate
    - likely time LT(task), optimistic time OT(task), pessimistic time PT(task)
    - deduce estimated time (= weighted average)

$$ET(task) = \frac{OT(task) + 4 \cdot LT(task) + PT(task)}{6}$$

    - Redo the critical path analysis with the estimated time ET
- Calculate Standard Deviation *per Task*
  + For each task, calculate the degree of uncertainty for the task time

$$S(task) = \frac{PT(task) - OT(task)}{6}$$

Example: Calculating Risk (1/2)

** Revised **
(Improved Formulas)

- Optimistic Time, Likely Time and Pessimistic Time is given
- deduce estimated time ET(task)
  + Redo the critical path analysis with ET

$$ET(task) = \frac{OT(task) + 4 \cdot LT(task) + PT(task)}{6}$$

- calculate standard deviation S(task)

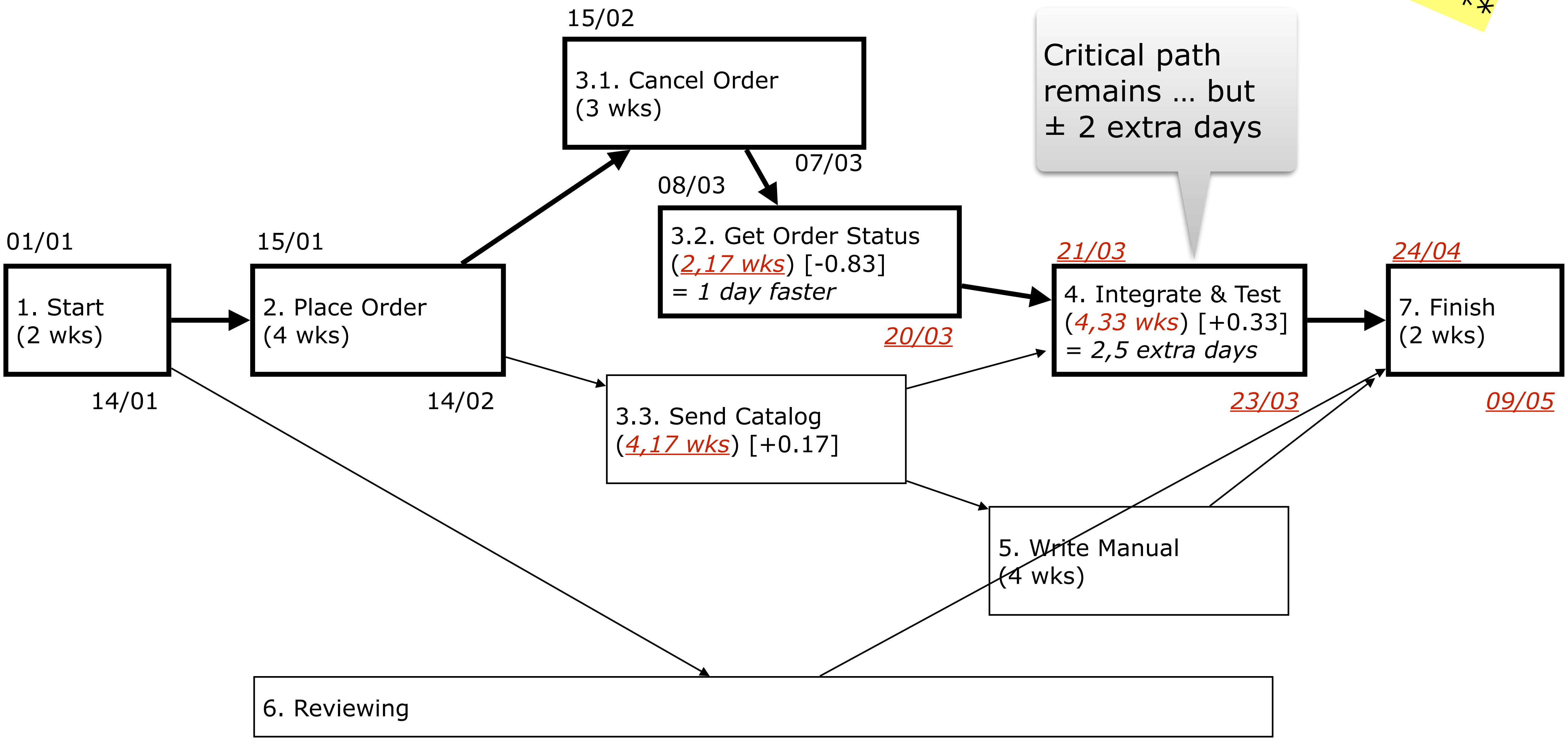
$$S(task) = \frac{PT(task) - OT(task)}{6}$$

	OT	LT	PT	ET	S
1.Start	2	2	2	2	0
2.Place Order	3	4	5	4	0,33
3.1.Cancel	2	3	4	3	0,33
3.2.Get Order	2	2	3	2,17	0,17
3.3.Send Catalogue	3	4	6	4,17	0,50
4.Test	4	4	6	4,33	0,33
5.Manual	3	4	5	4	0,33
7.Finish	2	2	2	2	0

Task 3.3 is riskiest task
(interface with legacy database)

Example: Redo Critical Path with ET

*** New ***



Critical path remains … but ± 2 extra days

15/02
3.1. Cancel Order (3 wks)
07/03
08/03
3.2. Get Order Status (2,17 wks) [-0.83] = 1 day faster
20/03
01/01
1. Start (2 wks)
14/01
15/01
2. Place Order (4 wks)
14/02
3.3. Send Catalog (4,17 wks) [+0.17]
21/03
4. Integrate & Test (4,33 wks) [+0.33] = 2,5 extra days
23/03
24/04
7. Finish (2 wks)
09/05
5. Write Manual (4 wks)
6. Reviewing

Calculating Risky Path (2/2)

** Revised **
(Improved Formulas)

- Forward Pass: Calculate Standard Deviation *per Path*
  + For each possible *path up until a given task n*
    - calculate the degree of uncertainty for the path execution time
      * Paths with a high deviation are likely to slip.

$$S(path) = \sqrt{\sum_{task \in path} S(task)^2}$$

  + For each task: compute standard deviation per path leading into the task
      * Degree to which a given task may end later than planned
      * = Maximum of all standard deviations for incoming paths

$$SP(task) = \max_{path \in incoming} S(path)$$

Results of Risky path Analysis

** Revised **
(Improved Formulas)

- Riskiest Task = the node with the highest risk for delay

  > Maximum for all S(task)

$$S(task) = \frac{PT(task) - OT(task)}{6}$$

- Risky Path = start-to-end path(s) with the highest standard deviation

  > Risky path applies to the whole PERT chart!

  > SP (end) := maximum of all incoming paths for end node

- Worst Case Delay: Applies to the risky path(s) only

  > = worst case impact the risky path may have on the end date

$$WorstCaseDelay(path) = \sum_{task \in path} PT(task) - LT(task)$$

Example: Calculating Risk (2/2)

** Revised ** (Improved Formulas)

- For each task n: compute standard deviation per path

  > = Maximum of all standard deviations for incoming paths

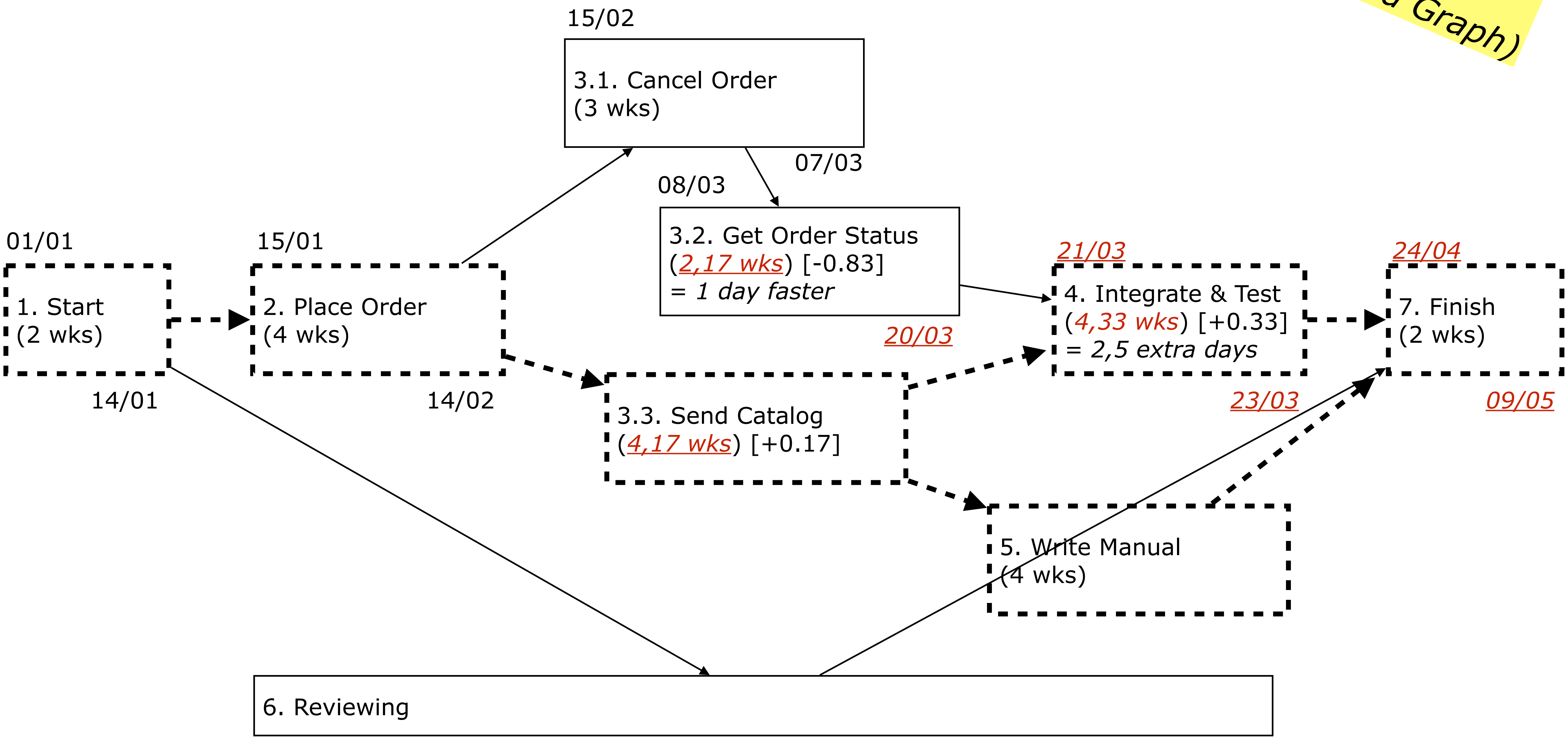
$$S(path) = \sqrt{\sum_{task \in path} S(task)^2}$$

End Node	path	$S(m_1)$	$S(m_2)$	$S(m_3)$	$S(m_4)$	$S(m_5)$	$S(m_6)$	$\sqrt{(\Sigma S(m_i)^2)}$	
1.Start	1	0						0	
2.Place O.	1,2	0	0,33					0,33	
3.1.Cancel	1,2,3.1	0	0,33	0,33				0,4667	
3.2.Get O.	1,2,3.1,3.2	0	0,33	0,33	0,17			0,4967	
3.3.Send C.	1,2,3.3	0	0,33	0,5				0,5991	
4.Test	1,2,3.1,3.2,4	0	0,33	0,33	0,17	0,33		0,5963	
	1,2,3.3,4	0	0,33	0,5	0,33			*0,684*	<< max
5.Manual	1,2,3.3,5	0	0,33	0,5	0,33			*0,684*	
7.Finish	1,2,3.1,3.2,4,7	0	0,33	0,33	0,17	0,33	0	0,5963	
	1,2,3.3,4,7	0	0,33	0,5	0,33		0	*0,684*	<< max
	1,2,3.3,5,7	0	0,33	0,5	0,33		0	*0,684*	<< max

⇒ Paths 1,2,3.3,4,7 and 1,2,3.3,5,7 represent largest risk!

Example: Risky Path

** Revised **
(Improved Graph)



- Worst case delay ("pessimistic time" minus "likely time" for all tasks on risky path)
  + 1,2,3.3,4,7: 0 + 1 + 2 + 2 + 0 = 5 extra weeks
  + 1,2,3.3,5,7: 0 + 1 + 2 + 1 + 0 = 4 extra weeks
- Risk analysis: can the project afford such delays? Customers decision; if not … no-go!

$$WorstCaseDelay(path) = \sum_{task \in path} PT(task) - LT(task)$$

Calculating Risk: exercise

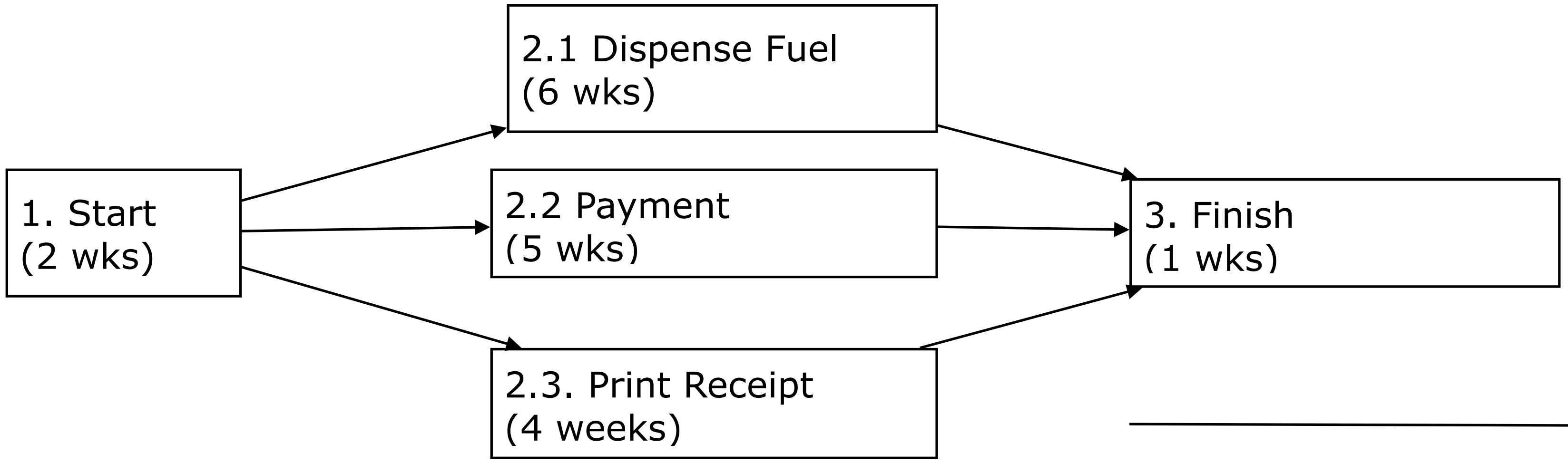
** New **

- What is the riskiest task?
- What is riskiest path?
- What is the worst case delay?

$$S(task) = \frac{PT(task) - OT(task)}{6}$$

$$S(path) = \sqrt{\sum_{task \in path} S(task)^2}$$

	OT	LT	PT	S	path	S(path)
1.Start	2	2	2		1	
2.1 Dispense Fuel	5	6	8		1,2.1	
2.2 Payment	4	5	8		1,2.2	
2.3 Print Receipt	3	4	5		1,2.3	
3. Finish	1	1	1		1,2.1, 3	
					1,2.2, 3	
					1,2.3, 3	



1. Start
(2 wks)

2.1 Dispense Fuel
(6 wks)

2.2 Payment
(5 wks)

2.3. Print Receipt
(4 weeks)

3. Finish
(1 wks)

Delays & Options

+ Assume that you have the following two options

Early with big risk for delay	Later with small risk for delay
delivery of project within 4 (four) months … but can be 1 month early … or 4 months late!	delivery of project within 5 (five) months … at maximum 1 week late … or 1 week early.

+ What would you choose?
+ What do you think upper management would choose? (*)

(*)Most managers would choose option 2!

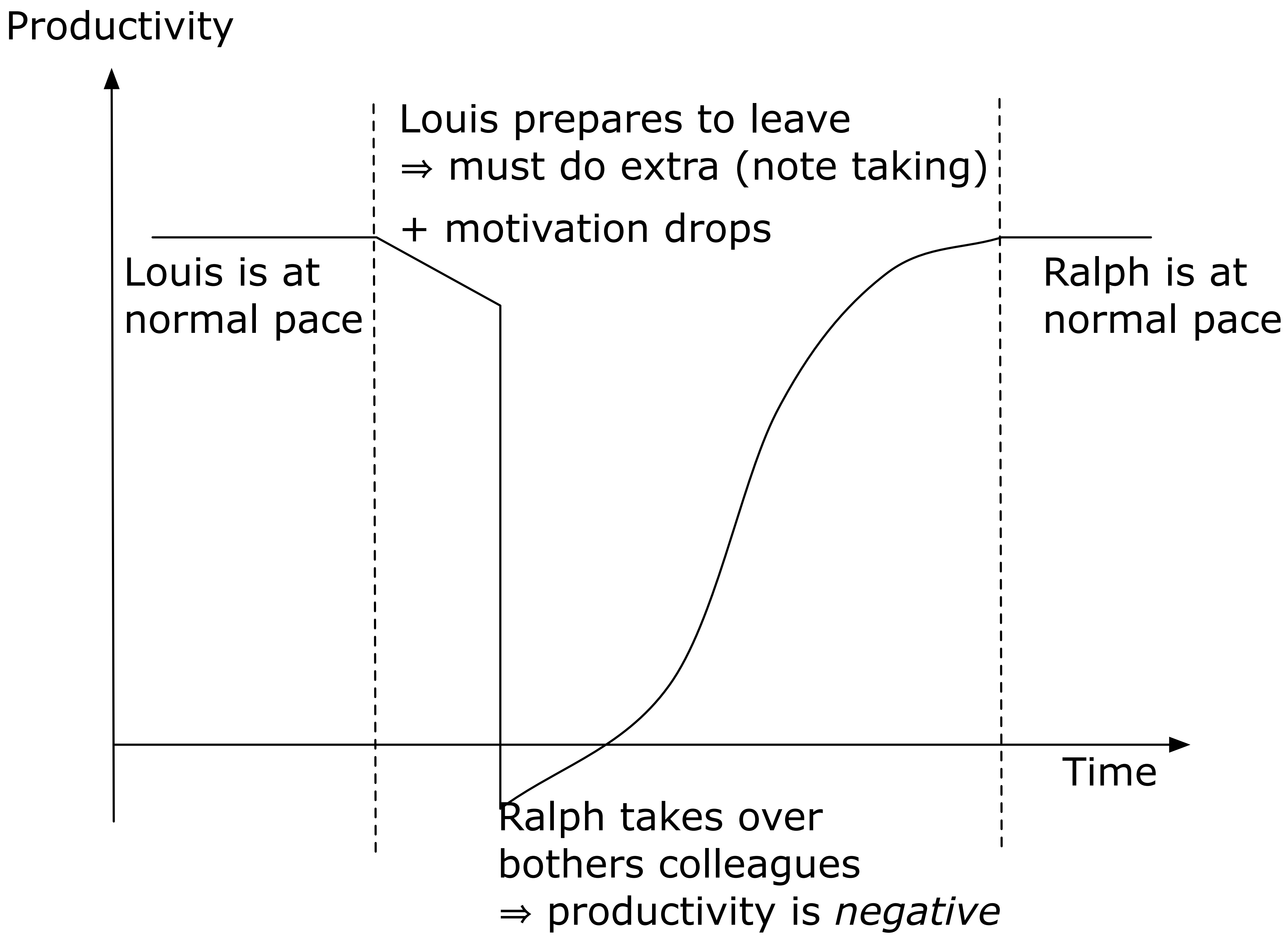
Delays

- Myth:
  + "If we get behind schedule, we can add more programmers and catch up."
- Reality:
  + Adding more people typically slows a project down.

- Scheduling Issues
  + Estimating the difficulty of problems and the cost of developing a solution is hard
  + The unexpected always happens. Always allow contingency in planning
  + Productivity is not proportional to the number of people working on a task
    - Productivity does not depend on raw man-power but on intellectual power
    - Adding people to a late project makes it later due to communication overhead.
  + Cutting back in testing and reviewing is a recipe for disaster
  + Working overnight? Only short term benefits …

Cost of Replacing a Person

(See [Dema98], chapter 13. The Human Capital)



Dealing with Delays

- Spot potential delays as soon as possible
  + ... then you have more time to recover

- How to spot?
  + Earned value analysis
    * planned time is the project budget
    * time of a completed task is credited to the project budget

- How to recover?
  + A combination of following 3 actions
    - Adding senior staff for well-specified tasks
      * outside critical path to avoid communication overhead
    - Prioritize requirements and deliver incrementally
      * deliver most important functionality on time
      * testing remains a priority (even if customer disagrees)
    - Extend the deadline

Calculating Earned Value (= Tasks Completed)

- The 0/100 Technique
  + earned value := 0% when task not completed
  + earned value := 100% when task completed
    * tasks should be rather small
    * gives a pessimistic impression

- The 50/50 Technique
  + earned value := 50% when task started
  + earned value := 100% when task completed
    * tasks should be rather large
    * may give an optimistic impression
    * variant with 20/80 gives a more realistic impression

- The Milestone Technique
  + earned value := number of milestones completed / total number of milestones
    * tasks are large but contain lots of intermediate milestones
    * Good for summary views on large schedules

Calculating Earned Value (= Time sheets)

Organizations usually require staff to maintain time sheets
= bookkeeping of time spent by an individual for a particular task in a project

Time Sheet
Name: Laura Palmer_____ Week ending: March, 3rd 2000_
Rechargeable hours

Project	Task	Activity	Description	Hours	Delay?
C34	5	5.3	Chapter 3	25	-
C34	5	5.4	Chapter 4	5	+
C34	6	6.0	Reviewing	4	-

Non-rechargeable hours

Hour	Description	Authorized
8	Use-case training	J.F. Kennedy

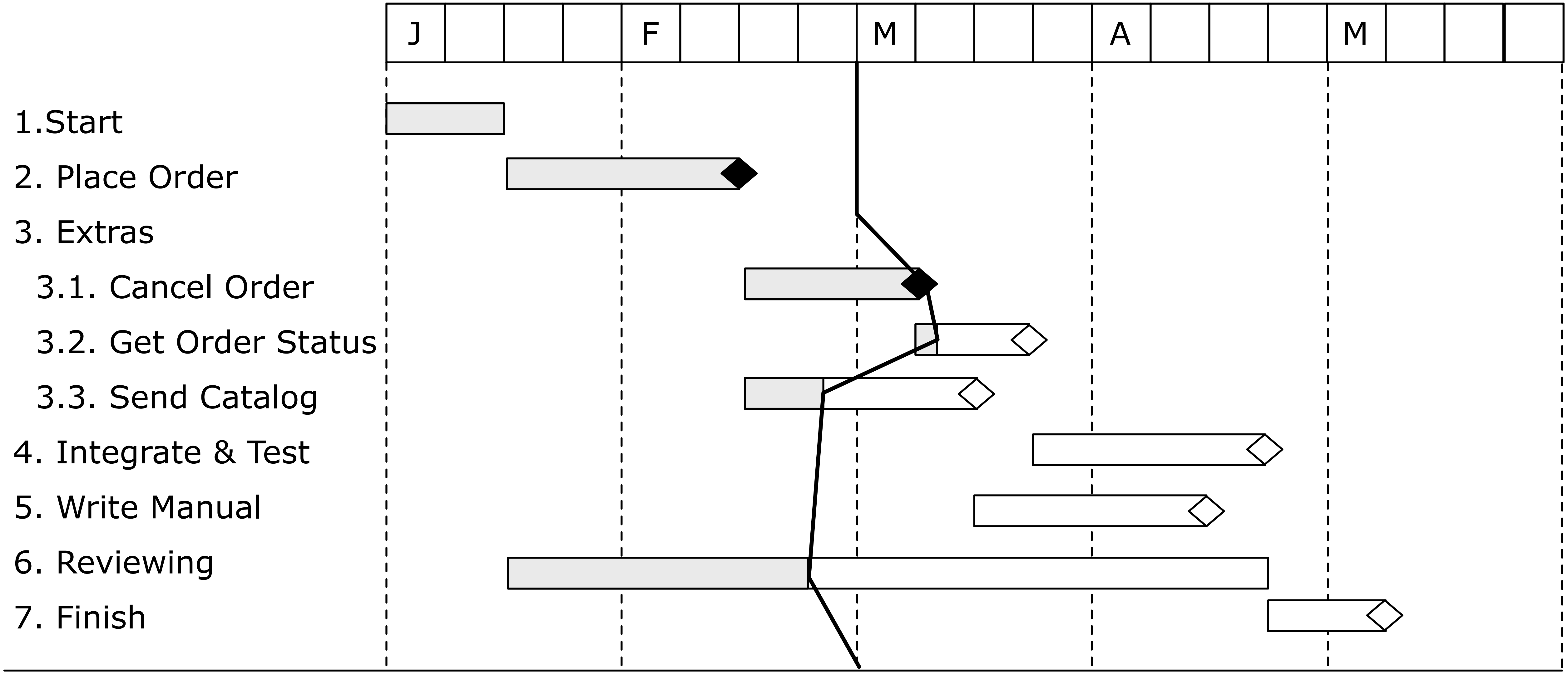
Opportunity to monitor team occupation

- Compare time spent (= earned value) vs. time planned
- Ask staff member if delay for this task is expected

Monitoring Delays – Slip Line (Gantt chart)

Visualise percentage of task completed via shading

- draw a slip line at current date, connecting endpoints of the shaded areas
- bending to the right = ahead of schedule, to the left = behind schedule



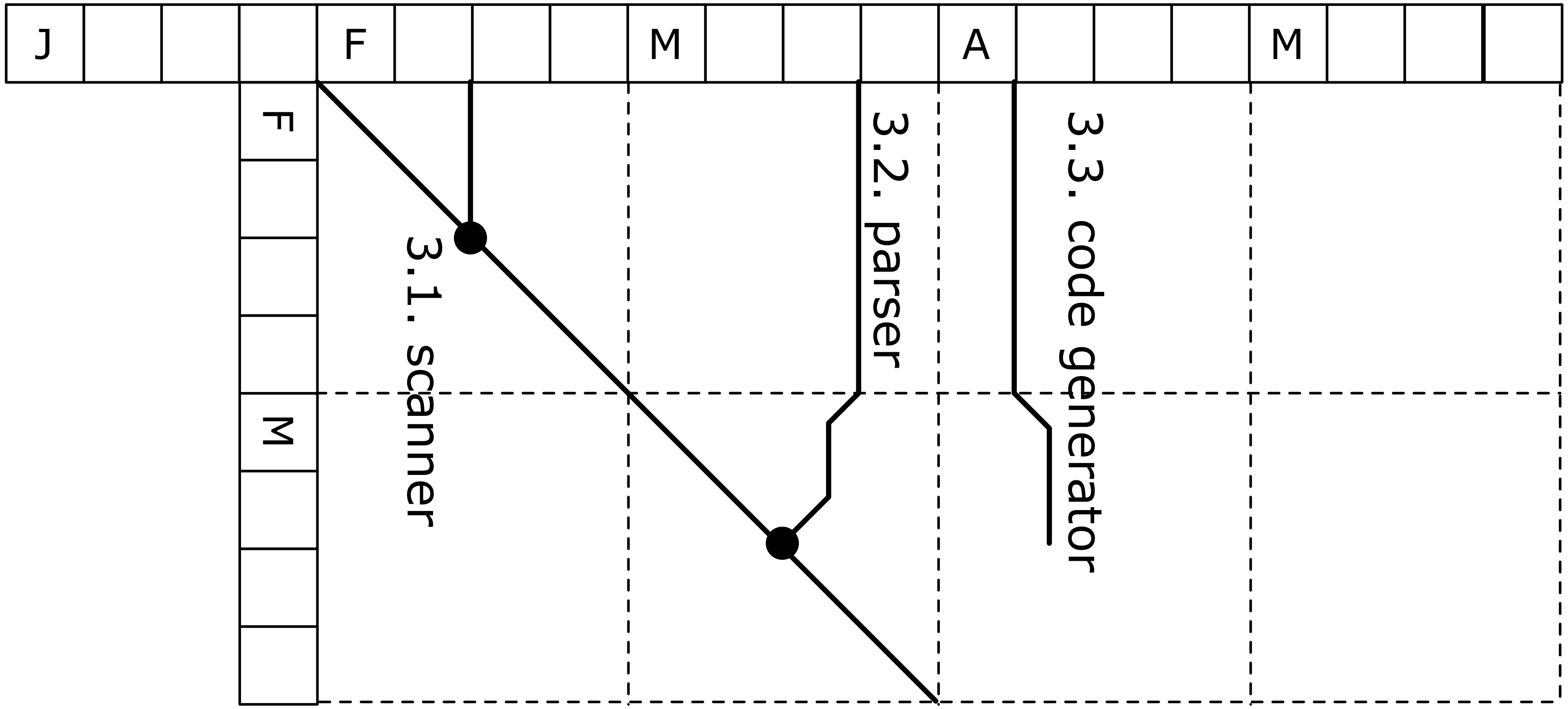
Interpretation

- Today is 1rst of March
- Task 3.1 is finished ahead of schedule and task 3.2 is started ahead of schedule
- Tasks 3.3 and 6 seem to be behind schedule (i.e., less completed than planned)

Monitoring Delays – Timeline Chart

Visualise slippage evolution

- downward lines represent planned completion time as they vary in current time
- bullets at the end of a line represent completed tasks



Interpretation (end of October)

- Task 3.1 is completed as planned.
- Task 3.2 is rescheduled 1/2 wk earlier end of February and finished 1 wk ahead of time.
- Tasks 3.3 rescheduled with one week delay at the and of February

Slip Line vs. Timeline

- Slip Line

  + Monitors current slip status of project tasks

  - many tasks
  - only for 1 point in time

    > include a few slip lines from the past to illustrate evolution

- Timeline

  + Monitors how the slip status of project tasks evolves

  - few tasks

    > crossing lines quickly clutter the figure

    > colors can be used to show more tasks

  - complete time scale

An afterthought …

All projects that finish late have this one thing in common: they started late.

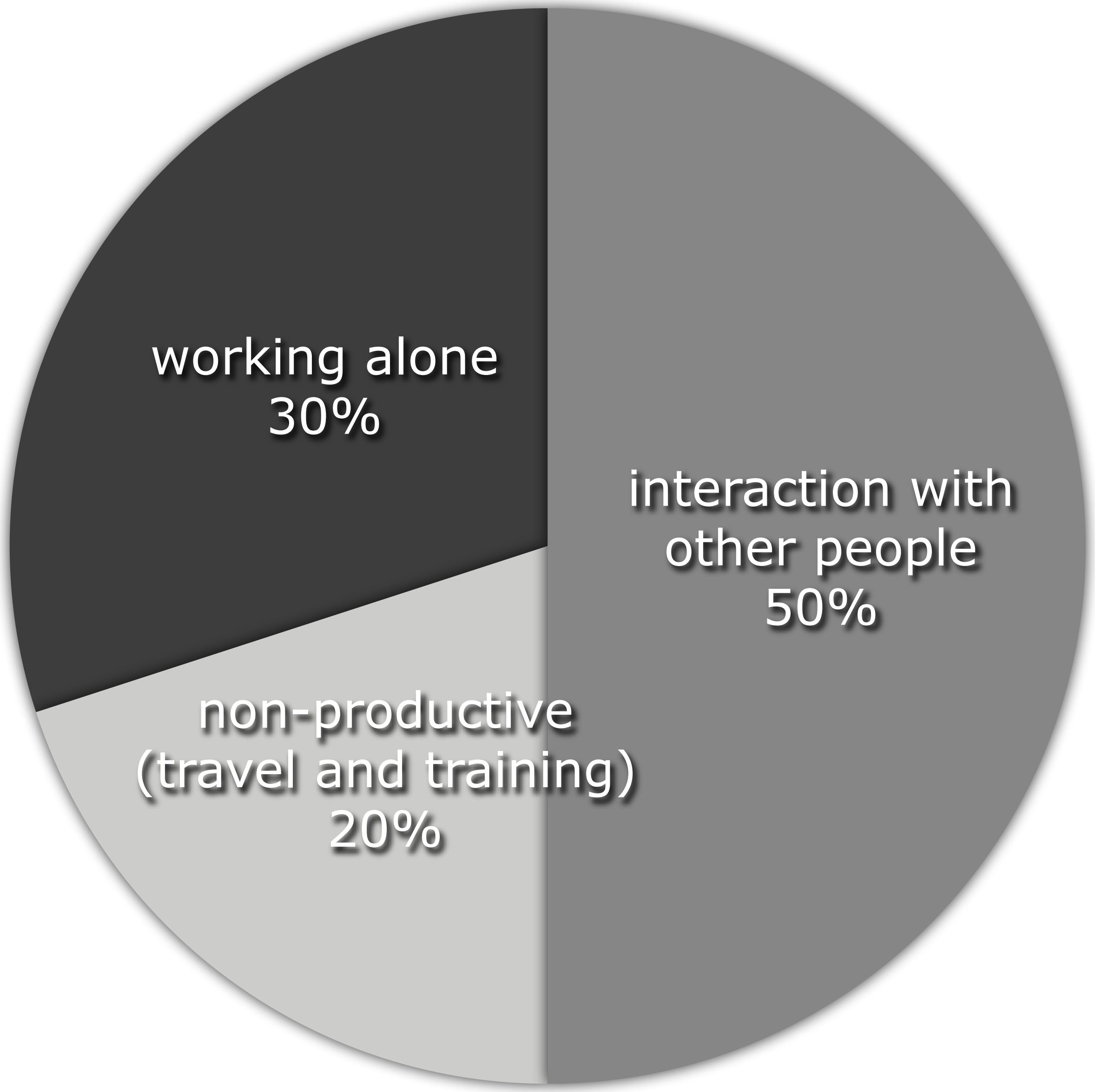
- [Dema11] Tom De Marco"All Late Projects Are the Same," IEEE Software, pp. 102-103, November/December, 2011

- 1. Nobody had the guts to kick off the project until the competition proved it doable and desirable; by then, the project was in catch-up mode and had to be finished lickety-split.
  ⇒ Business failure: blame marketing
- 2. If the project were started long enough before its due date to finish on time, all involved would have had to face up to the fact from the beginning that it was going to cost a lot more than anyone was willing to pay.
  + On the surface: poor risk analysis and cost estimation
  + What if gains would be orders of magnitude larger than the cost?
  + Who decides to start an expensive project with marginal gains?
  ⇒ Management failure: blame decision makers
- 3.No one knew that the project needed to be done until the window of opportunity was already closing.
  ⇒ Business failure + Management failure

Individuals work in Teams

Distribution of a software engineer’s time, as logged within IBM

- [McCu78] G M McCue, "IBM's Santa Teresa Laboratory — Architectural Design for Program Development,” IBM Systems Journal, 17, 1, pp. 4-25, 1978]



IMPLICATIONS?

- You cannot afford too many solo-players in a team
- Complementary personalities are as important as technical skills
- More women are necessary

Belbin Team Roles

“Do you want a collection of brilliant minds or a brilliant collection of minds?”
[Dr. Raymond Meredith Belbin (1926)]

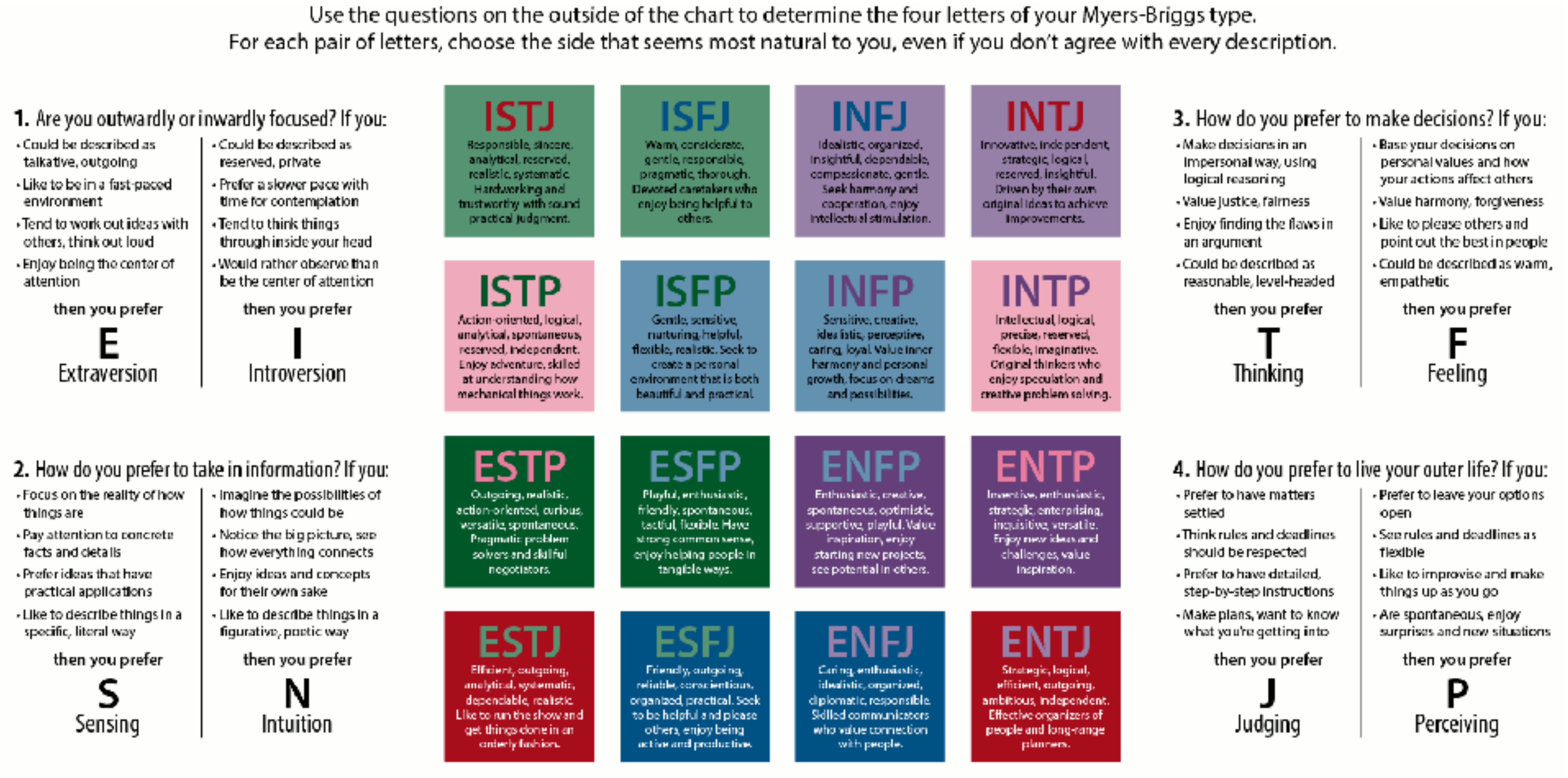
Action Oriented Roles	Shaper	Challenges the team to improve
	Implementer	Puts ideas into action
	Completer Finisher	Ensures thorough, timely completion
People Oriented Roles	Coordinator	Acts as a chairperson
	Team Worker	Encourages cooperation
	Resource Investigator	Explores outside opportunities
Thought Oriented Roles	Plant	Presents new ideas and approaches
	Monitor-Evaluator	Analyzes the options
	Specialist	Provides specialized skills

An *effective* team has members that cover nine classic team roles.

Overlap is possible!

Myers Briggs Type Inventory (MBTI)

** New **



Use the questions on the outside of the chart to determine the four letters of your Myers-Briggs type.
For each pair of letters, choose the side that seems most natural to you, even if you don't agree with every description.

1. Are you outwardly or inwardly focused? If you:

• Could be described as talkative, outgoing	• Could be described as reserved, private
• Like to be in a fast-paced environment	• Prefer a slower pace with time for contemplation
• Tend to work out ideas with others, think out loud	• Tend to think things through inside your head
• Enjoy being the center of attention	• Would rather observe than be the center of attention
then you prefer **E** Extraversion	then you prefer **I** Introversion

2. How do you prefer to take in information? If you:

• Focus on the reality of how things are	• Imagine the possibilities of how things could be
• Pay attention to concrete facts and details	• Notice the big picture, see how everything connects
• Prefer ideas that have practical applications	• Enjoy ideas and concepts for their own sake
• Like to describe things in a specific, literal way	• Like to describe things in a figurative, poetic way
then you prefer **S** Sensing	then you prefer **N** Intuition

ISTJ Responsible, sincere, analytical, reserved, realistic, systematic. Hardworking and trustworthy with sound practical judgment.	**ISFJ** Warm, considerate, gentle, responsible, pragmatic, thorough. Devoted caretakers who enjoy being helpful to others.	**INFJ** Idealistic, organized, insightful, dependable, compassionate, gentle. Seek harmony and cooperation, enjoy intellectual stimulation.	**INTJ** Innovative, independent, strategic, logical, reserved, insightful. Driven by their own original ideas to achieve improvements.
ISTP Action-oriented, logical, analytical, spontaneous, reserved, independent. Enjoy adventure, skilled at understanding how mechanical things work.	**ISFP** Gentle, sensitive, nurturing, helpful, flexible, realistic. Seek to create a personal environment that is both beautiful and practical.	**INFP** Sensitive, creative, idealistic, perceptive, caring, loyal. Value inner harmony and personal growth, focus on dreams and possibilities.	**INTP** Intellectual, logical, precise, reserved, flexible, imaginative. Original thinkers who enjoy speculation and creative problem solving.
ESTP Outgoing, realistic, action-oriented, curious, versatile, spontaneous. Pragmatic problem solvers and skillful negotiators.	**ESFP** Playful, enthusiastic, friendly, spontaneous, tactful, flexible. Have strong common sense, enjoy helping people in tangible ways.	**ENFP** Enthusiastic, creative, spontaneous, optimistic, supportive, playful. Value inspiration, enjoy starting new projects, see potential in others.	**ENTP** Inventive, enthusiastic, strategic, enterprising, inquisitive, versatile. Enjoy new ideas and challenges, value inspiration.
ESTJ Efficient, outgoing, analytical, systematic, dependable, realistic. Like to run the show and get things done in an orderly fashion.	**ESFJ** Friendly, outgoing, reliable, conscientious, organized, practical. Seek to be helpful and please others, enjoy being active and productive.	**ENFJ** Caring, enthusiastic, idealistic, organized, diplomatic, responsible. Skilled communicators who value connection with people.	**ENTJ** Strategic, logical, efficient, outgoing, ambitious, independent. Effective organizers of people and long-range planners.

3. How do you prefer to make decisions? If you:

• Make decisions in an impersonal way, using logical reasoning	• Base your decisions on personal values and how your actions affect others
• Value justice, fairness	• Value harmony, forgiveness
• Enjoy finding the flaws in an argument	• Like to please others and point out the best in people
• Could be described as reasonable, level-headed	• Could be described as warm, empathetic
then you prefer **T** Thinking	then you prefer **F** Feeling

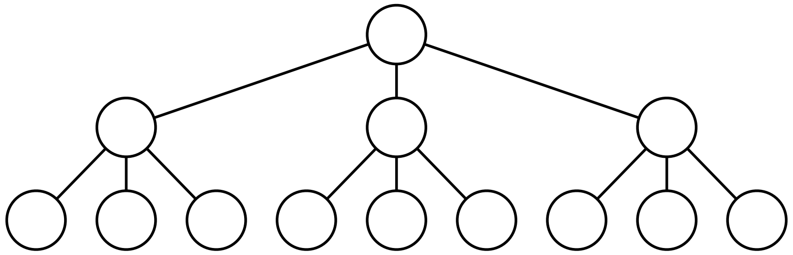
4. How do you prefer to live your outer life? If you:

• Prefer to have matters settled	• Prefer to leave your options open
• Think rules and deadlines should be respected	• See rules and deadlines as flexible
• Prefer to have detailed, step-by-step instructions	• Like to improvise and make things up as you go
• Make plans, want to know what you're getting into	• Are spontaneous, enjoy surprises and new situations
then you prefer **J** Judging	then you prefer **P** Perceiving

Typical Team Structures

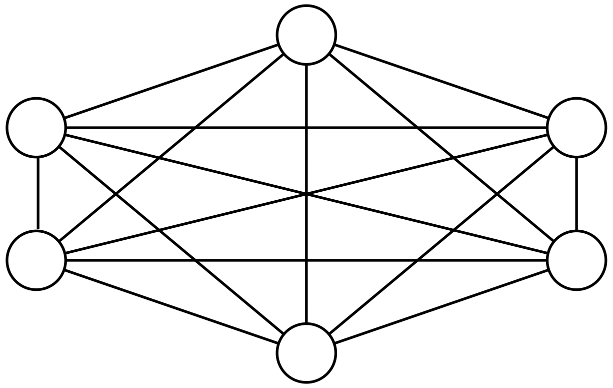
Hierarchical (Centralized)
e.g. Chief Programmer

- For well-understood problems
- Predictable, fast development
- Large groups



Consensus (Decentralized)
e.g. Egoless Programming Team

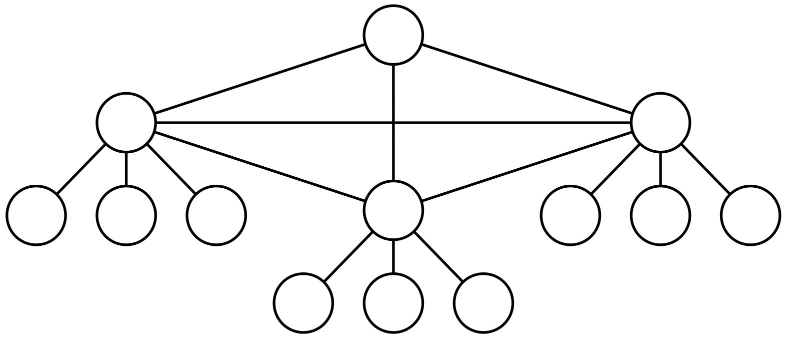
- For exploratory projects
- Fast knowledge transfer
- Small groups



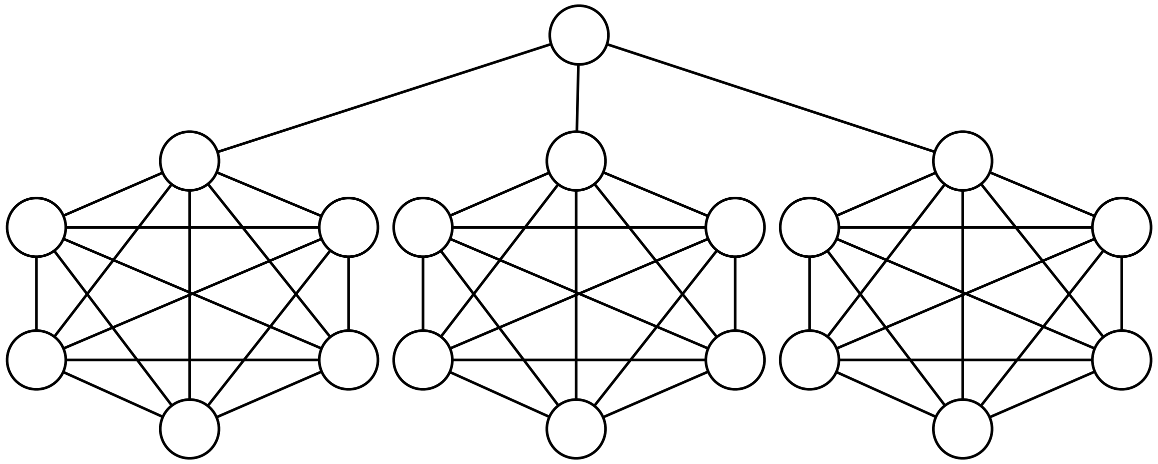
There is no “one size fits all” team structure!

Organize so that no one person has to talk to more then 8 (eight) persons in total!

Decentralized upper management
+ Centralized teams



Centralized upper management
+ Decentralized teams



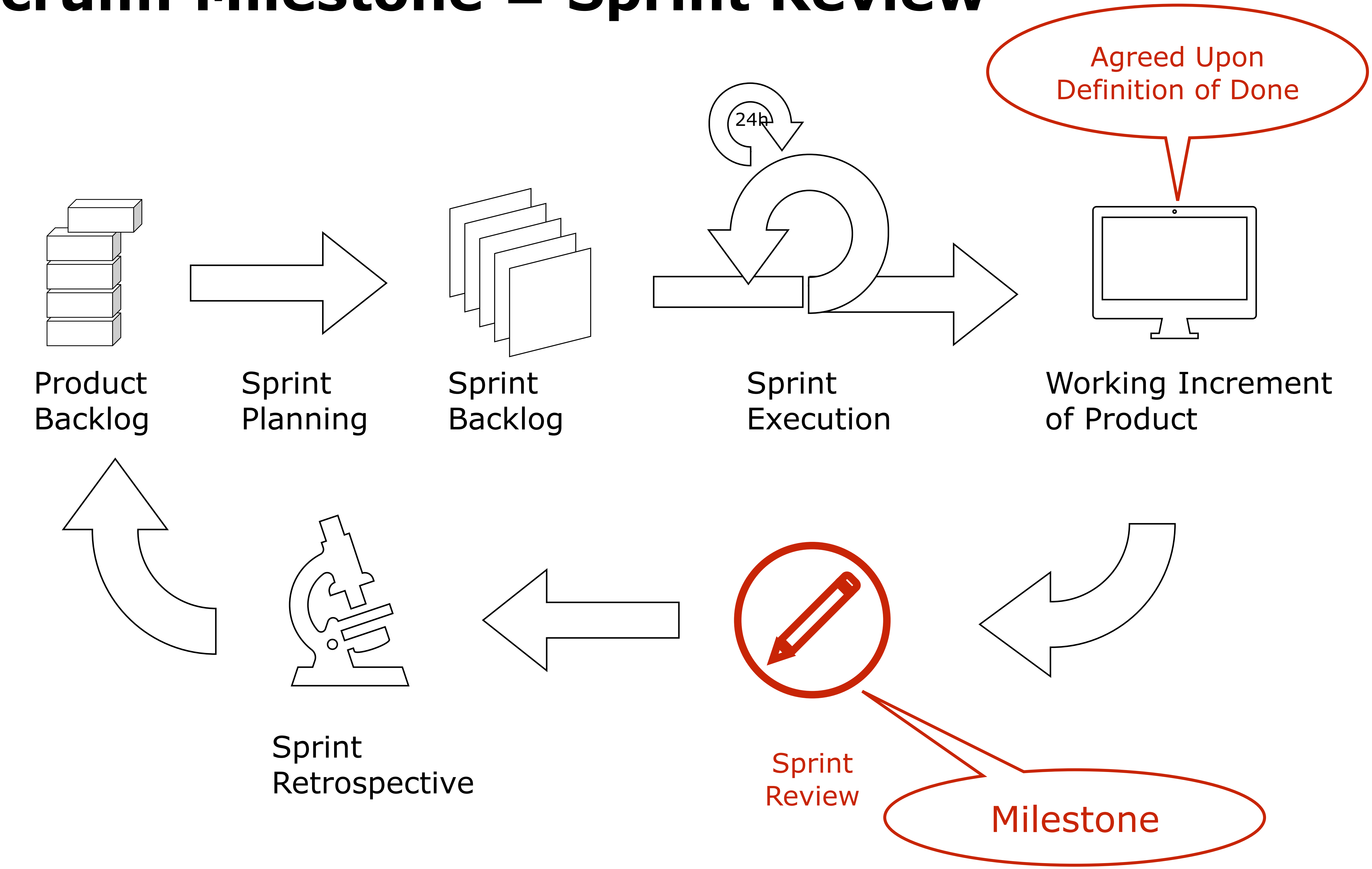
Directing Teams

Directing a team = the whole becomes more then the sum of its parts

- Managers serve their team
  - + Managers ensure that team has the necessary information and resources
    - > incl. pizza!
  - + Responsibility demands authority
    - - Managers must delegate
      - > Trust your own people and they will trust you.
  - + Managers manage
    - - Managers cannot perform tasks on the critical path
      - > Especially difficult for technical managers

  - + Developers control deadlines
    - - A manager cannot meet a deadline to which the developers have not agreed

Scrum: Milestone = Sprint Review



Agreed Upon Definition of Done

Product Backlog

Sprint Planning

Sprint Backlog

Sprint Execution

Working Increment of Product

Sprint Retrospective

Sprint Review

Milestone

Definition of Done

definition of done = a checklist of the types of work that the team is expected to successfully complete before it can declare its work to be potentially shippable.

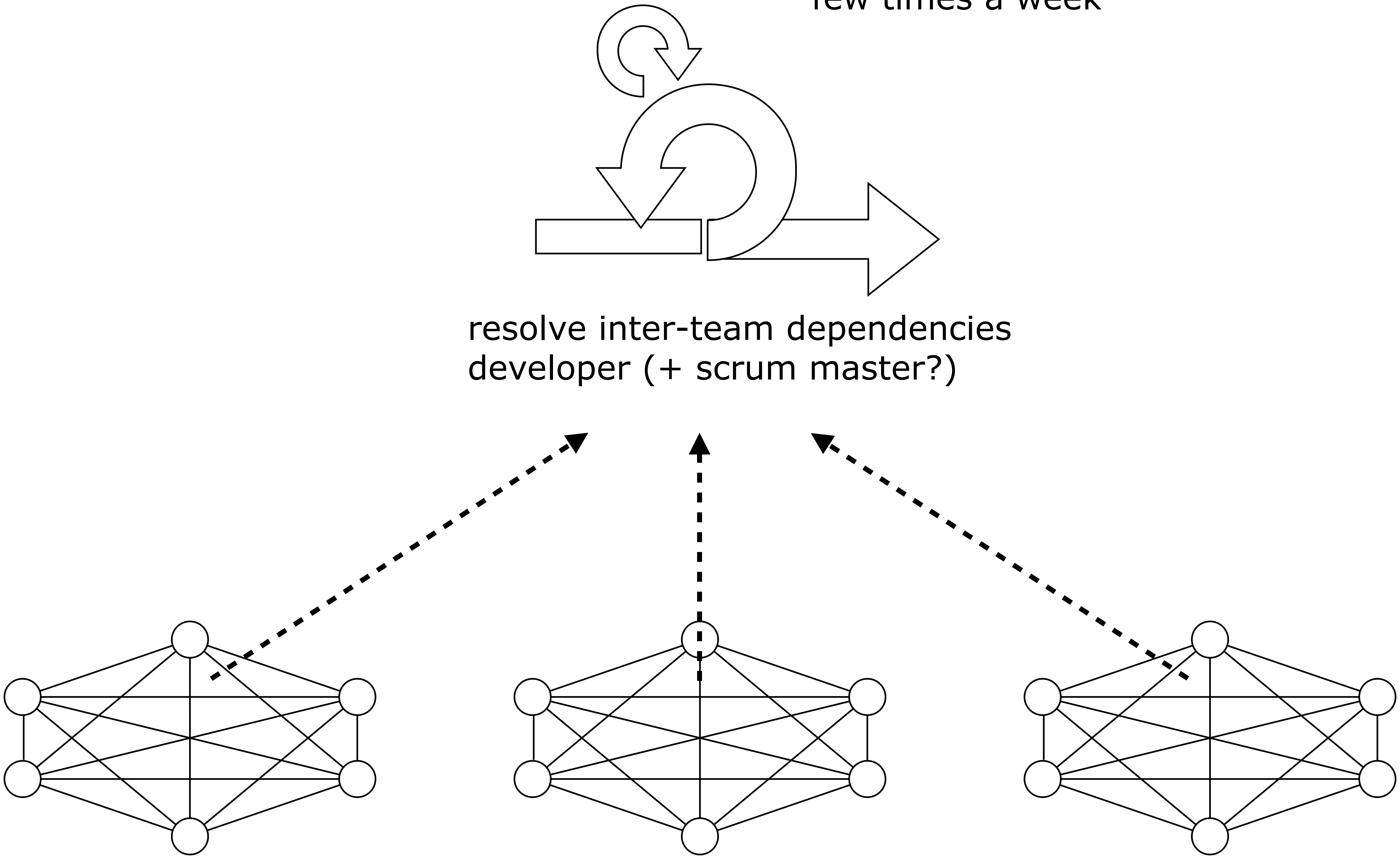
Different levels of “doneness”:

- Task level
- User story level
  - + (e.g. completed FIT acceptance tests with customer)
- Iteration level
  - + (e.g. all stories developed, all bugs closed)
- Release level
  - + (e.g. installation package created, stress testing completed)

- ✔ Design reviewed
- ✔ Code completed
  - ✔ Code refactored
  - ✔ Code in standard format
  - ✔ Code is commented
  - ✔ Code checked in
  - ✔ Code inspected
- ✔ End-user documentation
- ✔ Tested
  - ✔ Unit tested
  - ✔ Integration tested
  - ✔ Regression tested
  - ✔ Platform tested
  - ✔ Language tested
- ✔ Zero known defects
- ✔ Acceptance tested
- ✔ Live on production servers

Scaling Scrum: Scrum of Scrum

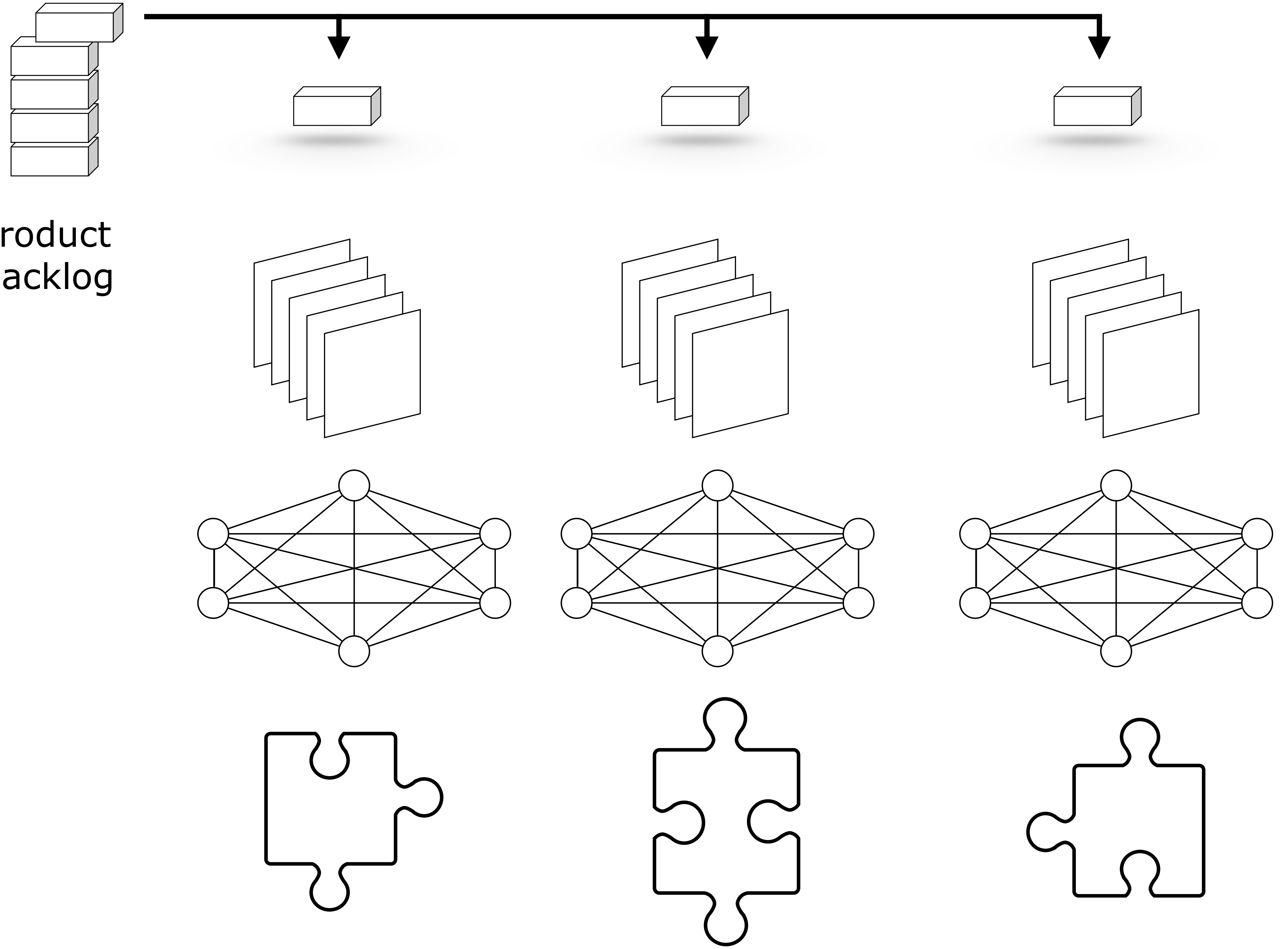
Synchronisation of work via “scrum of scrums”



few times a week

resolve inter-team dependencies
developer (+ scrum master?)

Scaling Scrum: Component Team



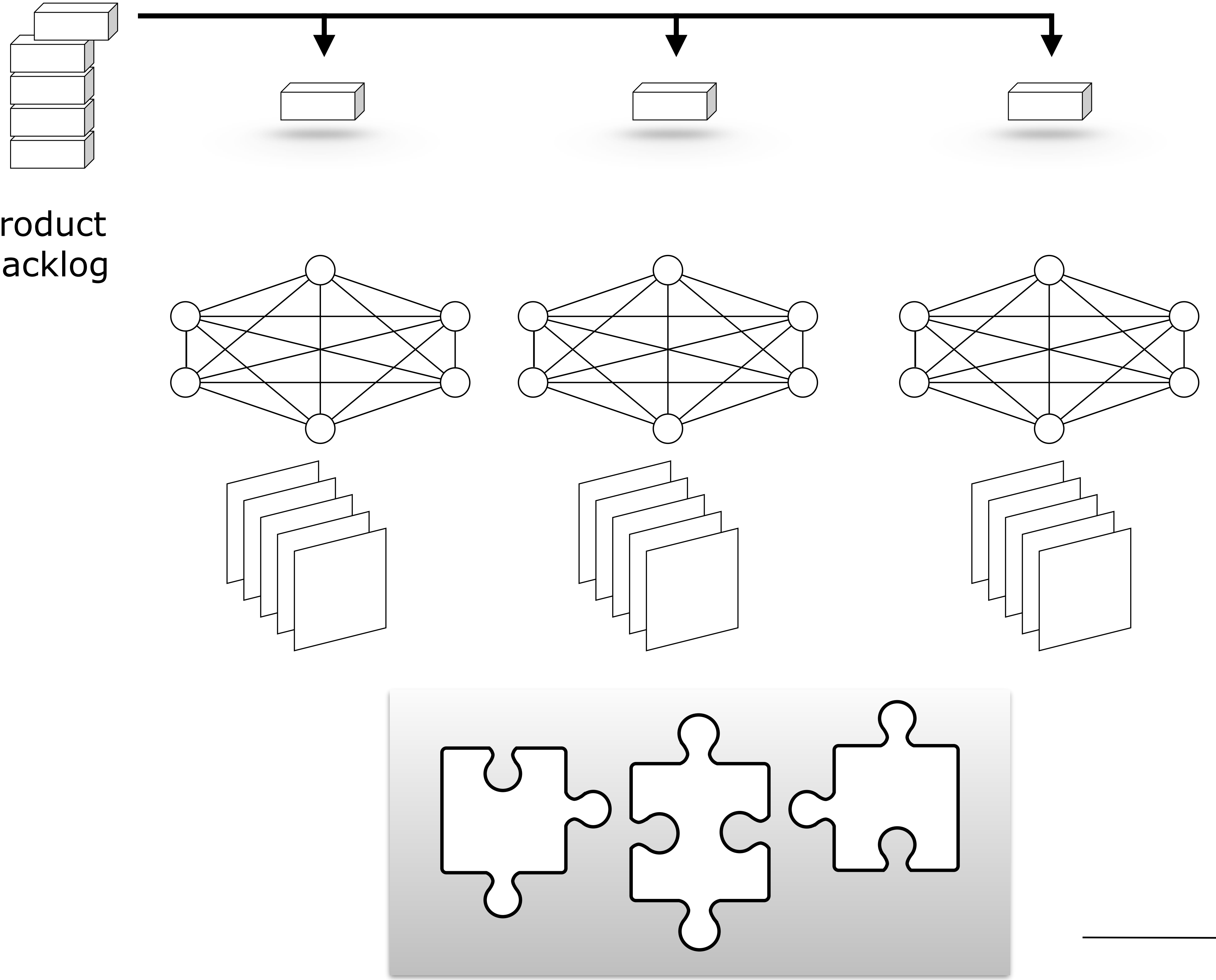
Product Backlog

Sprint Backlog

Multiple Scrum Teams …

… responsible for a single component

Scaling Scrum: Feature Team



Product Backlog

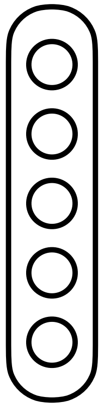
Multiple Scrum Teams …

… responsible for a single feature

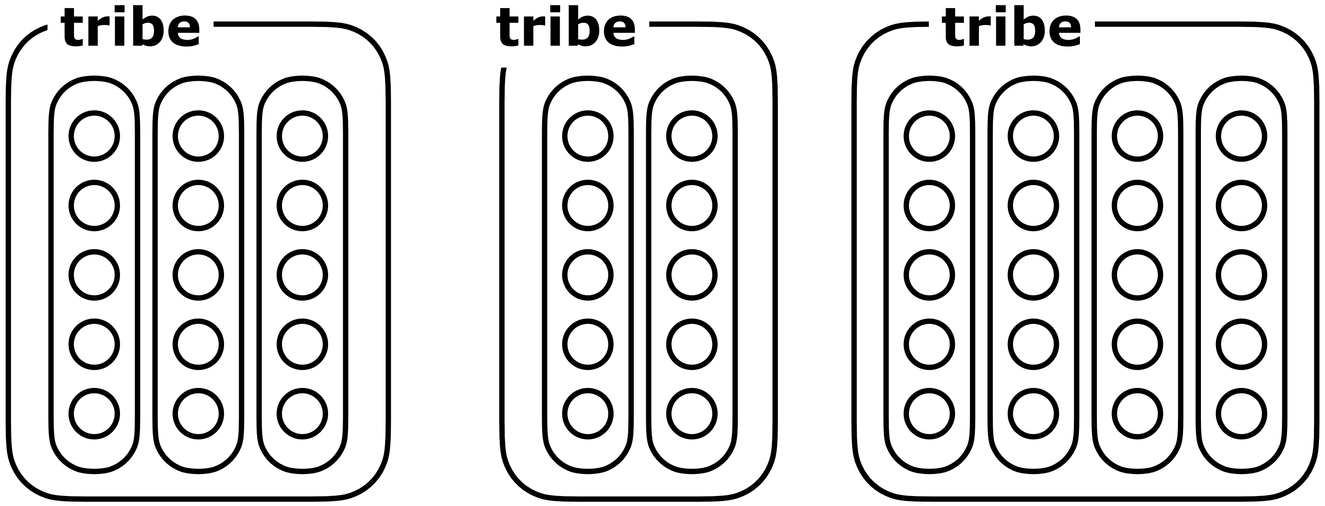
Joint code ownership

Spotify Scrum Model

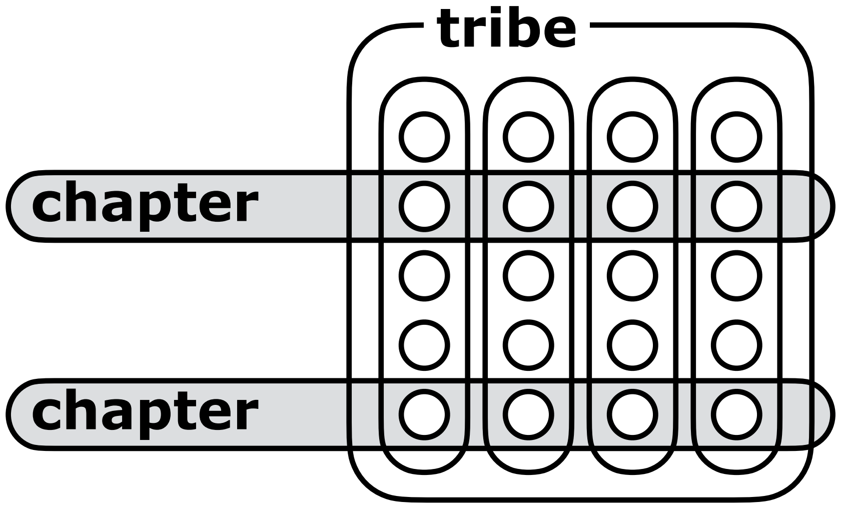
Squad =
Scrum Team



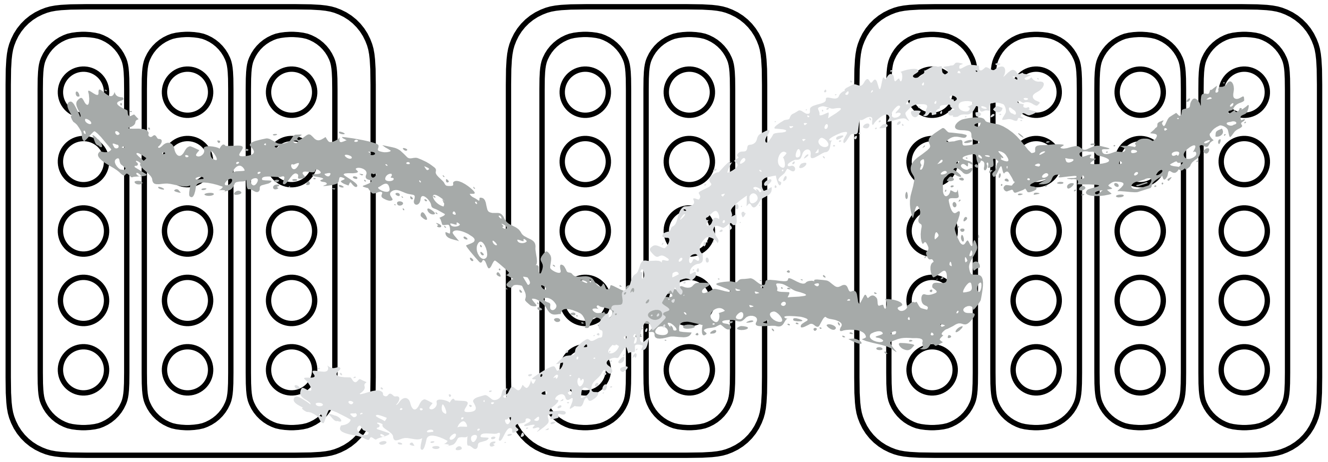
Tribe =
Loosely coupled Scrum Teams working
on related features/components



Chapter =
Team members with similar
expertise within a tribe.



Guild =
Team members with similar
interests across tribes.



Conclusion: Correctness & Traceability

- Correctness
  + The purpose of a plan is not correctness. 🙂
    - The purpose is to detect deviations as soon as possible
      ... and take appropriate actions
      * Adding people to a late project makes it later

  + Are we building the system right? 🙂
    - Deliver what’s required
      * ... on time within budget

- Traceability 😐
  + Plan ⇔ Requirements & System?
    - Only when done well
      * small tasks
      * milestones verifiable by customer

Summary (i)

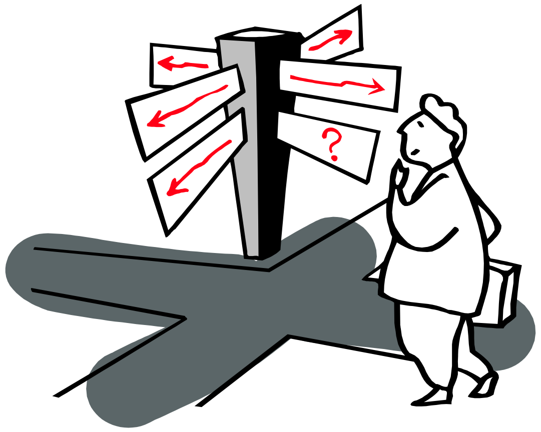
- You should know the answers to these questions
  + Name the five activities covered by project management.
  + What is a milestone? What can you use them for?
  + What is a critical path? Why is it important to know the critical path?
  + What can you do to recover from delays on the critical path?
  + How can you use Gantt-charts to optimize the allocation of resources to a project?
  + What is a “Known kown”, and “Unknown known” and an “Unknown Unknown”?
  + How do you use PERT to calculate the risk of delays to a project?
  + Why does replacing a person imply a negative productivity?
  + What’s the difference between the 0/100; the 50/50 and the milestone technique for calculating the earned value?
  + Why shouldn’t managers take on tasks in the critical path?
  + What is the “definition of done” in a Scrum project?
  + Give a definition for a Squad, Tribe, Chapter and Guild in the spotify scrum model.

- You should be able to complete the following tasks
  + draw a PERT Chart, incl. calculating the critical path and the risk of delays
  + draw a Gant chart, incl. allocating and optimizing of resources
  + draw a slip line and a timeline

Summary (ii)

- Can you answer the following questions?
  - + Name the various activities covered by project management. Which ones do you consider most important? Why?
  - + How can you ensure traceability between the plan and the requirements/system?
  - + Compare PERT-charts with Gantt charts for project planning and monitoring.
  - + How can you deal with “Unknown Unknowns” during project planning?
  - + Choose between managing a project that is expected to deliver soon but with a large risk for delays, or managing a project with the same result delivered late but with almost no risk for delays. Can you argue your choice?
  - + Describe how earned-value analysis can help you for project monitoring.
  - + Would you consider bending slip lines as a good sign or a bad sign? Why?
  - + You’re a project leader and one of your best team members announces that she is pregnant. You’re going to your boss, asking for a replacement and for an extension of the project deadline. How would you argue the latter request?
  - + You have to manage a project team of 5 persons for building a C++ compiler. Which team structure and member roles would you choose? Why?
  - + Can you give some advantages and disadvantages of scrum component teams and scrum feature teams.

CHAPTER 5 – Design by Contract



- Introduction
  + When, Why & What
  + Pre & Postconditions + Invariants
    - Example: Stack
- Implementation
  + Redundant Checks vs. Assertions
  + Exception Handling
  + Assertions are not…
- Theory
  + Correctness formula
  + Weak and Strong
  + Invariants
  + Subclassing and Subcontracting
    - The Liskov Substitution Principle
    - Behavioral subtyping
- Conclusion
  + How Detailed?
  + Tools: The Daikon Invariant Detector
  + Modern Application: Rest API
  + Example: Banking
  + Design by Contract vs. Testing

Literature

- [Ghez02], [Somm05], [Pres00]
  + Occurences of “contract”, “assertions”, “pre” and “postconditions”, via index
- [Meye97] Object-Oriented Software Construction, B. Meyer, Prentice Hall, Second Edn., 1997.
  + An excellent treatment on the do’s and don’ts of object-oriented development. Especially relevant are the chapters 6, 11-12

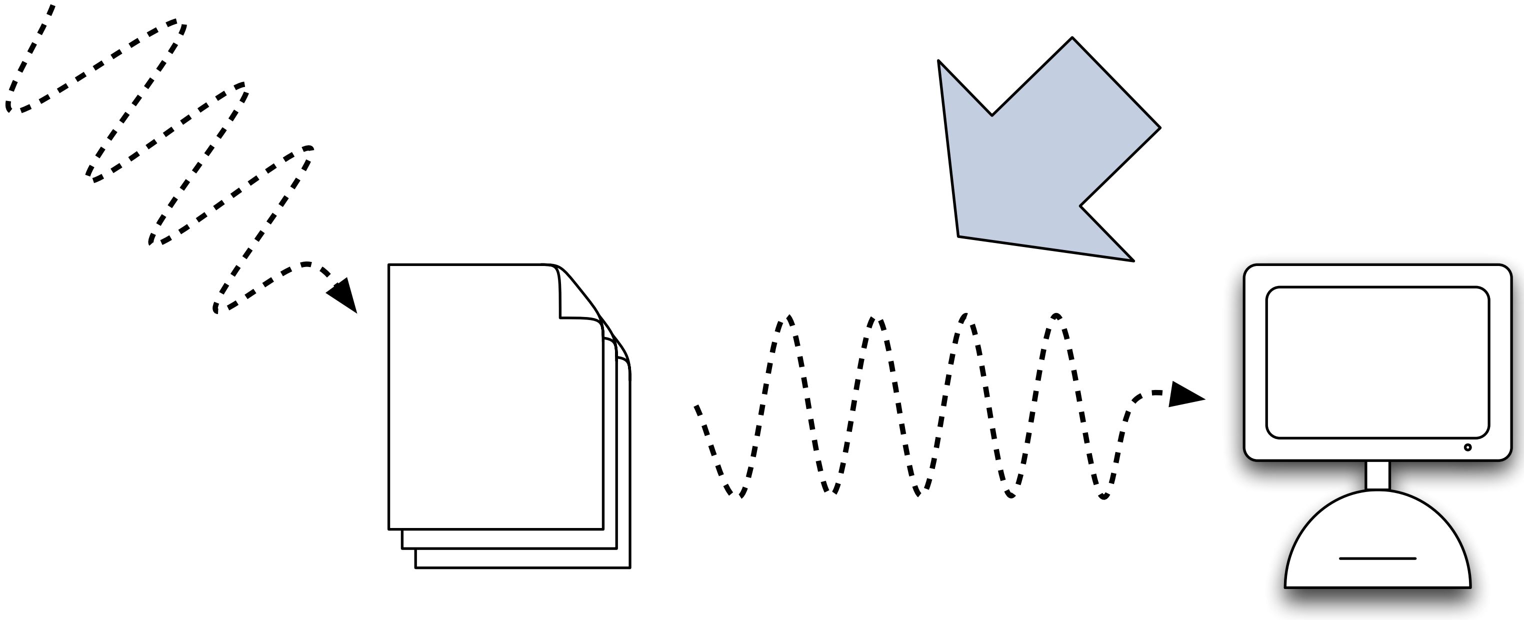
Copies of the following two articles are available from the course web-site.

- [Jeze97] “Put it in the contract: The lessons of Ariane”, Jean-Marc Jézéquel and Bertrand Meyer, IEEE Computer, January 1997, Vol30, no. 2, pages 129-130. A slightly different version of this article is available at http://www.irisa.fr/pampa/EPEE/Ariane5.html
  + A (heatedly debated) column arguing that Design by Contract would have prevented the crash of the first Ariane5 missile.
- [Garl97] “Critique of ‘Put it in the contract: The lessons of Ariane’”, Ken Garlington. See http://home.flash.net/~kennieg/ariane.html
  + An article providing counterarguments to the former. Yet by doing so gives an excellent comparison with Testing and Code Inspection.

Modern applications - Testing REST API

  + "Simplifying Microservice testing with Pacts", Ron Holshausen. https://dius.com.au/2014/05/19/simplifying-micro-service-testing-with-pacts/
    - Tutorial: https://docs.pact.io

When Design by Contract?



Mistakes are possible (likely!?)

- while transforming requirements into a system
- while system is changed during maintenance

Why Design By Contract?

- What's the difference with Testing?
  + Testing tries to *diagnose* (and cure) defects after the facts.
  + Design by Contract tries to *prevent* certain types of defects.
    > "Design by Contract" falls under Implementation/Design

- Design by Contract is particularly useful in an Object-Oriented context
  - (Or component-oriented, service-oriented, …)
  + preventing errors in interfaces between classes, components, services
    (incl. subclass and superclass via subcontracting)
  + preventing errors while reusing classes, components, services
    (incl. evolving systems, thus incremental and iterative development)
    * Example of the Ariane 5 crash

Use Design by Contract in combination with Testing!

What is Design By Contract?

"View the relationship between two classes as a formal agreement, expressing each party's rights and obligations." ([Meye97], Introduction to Chapter 11)

- Each party expects benefits (rights) and accepts obligations
- Usually, one party's benefits are the other party's obligations
- Contract is declarative: it is described so that both parties can understand *what* service will be guaranteed without saying *how*.

- Example: Airline reservation

	Obligations	Rights
Customer (Client Class)	- Be at Brussels airport at least 1 hour before scheduled departure time - Bring acceptable baggage - Pay ticket price	- Reach Chicago
Airline (Supplier Class)	- Bring customer to Chicago	- No need to carry passenger who is late, - has unacceptable baggage, - or has not paid ticket

Pre- and Post-conditions + Invariants

obligations are expressed via pre- and post-conditions

> "If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied."

pre-condition: {x >= 9}

post-condition: {x >= 13}

component: {x := x + 5}

... and invariants

> "For all calls you make to me, I will make sure the invariant remains satisfied."

invariant: {x >= y}

pre-condition: {x > 0, y > 0}

component: {x := x + y}

pre-condition: {x > 0, y < 0}

component: {x := x - y}

Quiz: Whose fault is it when a pre-condition is NOT satisfied?

Example: Stack

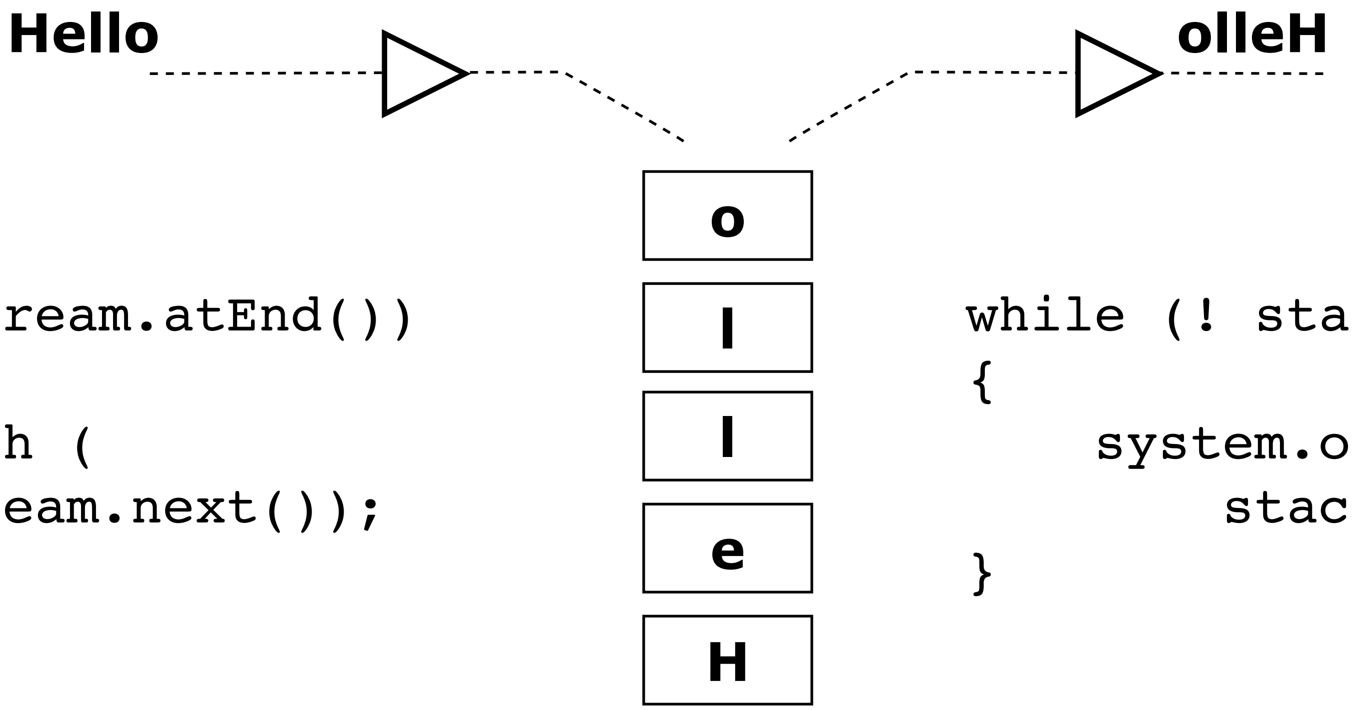
Given

A stream of characters, length unknown

Requested

Produce a stream containing the same characters but in reverse order

Specify the necessary intermediate abstract data structure



```
while (! inStream.atEnd())
{
    stack.push (
        inStream.next());
}
```

```
while (! stack.isEmpty())
{
    system.out.print (
        stack.pop());
}
```

Example: Stack Specification

```
class stack
    invariant: (isEmpty (this)) or
        (! isEmpty (this))
```

Implementors of stack promise that invariant will be true after all methods return (incl. constructors)

```
    public char pop ()
        require: ! isEmpty (this)
            ensure: true
```

Clients of stack promise precondition will be true before calling pop()

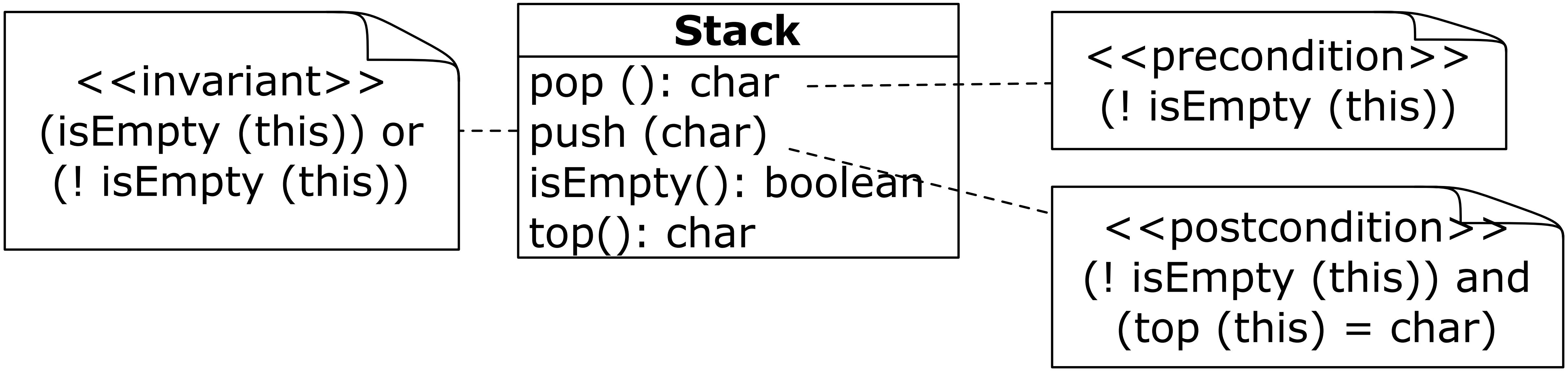
```
    public void push (char)
        require: true
        ensure: (! isEmpty (this))
            and (top (this) = char)
```

Implementors of stack promise postcondition will be true after push() returns

```
    public void top (char) : char
        require: ...
        ensure: ...
    public void isEmpty () : boolean
        require: ...
        ensure: ...
```

Left as an exercise

Design by Contract in UML



<<invariant>>
(isEmpty (this)) or
(! isEmpty (this))

Stack

pop (): char
push (char)
isEmpty(): boolean
top(): char

<<precondition>>
(! isEmpty (this))

<<postcondition>>
(! isEmpty (this)) and
(top (this) = char)

So what: isn’t this pure documentation?
Who will

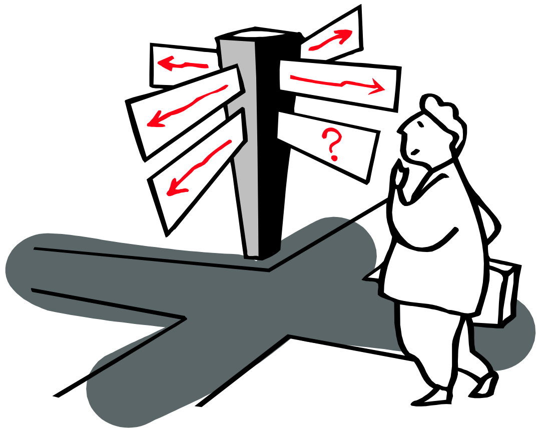
(a) Register these contracts for later reference (the book of laws)?
(b) Verify that the parties satisfy their contracts (the police)?

Answer

(a) The source code
(b) The running system

Quiz: What happens when a pre-condition is NOT satisfied?

CHAPTER 6 – Design by Contract



- Introduction
  + When, Why & What
  + Pre & Postconditions + Invariants
    - Example: Stack
- Implementation
  + Redundant Checks vs. Assertions
  + Exception Handling
  + Assertions are not…
- Theory
  + Correctness formula
  + Weak and Strong
  + Invariants
  + Subclassing and Subcontracting
    - The Liskov Substitution Principle
    - Behavioral subtyping
- Conclusion
  + How Detailed?
  + Tools: The Daikon Invariant Detector
  + Modern Application: Rest API
  + Example: Banking
  + Design by Contract vs. Testing

Redundant Checks

Redundant checks: naive way for including contracts in the source code

```
public char pop () {
    if (isEmpty (this)) {
        ... //Error-handling
} else {
        ...}
```

This is redundant code: it is the responsibility of the client to ensure the pre-condition!

Redundant Checks Considered Harmful

- Extra complexity

  Due to extra (possibly duplicated) code

  ... which must be verified as well.
- Performance penalty

  Redundant checks cost extra execution time.
- Wrong context

  How severe is the fault? How to remedy the situation? A service provider cannot asses the situation, only the consumer can.

Assertions

+ assertion = any boolean expression we expect to be true at some point.

- Assertions …
  + Help in writing correct software
    * formalizing invariants, and pre- and post-conditions
  + Aid in maintenance of documentation
    * specifying contracts IN THE SOURCE CODE
    * tools to extract interfaces and contracts from source code
  + Serve as test coverage criterion
    * Generate test cases that falsify assertions at run-time
  + Should be configured at compile-time
    * to avoid performance penalties with trusted parties
    * What happens if the contract is not satisfied?

Quiz: What happens when a pre-condition is NOT satisfied?
 > = What should an object do if an assertion does not hold?
   * ***Throw an exception.***

Assertions in Source Code

```
public char pop() throws AssertionException {
    char tempResult;
    my_assert(!this.isEmpty());
    tempresult = _store[_size--];
    my_assert(invariant());
    my_assert(true); //empty postcondition
     return tempResult;
}


private boolean invariant() {
    return (_size >= 0) && (_size <= _capacity);}


private void my_assert(boolean assertion)
    throws AssertionException {
    if (!assertion) {
        throw new AssertionException
            ("Assertion failed");}
}
```

Should be turned on/off via compilation option

Exception Handling

```
public class AssertionException extends Exception {
    AssertionException() { super(); }
    AssertionException(String s) { super(s); }
}
```

```
static public boolean testEmptyStack() {
    ArrayStack stack = new();
    try {
        // pop() will raise an exception
        stack.pop();


    } catch(AssertionException err) {
        // we should get here!
        return true;
    };

    return false;
}
```

If an ‘AssertionException’ is raised within the try block …

… we will fall into the ‘catch’ block

Assertions are not…

- Assertions look strikingly similar yet are different from …

  + Redundant Checks
    - Assertions become part of a class interface
    - Compilation option to turn on/off

  + Checking End User Input
    - Assertions check software-to-software communication, not software-to-human

  + Control Structure
    - Raising an exception is a control structure mechanism
    - … violating an assertion is a fault
      > precondition violation: responsibility of the client of the class
      > postcondition violation: responsibility of the supplier of the class
        * Only turn off assertions with trusted parties
        * Tests must verify whether exceptions are thrown

Programming Language Support

- Eiffel
  + Eiffel is designed as such … but only used in limited cases

- C++
  + assert() in C++ assert.h does not throw an exception
  + It’s possible to mimic assertions (incl. compilation option) in C++
  + Documentation extraction is more difficult but feasible

- Java
  + ASSERT is standard since Java 1.4
    ... however limited “design by contract” only; acknowledged by Java designers
    - https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html
  + Documentation extraction using JavaDoc annotations

- … Other languages
  + Possible to mimic; compilation option requires language idioms
  + Documentation extraction is possible (style Javadoc)

Two Implementation Issues

- 1) The use of ‘previous’ or ‘old’ state
  + sometimes postconditions compare exit state with starting state

```
public char pop ()
  require: ! isEmpty (this)
  ensure: (top (old) = char)
    and (size (old) = size (this) + 1)
```

Use ‘old’ as a way to refer to the starting state of the receiver

  + Eiffel has a pseudo variable ‘old’
  + Mimicking assertions in other languages?
    - store ‘old’ state in temporary variables

- 2) Invoking operations within assertions
  + Assertions may invoke operations with pre- and postconditions
    - overhead + cycles lead to infinite loops
  + Eiffel switches off assertions when checking assertions
  + Mimicking assertions in other languages?
    - Cumbersome using language idioms and class variable
      … best to avoid cycles

Testing Issues

+ Pre- and post-conditions are part of the interface of a component.
    - Part of black-box testing, not white-box testing
        > Do *not* include assertions in basis-path testing
        > Borderline case: include assertions in condition testing

+ Example

```
public char pop() throws AssertionException {
    assert(!this.isEmpty());
    return _store[_size--];
}
```

+ basis-path testing: cyclomatic complexity = 1; 1 path can cover the control-flow
    - (test case 1 = non-empty stack / value on the top)
+ condition testing: 2 inputs cover all conditions
    - (test case 1 = non-empty stack / value on the top
    - (test case 2 = empty stack / assertion exception)

See Next Week
(Chapter 6. Testing)

Compiler Checks?

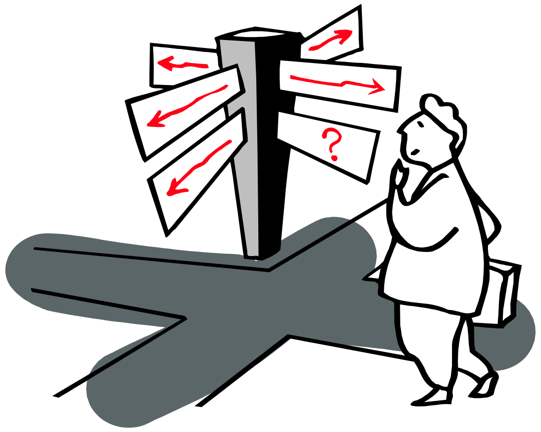
How much assertion monitoring is needed?

All
Especially during development
Too costly during production runs

None
Fully trusted system
Metaphor “sailing without life-jacket”

- Rule of thumb
  + *** At least monitor the pre-conditions.
    - Make sure that verifying pre-conditions is fast!
    - Do not rely on switching off monitoring to gain efficiency
    - Profile performance to see where you loose efficiency
      > First do it, then do it right, then do it fast!

CHAPTER 6 – Design by Contract



- Introduction
  - + When, Why & What
  - + Pre & Postconditions + Invariants
    - - Example: Stack
- Implementation
  - + Redundant Checks vs. Assertions
  - + Exception Handling
  - + Assertions are not…
- Theory
  - + Correctness formula
  - + Weak and Strong
  - + Invariants
  - + Subclassing and Subcontracting
    - - The Liskov Substitution Principle
    - - Behavioral subtyping
- Conclusion
  - + How Detailed?
  - + Tools: The Daikon Invariant Detector
  - + Modern Application: Rest API
  - + Example: Banking
  - + Design by Contract vs. Testing

Correctness Formula

a.k.a. *Hoare triples*

Let:

A be an *operation* (defined on a class C)

{P} and {Q} are *properties* (expressed via predicates, i.e functions returning a boolean)

Then:

{P} A {Q}

is a *Correctness Formula* meaning

"Any execution of A starting in a state where P holds, will terminate in a state where Q holds"

Example: ∀ x positive Integer

{x >= 9} x := x + 5 {x >=13}

See within 2 weeks (Chapter 7. Formal Specification)

Weak and Strong

- (Note: “weaker” and “stronger” follow from logic theory)

- Let {P1} and {P2} be conditions expressed via predicates
  - + {P1} is stronger then {P2} iff
    - {P1} <> {P2}
    - {P1} ⇒ {P2}
  - + example
    - {x >= 9} is stronger then {x >= 3}
  - + {false} is the strongest possible condition
    [(not {false}) or {X} is always true]
  - + {true} is the weakest possible condition
    [(not {X}) or {true} is always true]

- *Remember: {P1} ⇒ {P2}*

  is the same as (not {P1}) or {P2}

P1	TRUE	FALSE	TRUE	FALSE
P2	TRUE	TRUE	FALSE	FALSE
{P1} ⇒ {P2}	TRUE	TRUE	FALSE	TRUE
(not {P1}) or {P2}	TRUE	TRUE	FALSE	TRUE

Weak and Strong: Quiz

- {P} A {Q} is a specification for operation A
  + You, as a developer of A must guarantee that once {P} is satisfied and A is invoked it will terminate in a situation where {Q} holds
  + If you are a lazy developer, would you prefer
    - a weak or a strong precondition {P}?
    - a weak or a strong postcondition {Q}?

	weak	strong	don’t know
precondition {P}			
postcondition {Q}			

Weak or Strong (Preconditions)

- Given correctness formula: {P} A {Q}

- If you are a lazy developer, would you prefer a weak or a strong precondition {P}?
  - weak {P} ⇒ the starting situation is not constrained
  - strong {P} ⇒ little cases to handle inside the operation
    - The stronger the precondition, the easier it is to satisfy the postcondition
- Easiest Case
  - {false} A {...}
    {false} ⇒ {X} is true for any X

    [because (not {false}) or {X} is always true]
    - if {...} does not hold after executing A, you can blame somebody else because the precondition was not satisfied
    - ... independent of what happens inside A

- **Quiz:** If you are client of that class, would you prefer a weak or strong precondition?

Weak or Strong (Postconditions)

- Given correctness formula: {P} A {Q}

- If you are a lazy developer, would you prefer a weak or a strong postcondition {Q}?
  - + weak {Q} ⇒ the final situation is not constrained
  - + strong {Q} ⇒ you have to deliver a lot when implementing A
    - * The weaker the postcondition, the easier it is to satisfy that postcondition

- Easiest Case
  - + {...} A {true}

    {X} ⇒ {true} is true for any X

    [because (not {X}) or {true} is always true]
    - {true} will always hold after executing A
    - ... given that A *terminates* in a finite time

- **Quiz:** If you are client of that class, would you prefer a weak or strong postcondition?

Weak or Strong (Pre- vs. Post-conditions)

- Remember
  - + {false} A {...} is easier to satisfy then {...} A {true}
    - With the strong precondition you may go in an infinite loop
    - The weak postcondition must be satisfied in finite time

Invariants

- Invariants correspond to the general clauses in a legal contract, i.e. properties that always must be true for a given domain.

- {I} is an invariant for class C
  + After invoking a constructor of C, {I} is satisfied
    - Default constructors as well!
  + All public operations on C guarantee {I} when their preconditions are satisfied

- Thus, for each operation A defined on class C with invariant {I}
  + {P} A {Q} should be read as {I and P} A {I and Q}
    - strengthens the precondition ⇒ implementing A becomes easier
    - strengthens the postcondition ⇒ implementing A becomes more difficult

Contracts and Inheritance

- class C with invariant {I}
  + and operations $\{P_i\}$ m_i $\{Q_i\}$ where i: 1 .. n
- class C' extends C with invariant {I'}
  + and operations $\{P_i'\}$ m_i $\{Q_i'\}$ where i: 1 .. n
- [We ignore the case where C' extends the interface of C]

- Quiz: What's the relationship between the contract in C and the contract in C'
  + Invariant: Is {I'} stronger, weaker or equal to {I}
  + Precondition: Is {P'} stronger, weaker or equal to {P}
  + Postcondition: Is {Q'} stronger, weaker or equal to {Q}

- Answer according to the *Liskov Substitution Principle*
  + *** You may substitute an instance of a subclass for any of its superclasses.

Contracts and Inheritance

- `class C with invariant {I}`
  - + and operations `{Pi} mi {Qi}` where `i: 1 .. n`
- class `C'` extends `C` with invariant `{I'}`
  - + and operations `{Pi'} mi {Qi'}` where `i: 1 .. n`
- [We ignore the case where C' extends the interface of C]

- Quiz: What's the relationship between the contract in C and the contract in C'
  - + Invariant: Is {I'} stronger, weaker or equal to {I}
  - + Precondition: Is {P'} stronger, weaker or equal to {P}
  - + Postcondition: Is {Q'} stronger, weaker or equal to {Q}

VOTES	stronger	weaker	equal	don't know
{I'} vs. {I}				
{P'} vs. {P}				
{Q'} vs. {Q}				

Sidetrack: ACM Turing Award Barbara Liskov

Press release — NEW YORK, March 10, 2009
– ACM, the Association for Computing Machinery

The ACM has named Barbara Liskov of the Massachusetts Institute of Technology (MIT) the winner of the 2008 ACM A.M. Turing Award. The award cites Liskov for her foundational innovations to designing and building the pervasive computer system designs that power daily life. Her achievements in programming language design have made software more reliable and easier to maintain. They are now the basis of every important programming language since 1975, including Ada, C++, Java, and C#. The Turing Award, widely considered the "Nobel Prize in Computing," is named for the British mathematician Alan M. Turing. The award carries a $250,000 prize, with financial support provided by Intel Corporation and Google Inc.

[…]

In another exceptional contribution, Liskov designed the CLU programming language, an object-oriented language incorporating "clusters" to provide coherent, systematic handling of abstract data types, which are comprised of a set of data and the set of operations that can be performed on the data. She and her colleagues at MIT subsequently developed efficient CLU compiler implementations on several different machines, an important step in demonstrating the practicality of her ideas. Data abstraction is now a generally accepted fundamental method of software engineering that focuses on data rather than processes, often identified as "modular" or "object-oriented" programming.

Liskov substitution principle revisited

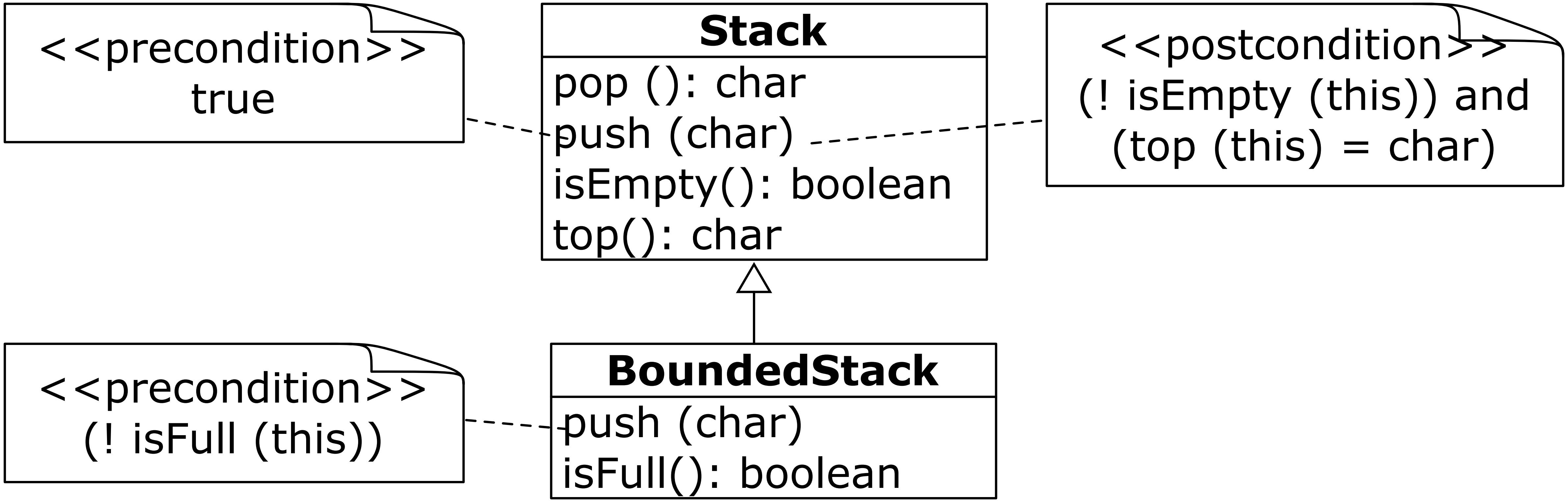
Subtype
Relationship
“is-a”

If it swims like a duck and quacks like a duck, then it’s a duck
(i.e.: the interfaces of the subtype and the supertype are equivalent)

If it swims like a duck and quacks like a duck,
but … needs batteries then it is NOT a duck.

(i.e.: mismatch between the interface of the subtype and the supertype)

Contracts and Inheritance: Example (1/2)



+ A client of Stack assumes a “true” pre-condition on push()
    - Any invocation on push() will deliver the post-condition
+ However, substituting a BoundedStack adds pre-condition
    - “! isFull(this)”
+ BoundedStack requires more from its clients
    - *You cannot substitute a BoundedStack for a Stack*

Contracts and Inheritance: Example (2/2)

As an illustration of the unsatisfied substitution principle, assume the following (test)code

testStack should work for any s: stack we pass !

```
testStack (s: Stack) {
    push(s, 99);
    if empty(s) {
        error(postC)};
    if pop(s) <> 99 {
        error(postC)};
}
```

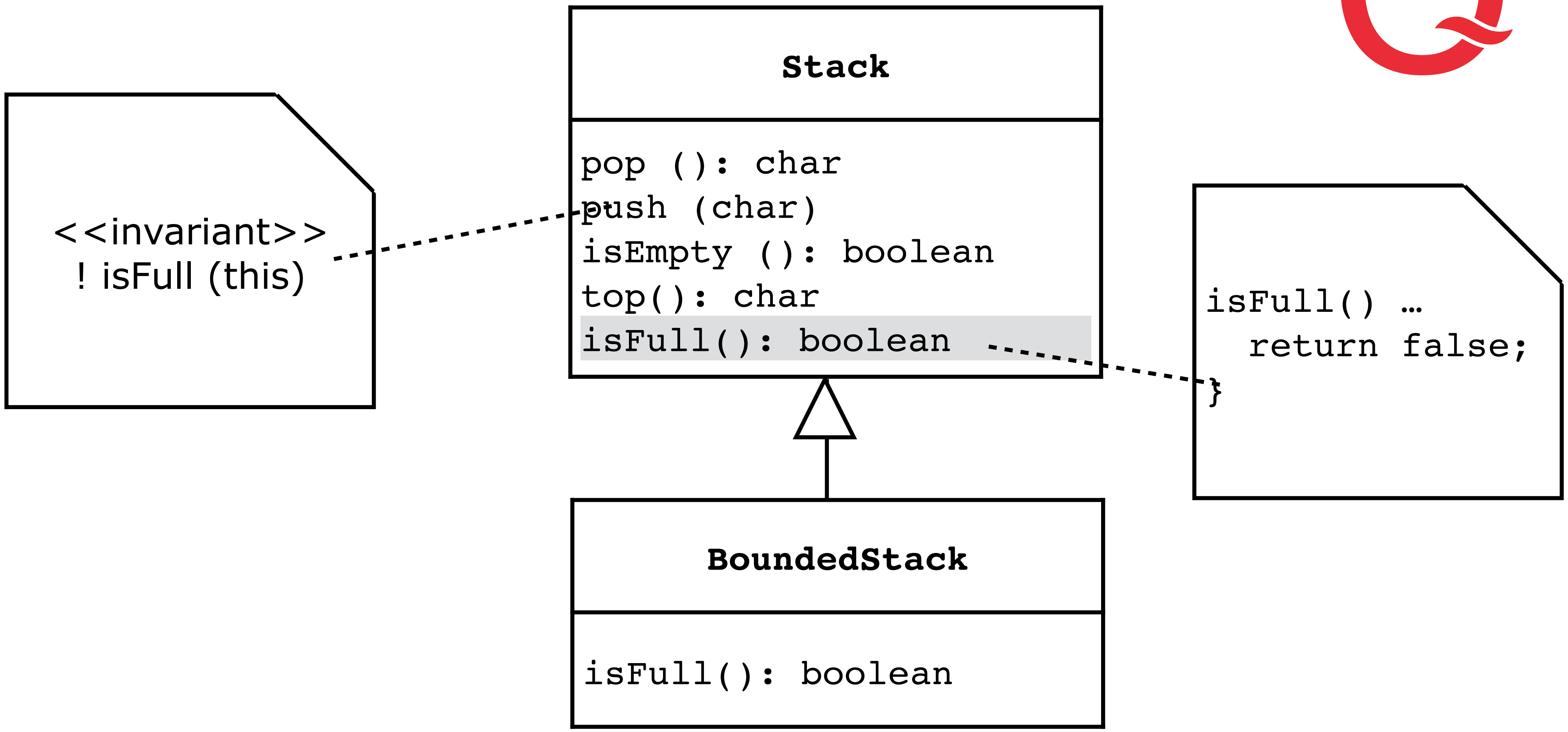
```
BoundedStack s;
s= new BoundedStack(3);
push(s, 1);
push(s, 2);
push(s, 3);
testStack (s);
```

However, it runs into a pre-condition error when we pass a bounded stack that is almost full.

⇒ substitution principle is *not* satisfied

How to fix?

What is the Fix?



- Push the “isFull” method higher and include as pre-condition for all “push” operations
- “isFull” usually returns false, but a BoundedStack overrides

Subclassing and Subcontracting

- Rule
  - + A subclass is a subcontractor of its parent class: it must at least satisfy the same contract

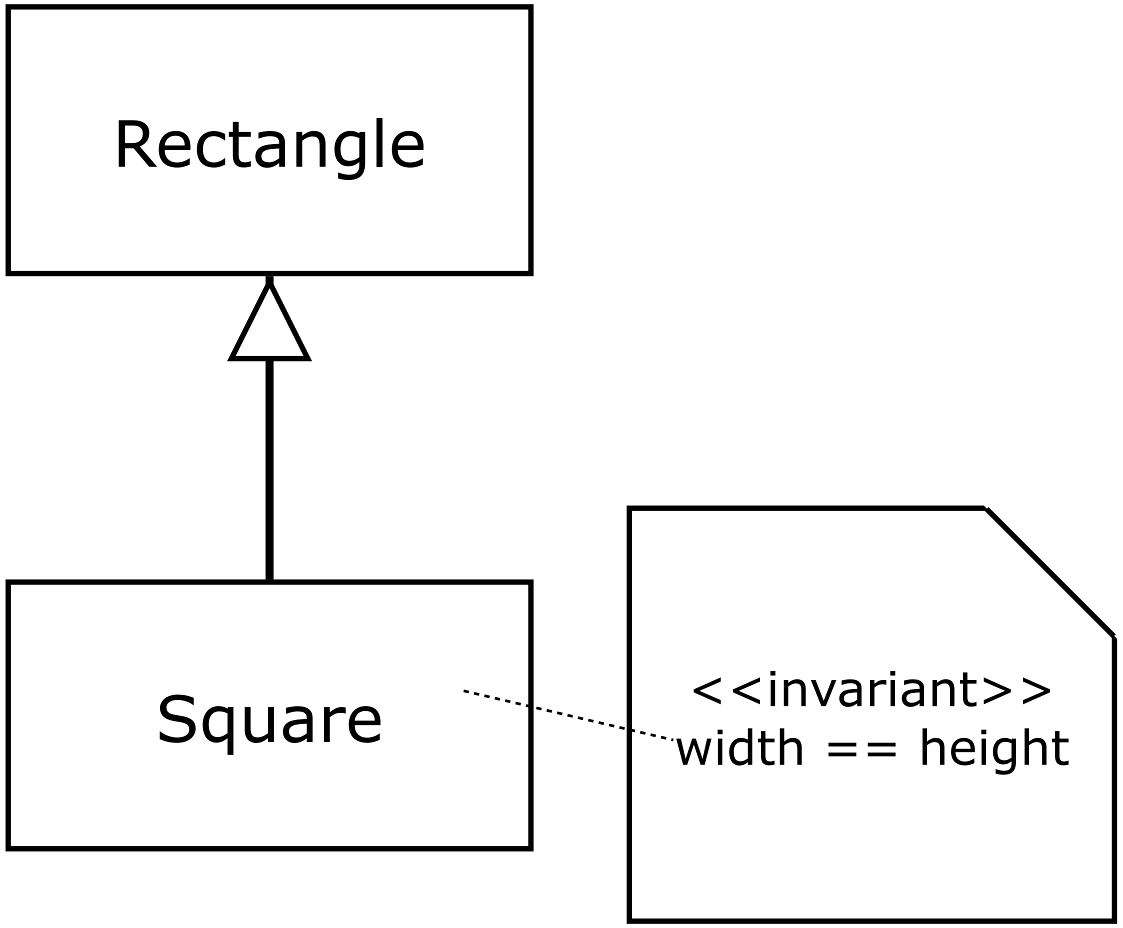
or

  - + If you subcontract, you must be willing to do the job under the original conditions, no less

- Thus
  - + Invariant: {I’} = {I}
    Invariant must remain equal (though may be expressed differently)
  - + Precondition: {P’} is weaker or equal to {P}
  - + Postcondition: {Q’} is stronger or equal to {Q}

- Implementation Issue
  - + Eiffel has special syntax for extensions of pre- and postconditions
    - * Compile-time guarantee that the substitution principle holds
  - + In other languages it is left to the programmer to ensure this rule

Behavioural Subtyping



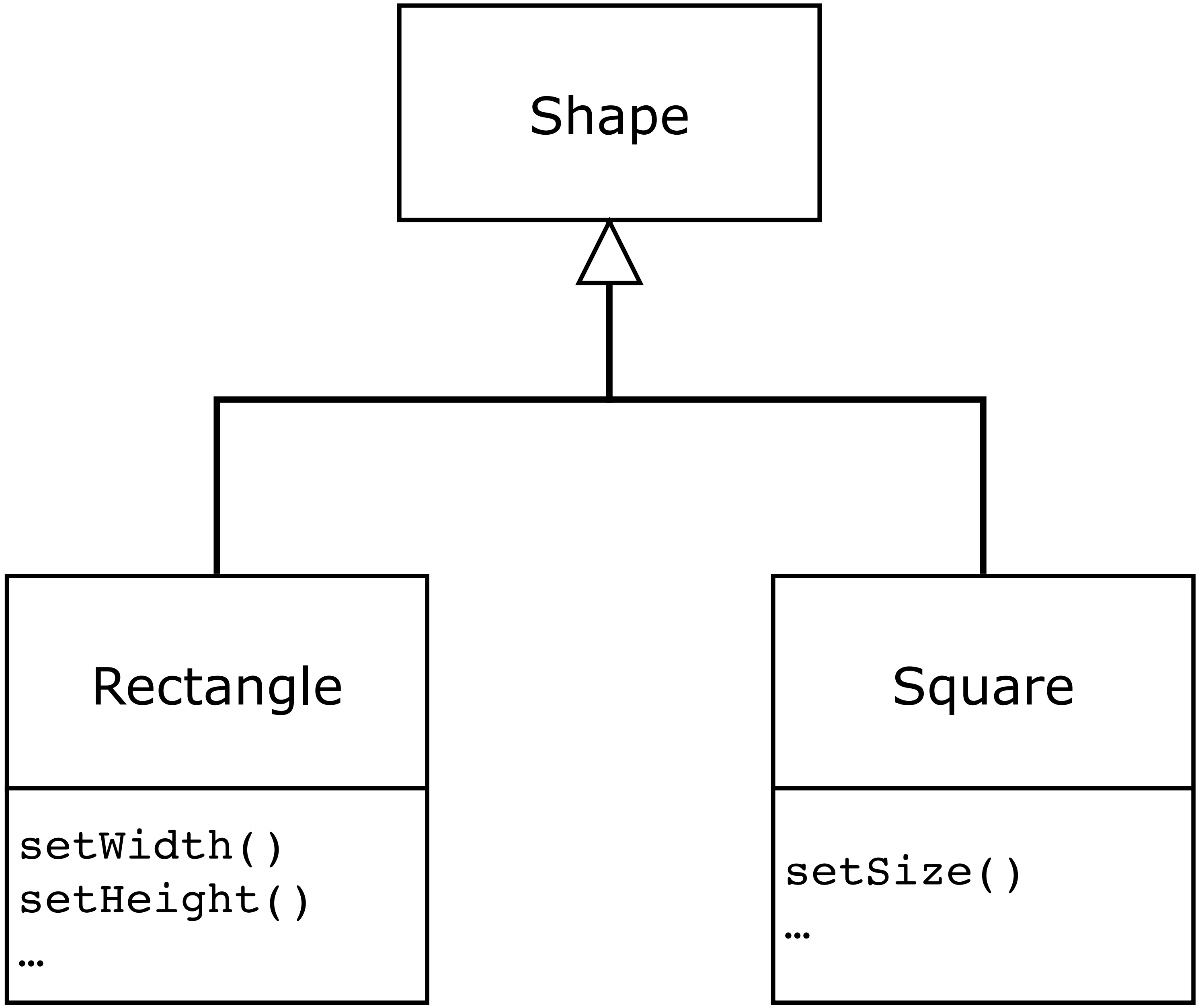
```
testRectangle(Rectangle r) {
  r.setWidth(2);
  r.setHeight(3);
  assert r.getWidth() == 2;
  assert r.getHeight() == 3;
}
```

```
Square s;
s = new Square(3);
testRectangle(s);
```

- a square “is a” rectangle
  + all square are rectangles; not all rectangles are squares
    > a square is a *subtype* of a rectangle
- but a square is not a *behavioural subtype* of a rectangle
    > a square does not respect the contractual obligations of rectangle
    > rectangle explicitly allows height and width to differ

How to fix?

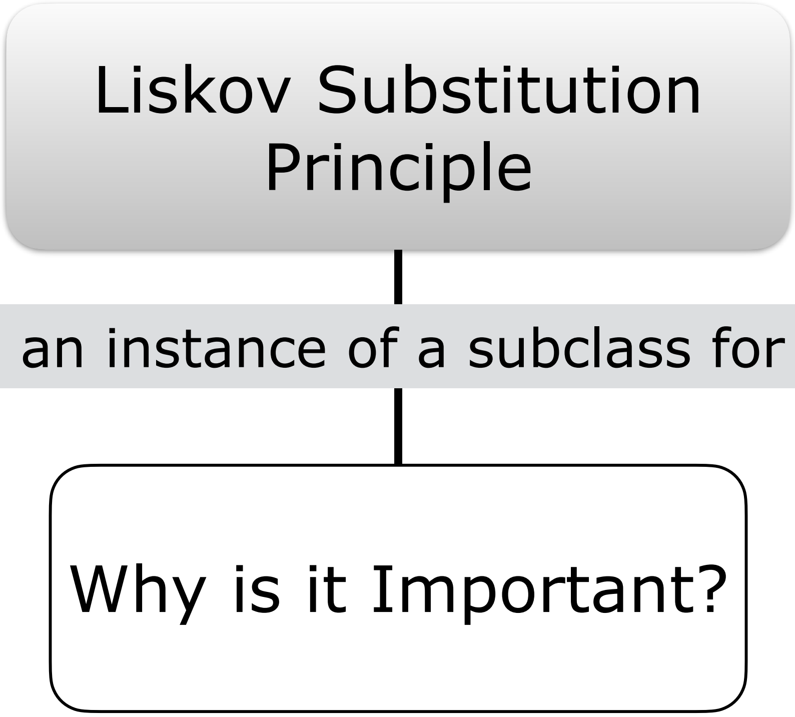
What is the Fix?



Rectangles and Square become siblings in an inheritance hierarchy

Exam Question

What’s the Liskov substitution principle?
Why is it important in OO development?



Liskov Substitution Principle

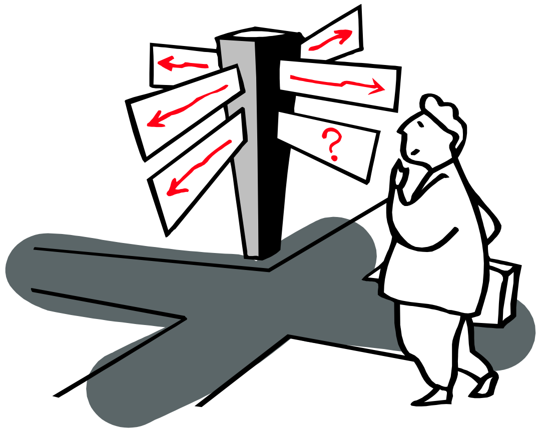
*** You may substitute an instance of a subclass for any of its superclasses.

Why is it Important?

It tells us what a subclass may do with pre- and post-conditions and invariants.

- Invariant: {I’} = {I}
  + Invariant must remain equal (though may be expressed differently)
- Precondition: {P’} is weaker or equal to {P}
- Postcondition: {Q’} is stronger or equal to {Q}

CHAPTER 6 – Design by Contract



- Introduction
  - + When, Why & What
  - + Pre & Postconditions + Invariants
    - Example: Stack
- Implementation
  - + Redundant Checks vs. Assertions
  - + Exception Handling
  - + Assertions are not…
- Theory
  - + Correctness formula
  - + Weak and Strong
  - + Invariants
  - + Subclassing and Subcontracting
    - The Liskov Substitution Principle
    - Behavioral subtyping
- Conclusion
  - + How Detailed?
  - + Tools: The Daikon Invariant Detector
  - + Modern Application: Rest API
  - + Example: Banking
  - + Design by Contract vs. Testing

How Detailed Should the Contract Be?

- Given correctness formula: {P} A {Q} for operation A
    + P := {false} is not desirable; nobody will invoke an operation like that
    + P := {true} looks promising... at first sight
        - A will do some computation + check for abnormal cases + take corrective actions and notify clients + produce a result anyway
            * It will be difficult to implement A correctly
            * It will be difficult to reuse A

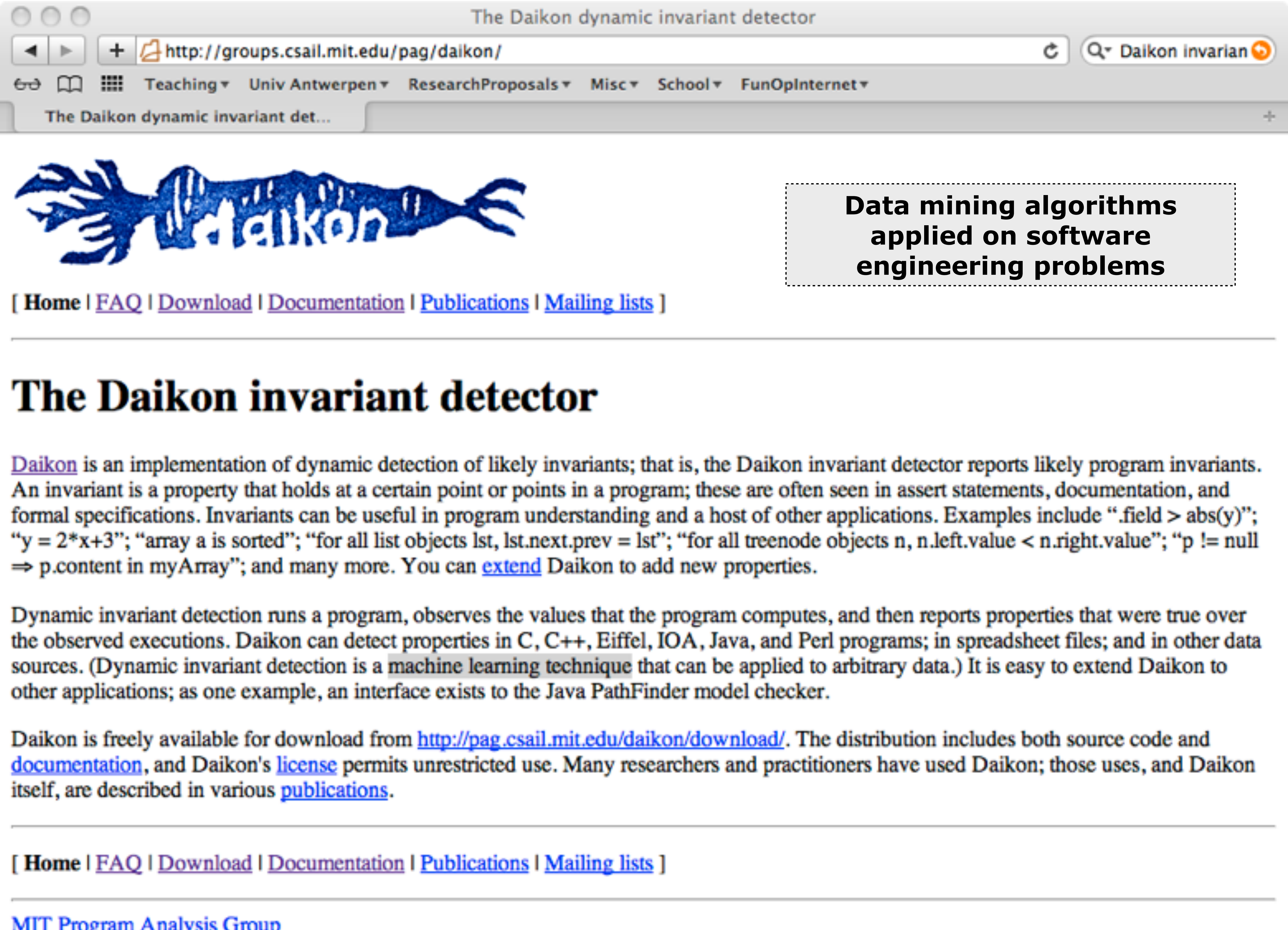
***** Strong preconditions make a component more reusable**

- Reasonable precondition: When designing a component with preconditions
    + It must be possible to justify the need for the precondition in terms of the requirements specification only
    + Clients should be able to satisfy and check the precondition
        - All operations used inside the precondition should be declared public

Q cfr. Question on slide — 24. Weak or Strong (Preconditions)

If you are client of that class, would you prefer a weak precondition?

> *We want a reasonable precondition*


The Daikon dynamic invariant detector
http://groups.csail.mit.edu/pag/daikon/
Daikon invarian
Teaching
Univ Antwerpen
ResearchProposals
Misc
School
FunOpInternet
The Daikon dynamic invariant det...
daikon

Data mining algorithms applied on software engineering problems

[**Home** | FAQ | Download | Documentation | Publications | Mailing lists]

The Daikon invariant detector

Daikon is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications. Invariants can be useful in program understanding and a host of other applications. Examples include ".field > abs(y)"; "y = 2*x+3"; "array a is sorted"; "for all list objects lst, lst.next.prev = lst"; "for all treenode objects n, n.left.value < n.right.value"; "p != null ⇒ p.content in myArray"; and many more. You can extend Daikon to add new properties.

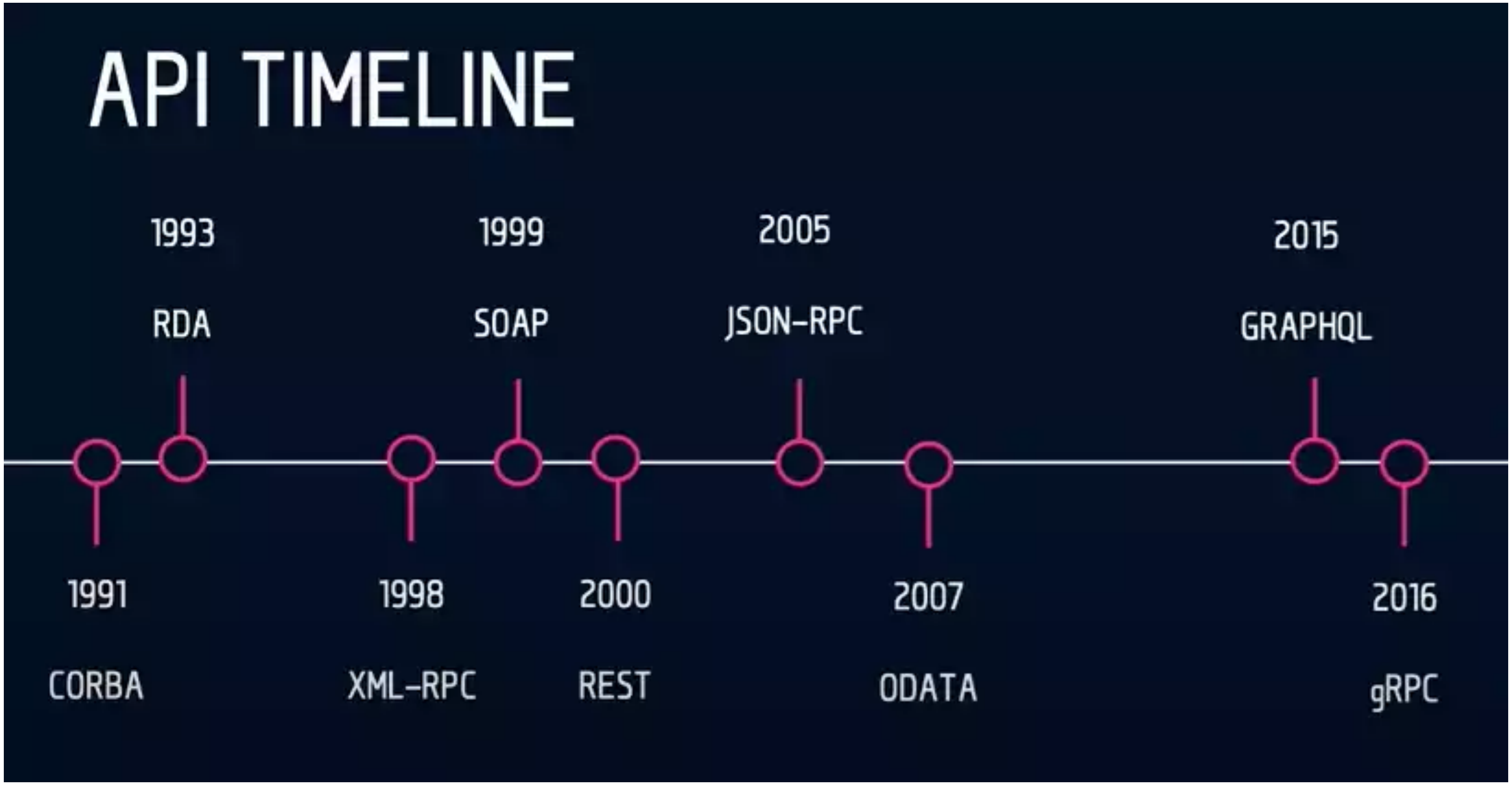
Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Daikon can detect properties in C, C++, Eiffel, IOA, Java, and Perl programs; in spreadsheet files; and in other data sources. (Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data.) It is easy to extend Daikon to other applications; as one example, an interface exists to the Java PathFinder model checker.

Daikon is freely available for download from http://pag.csail.mit.edu/daikon/download/. The distribution includes both source code and documentation, and Daikon's license permits unrestricted use. Many researchers and practitioners have used Daikon; those uses, and Daikon itself, are described in various publications.

[**Home** | FAQ | Download | Documentation | Publications | Mailing lists]

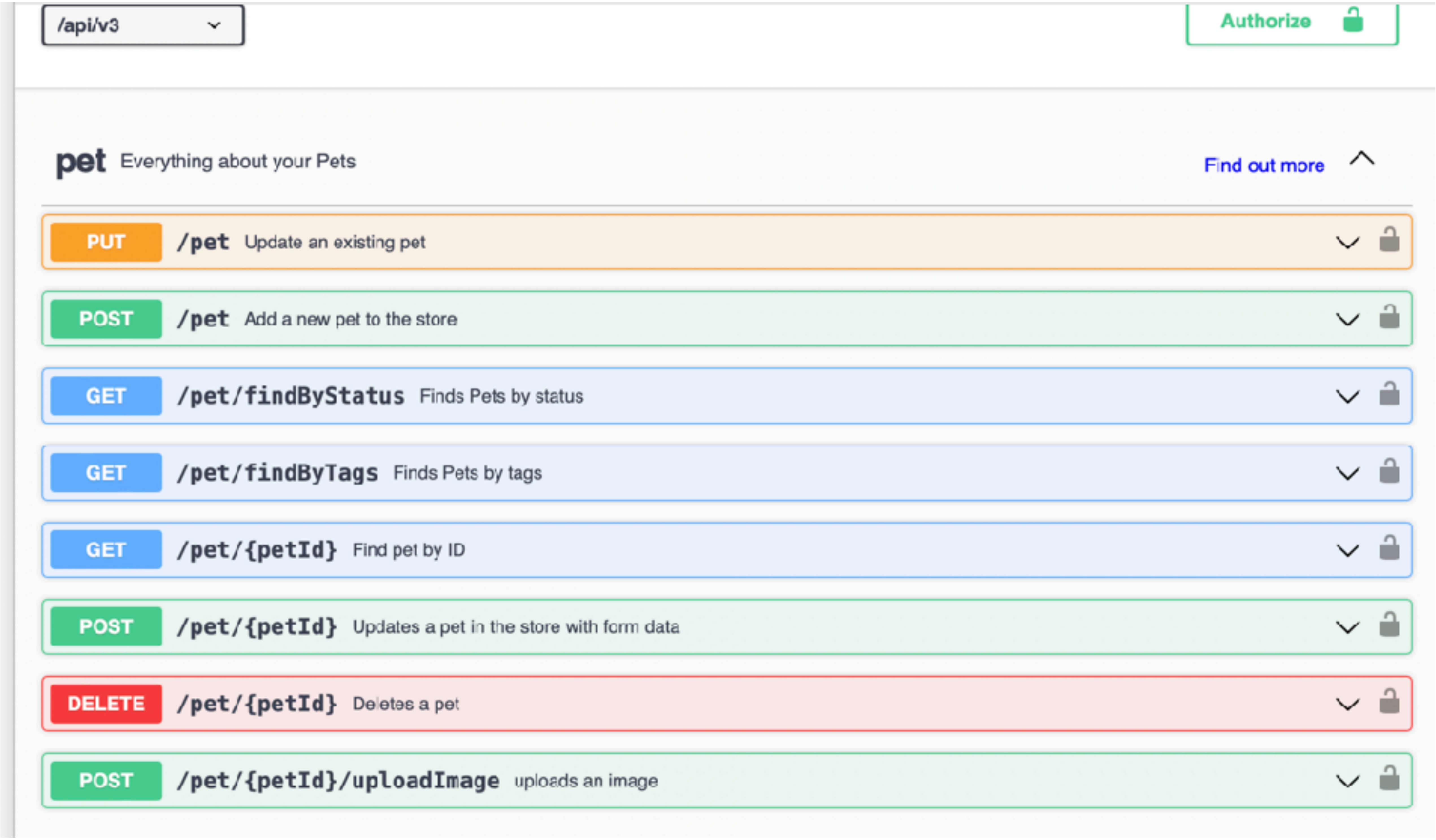
MIT Program Analysis Group

REST API — History



© API styles over time, Source: Rob Crowley

MicroService Example - Pet Store (REST API)



pet Everything about your Pets

PUT /pet Update an existing pet

POST /pet Add a new pet to the store

GET /pet/findByStatus Finds Pets by status

GET /pet/findByTags Finds Pets by tags

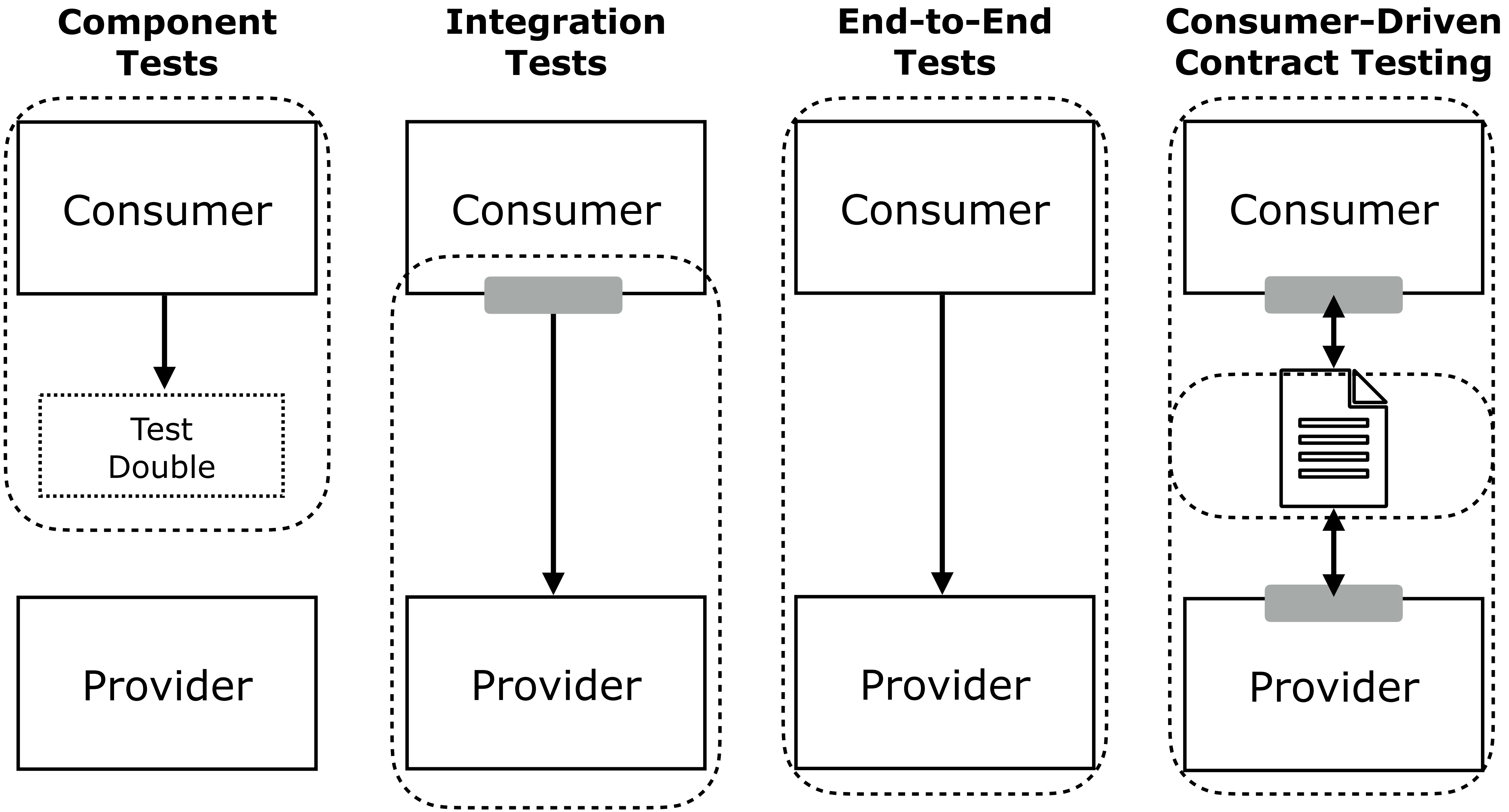
GET /pet/{petId} Find pet by ID

POST /pet/{petId} Updates a pet in the store with form data

DELETE /pet/{petId} Deletes a pet

POST /pet/{petId}/uploadImage uploads an image

Test Strategies for Micro-Services

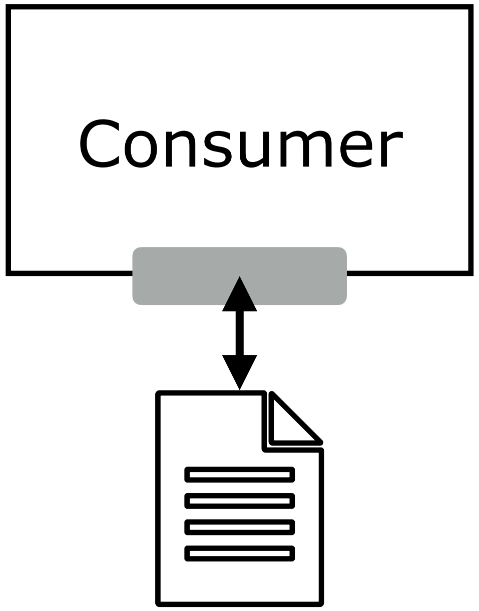


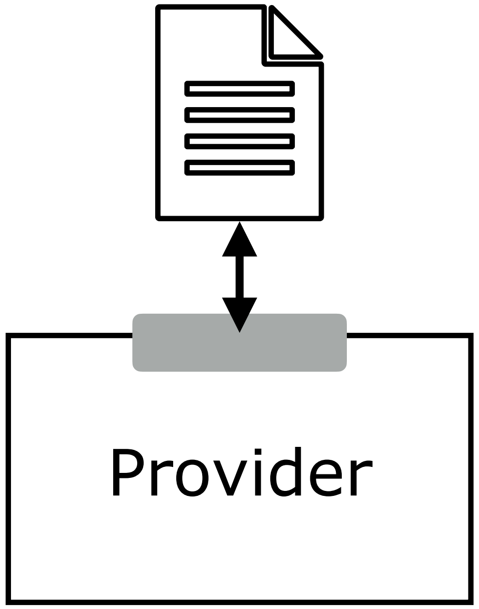
Lehvä, J., Mäkitalo, N., Mikkonen, T. (2019). Consumer-Driven Contract Tests for Microservices: A Case Study. In: Franch, X., Männistö, T., Martínez-Fernández, S. (eds) Product-Focused Software Process Improvement. PROFES 2019. Lecture Notes in Computer Science(), vol 11915. Springer, Cham. https://doi.org/10.1007/978-3-030-35333-9_35

Consumer-Driven Contract Testing

Test micro-services in isolation, solely based on the contractual obligations.

- Consumer
  + Explicit delivery of the (part of) the contract used.
  + Mocks the provider based on the (part of the) contract.
- Provider
  + Replay consumer requests against its API.
  + Verify responses against contract.





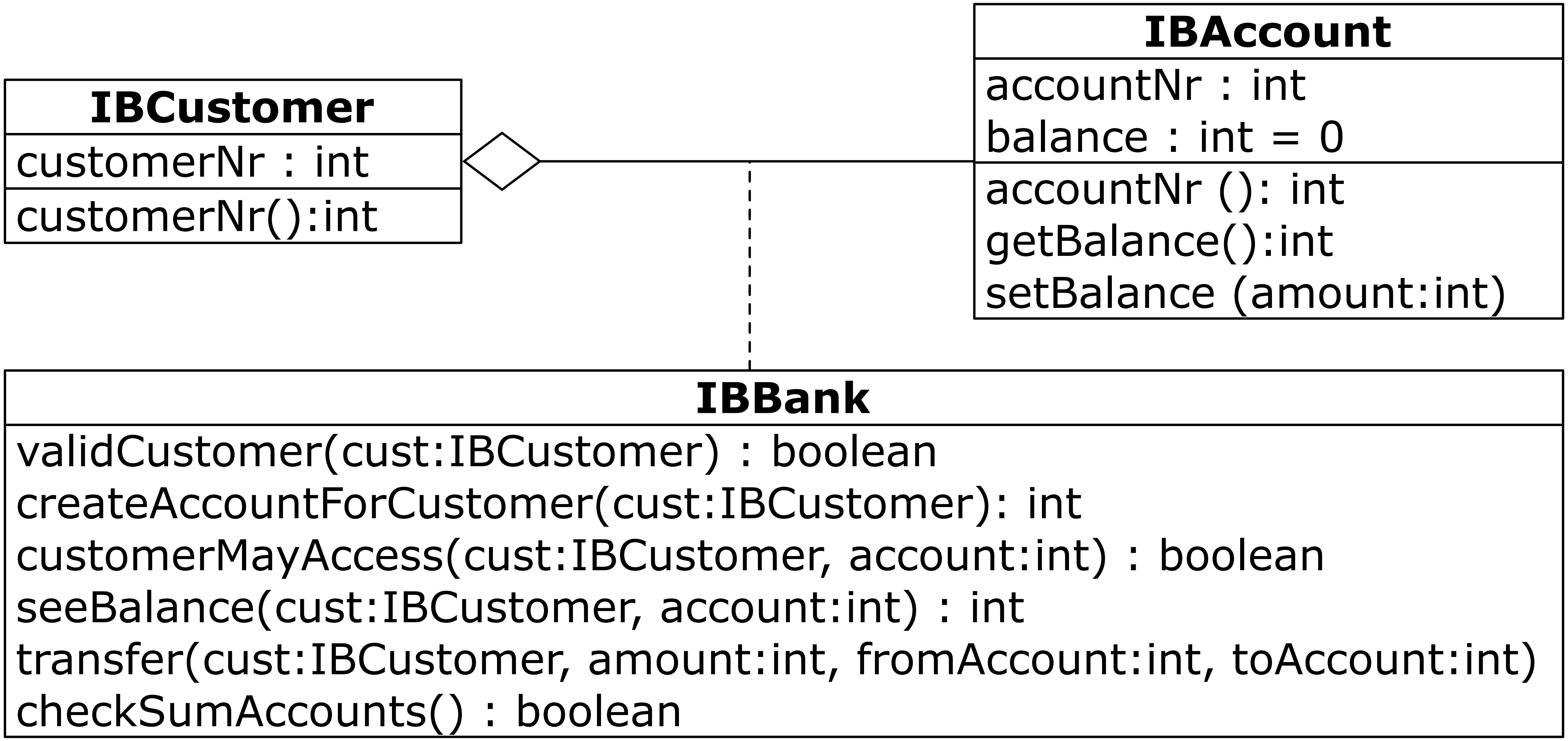
Provider is aware which parts of the contract are actually used by consumers.
> Breaking contracts is explicitly under control.

Example: Banking - Requirements

+ a bank has customers
+ customers own account(s) within a bank
+ with the accounts they own, customers may
  - deposit / withdraw money
  - transfer money
  - see the balance

- Non-functional requirements
  + *secure*: only authorised users may access an account
  + *reliable*: all transactions must maintain consistent state

Example: Banking - Class Diagram



IBCustomer
customerNr : int
customerNr():int
IBAccount
accountNr : int
balance : int = 0
accountNr (): int
getBalance():int
setBalance (amount:int)
IBBank
validCustomer(cust:IBCustomer) : boolean
createAccountForCustomer(cust:IBCustomer): int
customerMayAccess(cust:IBCustomer, account:int) : boolean
seeBalance(cust:IBCustomer, account:int) : int
transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
checkSumAccounts() : boolean

Example: Banking - Contracts

Ensure the "secure" and "reliable" requirements.

```
IBBank
    invariant: checkSumAccounts()


IBBank::createAccountForCustomer(cust:IBCustomer): int
    precondition: validCustomer(cust)
    postcondition: customerMayAccess(cust, <<result>>)


IBBank::seeBalance(cust:IBCustomer, account:int) : int
    precondition: (validCustomer(cust)) AND
        (customerMayAccess(cust, account))
    postcondition: true


IBBank::transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
    precondition: (validCustomer(cust))
        AND (customerMayAccess(cust, fromAccount))
        AND (customerMayAccess(cust, toAccount))
    postcondition: true
```

Example: Banking - CheckSum

Bookkeeping systems always maintain two extra accounts, “incoming” and “outgoing”

- ⇒ the sum of the amounts of all transactions is always 0 ⇒ consistency check

Incoming

date	amount
1/1/2000	-100
1/2/2000	-200

MyAccount

date	amount
1/1/2000	+100
1/2/2000	+200
1/3/2000	-250

OutGoing

date	amount
1/3/2000	+250

Correctness & Traceability

- Design by contract prevents defects
- Testing detect defects
  + One of them should be sufficient!?

- Design by contract and testing are *complementary*
  + None of the two guarantee correctness ...
    but the sum is more than the parts.
    - Testing *detects* wide range of coding mistakes
    - ... design by contract *prevents* specific mistakes
      (due to incorrect assumptions between provider and client)
  + design by contract ⇒ black box testing techniques
    - especially, equivalence partitioning & boundary value analysis
  + (condition) testing ⇒ verify whether parties satisfy their obligations
    - especially, whether all assertions are satisfied
  + consumer-driven contract testing ⇒ test distributed components in isolation

- Design by contract (and Testing) support Traceability
  + Assertions are a way to record requirements in the source code
  + (Regression) tests map assertions back to the requirements

Summary(i)

- You should know the answers to these questions
  - + What is the distinction between Testing and Design by Contract? Why are they complementary techniques?
  - + What's the weakest possible condition in logic terms? And the strongest?
  - + If you have to implement an operation on a class, would you prefer weak or strong conditions for pre- and postcondition? And what about the class invariant?
  - + If a subclass overrides an operation, what is it allowed to do with the pre- and postcondition? And what about the class invariant?
  - + Compare Testing and Design by contract using the criteria "Correctness" and "Traceability".
  - + What's the Liskov substitution principle? Why is it important in OO development?
  - + What is behavioral subtyping?
  - + When is a pre-condition reasonable?

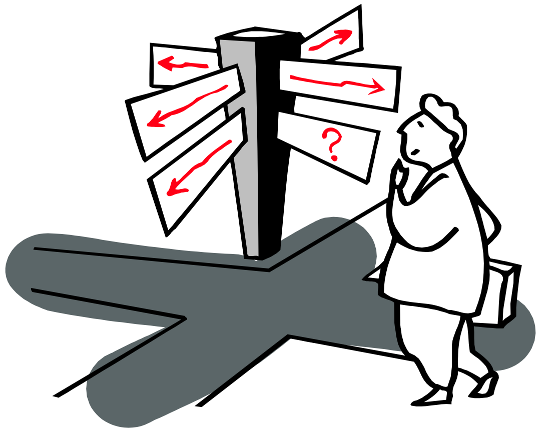
- You should be able to complete the following tasks
  - + What would be the pre- and post-conditions for the methods top and isEmpty in the Stack specification? How would I extend the contract if I added a method size to the Stack interface?
  - + Apply design by contract on a class Rectangle, with operations move() and resize().
  - + Write consumer-driven contracts for a given REST-API .

Summary(ii)

- Can you answer the following questions?
  - + Why are redundant checks not a good way to support Design by Contract?
  - + You’re a project manager for a weather forecasting system, where performance is a real issue. Set-up some guidelines concerning assertion monitoring and argue your choice.
  - + If you have to buy a class from an outsourcer in India, would you prefer a strong precondition over a weak one? And what about the postcondition?
  - + Do you feel that design by contract yields software systems that are defect free? If you do, argue why. If you don’t, argue why it is still useful.
  - + How can you ensure the quality of the pre- and postconditions?
  - + Why is (consumer-driven) contract testing so relevant in the context of micro-services?
  - + Assume you have an existing software system and you are a software quality engineer assigned to apply design by contract. How would you start? What would you do?

CHAPTER 5 – Testing

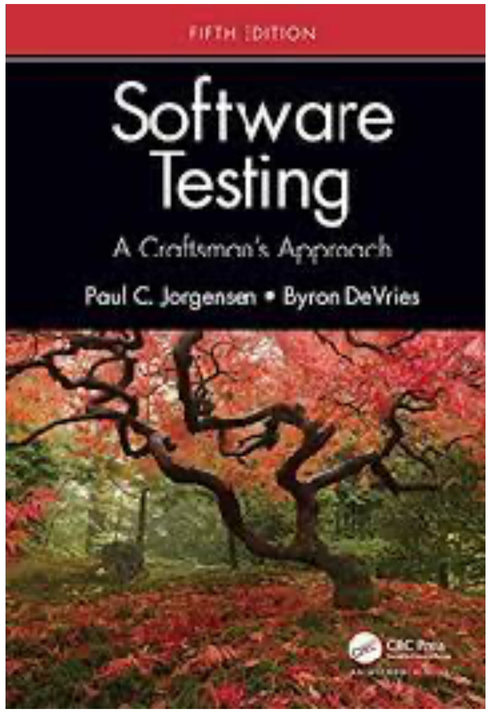
- Introduction
  + When, Why, What & Who?
    - The V-Model
  + What is “Correct”?
  + Terminology
- Testing Techniques
  + White Box
    - basis path, conditions, loops
  + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  + Black Box
    - equivalence partitioning
  + *Fuzz Testing*



- Testing Strategies
  + Unit & Integration Testing
  + Regression Testing
  + Acceptance Testing
  + More Testing Strategies
- Miscellaneous
  + When to Stop?
  + Tool Support
- Agile Testing (DevOps)
  + Flipping the V
  + 4-Quadrants
  + FIT Tables
- Conclusion
  + More Good Reasons

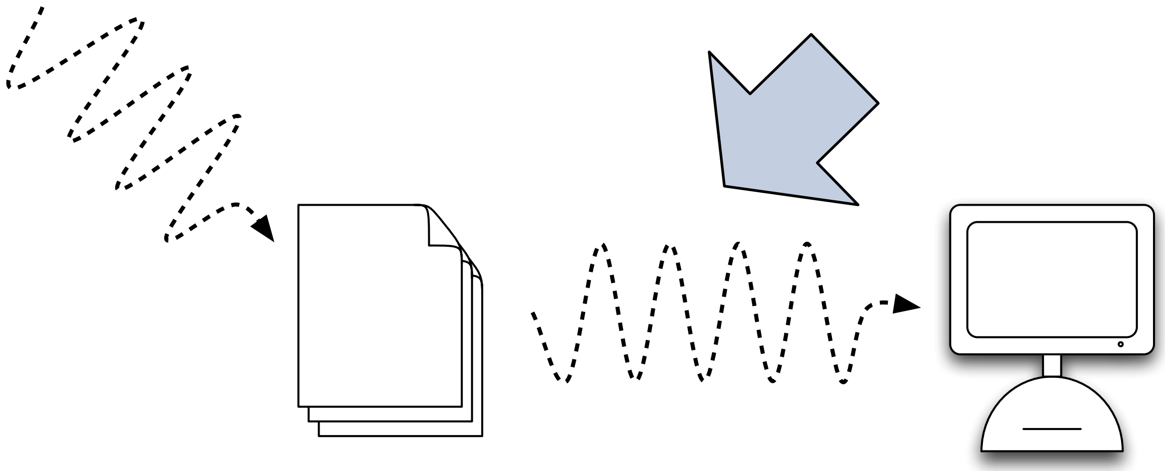
Literature

- Books
  + [Ghez02] Chapter on “Software Verification” is quite good with plenty of examples of the need for complementary testing techniques. Terminology used here differs from [Pres00] and [Somm05]
  + [Pres00] Chapter on “Software Testing Techniques” is very good with lots of concrete examples of the different techniques.
  + [Somm05] Chapter on “Verification and Validation” places Testing in a broader context.



- Specific Books
  + [Jorg21] Software Testing: A Craftsman’s Approach (5th edition)
    - Master course on Software Testing

When to Test?



Mistakes are possible (likely!?)

- while transforming requirements into a system
- while system is changed during maintenance

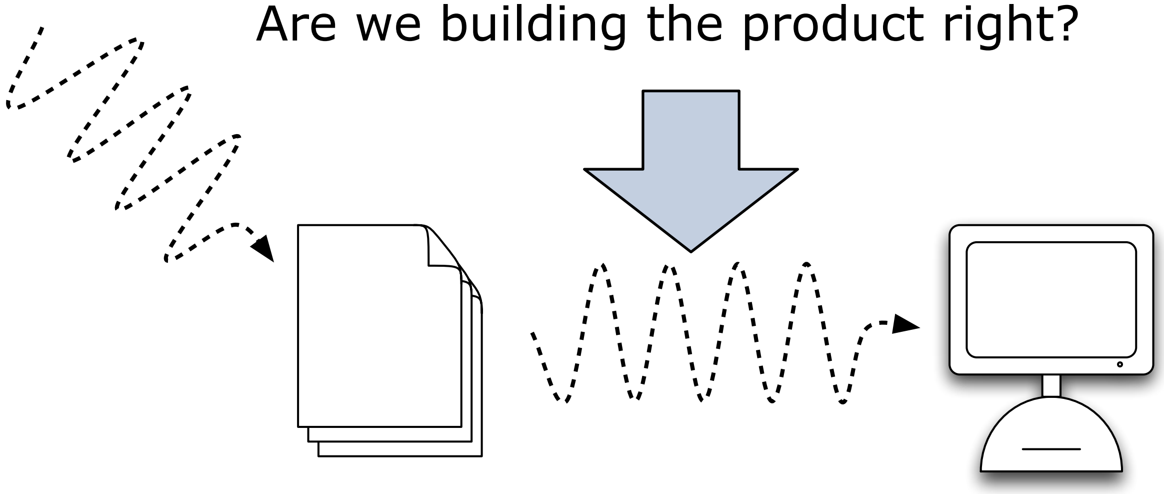
Correctness

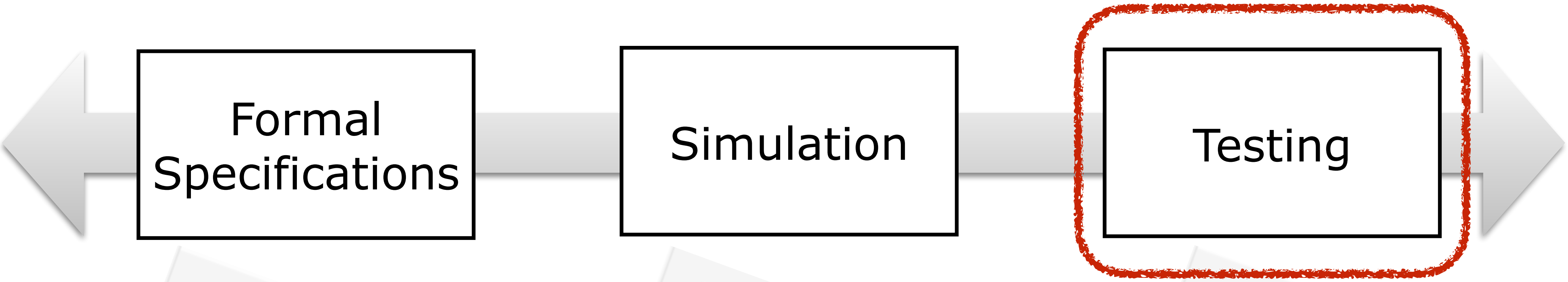
- ~~Are we building the right product? = VALIDATION~~
- Are we building the product right? = VERIFICATION

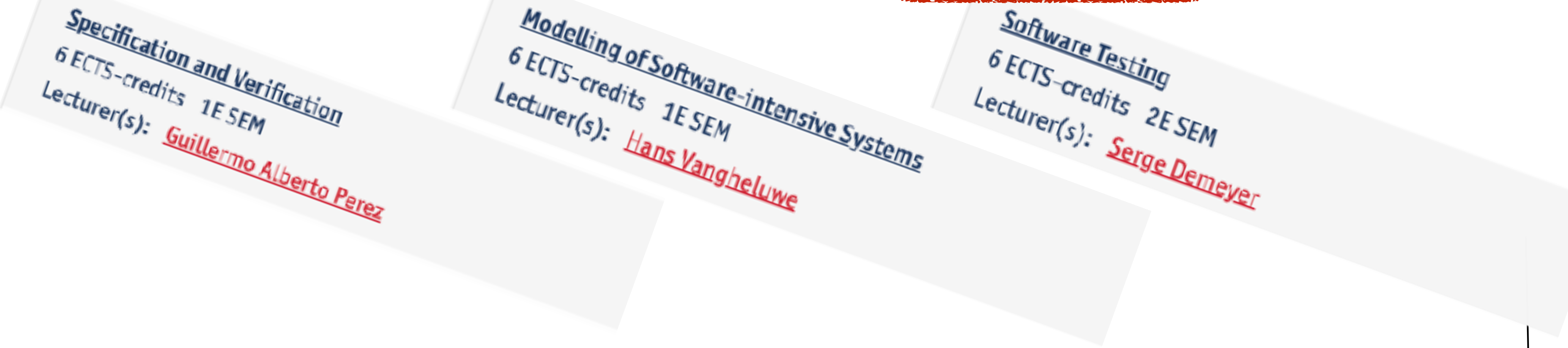
The Verification Landscape

New slide

Are we building the product right?







Specification and Verification
6 ECTS-credits 1E SEM
Lecturer(s): Guillermo Alberto Perez

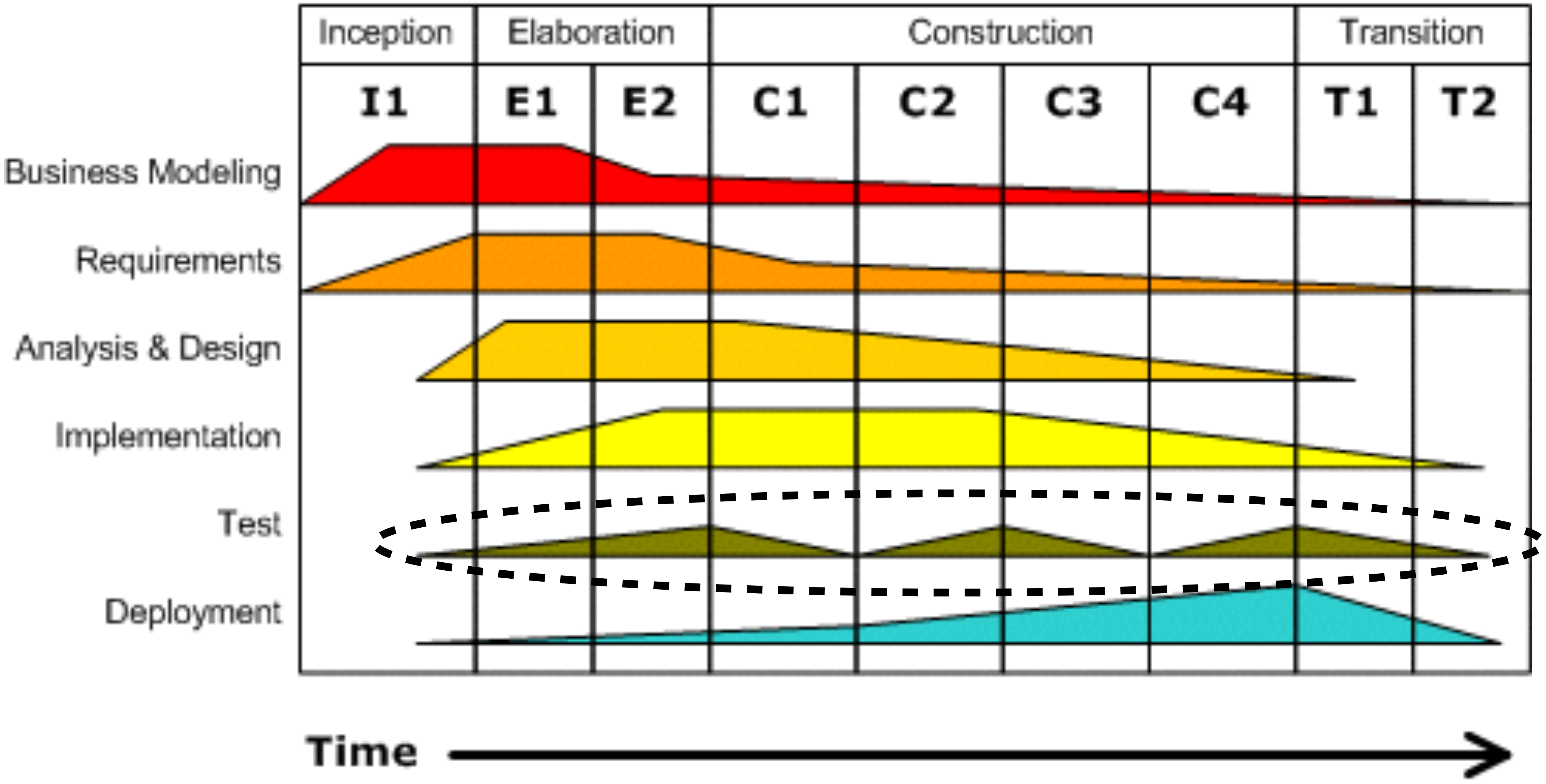
Modelling of Software-intensive Systems
6 ECTS-credits 1E SEM
Lecturer(s): Hans Vangheluwe

Software Testing
6 ECTS-credits 2E SEM
Lecturer(s): Serge Demeyer

When to Test? The Unified Process

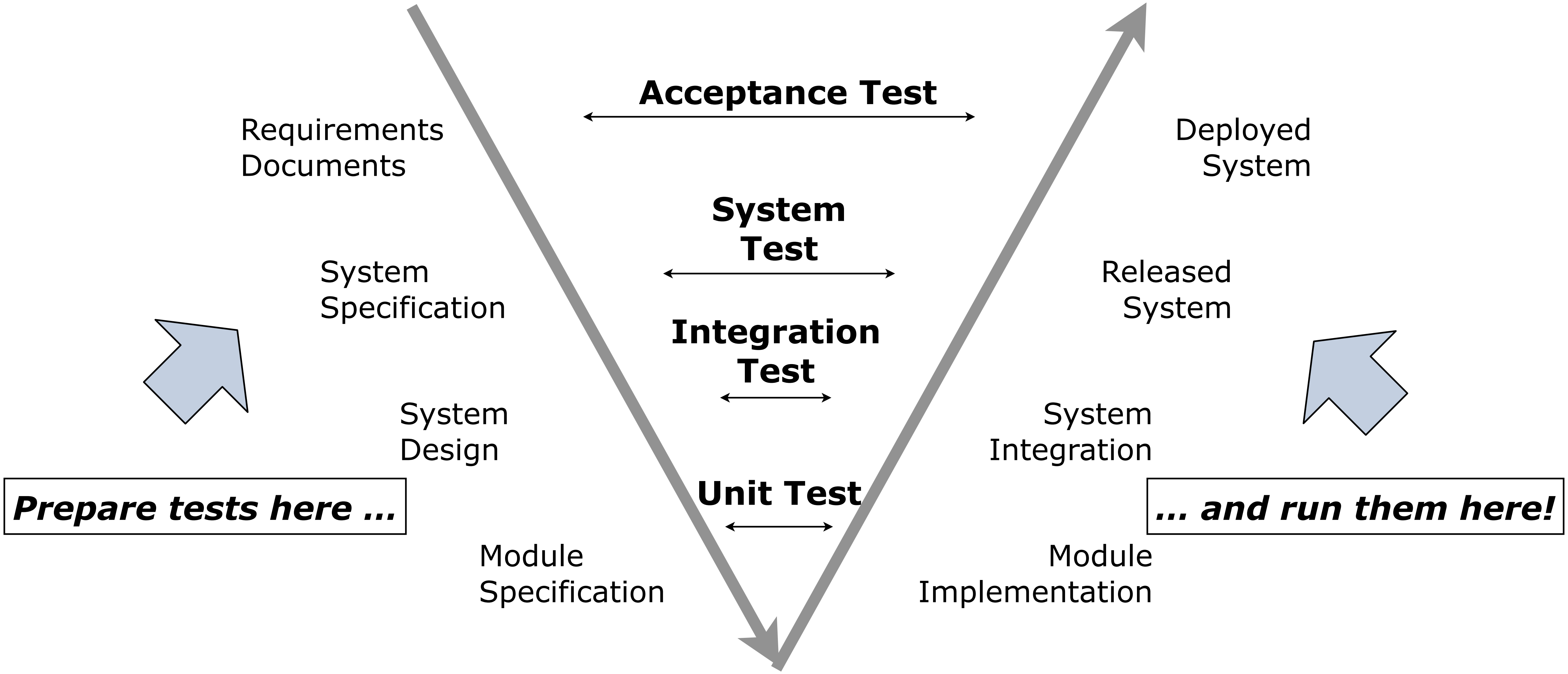
Testing is a *risk reduction* activity

- start as early as possible to assess & reduce risk towards the schedule
- repeat towards the end to assess & reduce risk towards reliability



DutchGuilder Wikipedia

When to Test? The V-model



Requirements Documents

System Specification

System Design

Module Specification

Acceptance Test

System Test

Integration Test

Unit Test

Deployed System

Released System

System Integration

Module Implementation

Prepare tests here …

… and run them here!

SPECIFY & DESIGN WITH TESTABILITY IN MIND

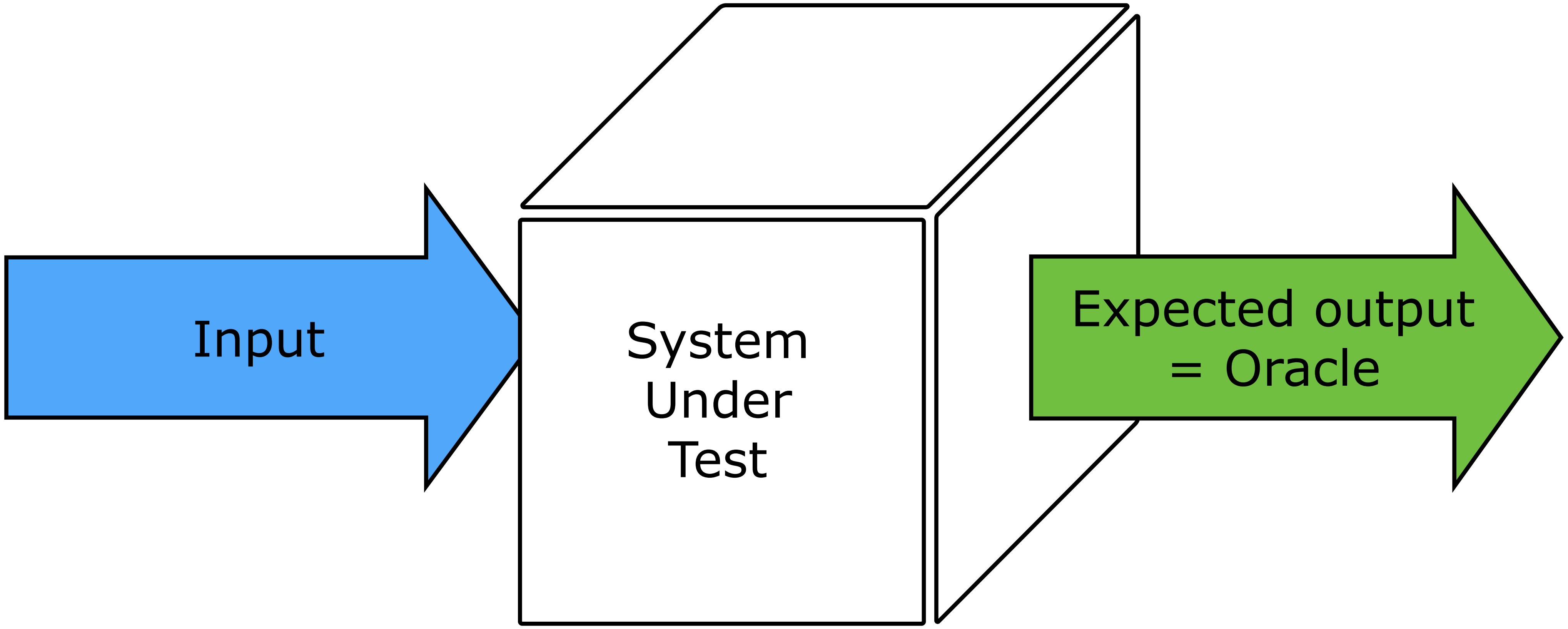
Why to Test?

> Program testing can be used to show the presence of defects, but never their absence.
>
> (E. W. Dijkstra)

- Perfect Excuse
  - + We should not invest in testing: our system will contain defects anyway
- Counter Arguments
  - + The more you test, the less likely such defects will cause harm
  - + The more you test, the more *confidence* you will have in the system
- Testing = Risk Management
  - + Testing is a risk *reduction* activity!
    - - Result of testing is a risk report to project management (Can we ship this product in good confidence?)
      - * Go / no-go decision

What is Testing? (1/3)

New slide



Software Testing is the process of executing a program or system
with the *intent* of finding errors.
(Myers, Glenford J., The art of software testing. Wiley, 1979)

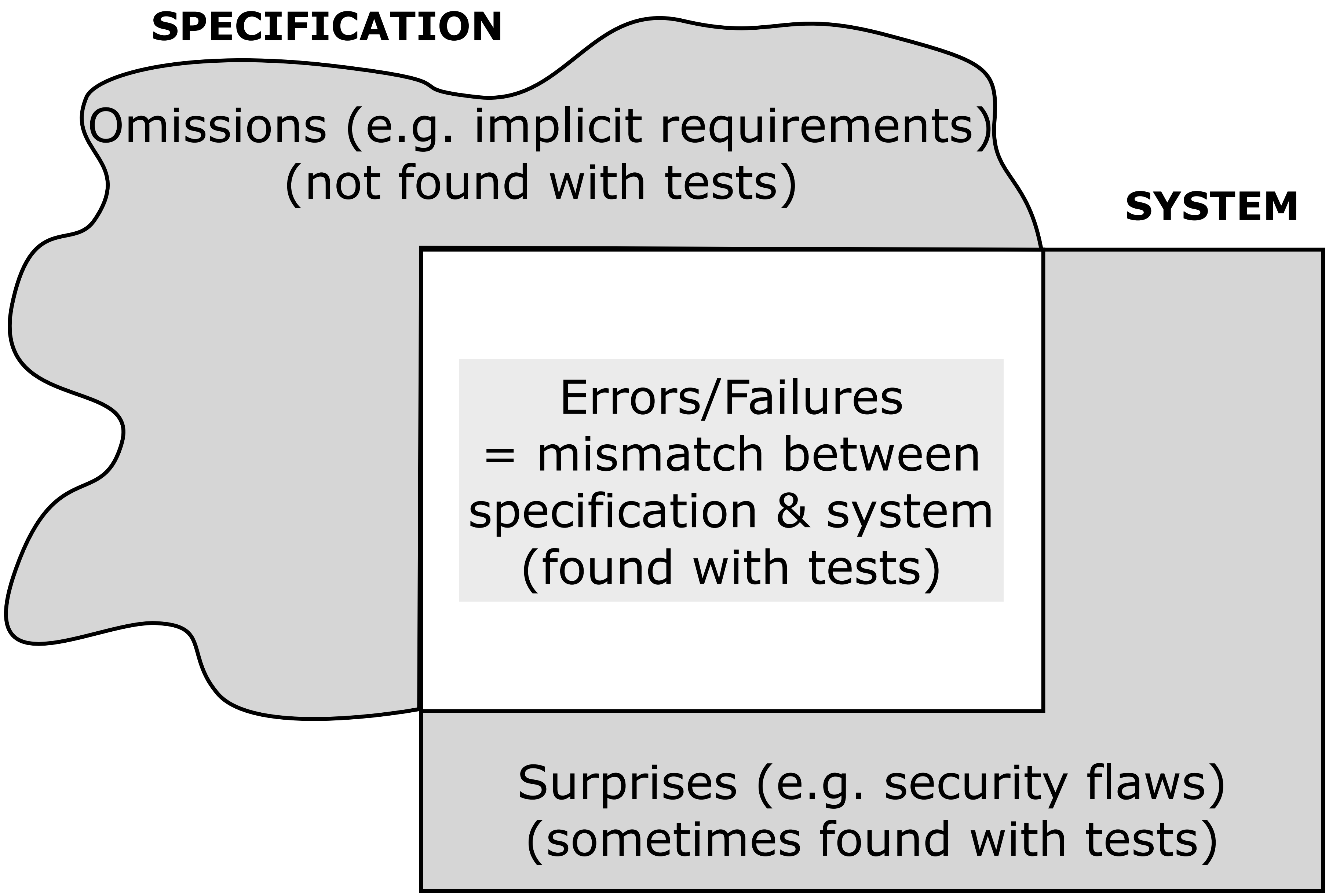
What is Testing? (2/3)

- Testing should
  + *verify* the requirements (Are we building the product right?)
  + NOT *validate* the requirements (Are we building the right product?)

- Definitions
  + Testing
    - Testing is the activity of executing a program with the intent of finding a defect
      > A successful test is one that finds defects!
  + Testing Techniques
    - Techniques with a high probability of finding an as yet undiscovered mistake
      > Criterion: *Coverage* of the code/requirements/model/risks/…
  + Testing Strategies
    - Tell you *when* you should perform *which* testing technique
      > Criterion: *Confidence* that you can safely proceed
      > Next activity = other testing until deployment

REMEMBER: Testing is a risk *reduction* activity!

What is Testing? (3/3)



Who should Test?

+ Programming is a constructive activity:
  - try to make things *work*

+ Testing is a destructive activity:
  - try to make things *fail*

Programmers are not necessarily the best testers!

- In practice
  + Testing is part of quality assurance
    - done by developers when finishing a component (unit tests)
    - done by a specialized test team when finishing a subsystem (integration tests / system tests / acceptance tests)

Unit tests …
not sufficient

- Interesting Tweet:
  All unit tests are passing

https://twitter.com/olafurw/status/1578704185809244160?s=11&t=mdYnxnMXgxYBEhH7anVCgQ

What is “Correct”?

See [Ghez02] — Representative Qualities

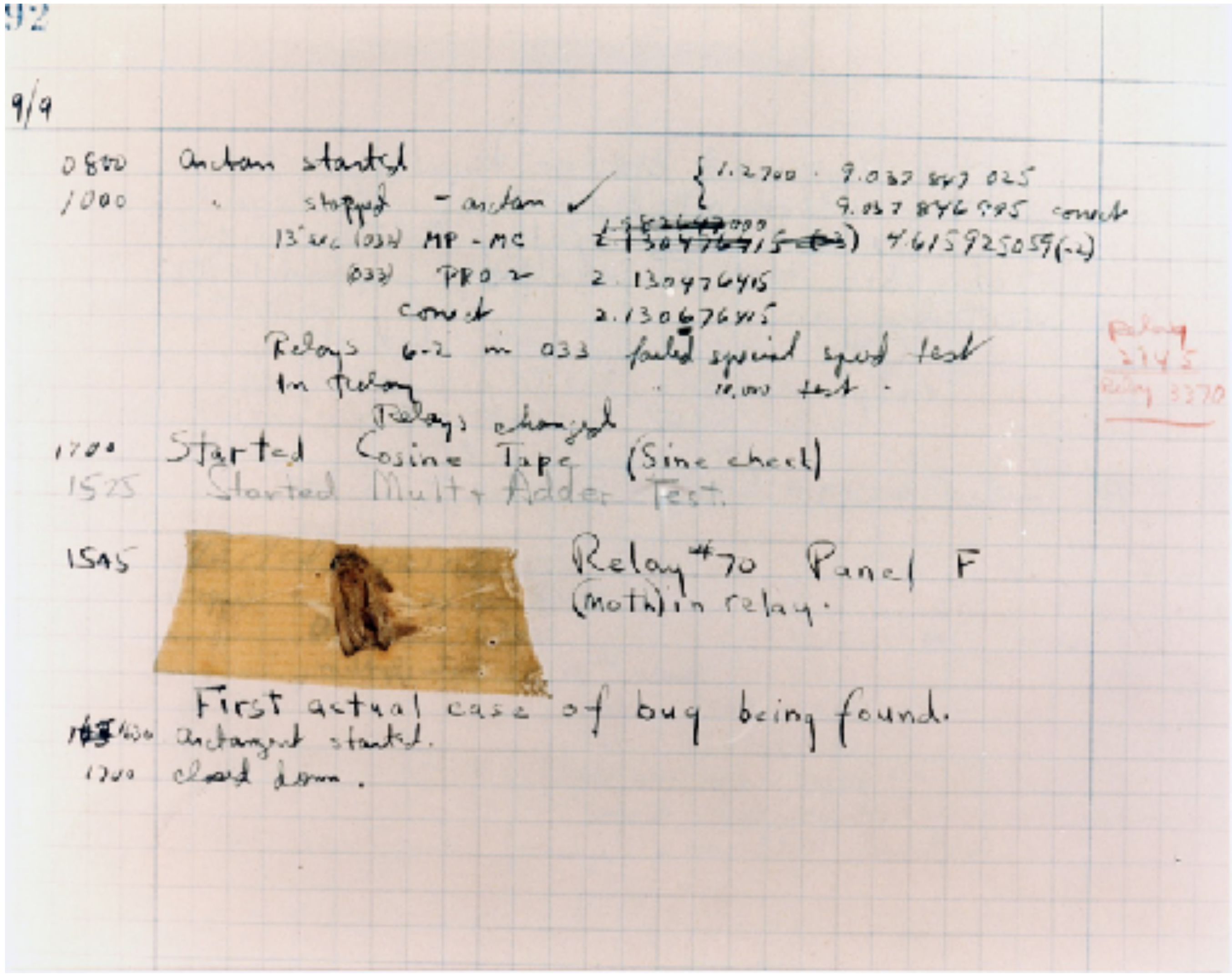
- Correctness
  + A system is correct if it behaves according to its specification
    > An *absolute* property
    (i.e., a system cannot be “almost correct”)
    > ... in theory and practice undecidable

- Reliability
  + The user may rely on the system behaving properly
  + The probability that the system will operate as expected over a specified interval
    > A *relative* property
    (a system has a mean time between failure of 3 weeks)

- Robustness
  + A system is robust if it behaves reasonably even in circumstances that were not specified
    > A *vague* property (once you specify the abnormal circumstances they become part of the requirements)

Terminology (1/3)

- Avoid the term “Bug” (*)
  + Implies mistakes creeping into the software from the outside
  + imprecise because mixes various “mistakes”



92

9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
{ 9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~
~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Relay
2145
Relay 3370

https://commons.wikimedia.org/wiki/File:First_Computer_Bug,_1945.jpg

Terminology (2/3)

To be more precise (Terminology not standard!) : IEEE Glossary / ISTQB

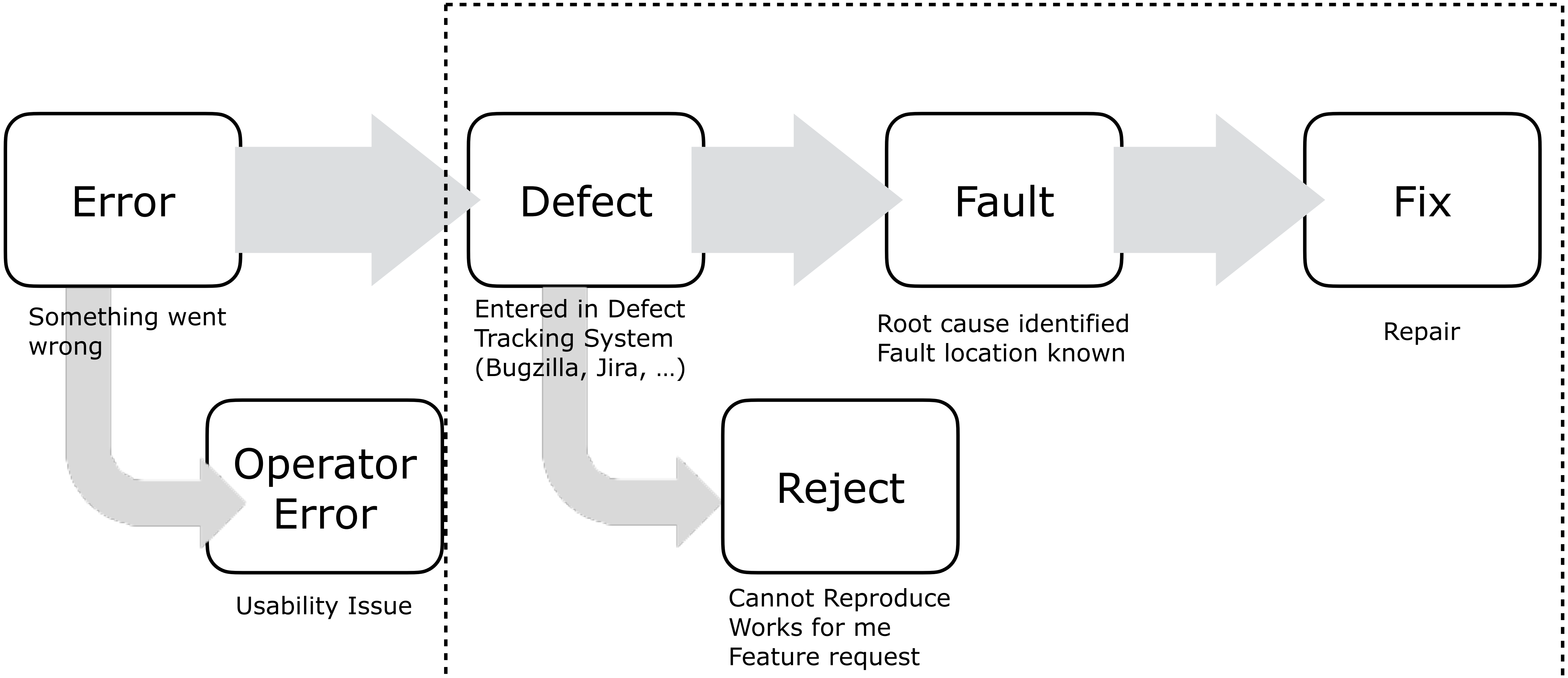
- Defect / Fault (NL = DEFECT, GEBREK, NALATIGHEID)
  - + A design or coding mistake that may cause abnormal behaviour
    - abnormal behaviour = deviations from specification (incl. surprises!)
  - + Faults by *omission*: something is missing in the design, model, code, …
  - + Faults by *commission*: incorrect entry in design, model, code, …

- Failure (NL = MISLUKKING, FALING)
  - + A deviation between the specification and the running system
  - + A manifestation of a defect during system execution
  - + Inability to perform required function within specified limits

- Error (NL = FOUT)
  - + The input that causes a failure
    - Transient occurs only with certain input combination
    - Permanent occurs with all inputs of a given class

Bug Tracking Workflow

Subtle deviations of terminology



Terminology (3/3)

- Component (Component under Test)
  + part of the system that can be isolated for testing
    - an object, a group of objects, one or more subsystems

- Test Case
  + set of inputs and expected results that exercise a component with the purpose of causing failures
    - predicate that answers “true” when the component answers with the expected results for the given input and “false” otherwise
      > “expected results” includes exceptions, error codes,...

- Test Stub
  + partial implementation of components on which the tested component depends
    - dummy code providing necessary input values and behaviour

- Test Driver
  + partial implementation of a component that depends on the tested component
    - a “main()” function that executes a number of test cases

- Test Fixture
  + fixed state of software under test, baseline for running test
    - all that is needed to set-up the appropriate test context

gTest Example: findLast

Find.cpp

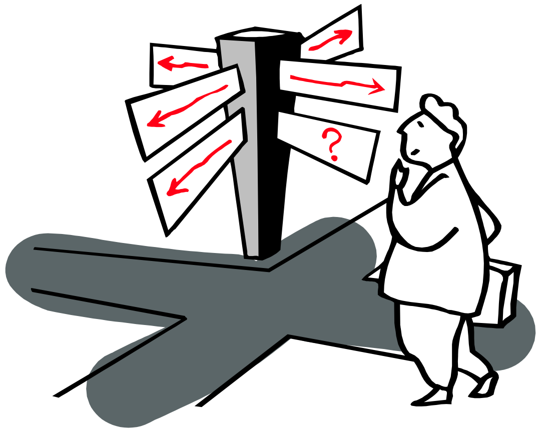
```
#include <vector>

int findLast(std::vector<int> x, int y) {
    if (x.size() == 0)
        return -1;
    for (int i = x.size() - 1; i >= 0; i–)
        if (x[i] == y)
            return i;
    return -1;
}
```

Tests.cpp

```
#include <vector>
#include <gtest/gtest.h>

#include "find.cpp"

TEST(FindLastTests, noOccurrence) {
    EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
}

TEST(FindLastTests, doubleOccurrence) {
    EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
}

TEST(FindLastTests, emptyVector) {
    EXPECT_EQ(-1, findLast({}, 3));
}
```

Are these tests sufficiently strong?
(Discuss with your neighbour)

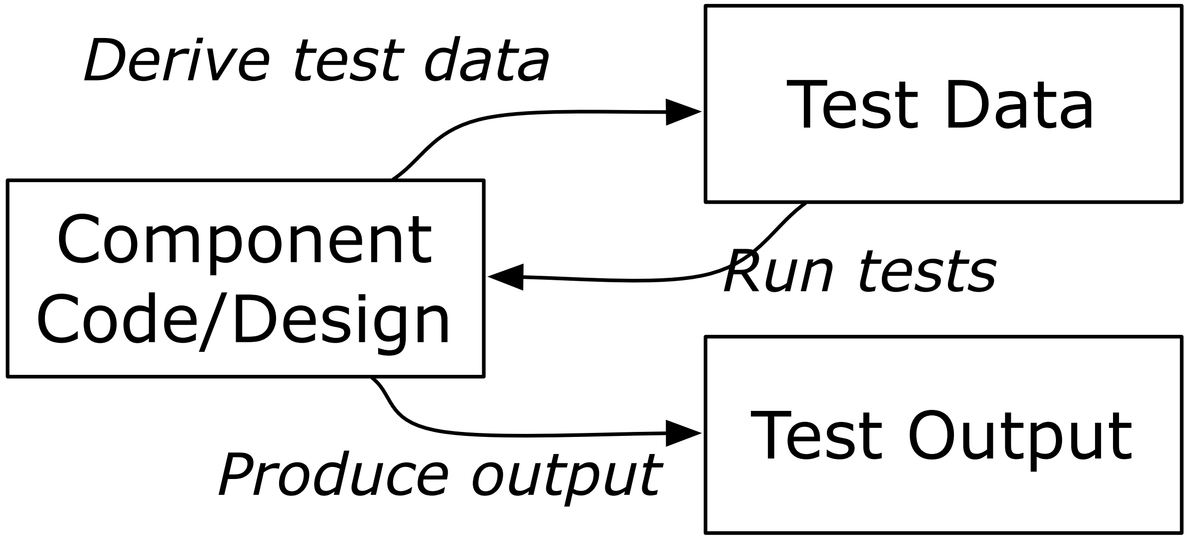
CHAPTER 5 – Testing



- Introduction
  + When, Why, What & Who?
    - The V-Model
  + What is “Correct”?
  + Terminology
- Testing Techniques
  + White Box
    - basis path, conditions, loops
  + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  + Black Box
    - equivalence partitioning
  + Fuzz Testing
- Testing Strategies
  + Unit & Integration Testing
  + Regression Testing
  + Acceptance Testing
  + More Testing Strategies
- Miscellaneous
  + When to Stop?
  + Tool Support
- Agile Testing (DevOps)
  + Flipping the V
  + 4-Quadrants
  + FIT Tables
- Conclusion
  + More Good Reasons

White Box Testing

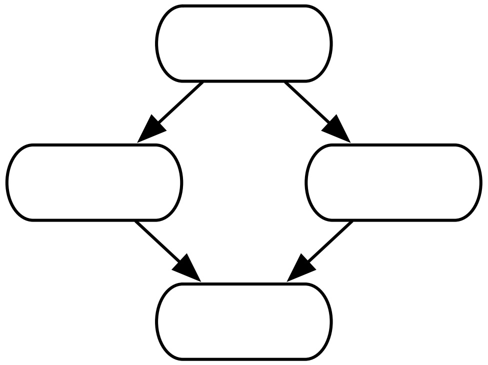
- a.k.a. Structural testing, Testing in the small
  - + Treat a component as a “white box”, i.e. you can inspect its internal structure
  - + Internal structure is also design specs; e.g. sequence diagrams, state charts, …
  - + Derive test cases to maximize coverage of that structure, yet minimize number of test cases



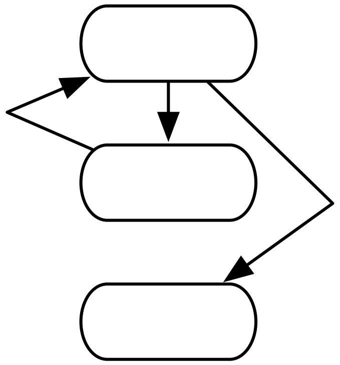
  - + Coverage criteria
    - every statement at least once
    - all portions of control flow (= branches) at least once
    - all possible values of compound conditions at least once
    - all portions of data flow at least once
    - all loops, iterated at least 0, once, and N times

Basis Path Testing (1/2)

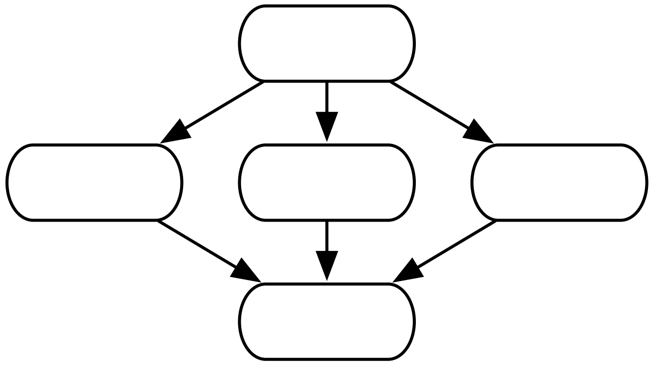
+ 1. Draw a control flow graph
    - nodes = sequences of non branching statements (assignments, procedure calls)
    - edges = control flow



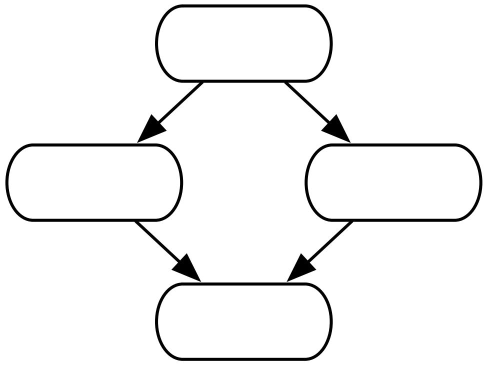
if-then-else
[cc = 2]



while
[cc = 2]



case-of
[cc = 3]

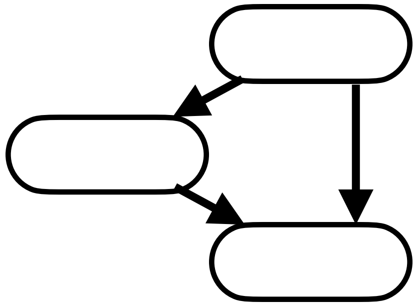


and/or
= if-then-else
[cc = 2]

Guiding principle: make sure that the control flow graphs stays as close as possible to the actual source code. This allows for better traceability when demonstrating that the test suite is well designed.

Clarification

+ Empty nodes (= an empty sequence of non-branching statement)
    - Removing them graph does not affect the cyclomatic complexity
    - But it hinders traceability
+ What with an if-then (without an else branch)
    - Then you can remove the empty else branch



Basis Path Testing (2/2)

+ …
+ 2. Compute the Cyclomatic Complexity
  = #(edges) - #(nodes) + 2
  = number of binary conditions + 1
  = # regions
+ 3. Determine a set of *independent* paths (= at least one new edge in every path)
  [name *independent* stems from a mathematical vector *basis* for the complete graph]
  - Several possibilities: upper bound = Cyclomatic Complexity
+ 4. Prepare test cases that force each of these paths
  - Choose values for all variables that control the branches.
  - Predict the result in terms of values and/or exceptions raised
+ 5. Write test driver for each test case

Example - Code

```
 public boolean find(int key) {                          //Binary Search
   int bottom = 0;                                       // (1)
   int top = _elements.length-1;
   int lastIndex = (bottom+top)/2;
   int mid;
   boolean found = key == _elements[lastIndex];
   while ((bottom <= top) && !found) {                  // (2) (3)
     mid = (bottom + top) / 2;
     found = key == _elements[mid];
     if (found) {                                       // (5)
       lastIndex = mid;                                 // (6)
     } else {
       if (_elements[mid] < key) {                      // (7)
         bottom = mid + 1;                              // (8)
       } else {
         top = mid - 1; }                               // (9)
     }                                                  // (10) (11)
   }                                                    // (4) (12)
   return found;                                        // (13)
}
```

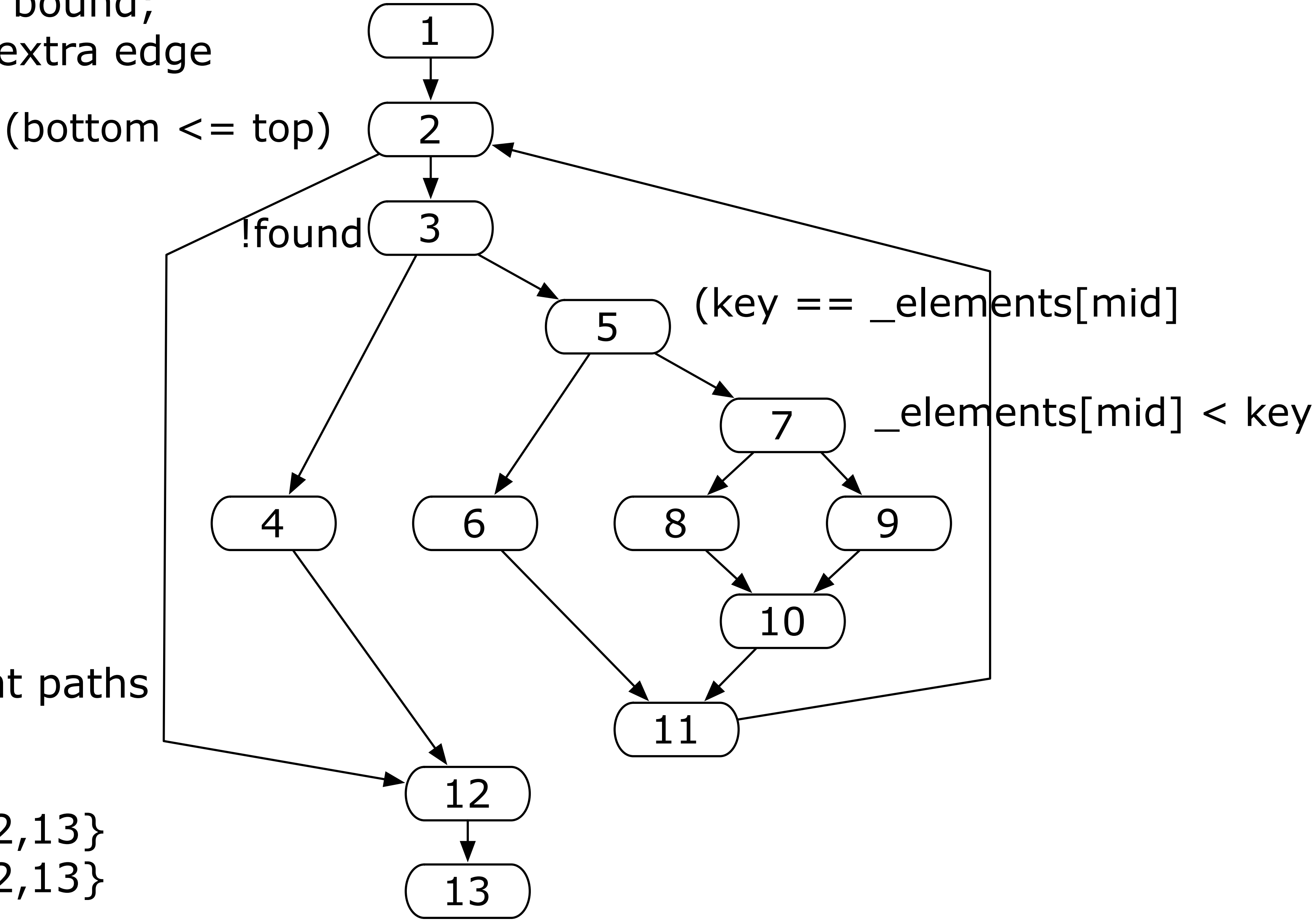
(*) (4) and (12) are needed to close the control flow path because the condition in (2) and (3) must be split up in two primitive conditions: (2) (bottom <= top) and (3) !found.
Remember: A boolean expression involving an "and" or "or" is equivalent to an if statement.

Example - Flow Graph

set of independent paths of a flow graph ⇒ try to cover all the edges in the graph.

Heuristic for constructing such a set

- upper bound for size = 16 - 13 + 2 = 4 + 1 = 5
- pick most simple entry/exit path: {1,2,12,13}
- add new paths until upper bound; each addition includes an extra edge



- possible set of independent paths
  + {1, 2, 3, 4, 12,13}
  + {1,2,3,5,6,11,2,12,13}
  + {1,2,3,5,7,8,10,11,2,12,13}
  + {1,2,3,5,7,9,10,11,2,12,13}

Example - Test Cases

Path	Input	Output
{1,2,12,13}	_elements = []; key = 5	false / index out of bounds
{1,2,3,4,12,13}	_elements = [1, 5, 9]; key = 5	TRUE

{1,2,3,5,6,11,2,12,13}
{1,2,3,*5,7,9,10,11,2,3,*
5,6,11,2,12,13}
	_elements = [1, 5, 9]; key = 1
	actual path is not intended path(*)
		TRUE

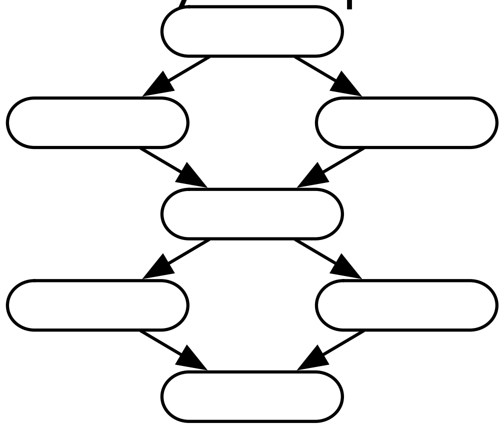
{1,2,3,5,7,8,10,11,2,12,13}	_elements = [5]; key = 9	FALSE
{1,2,3,5,7,9,10,11,2,12,13}	_elements = [5]; key = 1	FALSE

(*) The *intended* path resulting from the heuristic is {1,2,3,5,6,11,2,12,13}.
However, this path can never be forced by any input value.
Therefore the *actual* path is a little different and takes an extra cycle.

Basis Path Testing: Evaluation

- Pros
  - + coverage = (most of the times) every statement + all portions of control flow (branches)
    - * reasonable coverage for reasonable effort
  - + tool support exists (computing cyclomatic complexity + drawing flow graph)
    - * possibility to estimate testing complexity
- Cons
  - + construction is a heuristic: does not necessarily result in set of independent paths
  - + it is possible to get the same coverage with less paths
  - + it is sometimes not feasible to exercise all required paths
  - + it does not necessarily cover all entry-exit paths

```
if (x + y < 3)
   {x := 3} else {x := 5};
if (x + y < 3)
   {y := 3} else {y := 5};
```



- cc = 3 but 4 different entry-exit paths !
- Situation gets worse with nested conditionals

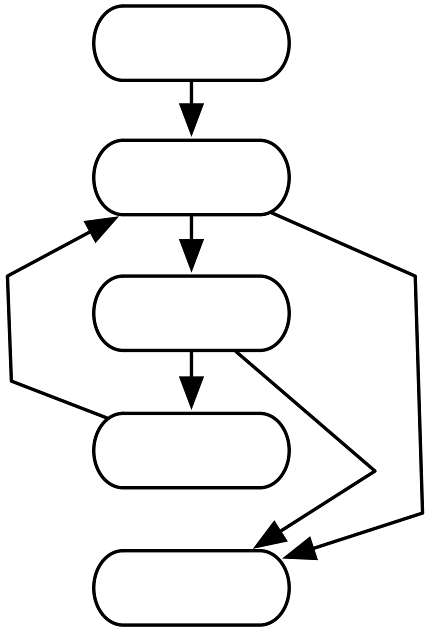
+ not all cc independent paths will cover all statements and all branches (see “Summary”; Perform basis path with a nested conditional of 2 levels deep.)

For crucial code, complement basis path with condition and loop testing

Condition Testing

- For complex boolean expressions, Basis Path Testing is not enough!

```
public void helloWorld (int x, y, z) {
assert((x <> y) && (x <> z));
while (x > y) && (x > z) {
printf(''Hello World'');
x = x - 1;
};
assert((x == y) || (x == z));
}
```



- Input
  + {x = 3, y=4, z = 4}, {x = 4, y=3, z = 3}, {x = 4, y=4, z = 3}
  + exercises all paths ...
    * but several important conditions (assertions) are not triggered (e.g. {x = 3, y=3, z=3})

- Condition Testing
  + *Condition coverage*: all true/false combinations for whole condition expressions
  + *Multiple condition coverage*: all true/false combinations for all simple conditions
  + *Domain Testing*: all combinations of true/false + almost "true/false" for each occurrence of a < b, a <= b, a == b, a <> b, a >= b, 3 tests
    * test cases {a < b; a == b; a > b}

Condition Testing - Test Cases

Condition Coverage
line 2: (x <> y) && (x <> z): {x = 3, y=3, z = 3} and {x = 4, y=3, z = 3}
line 3: (x > y) && (x > z) and line 7: (x == y) || (x == z) are exercised by same values

Multiple Condition Coverage
line 2: {x = 3, y=3, z = 3}, {x = 4, y=3, z = 3}, {x = 4, y=4, z = 3},
{x = 2, y = 3, z = 4}
line 3 and line 7: are exercised by same values

Domain Testing

	x = z	x < z	x > z
x = y	x = 3, y = 3, z = 3	x = 2, y = 2, z = 3	x = 4, y = 4, z = 3
x < y	x = 2, y = 3, z = 2	y = z: x = 2, y = 3, z = 3	y = z: --- not possible
		y < z: x = 2, y = 3, z = 4	y < z: --- not possible
		y > z: x = 2, y = 4, z = 3	y > z: x = 3, y = 4, z = 2
x > y	x = 4, y = 3, z = 4	y = z: --- not possible	y = z: x = 4, y = 3, z = 3
		y < z: x = 3, y = 2, z = 4	y < z: x = 4, y =2, z = 3
		y > z: --- not possible	y > z: x = 4, y = 3, z = 2

Loop Testing

for all loops L, with n allowable passes:

- (i) skip the loop;
- (ii) 1 pass through the loop;
- (iii) 2 passes through the loop;
- (iv) m passes where 2 < m < n;
- (v) n-1, n, n+1 passes

Test cases for binary search: $n = \log_2(\text{size}(\_elements)) = \log_2(16) = 4$

Path	Input	Output (*)
skip the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
1 pass through the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
2 passes through the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
m passes where 2 < m < n	_elements = [1, 3, ..., 29, 31]; key = ...	
n-1	_elements = [1, 3, ..., 29, 31]; key = ...	
n passes	_elements = [1, 3, ..., 29, 31]; key = ...	
n+1 passes	_elements = [1, 3, ..., 29, 31]; key = ...	

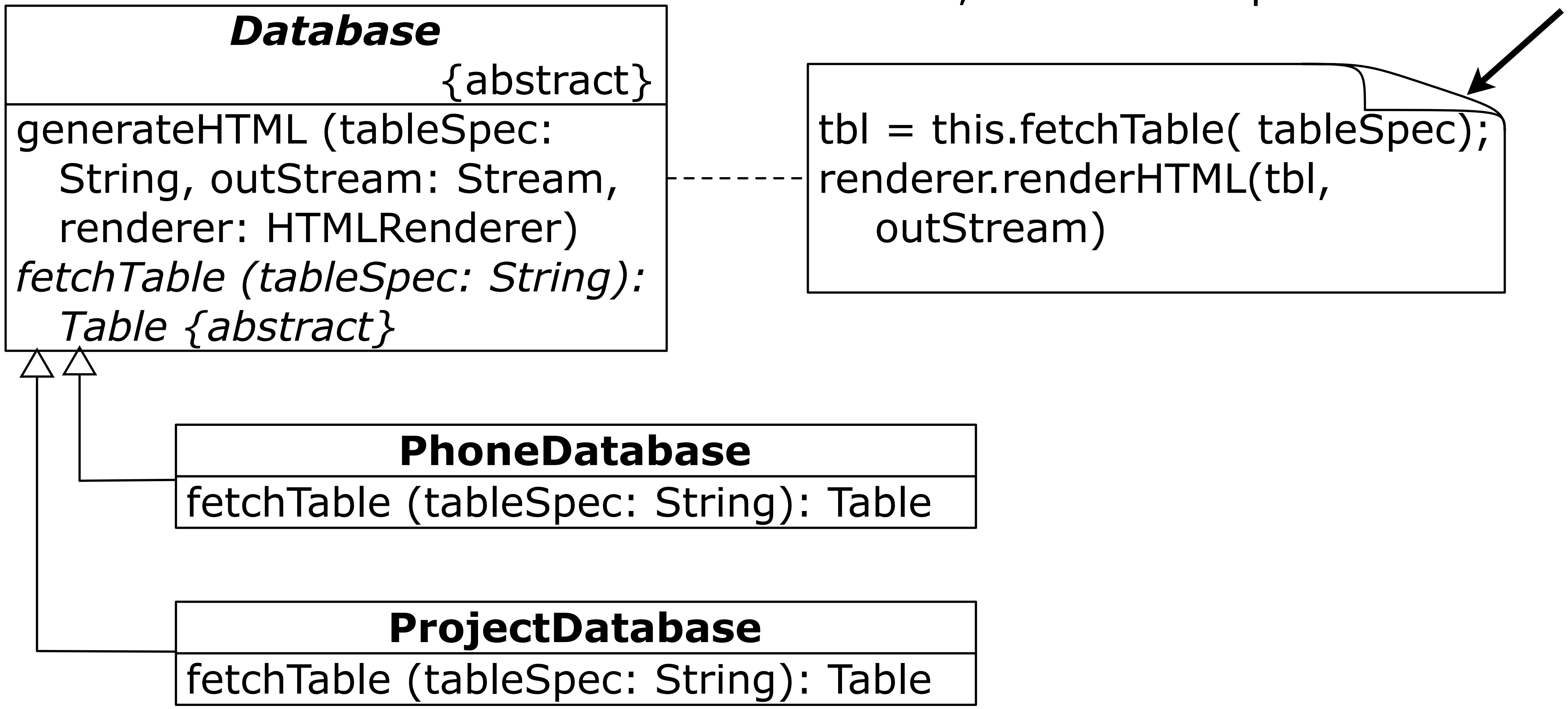
(*) The actual test cases are left as an exercise

White Box Testing and Objects (1/2)

Pure white box testing is less relevant in an object-oriented context.

- Internal structure embedded in object compositions and polymorphic method invocations

Number of paths, conditions, loops = 1
Yet, masks an important conditional



Database
{abstract}

generateHTML (tableSpec: String, outStream: Stream, renderer: HTMLRenderer)
fetchTable (tableSpec: String): Table {abstract}

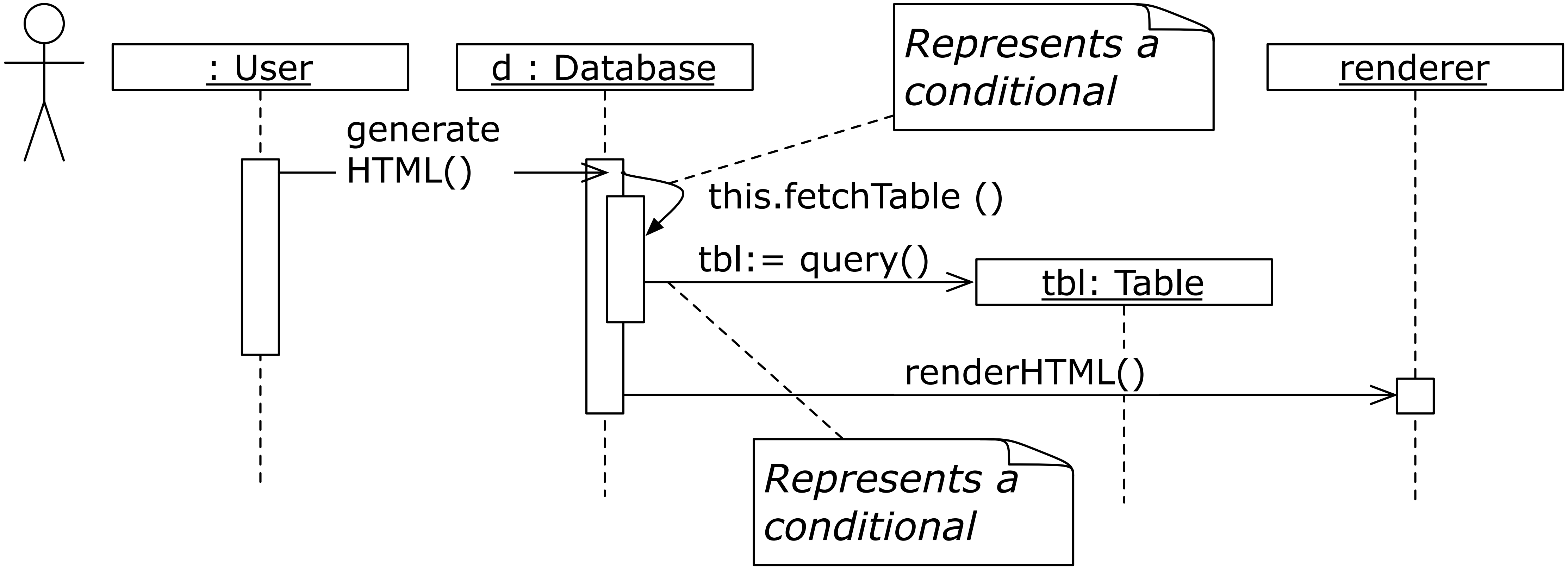
PhoneDatabase
fetchTable (tableSpec: String): Table

ProjectDatabase
fetchTable (tableSpec: String): Table

```
tbl = this.fetchTable( tableSpec);
renderer.renderHTML(tbl,
    outStream)
```

White Box Testing and Objects (2/2)

… but: sequence & collaboration diagrams may serve better



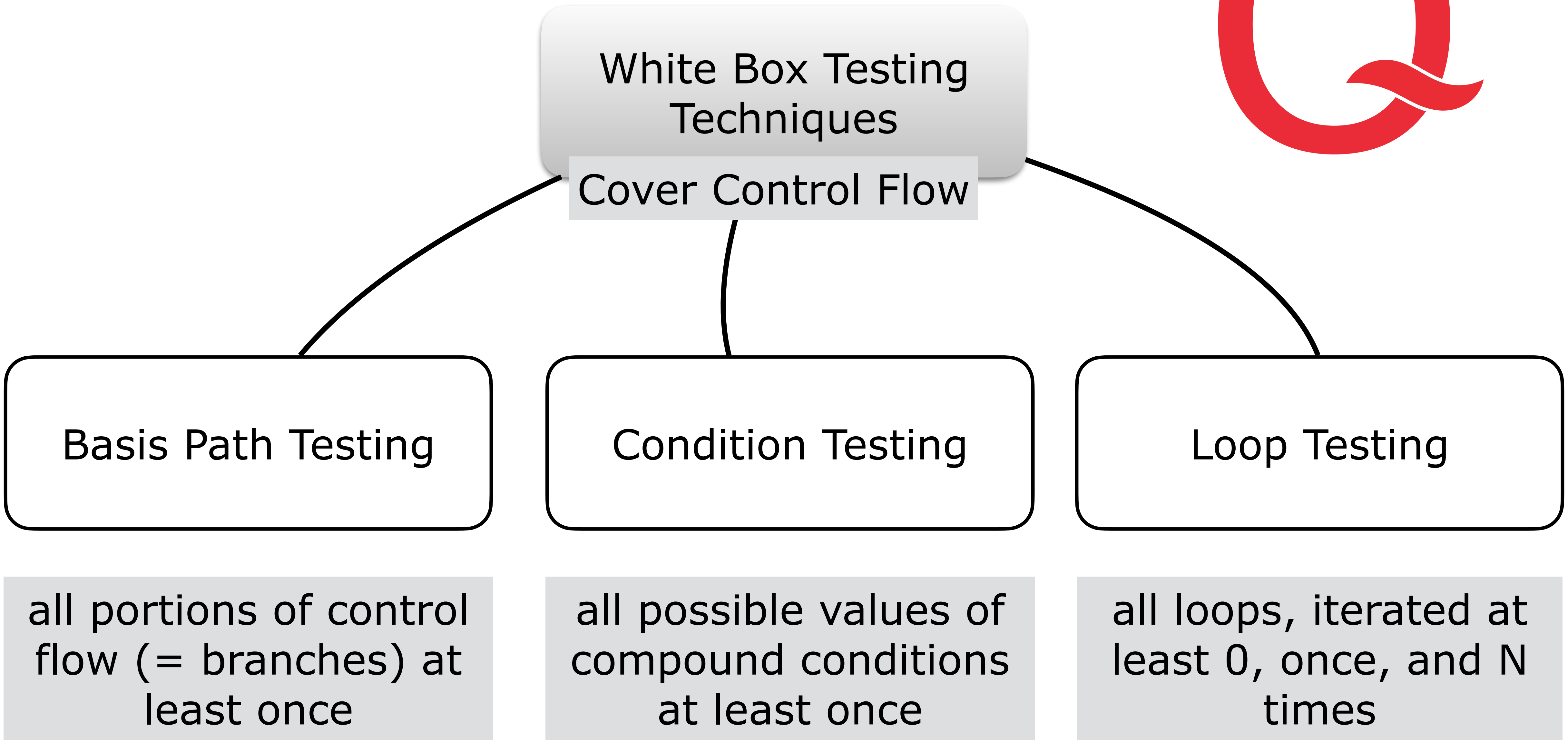
⇒ Identify polymorphic messages representing a conditional

⇒ plug-in instances of appropriate subclasses to exercise branches

The distinction between white-box and black-box testing is not that sharp.

Question

What are the differences and similarities between basis path testing, condition testing and loop testing?



White Box Testing Techniques

Cover Control Flow

Basis Path Testing

Condition Testing

Loop Testing

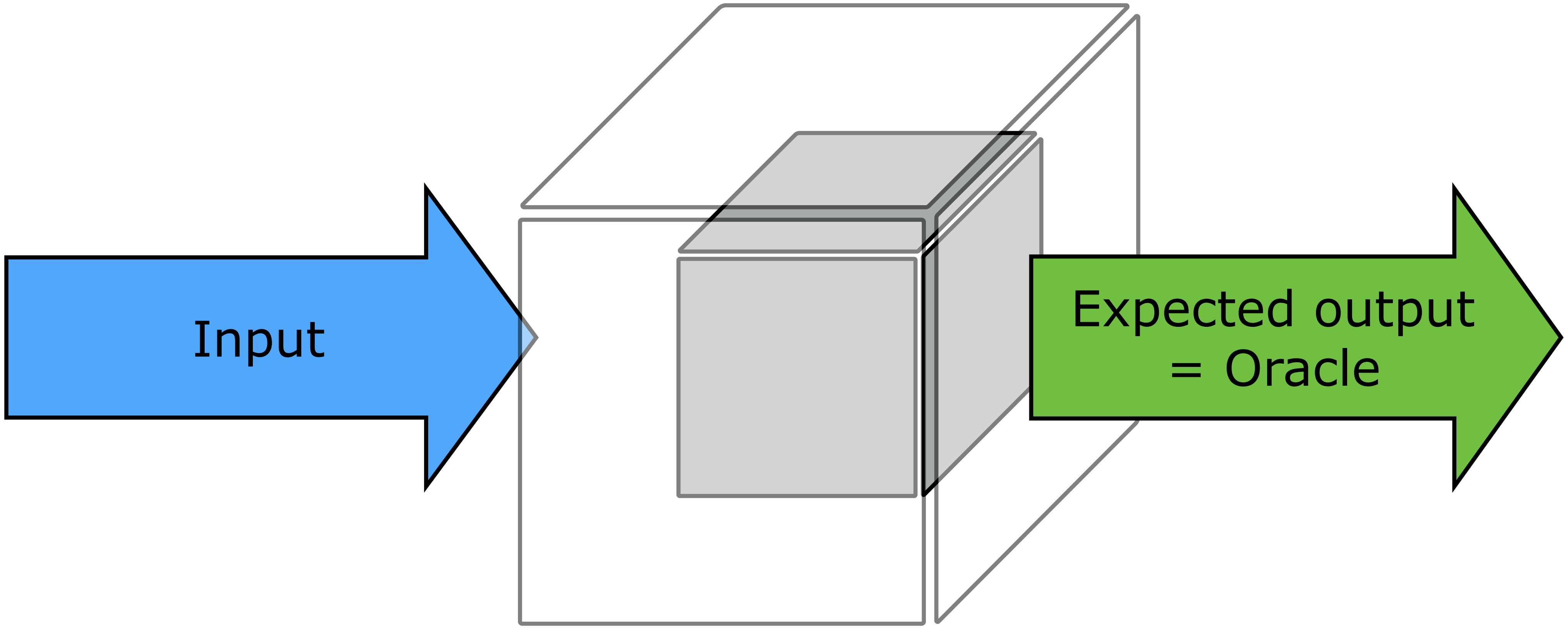
all portions of control flow (= branches) at least once

all possible values of compound conditions at least once

all loops, iterated at least 0, once, and N times

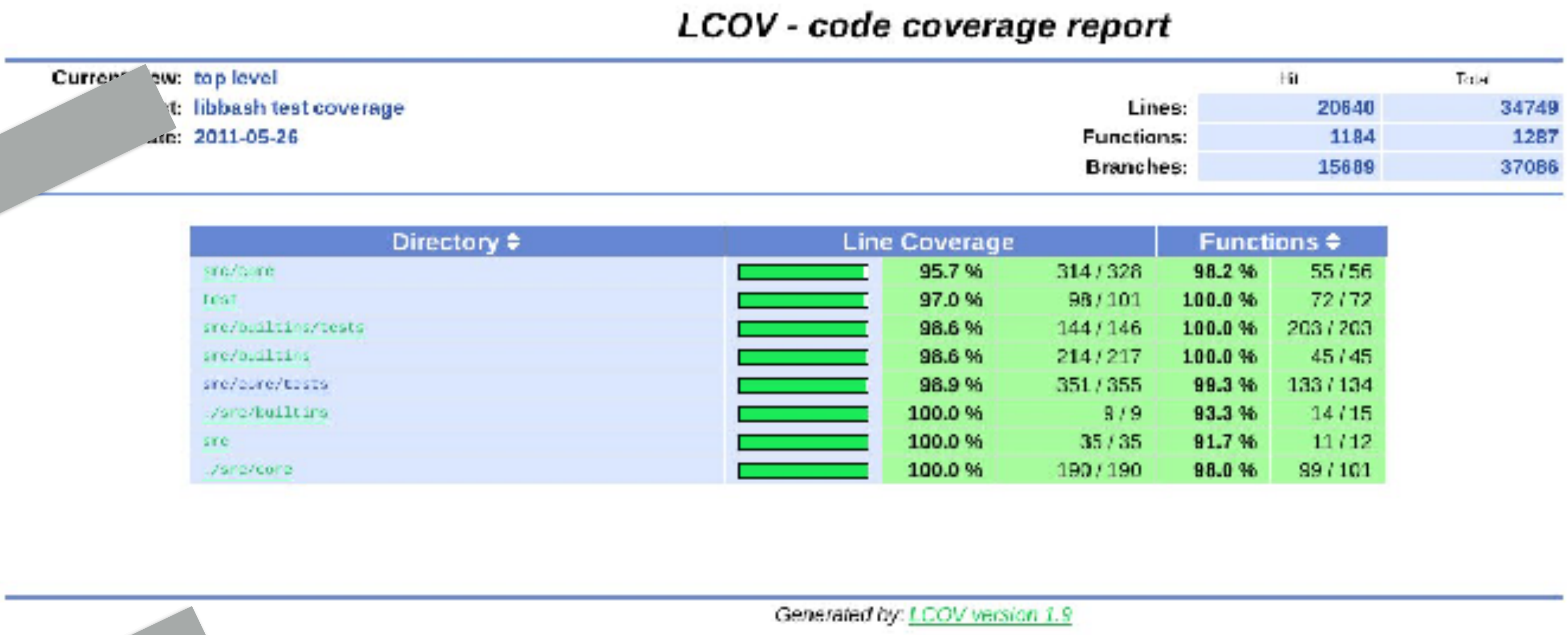
Code Coverage: Strength of a Test Suite

New slide



Code Coverage:
The degree to which code is exercised by a test suite, expressed as a percentage.

Code Coverage



c++

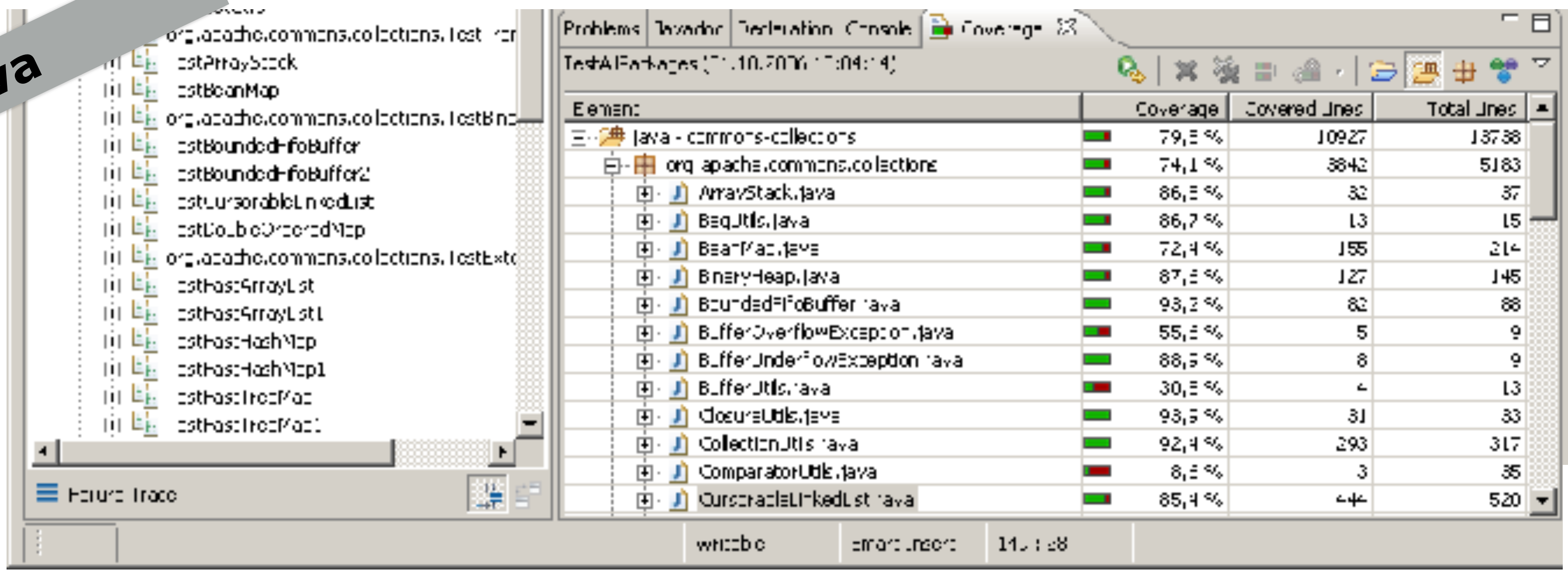
LCOV - code coverage report

Current view: top level
Test: libbash test coverage
Date: 2011-05-26

	Hit	Total
Lines:	20640	34749
Functions:	1184	1287
Branches:	15689	37086

Directory ⇕	Line Coverage			Functions ⇕	
src/core	🟩	95.7 %	314 / 328	98.2 %	55 / 56
test	🟩	97.0 %	98 / 101	100.0 %	72 / 72
src/builtins/tests	🟩	98.6 %	144 / 146	100.0 %	203 / 203
src/builtins	🟩	98.6 %	214 / 217	100.0 %	45 / 45
src/core/tests	🟩	98.9 %	351 / 355	99.3 %	133 / 134
./src/builtins	🟩	100.0 %	9 / 9	93.3 %	14 / 15
src	🟩	100.0 %	35 / 35	91.7 %	11 / 12
./src/core	🟩	100.0 %	190 / 190	98.0 %	99 / 101

Generated by: LCOV version 1.9



java

org.apache.commons.collections.Test[illegible]
TestArrayStack
TestBeanMap
org.apache.commons.collections.TestBin[illegible]
TestBoundedFifoBuffer
TestBoundedFifoBuffer2
TestCursorableLinkedList
TestDoubleOrderedMap
org.apache.commons.collections.TestExt[illegible]
TestFastArrayList
TestFastArrayList1
TestFastHashMap
TestFastHashMap1
TestFastTreeMap
TestFastTreeMap1

Failure Trace

Problems | Javadoc | Declaration | Console | Coverage

TestAllPackages ([illegible].10.2006 [illegible]:04:[illegible])

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Writable | Smart Insert | 14 : 28

Tools to measure line coverage, statement coverage, function coverage, branch coverage readily exist

CAPSTONE PROJECT

Modified Condition/Decision Coverage (MC/DC)

- Condition ≈ Condition on Input to the function/component under test
- Decision ≈ Output of the function/component under test

MC/DC is required by most software standards for safety critical software.
(DO-178C: Avionics Safety Standard; ISO 26262: Road vehicles – Functional safety; ISO/IEC 62304: medical device software)

MC/DC requires all of the below during testing:

- Each entry and exit point is invoked.
- Each decision takes every possible outcome.
- Each condition in a decision takes every possible outcome.
- Each condition in a decision is shown to *independently affect* the outcome of the decision.
  + Independence of a condition is shown by proving that only one condition changes at a time.

MC/DC Example

```
int isReadyToTakeOff(int a, int b, int c, int d) {
 if(((a == 1) ||(b == 1)) && ((c == 1) || (d == 1))) return 1; else return 0;
}
```

2 decisions: “return 1” or “return 0”
4 inputs: a, b, c, d
4 conditions: (a == 1) / (b == 1) / (c == 1) / (d == 1)

Decision Coverage
- 2 test cases, one for each decision

Condition Coverage
- 2 test cases, one for all conditions to be true, one for all conditions to be false

Condition/Decision Coverage
- 3 test cases, all decisions at least once + all conditions once true, once false

Modified Condition/Decision Coverage
- n + 1 test cases (for a decision with n conditions)

Multiple Condition Coverage
- 2^n test cases (for a decision with n conditions)
  + Usually too large to handle

Condition/Decision Coverage

a==1	**b==1**	**c==1**	**d==1**	**decision**
FALSE	TRUE	TRUE	TRUE	return 1
FALSE	FALSE	FALSE	TRUE	return 0
TRUE	FALSE	FALSE	FALSE	return 0

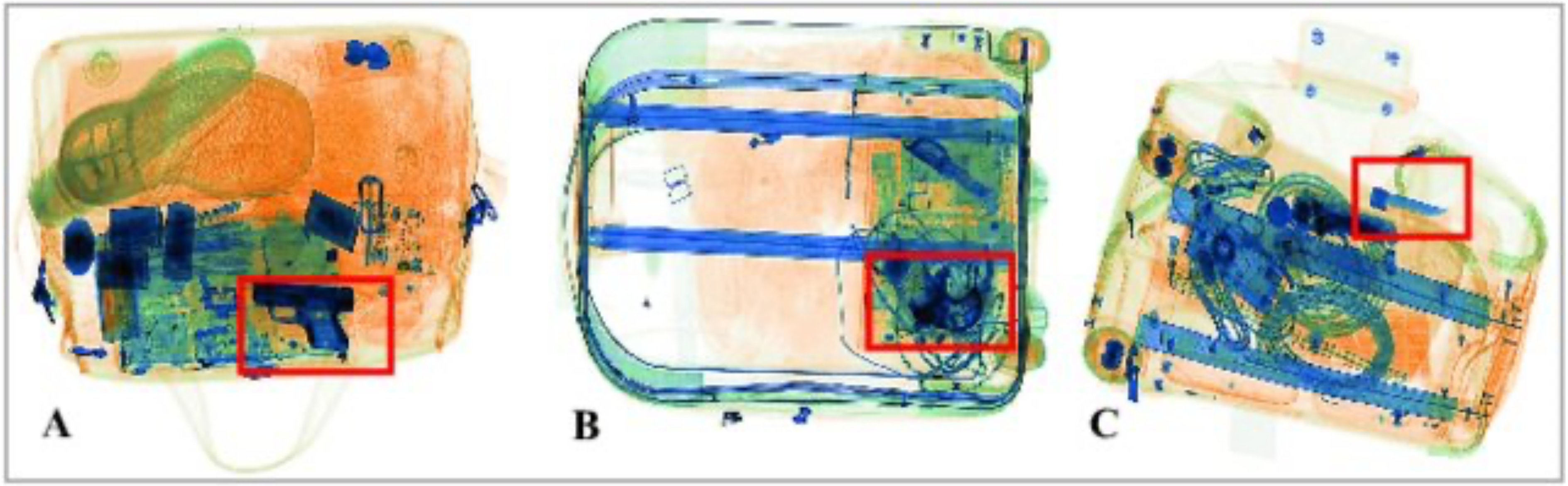
Modified Condition/Decision Coverage

a==1	**b==1**	**c==1**	**d==1**	**decision**	
TRUE	FALSE	TRUE	FALSE	return 1	+ row 4 shows effect of c
TRUE	FALSE	FALSE	TRUE	return 1	+ row 5 shows effect of a
FALSE	TRUE	FALSE	TRUE	return 1	+ row 5 shows effect of b
TRUE	FALSE	FALSE	FALSE	return 0	+ row 2 shows effect of d
FALSE	FALSE	FALSE	TRUE	return 0	

Mutation Testing: Metaphor

© Brussels Airlines

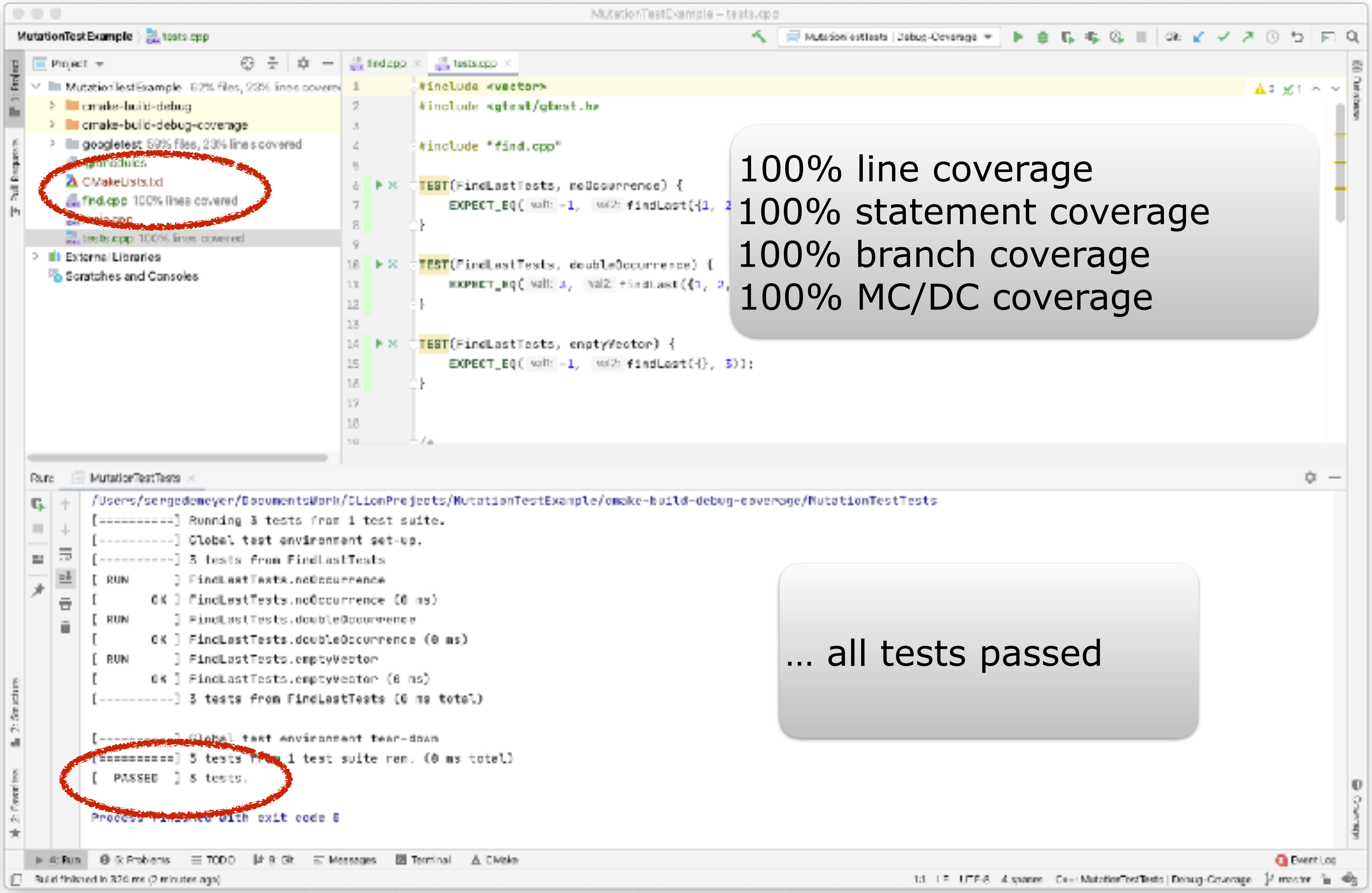
How to test the quality assurance?
Inject synthetic problematic items.



© "The Good, the Bad and the Ugly: Evaluating Convolutional Neural Networks for Prohibited Item Detection Using Real and Synthetically Composite X-ray Imagery" Neelanjan Bhowmik, Qian Wang, Yona Falinie A. Gaus, Marcin Szarek, Toby P. Breckon

Code Coverage

gTest Example: findLast



MutationTestExample – tests.cpp
MutationTestExample
tests.cpp
Project
MutationTestExample 67% files, 23% lines covered
cmake-build-debug
cmake-build-debug-coverage
googletest 59% files, 23% lines covered
gmodules
CMakeLists.txt
find.cpp 100% lines covered
tests.cpp 100% lines covered
External Libraries
Scratches and Consoles
find.cpp
tests.cpp
Run: MutationTestTests
4: Run
6: Problems
TODO
9: Git
Messages
Terminal
CMake
Event Log
Build finished in 324 ms (2 minutes ago)

```
#include <vector>
#include <gtest/gtest.h>

#include "find.cpp"

TEST(FindLastTests, noOccurrence) {
    EXPECT_EQ( val1: -1,  val2: findLast({1, 2
}

TEST(FindLastTests, doubleOccurrence) {
    EXPECT_EQ( val1: 3,  val2: findLast({1, 2,
}

TEST(FindLastTests, emptyVector) {
    EXPECT_EQ( val1: -1,  val2: findLast({}, 5));
}
```

```
/Users/sergedemeyer/DocumentsWork/CLionProjects/MutationTestExample/cmake-build-debug-coverage/MutationTestTests
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from FindLastTests
[ RUN      ] FindLastTests.noOccurrence
[       OK ] FindLastTests.noOccurrence (0 ms)
[ RUN      ] FindLastTests.doubleOccurrence
[       OK ] FindLastTests.doubleOccurrence (0 ms)
[ RUN      ] FindLastTests.emptyVector
[       OK ] FindLastTests.emptyVector (0 ms)
[----------] 3 tests from FindLastTests (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.

Process finished with exit code 0
```

100% line coverage
100% statement coverage
100% branch coverage
100% MC/DC coverage

… all tests passed

Inject Mutant (Survived - Live)

```
int findLast(std::vector<int> x, int y) {
    if (x.size() == 0)
        return -1;
    for (int i = x.size() - 1; i > 0; i--)
        if (x[i] == y)
            return i;
    return -1;
}
```

Relational Operator Replacement (ROR)
"i >= 0" becomes "i > 0"

```
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
```

⇒ One of these tests should fail!

But all of them pass: the mutant survives.

Extra test *kills* the mutant

Find.cpp

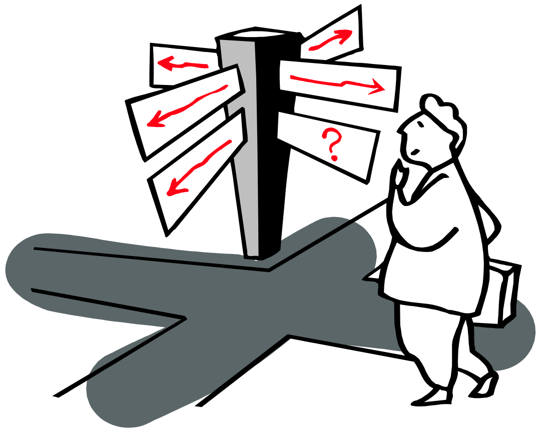
```
#include <vector>

int findLast(std::vector<int> x, int y) {
    if (x.size() == 0)
        return -1;
    for (int i = x.size() - 1; i >= 0; i–)
        if (x[i] == y)
            return i;
    return -1;
}
```

Tests.cpp

```
#include <vector>
#include <gtest/gtest.h>

#include "find.cpp"

TEST(FindLastTests, noOccurrence) {
    EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
}

TEST(FindLastTests, doubleOccurrence) {
    EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
}

TEST(FindLastTests, emptyVector) {
    EXPECT_EQ(-1, findLast({}, 3));
}
```

```
TEST(FindLastTests, occurrenceOnBoundary) {
    EXPECT_EQ(0, findLast({1, 2, 42, 42, 63}, 1));
}
```

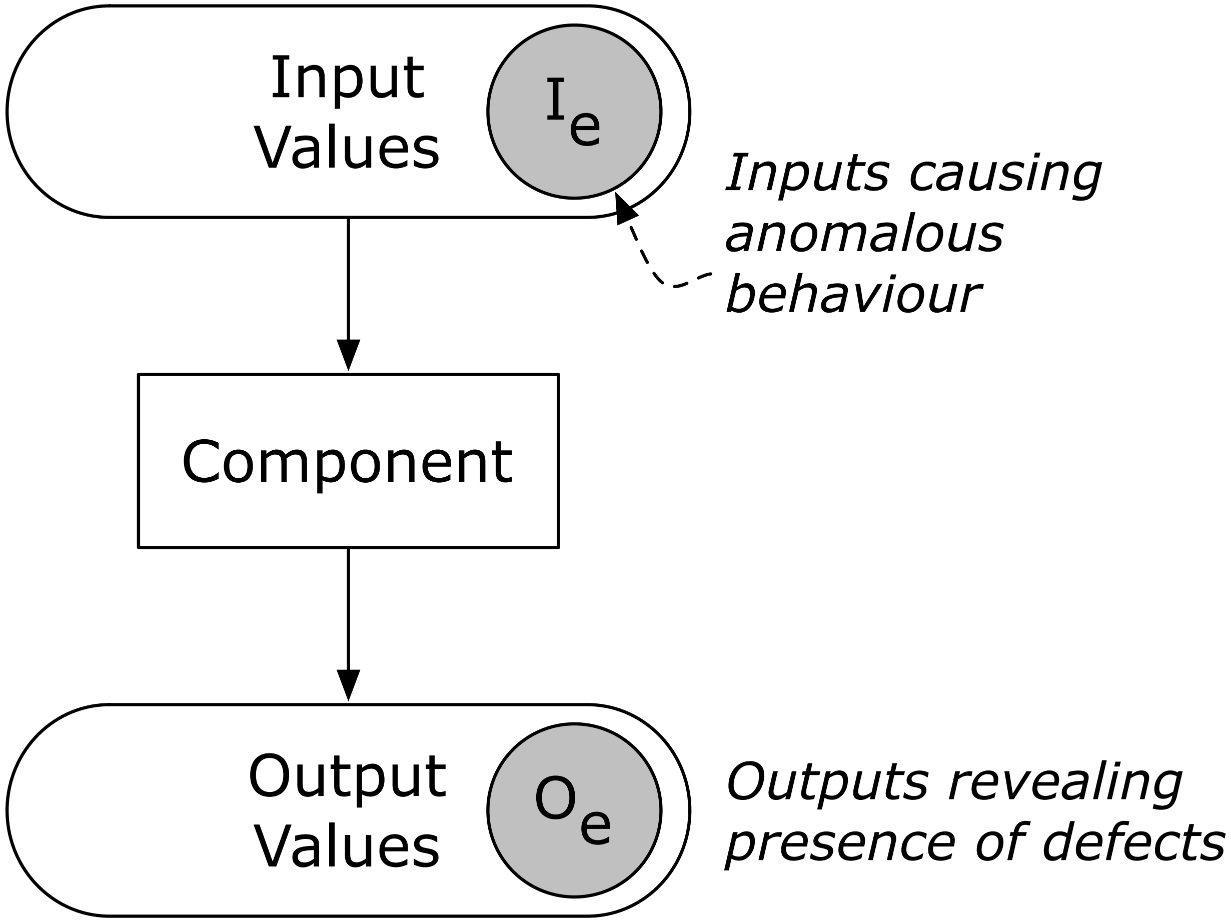
CHAPTER 5 – Testing



- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz testing
- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

Black Box Testing

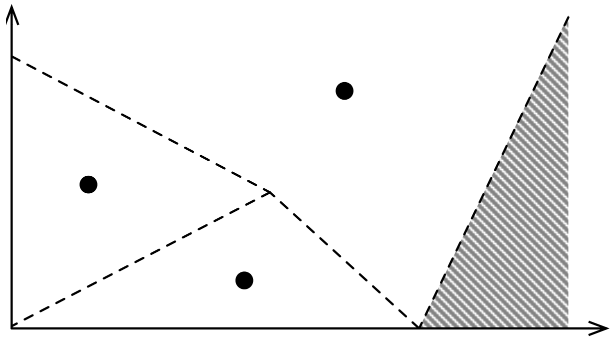
- a.k.a. Functional testing, Testing in the large
  - + Treat a component as a a “black box” whose behaviour can be determined only by studying its inputs and outputs.
  - + Test cases are derived from the external specification of the component
  - + Derive test cases to maximize coverage of elements in the spec, yet minimize number of test cases
  - + Coverage criteria
    ⇒ all exceptions


Input Values
I_e
Inputs causing anomalous behaviour
Component
Output Values
O_e
Outputs revealing presence of defects

Equivalence Partitioning

& Boundary Value Analysis

- 1. Divide input domain in classes of data, according to input condition.
  Input condition may require:
  + a range ⇒ 1 valid (in the range) and 2 invalid equivalence classes
  + a value ⇒ 1 valid (= value) and 2 invalid equivalence classes
  + a set ⇒ 1 valid (in the set) and 1 invalid equivalence class
  + a boolean ⇒ 1 valid and 1 invalid equivalence class



- 2. Choose test data corresponding to each equivalence class
  + Normal equivalence partitioning chooses test data at random
  + Boundary Value Analysis choose values at the “edge” of the class, e.g., just above and just below the minimum and maximum of a range
- 3. Predict the corresponding output and derive test case
- 4. Write test driver

You can partition the output domain as well and apply the same technique

Equivalence Partitioning : Example

- Example: Binary search

```
private int[] _elements;
public boolean find(int key) { ... }
•pre-condition(s)
    - Array has at least one element
    - Array is sorted
•post-condition(s)
    (The element is in _elements and the result is true)
    or (The element is not in _elements and the result is false)
```

- Check input partitions:
  + Do the inputs satisfy the pre-conditions?
  + Is the key in the array?
    * leads to (at least) 2x2 equivalence classes

- Check boundary conditions
  + Is the array of length 1?
  + Is the key at the start or end of the array?
    * leads to further subdivisions
    (not all combinations make sense)

Equivalence Partitioning: Test Data

Generate test data that cover all meaningful equivalence partitions.

Test Cases	Input	Output
Array length 0	key = 17, elements = { }	FALSE
Array not sorted	key = 17, elements = { 33, 20, 17, 18 }	exception
Array size 1, key in array	key = 17, elements = { 17 }	TRUE
Array size 1, key not in array	key = 0, elements = { 17 }	FALSE
Array size > 1, key is first element	key = 17, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key is last element	key = 33, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key is in middle	key = 20, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key not in array	key = 50, elements = { 17, 18, 20, 33 }	FALSE
…	…	…

Design by Contract — Tests

+ Pre- and post-conditions are part of the interface of a component.
  - Part of black-box testing, not white-box testing
    > Do *not* include assertions in basis-path testing
    > Borderline case: include assertions in condition testing

+ Example

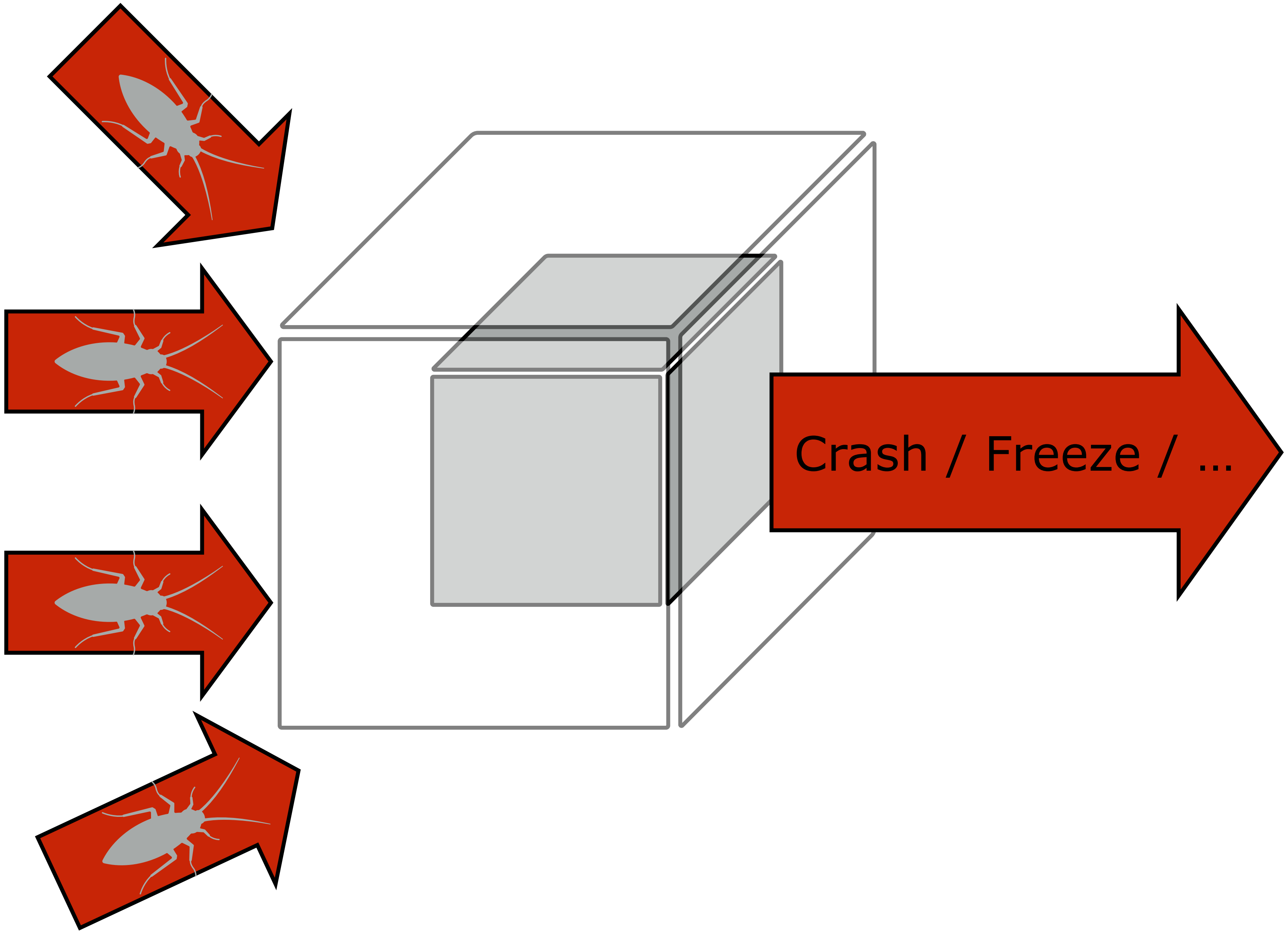
```
public char pop() throws AssertionException {
    assert(!this.isEmpty());
    return _store[_size--];
}
```

+ Equivalence partition with boolean
  - = condition testing: 2 inputs cover all conditions
  - (test case 1 = non-empty stack / value on the top
  - (test case 2 = empty stack / assertion exception)

Repeat from (Chapter 5. Design by Contract)

Fuzz Testing

New slide



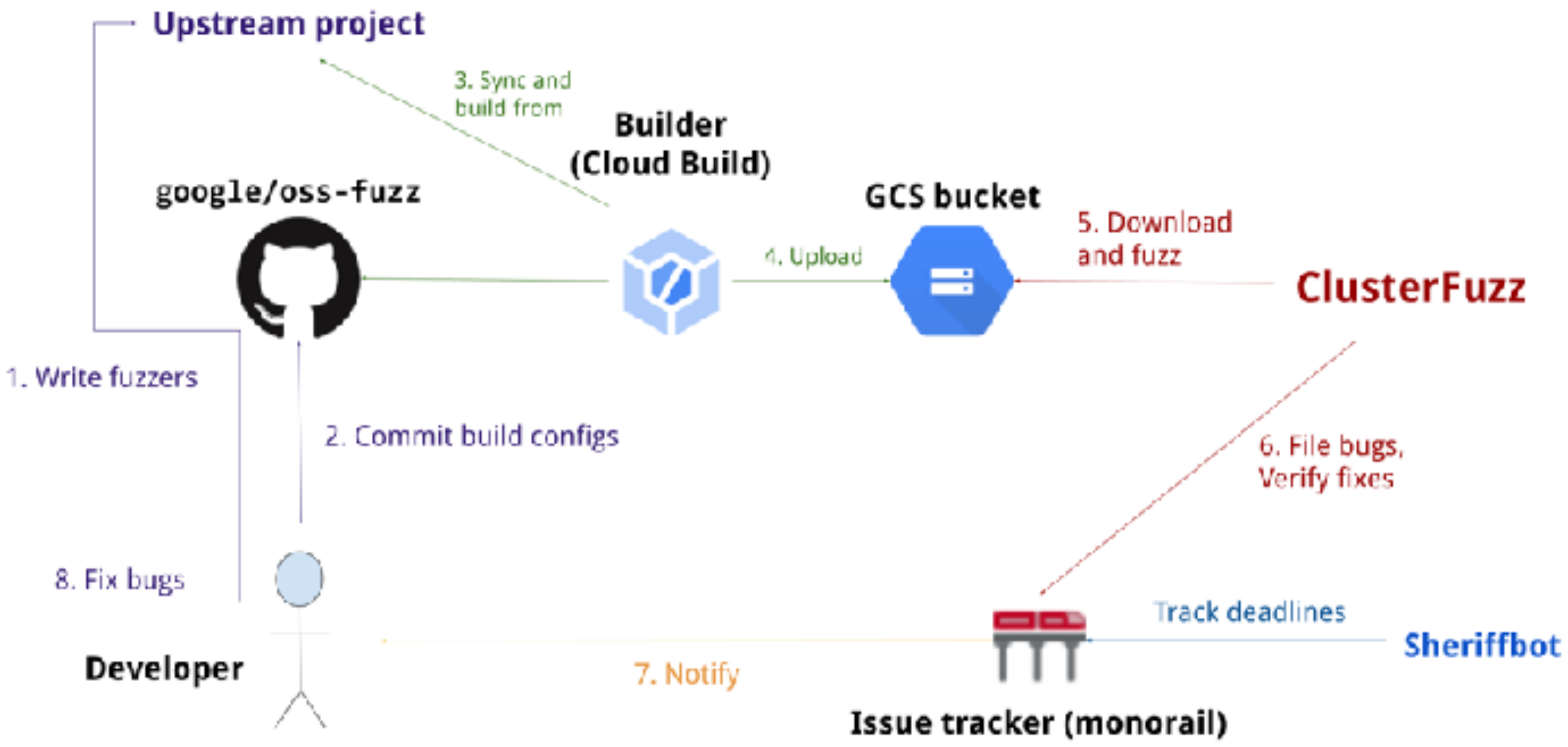
Fuzz Testing:
A software testing technique used to discover security vulnerabilities by inputting massive amounts of random data, called fuzz, to the component or system.

Fuzz-Testing: Open Source Libraries

New slide

OSS-Fuzz: Continuous Fuzzing for Open Source Software

https://github.com/google/oss-fuzz



Trophies

As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects.

Fuzz-Testing: REST-API

New slide

RESTler: first stateful REST API fuzzing tool

https://github.com/microsoft/restler-fu

CAPSTONE PROJECT

- **Use-after-free rule.** A resource that has been deleted must no longer be accessible.
- **Resource-hierarchy rule.** A child resource of a parent resource must not be accessible from another parent resource.
- …

In an Azure service, we found the following use-after-free violation.

1) Create a new resource R (with a PUT request).
2) Delete resource R (with a DELETE request).
3) Create a new child resource of the deleted resource R

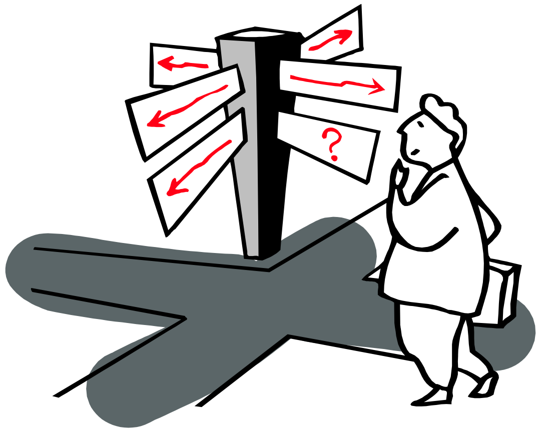
and of a specific type (with another PUT request).
This sequence of requests results in a “500 Internal Server Error”.

In an Office365 messaging service where users can post messages and then reply and edit these, the resource-hierarchy checker detected the following bug.

1) Create a first message msg-1 (with a request POST /api/posts/msg-1).
2) Create a second message msg-2 (with a request POST /api/posts/msg-2).
3) Create a reply reply-1 to the first message
   (with a request POST /api/posts/msg-1/replies/reply-1).
4) Edit the reply reply-1 with a PUT request using msg-2 as message identifier
   (with a request PUT /api/posts/msg-2/replies/reply-1).

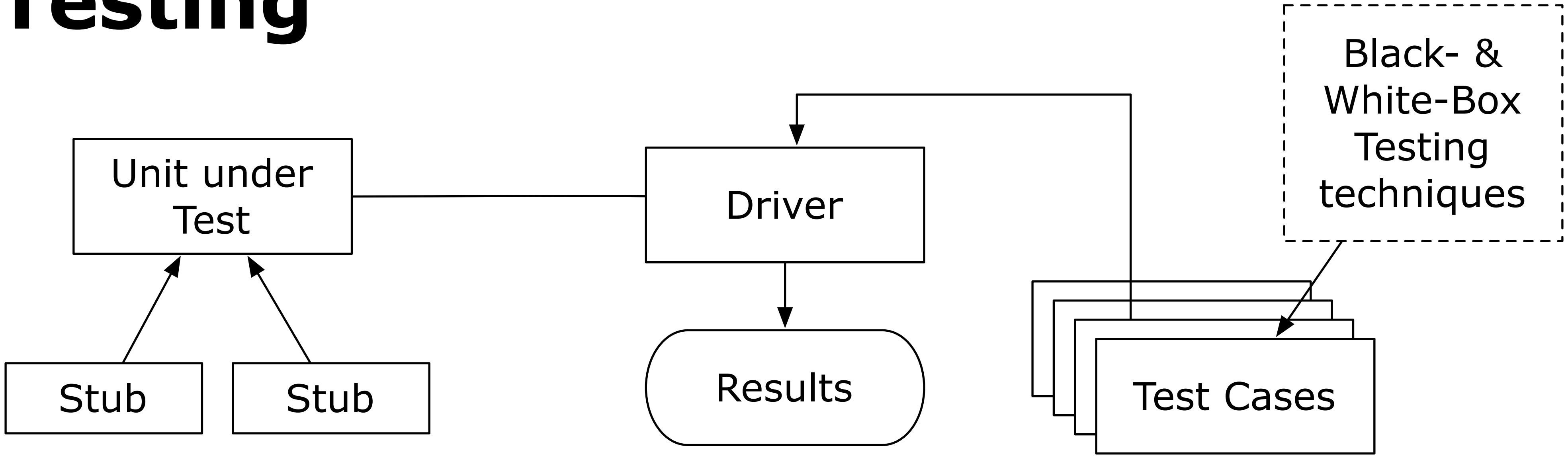
Surprisingly, the last request in Step 4 returns a “200 Allowed” response while it must have returned a “404 Not Found” response.

CHAPTER 5 – Testing



- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is “Correct”?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz Testing
- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

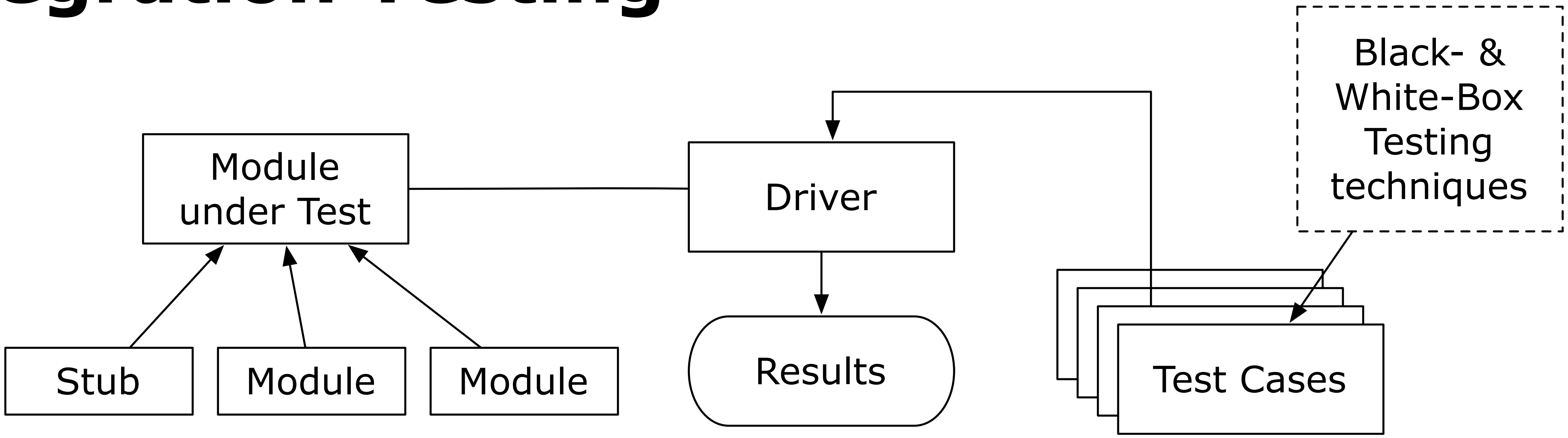
Unit Testing



+ Why?
    - Identify local defects (= within a unit) fast
+ Who?
    - Person developing the unit writes the tests.
+ When? At the latest when a unit is delivered to the rest of the team
    - No test ⇒ no unit
    - Test drivers & stubs are part of the system ⇒ configuration management
    - Today fully automated

- *** Write the test first,
    + i.e. before writing the unit.
    + It will encourage you to design the component interface right

Integration Testing



+ Why?
  - The sum is more then its parts,
    i.e. interfaces (and calls to them) may contain defects too.
+ Who?
  - Person developing the module writes the tests.
+ When?
  - Top-down: main module before constituting modules
  - Bottom-up: constituting modules before integrated module
  - In practice: a little bit of both

- ## The distinction between unit testing and integration testing is not that sharp!

Regression Testing

Regression Testing ensures that all things that used to work still work after changes.

- Regression Test
  + = re-execution of some subset of tests to ensure that changes have not caused unintended side effects
  + tests must avoid regression (= degradation of results)
  + Regression tests must be repeated often (after every change, every night, with each new unit, with each fix,...)
  + Regression tests may be conducted manually
    - Execution of crucial scenarios with verification of results
    - Manual test process is slow and cumbersome
      * preferably completely automated

- Advantages
  + Helps during iterative and incremental development
    + during maintenance
- Disadvantage
  + Up front investment in maintainability is difficult to sell to the customer

Acceptance Testing

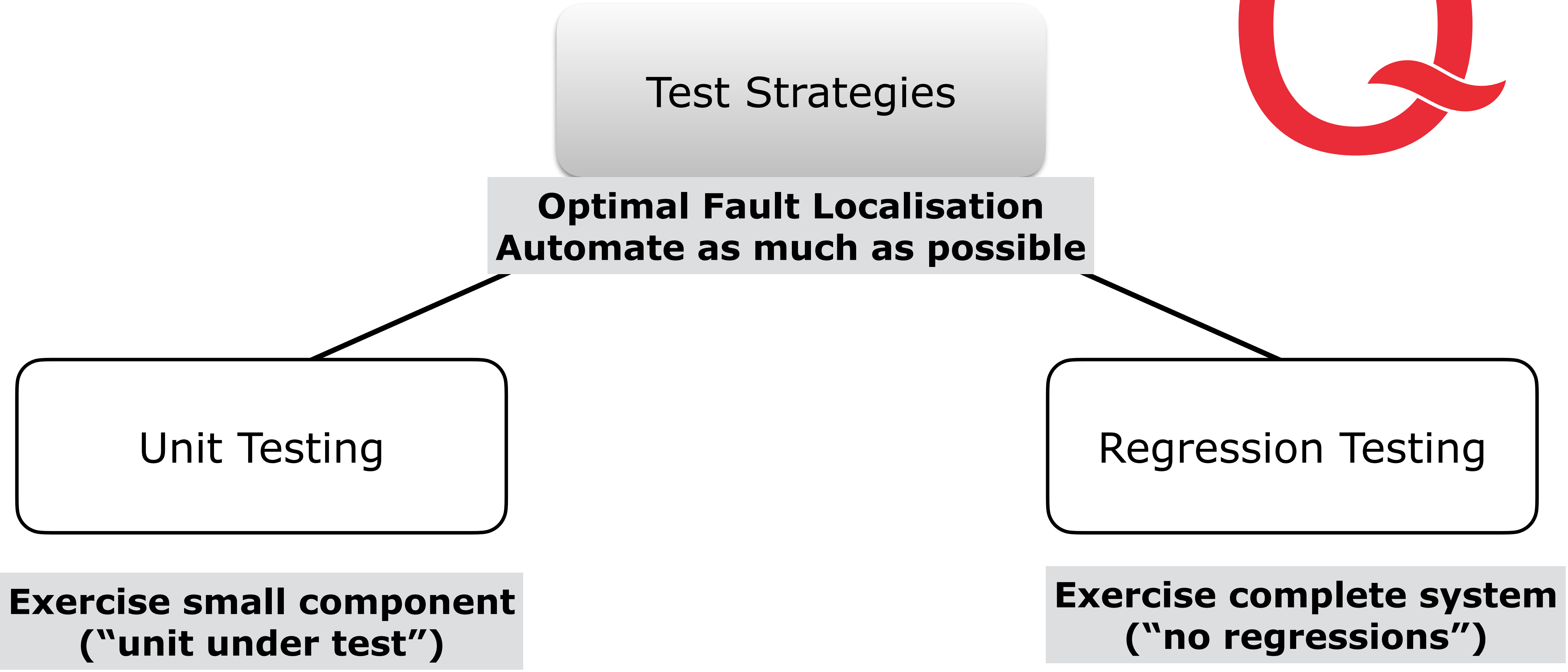
- Acceptance Tests
  + conducted by the end-user (representatives)
  + check whether requirements are correctly implemented
    - borderline between verification (“Are we building the system right?”) and validation (“Are we building the right system?”)

- Alpha- & Beta Tests
  + acceptance tests for “off-the-shelves” software (many unidentified users)
    - Alpha Testing
      > end-users are invited at the developer’s site
      > testing is done in a controlled environment
    - Beta Testing
      > software is released to selected customers
      > testing is done in “real world” setting, without developers present

Question

What are the differences and similarities between unit testing and regression testing?

New slide



Test Strategies

Optimal Fault Localisation
Automate as much as possible

Unit Testing

Exercise small component
(“unit under test”)

Regression Testing

Exercise complete system
(“no regressions”)

More Testing Strategies

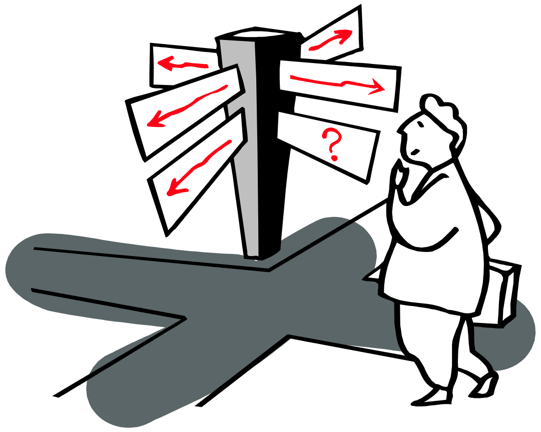
- Recovery Testing / Resilience Testing
  + Test forces system to fail and checks whether it recovers properly
    - For fault tolerant systems

- Stress Testing (Overload Testing)
  + Tests extreme conditions
    - e.g., supply input data twice as fast and check whether system fails

- Performance Testing
  + Tests run-time performance of system
    - e.g., time consumption, memory consumption
      > first do it, then do it right, then do it fast

- Back-to-Back Testing
  + Compare test results from two different versions of the system
    - requires N-version programming or prototypes

CHAPTER 5 – Testing



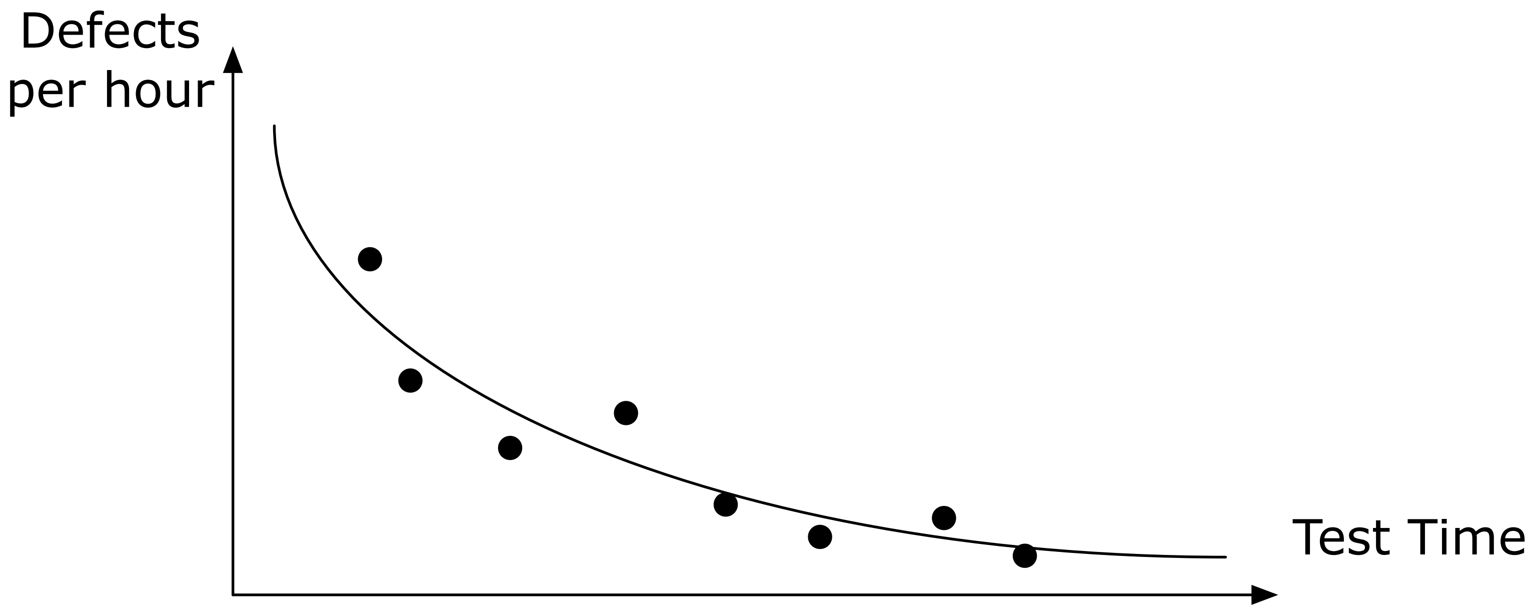
- Introduction
  - + When, Why, What & Who?
    - - The V-Model
  - + What is “Correct”?
  - + Terminology
- Testing Techniques
  - + White Box
    - - basis path, conditions, loops
  - + Coverage
    - - Code Coverage
    - - MC/DC Coverage
    - - Mutation Coverage
  - + Black Box
    - - equivalence partitioning
  - + Fuzz Testing
- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

When to Stop?

When are we done testing? When do we have enough tests?

- Cynical Answers (sad but true)
  - + You’re never done: each run of the system is a new test
    - > Each bug-fix should be accompanied by a new test
  - + You’re done when you are out of time/money
    - > Include test in project plan
      AND DO NOT GIVE IN TO PRESSURE
    - > ... in the long run, tests SAVE time

- Statistical Testing
  - + Test until you’ve reduced failure rate under risk threshold
    - * Testing is like an insurance company calculating risks



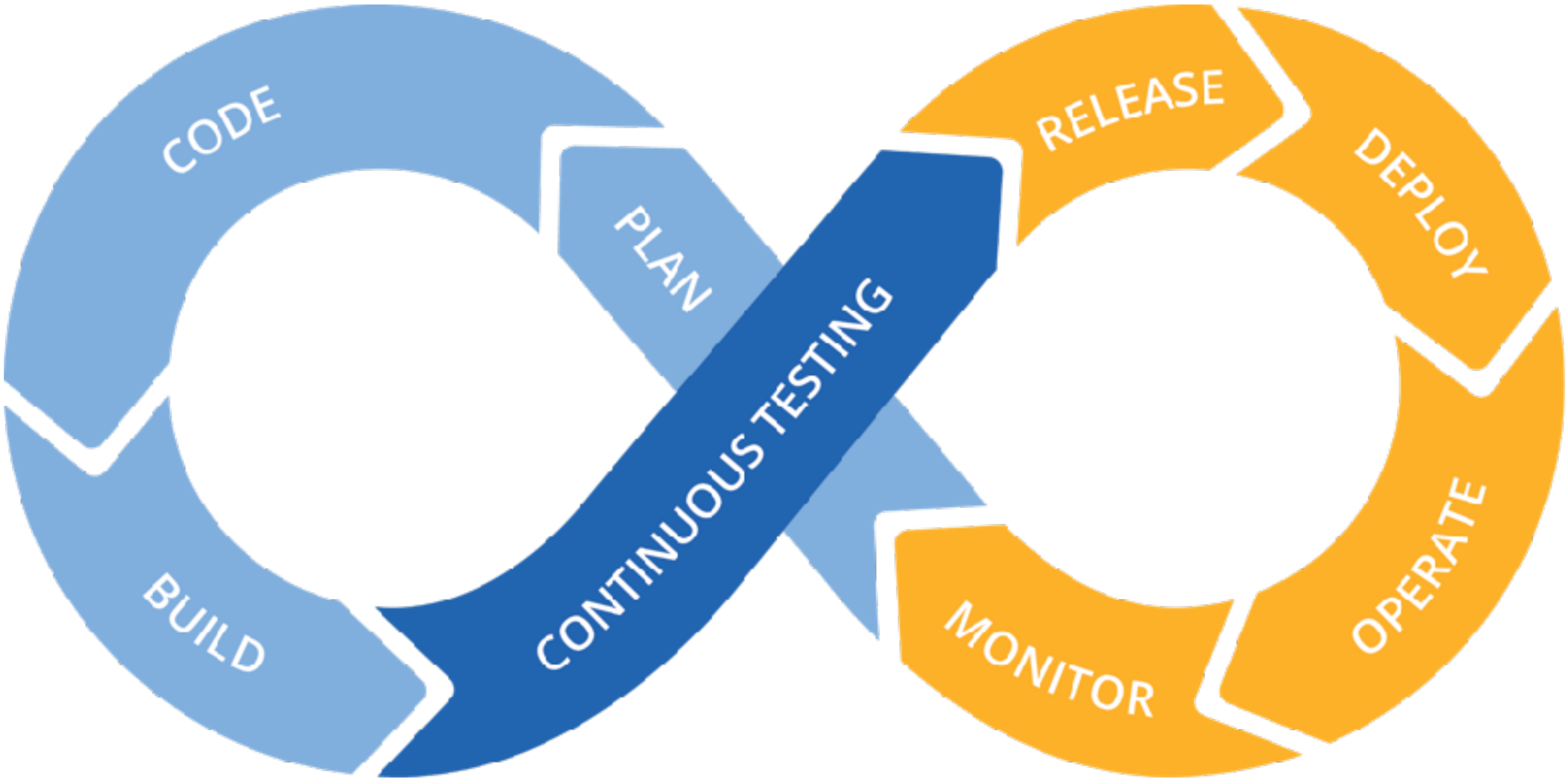
Tool Support for Testing

- Test Harness
  - + Deterministic tests without any user intervention
    - - all input is generated by stubs/all output is absorbed by stubs
    - - input/output behaviour is entirely predictable
  - + A test-case is a predicate taking one parameter; an output stream
    - - Answers true (component passed test successfully) or false (component did not pass the test + report on the output stream)
    - - For each change in requirements, for each bug report
      - > Adapt test cases
        - * Takes a lot of work: more test code than production code

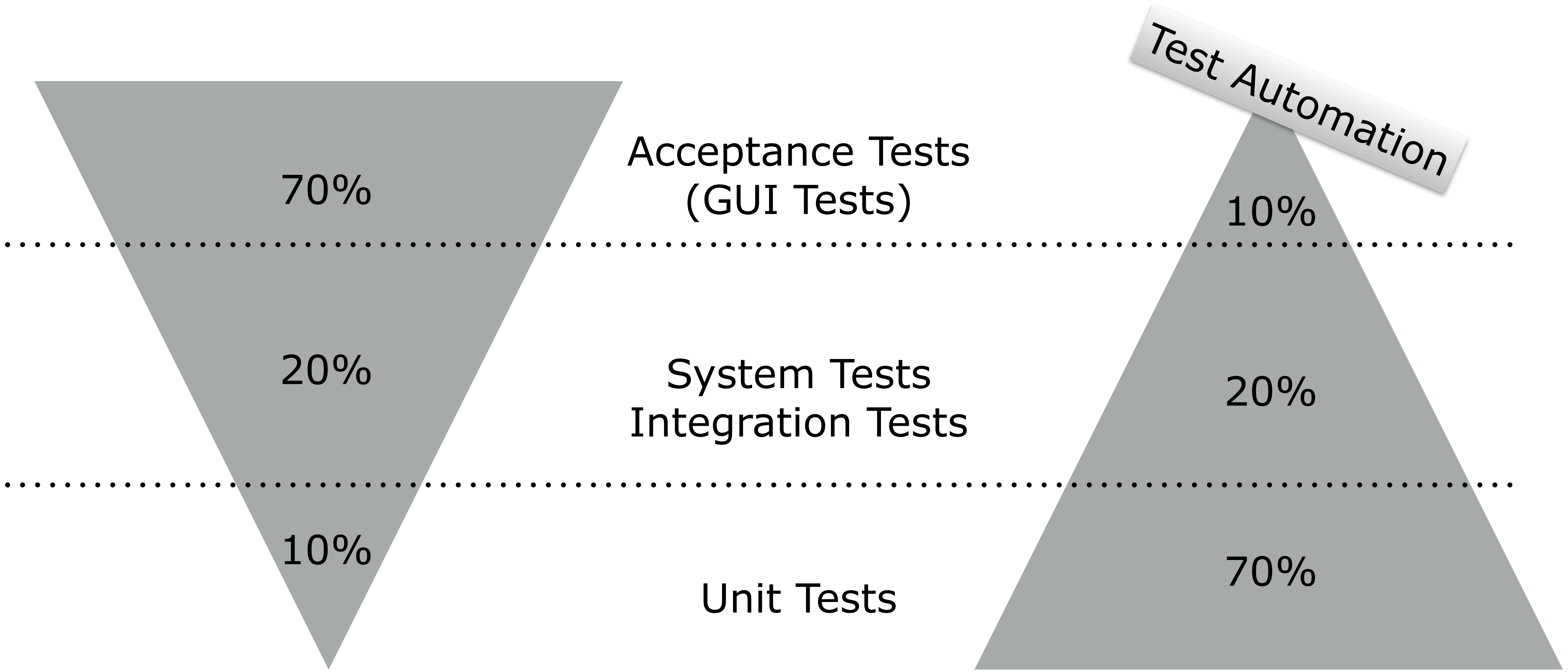
- Code coverage tools
  - + Instrument code to see which parts are (not) executed by a test suite
    - - More coverage ≠ revealing more defects
  - + Mutation coverage
    - - Systematically inject faults and execute test suite

- Capture-playback tools
  - + A tool records all UI-actions and their results
  - + Possibility to replay recordings and verify results
        - * Vulnerable to modifications in UI

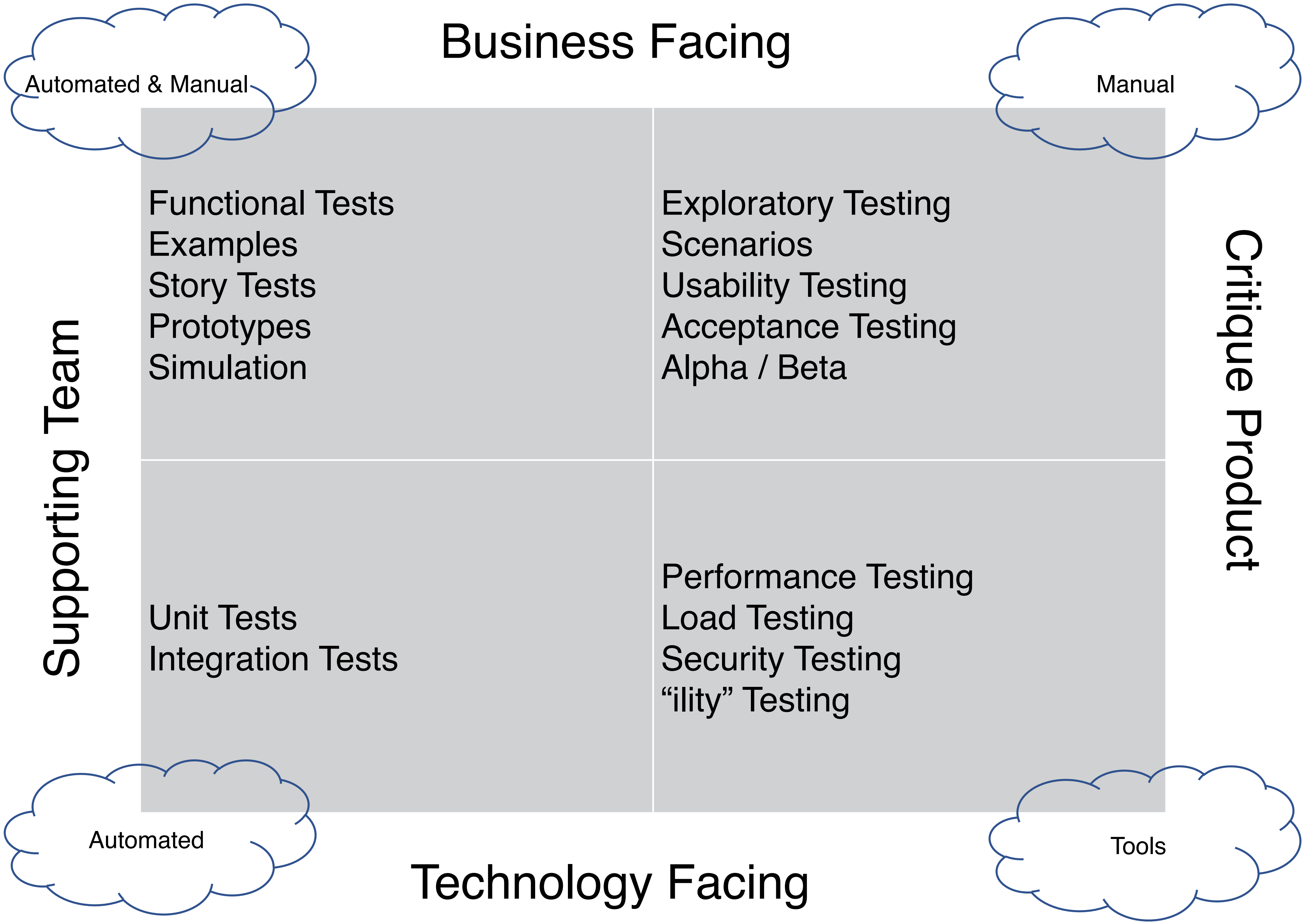
Agile Testing (DevOps)



Flipping the V



4 Quadrants



Business Facing

Technology Facing

Supporting Team

Critique Product

Automated & Manual

Functional Tests
Examples
Story Tests
Prototypes
Simulation

Manual

Exploratory Testing
Scenarios
Usability Testing
Acceptance Testing
Alpha / Beta

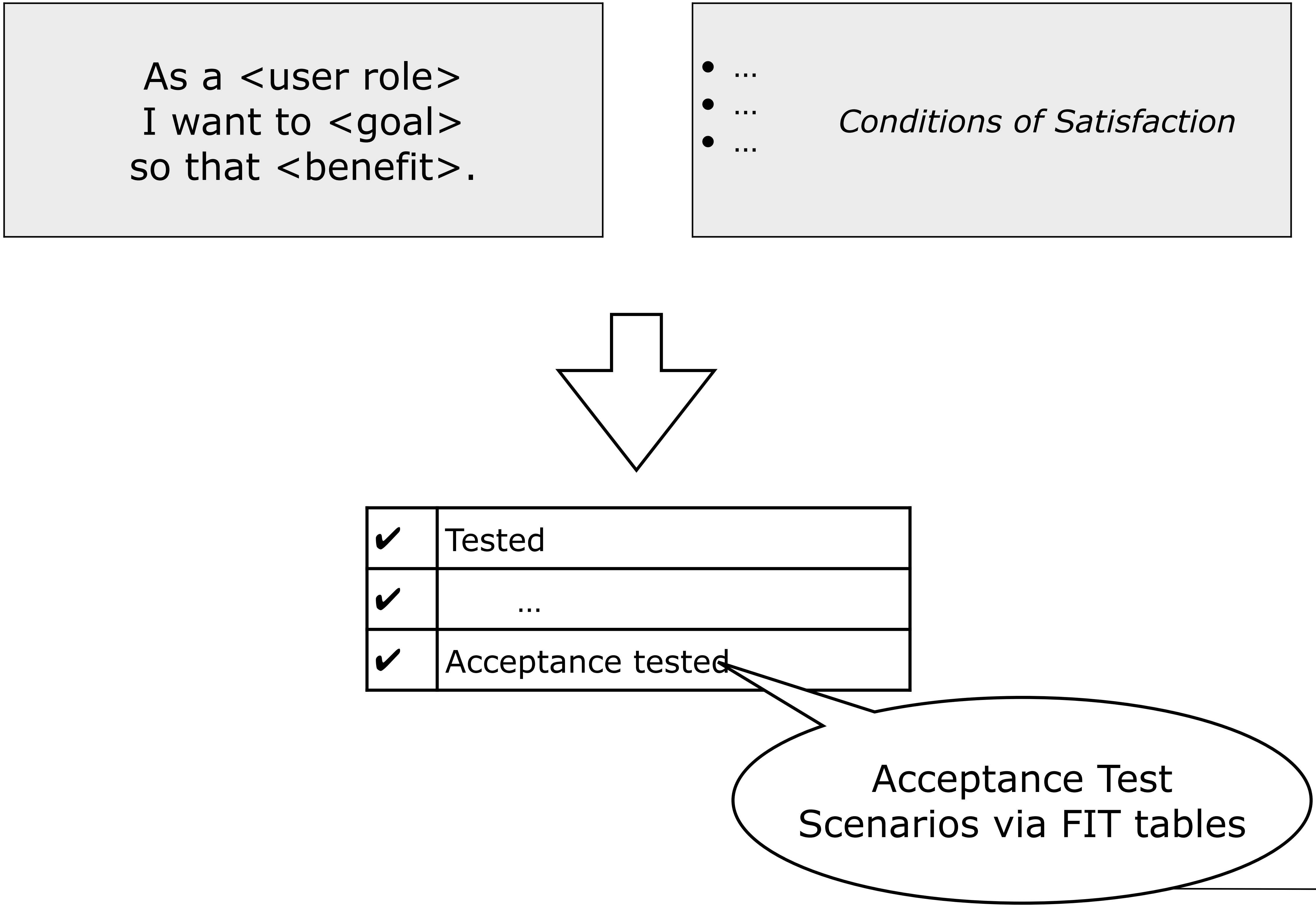
Automated

Unit Tests
Integration Tests

Tools

Performance Testing
Load Testing
Security Testing
“ility” Testing

Definition of Done



As a <user role>
I want to <goal>
so that <benefit>.

- …
- …
- …

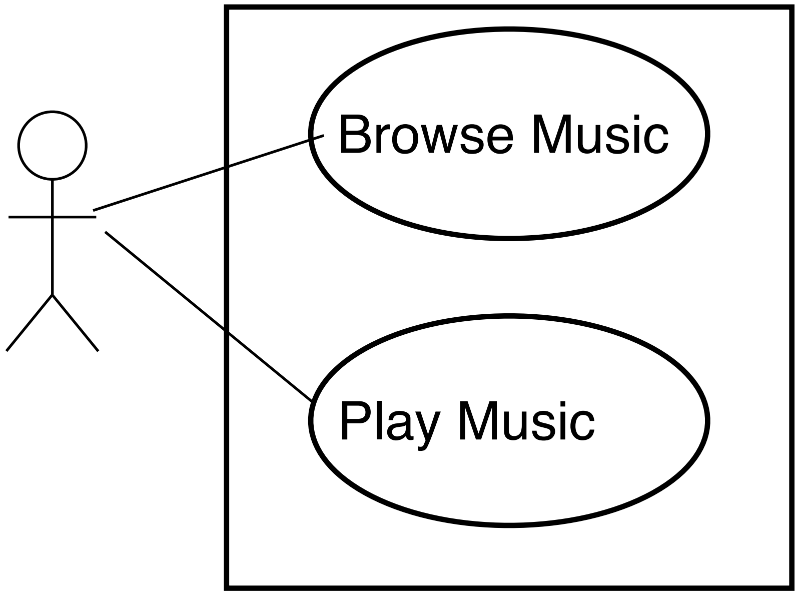
Conditions of Satisfaction

✔	Tested
✔	…
✔	Acceptance tested

Acceptance Test
Scenarios via FIT tables

FIT(*) Tables

Example: Acceptance Test Cases
http://fit.c2.com



Browse Music		
start	eg.music.browser	
enter	library	
check	total songs	37

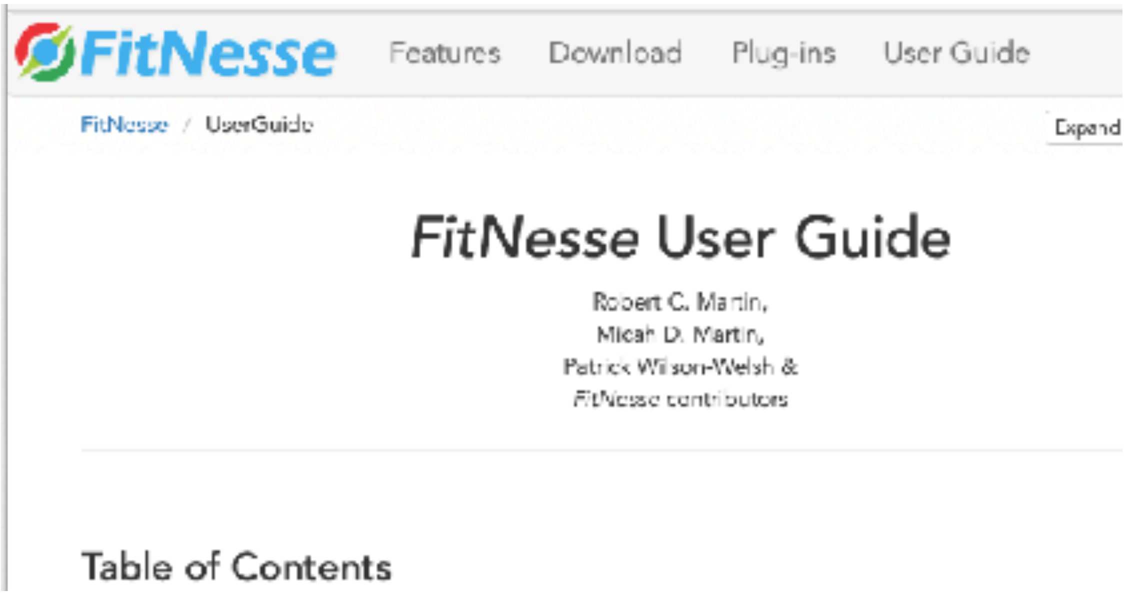
Browse Music		
enter	select	1
check	title	Akila
check	artist	Toure Kunda
enter	select	2
check	title	American Tango
check	artist	Weather Report
check	album	Mysterious Traveller
check	year	1974

Play Music		
start	eg.music.Realtime	
press	play	
check	status	loading
pause	2	
check	status	playing

(*) FIT = Framework for Integrated Testing

Tool Support

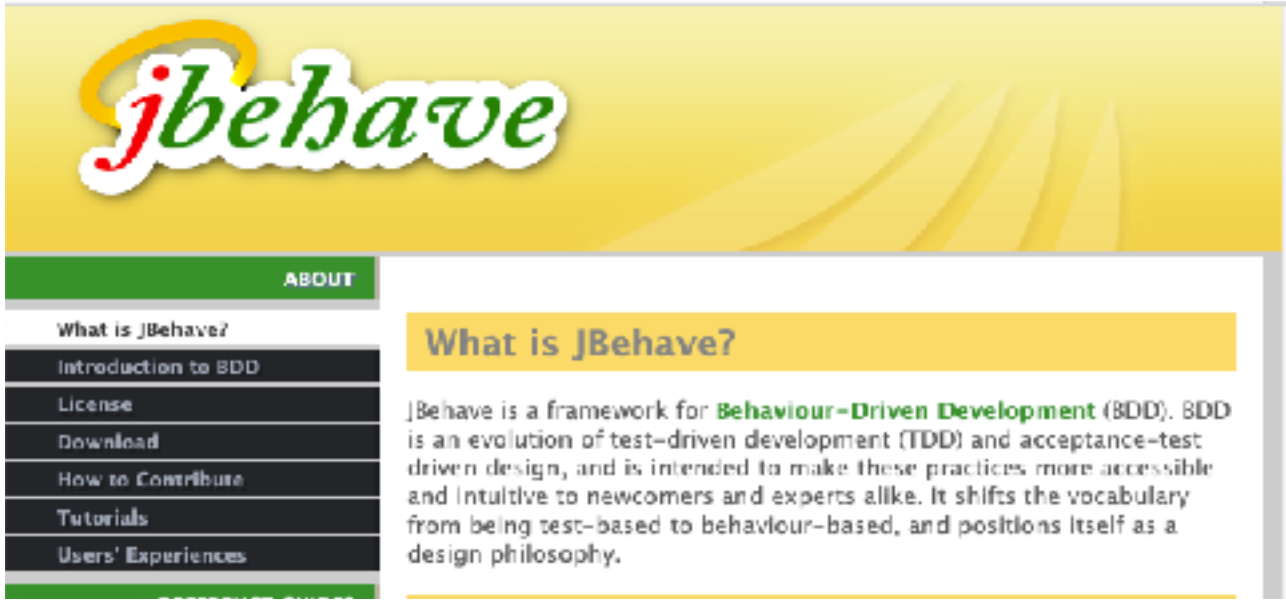
CAPSTONE PROJECT



FitNesse User Guide

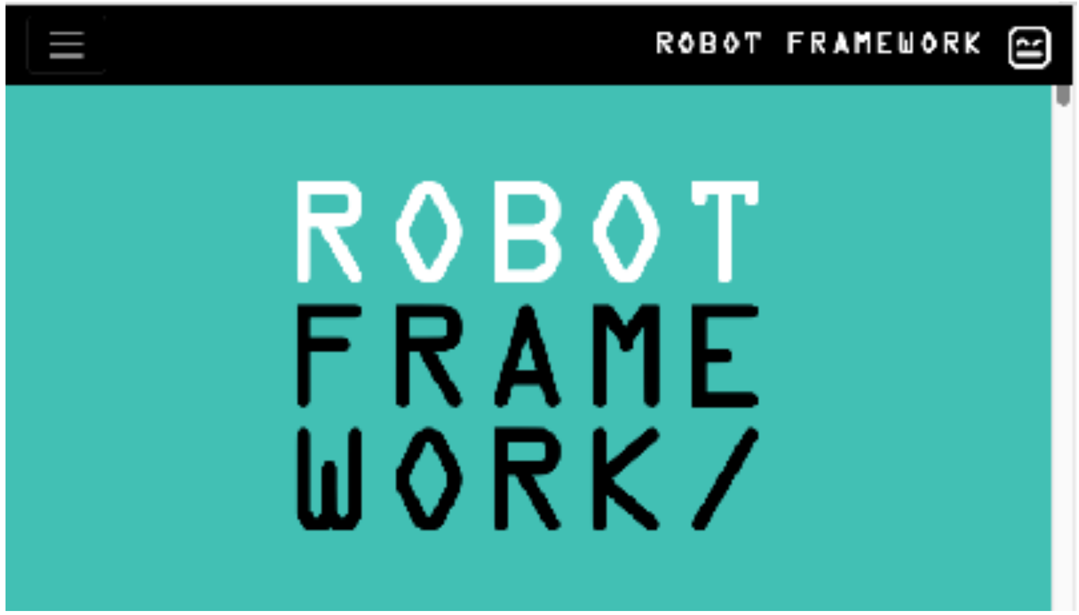
Robert C. Martin,
Micah D. Martin,
Patrick Wilson-Welsh &
FitNesse contributors

Table of Contents



What is JBehave?

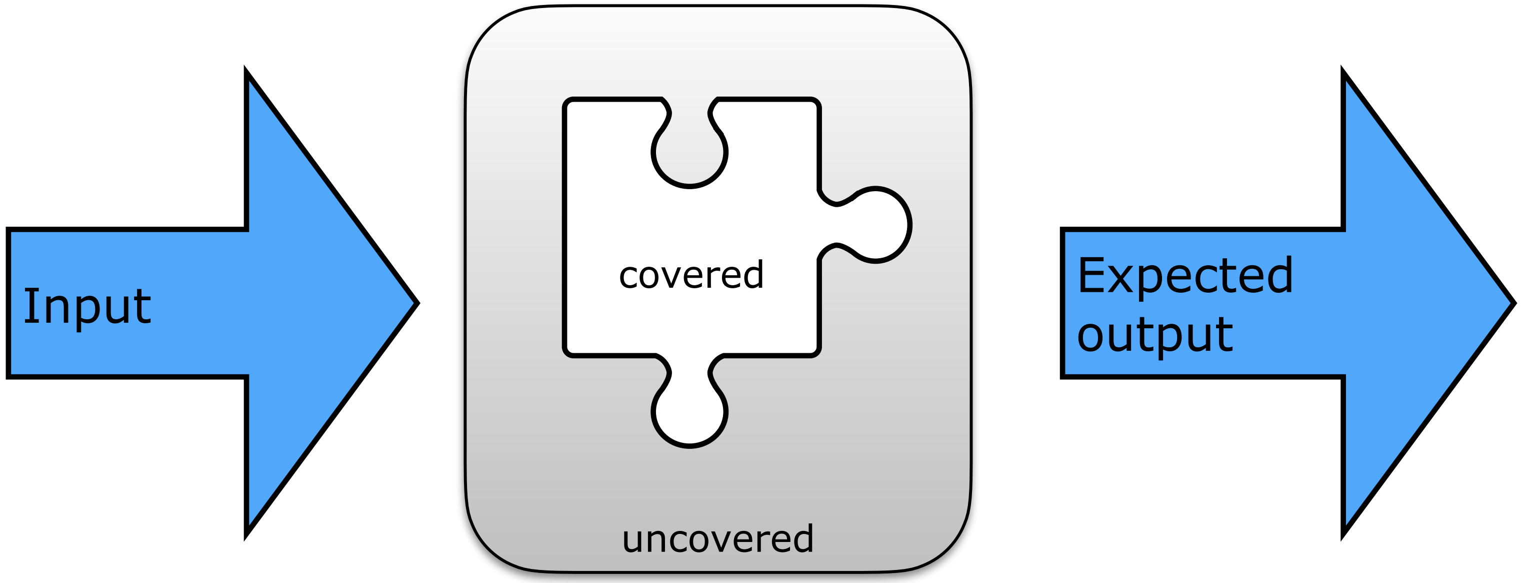
JBehave is a framework for **Behaviour-Driven Development** (BDD). BDD is an evolution of test-driven development (TDD) and acceptance-test driven design, and is intended to make these practices more accessible and intuitive to newcomers and experts alike. It shifts the vocabulary from being test-based to behaviour-based, and positions itself as a design philosophy.



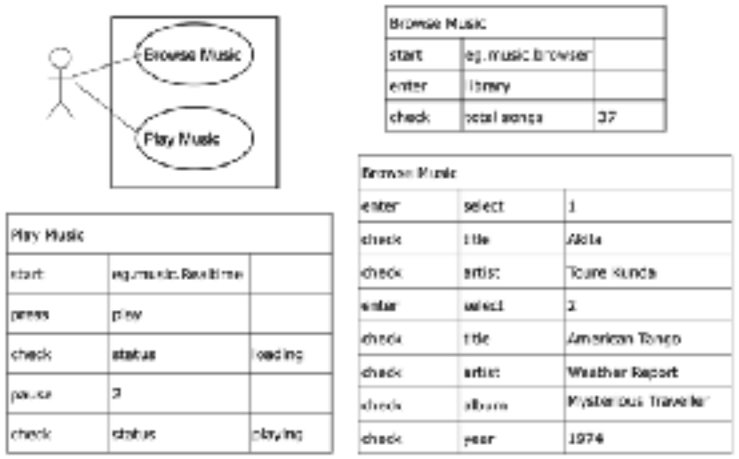
Tools & techniques that elevate teams to greatness

Test Coverage ≠ Code Coverage

Code Coverage
line, statement, …,
MC/DC, mutation



Requirement Coverage
FIT-tables, FMEA-tables


Browse Music
Play Music

Browse Music		
start	eg.music.browser	
enter	library	
check	total songs	37

Play Music		
start	eg.music.Realtime	
press	play	
check	status	loading
pause	2	
check	status	playing

Browse Music		
enter	select	1
check	title	Aida
check	artist	Toure Kunda
enter	select	2
check	title	American Tango
check	artist	Weather Report
check	album	Mysterious Traveller
check	year	1974

Test Coverage
Test Plan

- How many of the planned test cases did we specify?
- How many of the specified test cases did we execute?

Conclusion: Correctness & Traceability & …

- Correctness
  + Obviously (are we building the product right)

Besides verifying that the implementation corresponds with the specification, there are other good reasons to test

- Traceability
  + Naming conventions between tests and requirements specification is a way to trace back from components to the requirements that caused their presence

- Maintainability
  + Regression tests verify that post-delivery changes do not break anything

- Understandability
  + If you are a newcomer to the system, reading the test code is a good place to see what it actually does
  + *Write the tests first*, and you'll be the first user of your component interface, encouraging you to make it very readable

Summary (i)

You should know the answers to these questions

- What is (a) Testing, (b) a Testing Technique, (c) a Testing Strategy
- What is the difference between an error, a failure and a defect?
- What is a test case? A test stub? A test driver? A test fixture?
- What are the differences and similarities between basis path testing, condition testing and loop testing?
- How many tests should you write to achieve MC/DC coverage? And multiple condition coverage?
- Where do you situate alpha/beta testing in the four quadrants model?
- What are the differences and similarities between unit testing and regression testing?
- How do you know when you tested enough?
- What is Alpha-testing and Beta-Testing? When is it used?
- What is the difference between stress-testing and performance testing?

You should be able to complete the following tasks

- Complete test cases for the Loop Testing example (Loop Testing on page 19).
- Rewrite the binary search so that basis path testing and loop testing becomes easier.
- Write a piece of code implementing a quicksort. Apply all testing techniques (basis path testing, conditional testing [3 variants], loop testing, equivalence partitioning) to derive appropriate test cases.

- Write FIT test cases for the user stories in you Bachelor Capstone Project
- *Apply fuzz testing to the REST-API of your project*

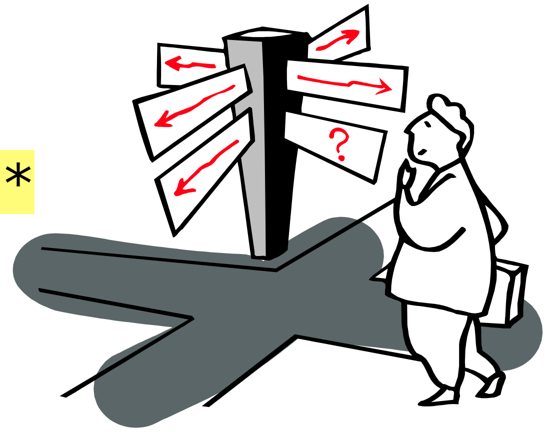
CAPSTONE PROJECT

Summary (ii)

Can you answer the following questions?

- You’re responsible for setting up a test program. To whom will you assign the responsibility to write tests? Why?
- Why do we distinguish between several levels of testing in the V-model?
- Explain why basis path testing, condition testing and loop testing complement each other.
- Why is mutation coverage a better criterion for assessing the strength of a test suite?
- *Explain fuzzing (fuzz testing) in your own words.*
- Explain what FIT tables are.
- When would you combine top-down testing with bottom-up testing? Why?
- When would you combine black-box testing with white-box testing? Why?
- Is it worthwhile to apply white-box testing in an OO context?
- What makes regression testing important?
- Is it acceptable to deliver a system that is not 100% reliable? Why (not)?
- Explain the subtle difference between code coverage and test coverage.

CHAPTER 7 – Formal Specification



Chapter completely revised

- Introduction
  + When, Why and What?
  ⇒ Design by contract & Testing
- Input/Output Specifications
  + Pre- and postconditions + invariants
  + Theorem proving
  + Weakest Possible Precondition
    - Statements, if-statements, loops, function calls
  + Experience report: JDK sort method
- State-Based Specifications
  + Statecharts
  + Guards, Nested states
  + Complete, Consistent, Unambiguous
  + Deduce test cases
- Formal Verification in Practice
- Conclusion
  + Correctness & Traceability

Literature

Books

- [Ghez02] In particular, chapters “Specification” and “Verification - Analysis”
- [Somm05] In particular, chapters “Formal Specification” & “Verification and Validation”
- [Pres00] In particular chapters “Formal Methods” & “Cleanroom Software Engineering“

Articles

- D. Cofer et al., "A Formal Approach to Constructing Secure Air Vehicle Software," in Computer, vol. 51, no. 11, pp. 14-23, Nov. 2018, doi: 10.1109/MC.2018.2876051.

- de Gouw, S., de Boer, F.S., Bubel, R. et al. “Verifying OpenJDK’s Sort Method for Generic Collections.” Journal of Automated Reasoning 62, 93–126 (2019). doi.org/10.1007/s10817-017-9426-4

Your Opinion?

What was the most effective means to achieve “provably secure against cyberattacks”?

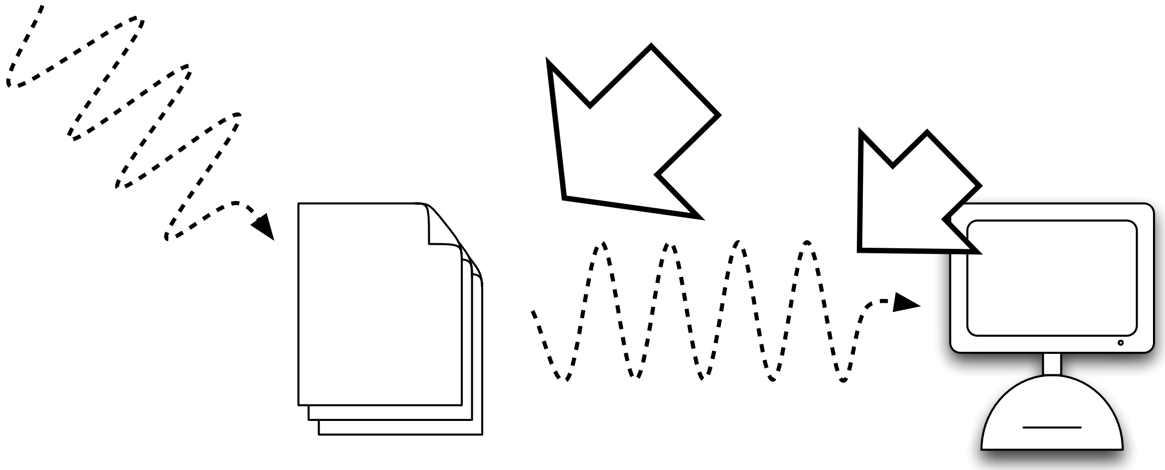
- ✓ 1. Modeling the system architecture and formal verification of its key security and safety properties.
- ✓ 2. synthesis of software components using languages that guarantee important security properties.
- ✓ 3. use of a formally verified micro-kernel to guarantee enforcement of communication and separation constraints specified in the architecture.
- ✓ 4. automatically building the final system from the verified architecture model and component specifications.
- ✓ 5. To assess the security of the software produced, we worked with a Red Team of professional penetration testers who evaluated our software and attempted to identify vulnerabilities.

A Formal Approach to Constructing Secure Air Vehicle Software

When Formal Specification?

Correctness

- ~~Are we building the right product? = VALIDATION~~
- Are we building the product right? = VERIFICATION



Mistakes are possible (likely!?)

- while transforming requirements into a system

Formal Specification is used for

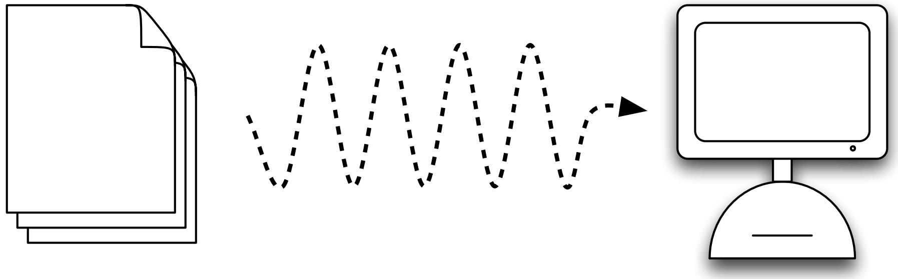
- (detailed) design
  + specify and verify key properties of *system under design*
    - e.g. State-based Specifications
- formal verification
  + mathematical proof: code is "correct"
    - e.g. Input/output specifications

Why Formal Specification?

- Software projects rely more and more on “Buy” than on “Build”
  - + Cheaper, more reliable, ...
  - + Companies focus in-house development on core business
    - - Buy 3rd party components for functionality outside the core
  - + 3rd party components evolve
    - - require well-specified interface

- Buy vs. Build
  - + But if we buy we need to specify
    - - clearly
    - - unambiguously
    - - completely

⇒ **Formally**



Why do we Care?

It is possible to build high-quality products without formal specifications!
(And it is possible to build low-quality products with formal specifications)

- **Fact**
  + For most systems it is more cost effective to apply other techniques (reviews, tests, ...)
    - business systems, information systems, ...
    - Most software development is in that area

- **Fact**
  + For some areas, the benefits more than outweigh the costs
    - high-risk systems: human lives depend on the software
      > (because reliability is such a big issue)
    - embedded systems: software controlling hardware
      > (because components evolve at different rates)

    } **Cyber-Physical Systems**

    - standards: defining information exchange protocols
      > (because the same specification is reused by a lot of implementations)

Sooner or later you will be confronted with one of these!

What are Formal Specifications?

- What is a ...

  + **Specification**. A description of desired system properties.
    - Preferably the “what” and not the “how”

  + **Informal specification**. Specification in natural language.
    - augmented with figures, tables, examples, scenarios

  + **Semi-formal specification**. Specification based on a notation with precise syntax but loose semantics.
    - e.g. UML class & sequence diagrams

  + **Formal Specification**. Specification based on a formal model with precise syntax & semantics.

Testing and Design by Contract

Formal foundation: formal syntax + formal semantics

- Possible to mathematically prove that a given system satisfies the specification

A system is correct with respect to its specification !
Note: faults (omissions!) in the specification are still possible

Testing

- Formal specifications ⇒ black-box testing

  test-cases: complete coverage, thus highest probability of finding mistakes

Design by Contract

- Formal specifications ⇒ natural pre- and post conditions

Is this Valid? Output?

```
<p>
  <strong>
    <em>
      Hello World
    </em>
  </strong>
</p>
```

```
 <p>
  <strong>
    <em>
      Hello World
    </strong>
  </em>
</p>
```

A) Input/Output Specifications

- include logic assertions (pre & post-conditions + invariants) inside an algorithm
- verify termination and correctness via stepwise formal reasoning

Example: Input/Output Specification for a binary search procedure

```
procedure Binary_search (Key : ELEM ; T: ELEM_ARRAY;
    Found : out BOOLEAN; L: out ELEM_INDEX) ;

Pre-condition
    T’LAST - T’FIRST ≥ 0 and   -- not empty
        for_all i, -- universal qualifier
            T’FIRST ≤ i ≤ T’LAST-1, T (i) ≤ T (i + 1) --sorted

Post-condition
    (Found and T (L) = Key)  or
        (not Found and
        not (exists i, -- existential qualifier
            T’FIRST ≤ i ≤ T’LAST, T (i) = Key))
```

Proving Correctness

Goal:

- mathematically prove that post-condition is always satisfied when pre-condition is true

Termination?

- While loop terminates if Found or Bott > Top
- If an element = key exists, Found is set true
- In a loop execution either Found := true, Bott >> or Top <<
- Initially, Top > Bott thus (if Found remains false) eventually Bott > Top

Correctness?

- Loop invariant is “true” on entry to the loop.
- Assertion 2 follows because of the successful test Key = Mid
- Assertion 3 follows because the array is ordered. If T (Mid) < Key all values up to T (Mid) must also be less than the key
- Assertion 4 follows by substituting Bott-1 for Mid (if T(mid) != Key)
- Assertions 5 and 6. Similar argument to 3 and 4
- After loop execution, either the key has been found or there is no value in the array which has been searched which matches the key. However, Bott > Top so all the array has been searched

⇒ Therefore, the binary search routine code conforms to its specification

Intermediate Assertions

```
Bott := T’FIRST; Top := T’LAST ;
L := ( T’FIRST + T’LAST ) mod 2; Found := T( L ) = Key;
-- 1 . ASSERT (Found and T(L) = Key) or ((not Found)
-- and (not Key in T(T’FIRST..Bott-1, Top+1..T’LAST)));
while Bott <= Top and not Found loop
        Mid := (Top + Bott) / 2;
        if T( Mid ) = Key then
            Found := true; L := Mid;
        --  2. ASSERT Key = T(Mid) and Found;
        elsif T( Mid ) < Key then
            --  3. ASSERT not Key in T(T’FIRST..Mid);
            Bott := Mid + 1;
        --  4. ASSERT not Key in T(T’FIRST..Bott-1);
        else
            --  5. ASSERT not Key in T( Mid..T’LAST );
            Top := Mid - 1;
        --  6. ASSERT not Key in T(Top+1..T’LAST);
end if;
end loop;
```

Automated Theorem Provers

Compiler support to prove that
when pre-condition is true post-condition will (should) always be satisfied.

(Sometimes counter examples when proof does not hold.)

The KeY Project

Java

ethereum

solidity

VeriFast

LEAN THEOREM PROVER

TLA+

intel

aws

Microsoft Azure

elastic

Dafny

https://dafny.org/

Hoare Logic (revisited)

Let:

S series of statements

{P} and {Q} are *properties*

{P} is the precondition

{Q} is the postcondition

Then:

{P} S {Q}

is a *Hoare Triple* meaning

“Any execution of A starting in a state where P holds, ~~must~~ *should* terminate in a state where Q holds”

Example: ∀ x positive Integer

{x = 5} x := x * 2 {x > 0}

Adapted from
Chapter 5. Design by
Contract

Partially Correct / Totally Correct

Then:

{P} S {Q}

is a Hoare Triple meaning

"Any execution of A starting in a state where P holds, *should* terminate in a state where Q holds"

The implementation of S with respect to its specification is …

- **Partially correct.**
  + Assuming the precondition is true just before the function executes, then *if the function terminates*, the postcondition is true.
    - Infinite loops, raising exceptions, … is allowed

- **Totally correct.**
  + Again assuming the precondition is true before function executes, the function *is guaranteed to terminate* and when it does, the postcondition is true.

Stronger (and Weaker)

Let {P1} and {P2} be conditions expressed via predicates

- {P1} is *stronger* then {P2} iff
  + $\{P1\} \neq \{P2\}$
  + $\{P1\} \Rightarrow \{P2\}$
- {P1} is *weaker* then {P2} iff
  + $\{P1\} \neq \{P2\}$
  + $\{P2\} \Rightarrow \{P1\}$
- example
  + {x = 5} x := x * 2 {x > 0}
  + {x = 5} x := x * 2 {x > 5 and X < 20}

    - {x > 5 and X < 20} is stronger than {x > 0}
      > stronger is better for a post-condition
      (it is more precise about the outcome)

Adapted from
Chapter 5. Design by
Contract

Strongest Postcondition

Consider the Hoare triple
  {P} S {Q}

if ∀ Q′ such that {P} S {Q′}, Q ⇒ Q′

  then Q is the *strongest postcondition* of S with respect to P

Denoted with sp(S, Q)

Quizz

- example
  + {x = 5} x := x * 2 {x > 0}
  + {x = 5} x := x * 2 {x > 5 and X < 20}

What is the *strongest postcondition* for this Hoare triple?

+ {x = 5} x := x * 2 {……………………}

Deducing the Strongest Postcondition

Consider the Hoare triple

{P} S {Q}

When we know {P} and S we can deduce sp(S,P)

For assignment

{P} x:= E {x = E}

Assignment with operation

{x+y = 5} x := x + z {x’ + y = 5 and x = x’ + z}

x’ represents the “old” value of x, thus before S executes

Weakest Precondition

For proving correctness it makes more sense(*) to calculate the inverse:

- Given a statement S and a postcondition Q,
  + what is the weakest possible precondition?

Consider the Hoare triple
{P} S {Q}

If ∀ P′ such that {P′} S {Q}, P′ ⇒ P, then

P is the *weakest precondition* of S with respect to Q.

Denoted with wp(S, Q)

(*) Why does it make more sense to find the weakest possible precondition?

- It represents the least amount of work to prove correctness
- Too strong a pre-condition may imply that we cannot prove correctness

Weakest Precondition (assignment)

Consider the Hoare triple
{P} x:= E {Q}

Then the weakest precondition means that we should

- replace each occurrence of x in Q with E
  - denoted with [E/x] Q
- and then substitute in the precondition.

Thus {[E/x] Q} x := E; {Q}

{………} x := x - 2; {x > 0}

Fill in the weakest precondition for {………}

Weakest Precondition (multiple statements)

Consider the Hoare triple

$\{P\}\ S_1;\ S_2;\ \ldots\ S_n\ \{Q\}$

Then the weakest precondition is deduced backwards.

$\{P\} = wp(S_1;\ S_2;\ \ldots\ S_n,\ Q)$

$= wp(S_1;\ S_2;\ \ldots S_{n-1},\ wp(S_n, Q))$

$= wp(S_1,\ wp(S_2,\ wp(\ \ldots\ wp(S_n, Q)\ldots)))$

$\{P\}\ x := z + 1;\ y := x + y\ \{y > 5\}$

step 1: $wp(y := x + y,\ y > 5)$

$[x + y\ /\ y]\ y > 5$

$(x + y > 5)$

step 2: $wp(x := z + 1,\ x + y > 5)$

[………… / …………] $x + y > 5$

(……………………… > 5)

Fill in the weakest precondition for {P}

Weakest Precondition (if statement)

Consider the Hoare triple

$\{P\}$ if C then S else T; $\{Q\}$

Then the weakest precondition means that we should

- calculate the result depending on C being true or false
- and then substitute in S or T branch.

Thus wp (if C then S else T, Q)

$$= (c \Rightarrow wp(S, Q)) \wedge (\neg c \Rightarrow wp(T, Q))$$

$$= (c \wedge wp(S, Q)) \vee (\neg c \wedge wp(T, Q))$$

Legend

$\wedge$	logical and
$\vee$	logical or
$\neg$	negation (not)

{………} if (x> y) then z := x else z := y; {z = max(x,y)}

Fill in the weakest precondition for {………}

Loops

Consider the Hoare triple

  {P} while C do S; {Q}

Proof by induction - introduce (i) loop invariant and (ii) loop variants

- (i) loop invariant I — what will ensure the postcondition?
  + The invariant is initially true (base case): $P \Rightarrow I$
  + Each loop step preserves the invariant (inductive step): $\{I \wedge C\}$ S $\{I\}$
  + After the loop terminates the postcondition is true: $\{\neg C \wedge I) \Rightarrow Q$
- (ii) loop variant — what guarantees that the loop terminates?
  = a monotonically decreasing function integer-value function v
  + a strictly decreasing with every step: $\{I \wedge C \wedge v = V\}$ S $\{I \wedge v < V\}$
  + when v reaches zero the loop terminates: $\{I \wedge v \leq 0\} \Rightarrow \neg C$

Quizz

What is the ***weakest precondition*** for the following loop?

+ {………………} while (x > 0) do x := x-1; {x = 0}

in pseudo code

```
function int useless_loop(x int) {
    require (………………);
    while (x > 0)do x := x - 1;
    ensure (x == 0);
}
```

Quizz — step 1 - loop *in*variant

What is the *weakest precondition* for the following loop?

- + {……………….} while $(x > 0)$ do $x := x-1$; $\{x = 0\}$

- establish loop invariant I where $\{I \wedge C\}\ S\ \{I\}$
  - + $\{I \wedge (x > 0)\}\ x := x - 1\ \{I\}$
- look for the weakest pre-condition
  - + $I \wedge (x > 0) = wp(x := x - 1, I)$
  - + $I \wedge (x > 0) = [x - 1 / x]\ I$
- Which I would resolve the above?
  - + I = …………………
  - + ………………
- Does it terminate the loop? $\{\neg C \wedge I) \Rightarrow Q$
  - + $\{\neg C \wedge$ ……………) $\Rightarrow x = 0$

Quizz — step 2 - loop variant

What is the *weakest precondition* for the following loop?

+ {………………} while $(x > 0)$ do $x := x-1$; $\{x = 0\}$

- establish loop variant so that $\{I \wedge C \wedge v = V\}$ S $\{I \wedge v < V\}$
  + $\{(x \geq 0) \wedge (x > 0) \wedge v = V\}$ $x := x-1$; $\{x \geq 0 \wedge v < V\}$
- choose x for v
  + $\{(x \geq 0) \wedge (x > 0) \wedge x = V\}$ $x := x-1$; $\{x \geq 0 \wedge x < V\}$
- substitute x-1 for x in the postcondition; always true?
  + ……………………
  + ……………………
- when v reaches zero the loop terminates: $\{I \wedge v \leq 0\} \Rightarrow \neg C$

  + $\{(x \geq 0) \wedge x \leq 0\} \Rightarrow \neg(x > 0)$

  + …………………
  + …………………

Function (Procedure) Calls

Consider the Hoare triple

  {P} S_{prev}; F(…); S_{next}; {Q}

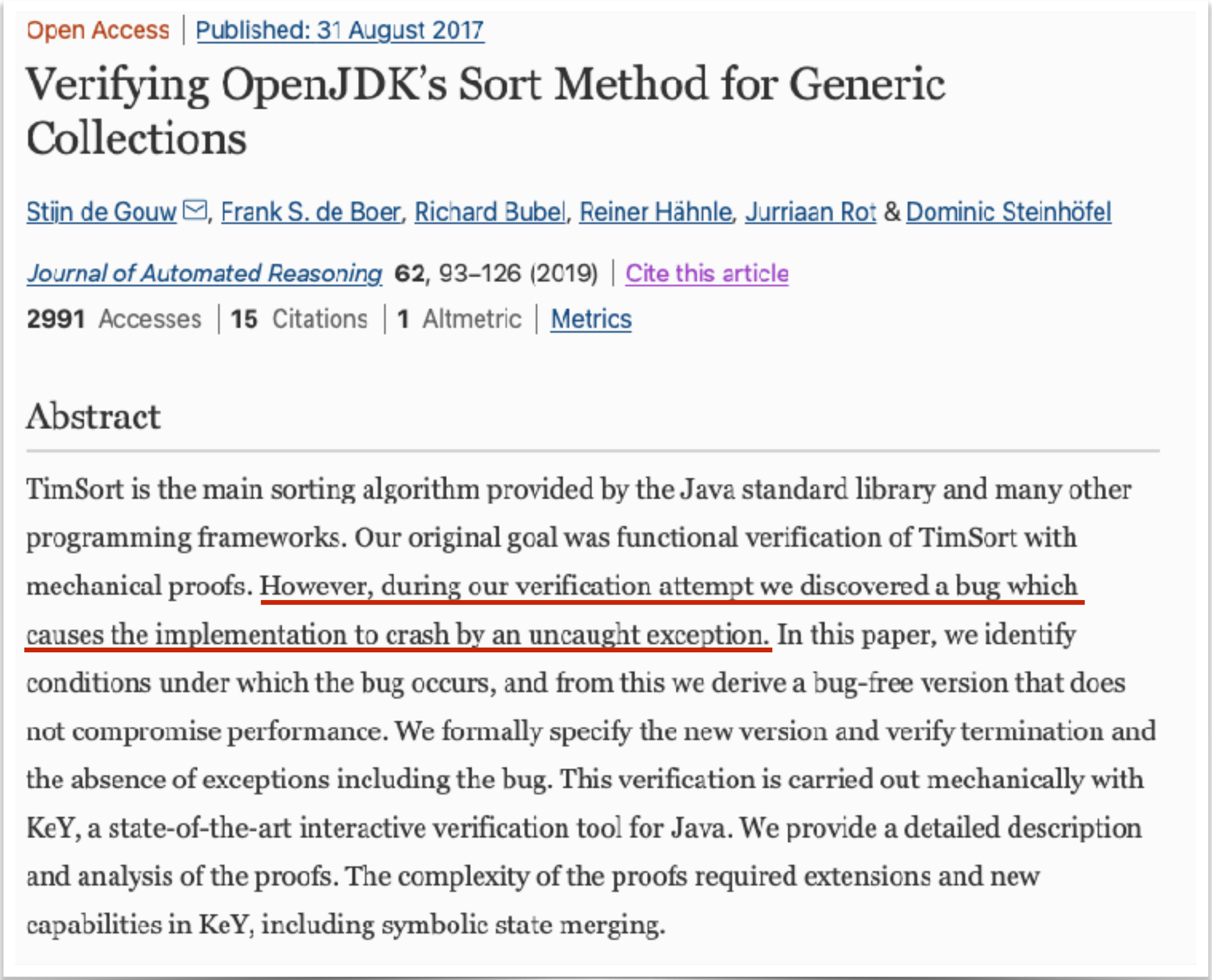
where F is a function call / procedure call / method invocation / …

> F is considered a black box
>
> We need a pre- and postcondition for F
>
> * Must be provided by the developer of F

Postcondition for F? = weakest precondition for S_{next}.

Postcondition for S_{prev}? = strongest precondition for F.

Example



Open Access | Published: 31 August 2017

Verifying OpenJDK's Sort Method for Generic Collections

Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot & Dominic Steinhöfel

Journal of Automated Reasoning **62**, 93–126 (2019) | Cite this article

2991 Accesses | **15** Citations | **1** Altmetric | Metrics

Abstract

TimSort is the main sorting algorithm provided by the Java standard library and many other programming frameworks. Our original goal was functional verification of TimSort with mechanical proofs. However, during our verification attempt we discovered a bug which causes the implementation to crash by an uncaught exception. In this paper, we identify conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise performance. We formally specify the new version and verify termination and the absence of exceptions including the bug. This verification is carried out mechanically with KeY, a state-of-the-art interactive verification tool for Java. We provide a detailed description and analysis of the proofs. The complexity of the proofs required extensions and new capabilities in KeY, including symbolic state merging.

Your Opinion?

What do you think happened with the bug report on the broken Java.utils.Collection.sort ()?

□ The suggested fix was correctly incorporated; the sort method now is provably correct.

□ They fixed the symptom and not the root cause; the risk is reduced but still not correct.

□ The bug report was ignored because the fault could not be reproduced (i.e. "Works for me”).

□ The bug report was closed without fix, because it was a low risk bug.

What actually happened …

We favored the second suggestion which is to formalize the invariant as originally intended and to fix the code of the method mergeCollapse that is responsible for reestablishing the invariant. We were able to formally and mechanically prove that this fixed version of the algorithm is correct in the sense that the stack lengths are sufficient and no ArrayIndexOutOfBoundsException is thrown. We describe this fix and its verification in Sect. 4.3 below.

In the aftermath of our discovery, it turned out that the bug was present in several implementations of TimSort. Besides in (Open)JDK, the bug was present in

(1) its original Python implementation,
(2) Android,
(3) an independent Java implementation used by Apache Lucene, as well as
(4) a Haskell implementation.

All of these projects fixed the bug within a short time frame. The OpenJDK project was the only one where the bug was fixed by just increasing the allocated array lengths, which is in our opinion sub-optimal, and there is no machine checked proof of that fix. All other projects implemented our second suggestion and fixed the underlying problem.

Cited from … (with slight lay-out changes)

- de Gouw, S., de Boer, F.S., Bubel, R. et al. “Verifying OpenJDK’s Sort Method for Generic Collections.” Journal of Automated Reasoning 62, 93–126 (2019). doi.org/10.1007/s10817-017-9426-4

(a) suggested fix was correctly incorporated

(b) fixed the symptom not the root cause

B) State-Based Specifications

Typically based on the notion of finite state machines

- describes the sequence of states a system is supposed to go through … in response to external stimuli (a.k.a. events)

StateCharts

- widely used: present in UML
- commonly used in real-time systems

- Definitions

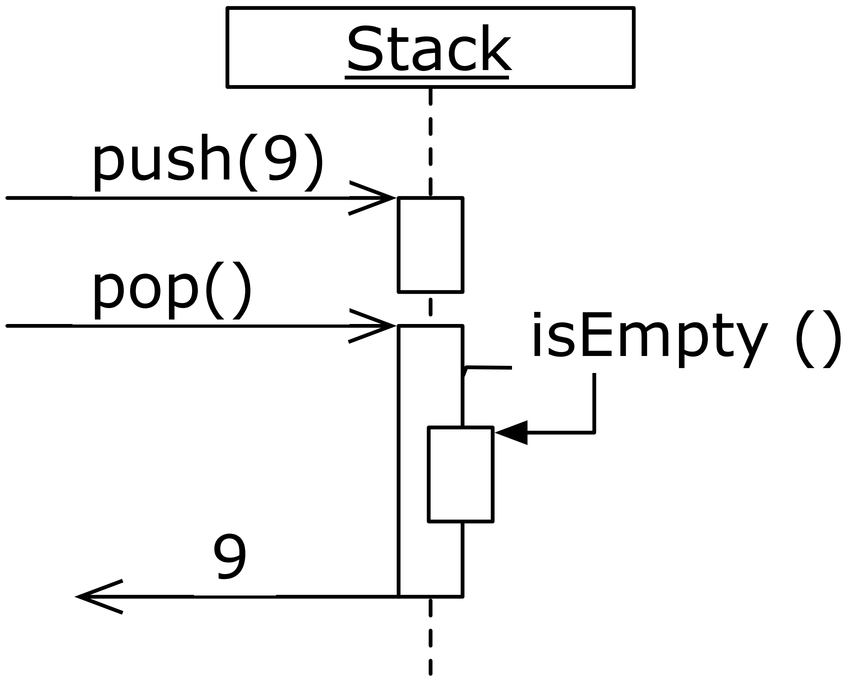
  + state = a condition an object satisfies
  (i.e. a predicate computing its result with the
  attribute values of the object)

  > The state can be observed from the outside !

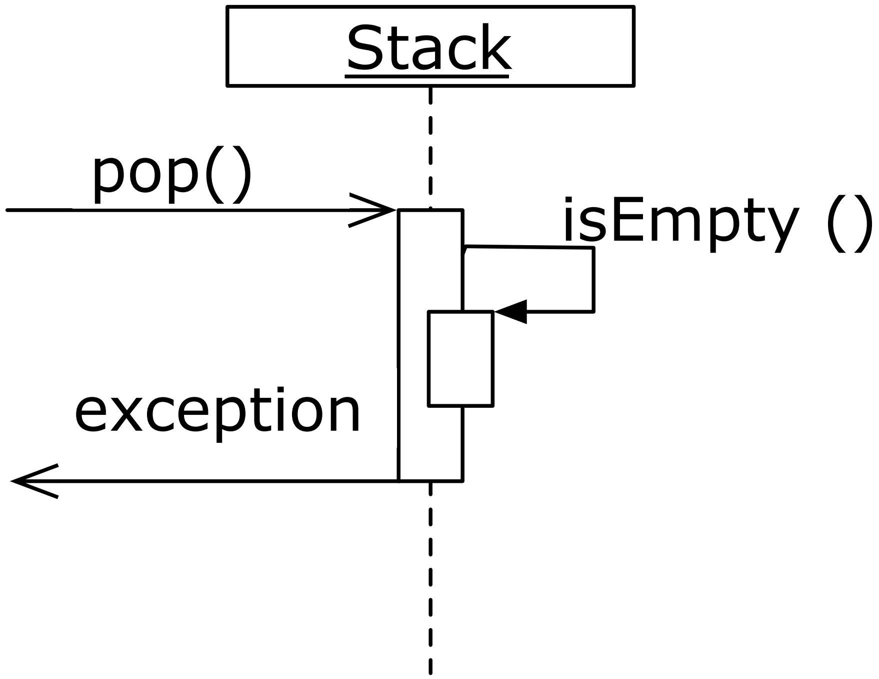
  + transition = a change of state triggered by an event, condition or time

Sequence Diagrams

a) happy day scenario: pop of a non-empty stack



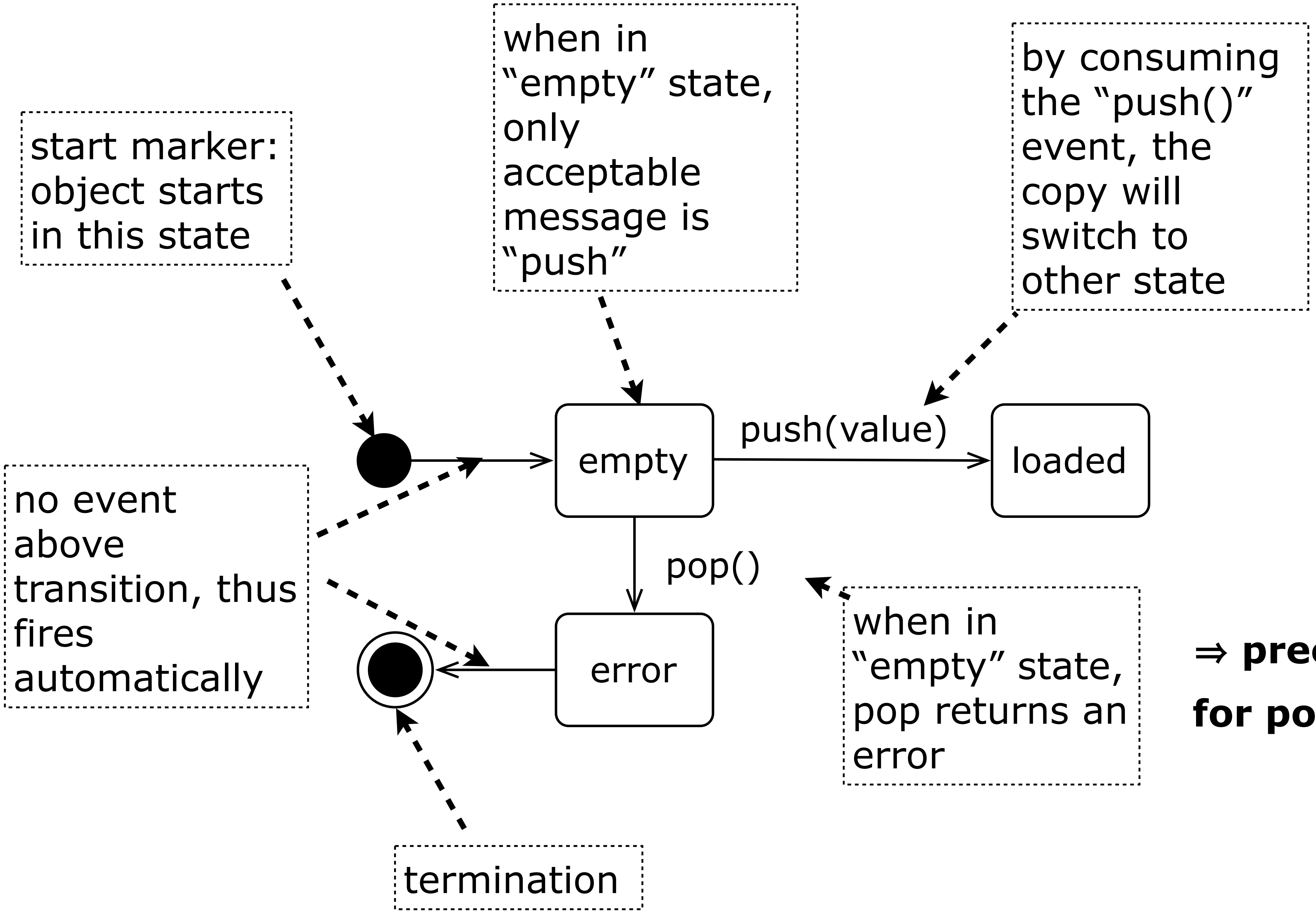
b) secondary scenario: pop of an empty stack



- What are acceptable message sequences for a stack?
- What is the union of all possible scenarios?

⇒ A statechart allows to specify all valid (also all invalid) scenarios

Statechart for a Stack



start marker: object starts in this state

when in “empty” state, only acceptable message is “push”

by consuming the “push()” event, the copy will switch to other state

no event above transition, thus fires automatically

when in “empty” state, pop returns an error

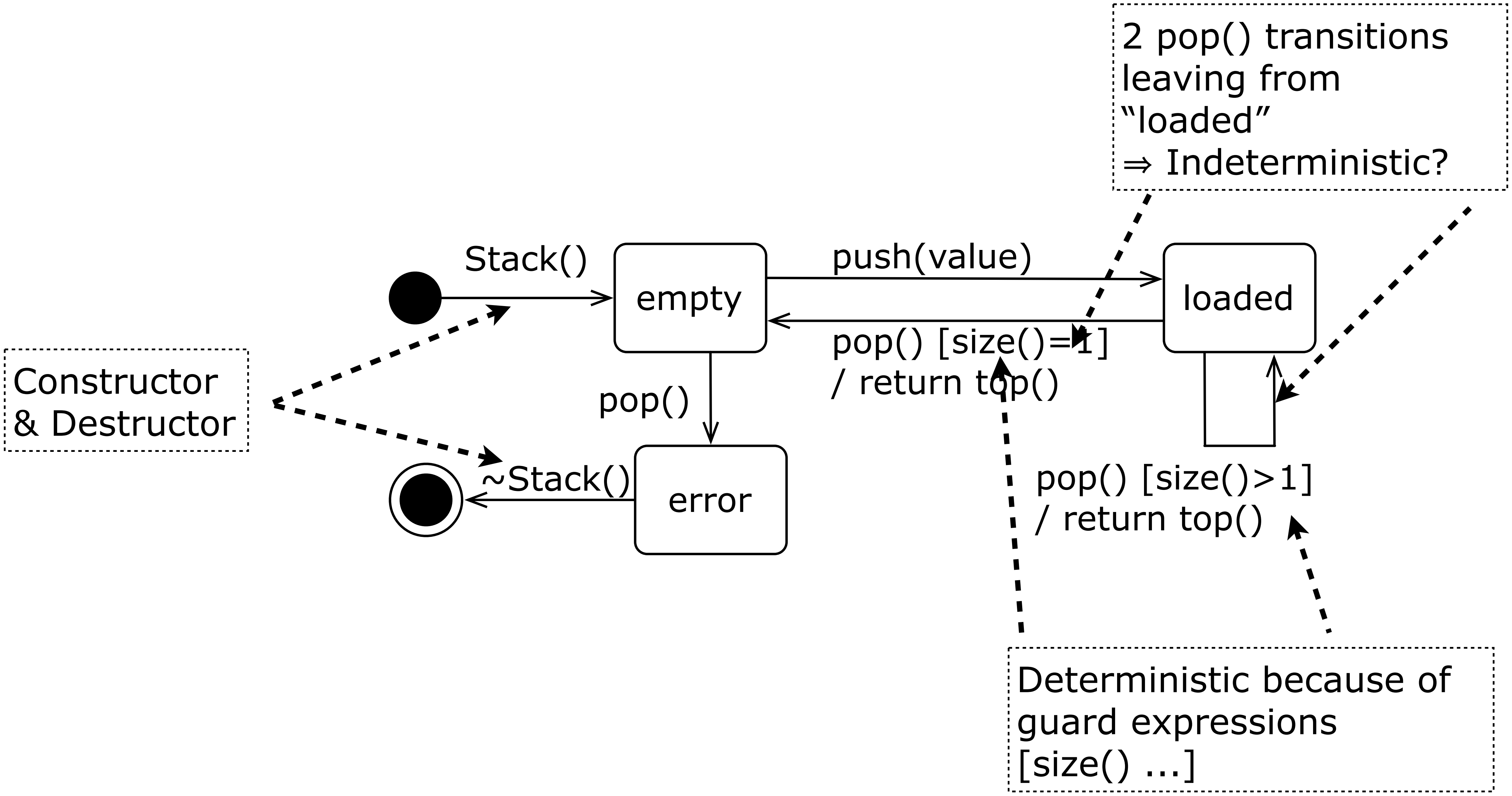
termination

⇒ **postcondition for push**

⇒ **precondition for pop**

What about a pop() on a loaded stack?

Guarded Transition



Constructor & Destructor

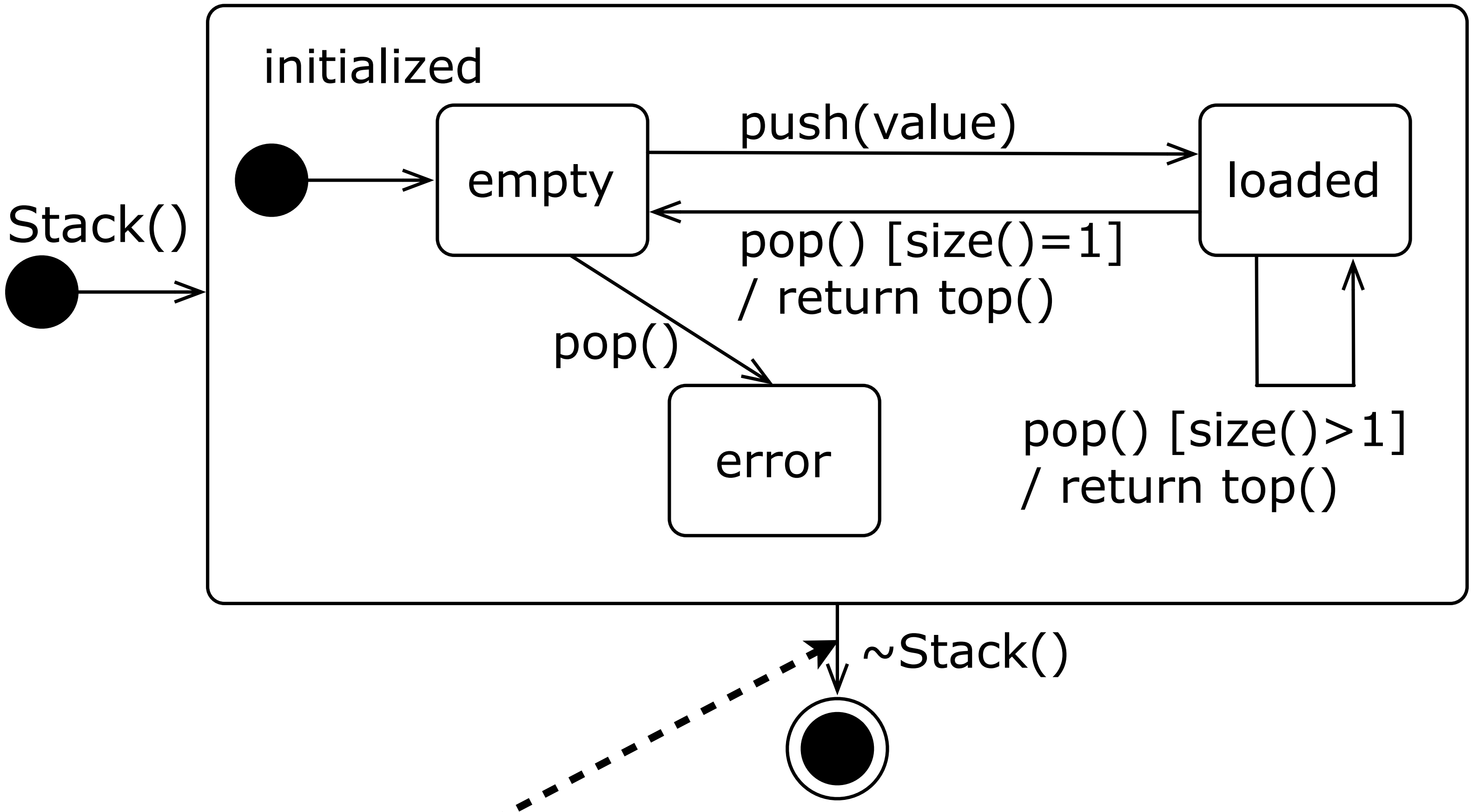
2 pop() transitions leaving from “loaded”
⇒ Indeterministic?

Deterministic because of guard expressions [size() ...]

What about destructors on empty/loaded state?

Nested State

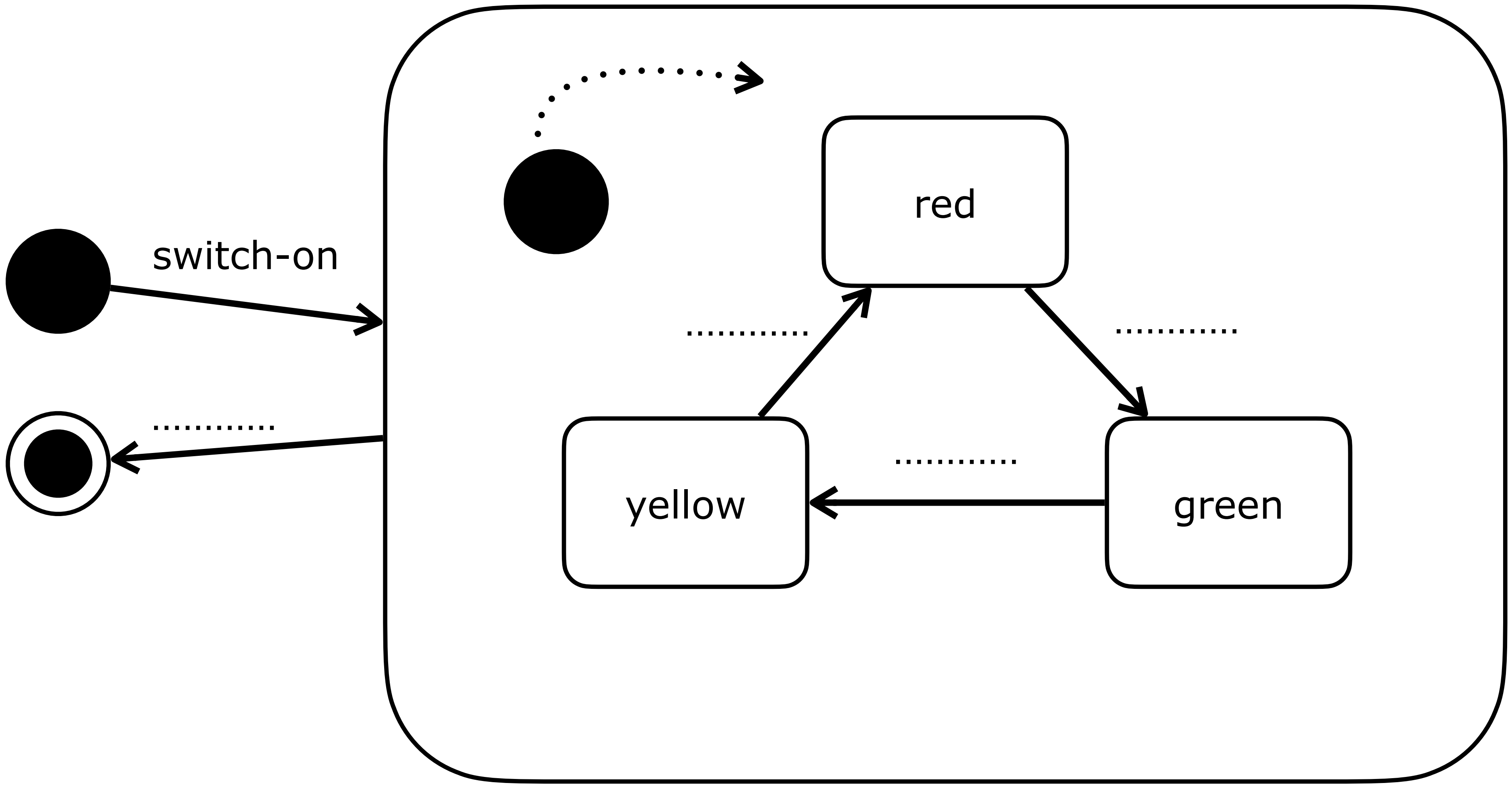
State with Nested States



shorthand for multiple transitions with same events and target states

Quiz

Complete the Traffic Light Statechart



Consistent / Complete / Unambiguous

When is a state-based specification …

- Complete
  + every event/state pair has a transition
    - Create table: events (incl. guards) x state
      one cell contains target state
    - all cells should have a target state

- Consistent
  + every state is reachable from initial state
    & final state is reachable from every other state
    - Breadth-first spanning tree; root is initial state
    - all leaf nodes of the graph should be terminal state

- Unambiguous (= deterministic)
  + same event (incl. guard) does not appear on more than one transition
    leaving any given state
    - Verify using table created in completeness

Deducing Test Cases

Test cases
+ cover all state transitions at least once (*)

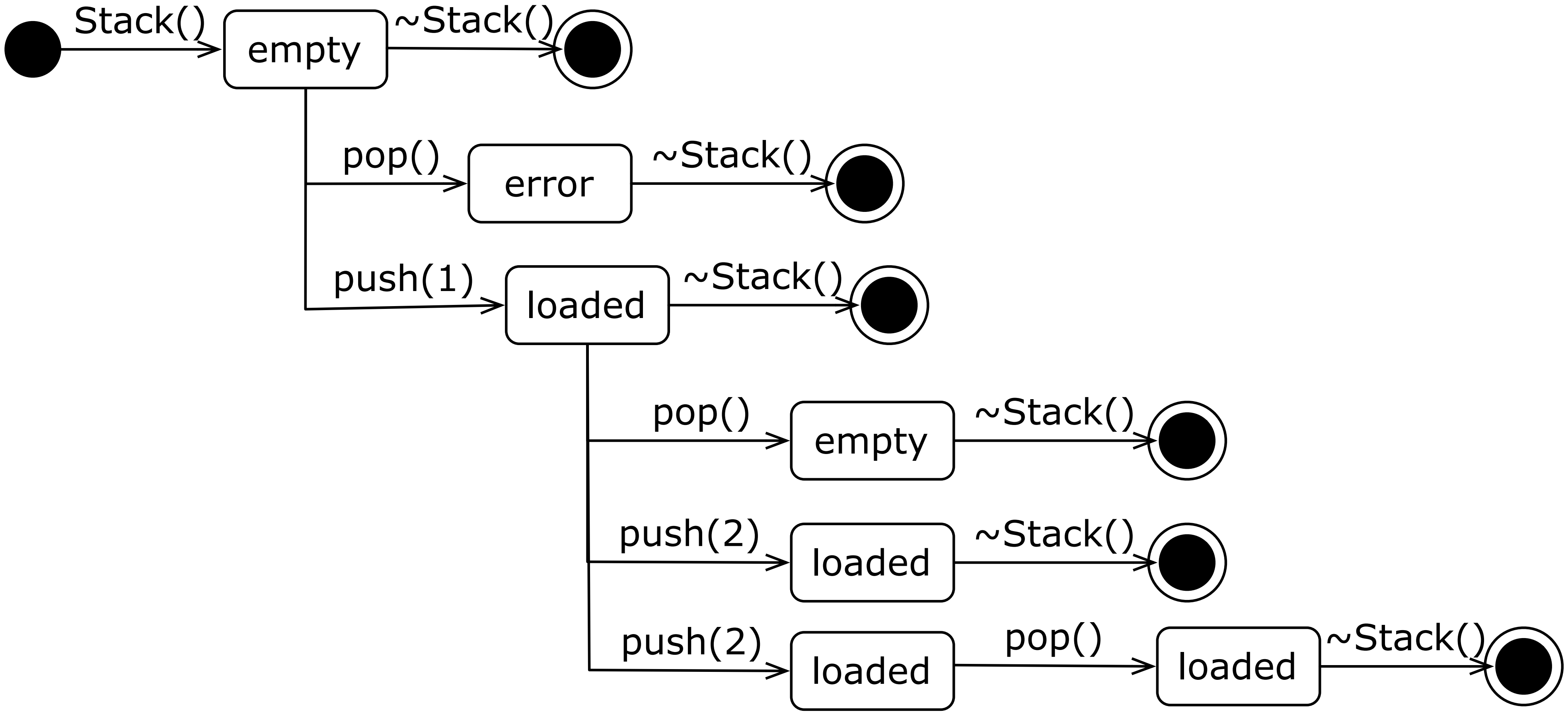
- define a predicate for each state,
  + which answers whether object is in that state
    - (Thus `initialized()` – `empty()` – `loaded()` – `error()`)

- test-cases must cover the breadth-first spanning tree
  + construct with same table used to verify completeness
    - rows and columns = events (incl. guards) x state
      cell contains target state

(*) Stronger coverage is possible:
- cover all sequences of state transitions of length n
- force all guards
- force all guards with boundary values
- …

Test cases for Statechart “Stack” (1/2)

	empty	loaded	error	initial
Stack()	empty (??)	loaded (??)	error	empty
push	loaded	loaded	error (??)	error
pop() [size()=1]	error	empty	error (??)	error
pop() [size()>1]	error	loaded	error (??)	error
~Stack	terminated	terminated	terminated	error



Test cases for Statechart “Stack” (2/2)

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
pop(s);
assertTrue(error(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
pop(s);
assertTrue(empty(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
push(s, 2);
assertTrue(loaded(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
push(s, 2);
assertTrue(loaded(s));
pop(s);
assertTrue(loaded(s));
s.~Stack();
```

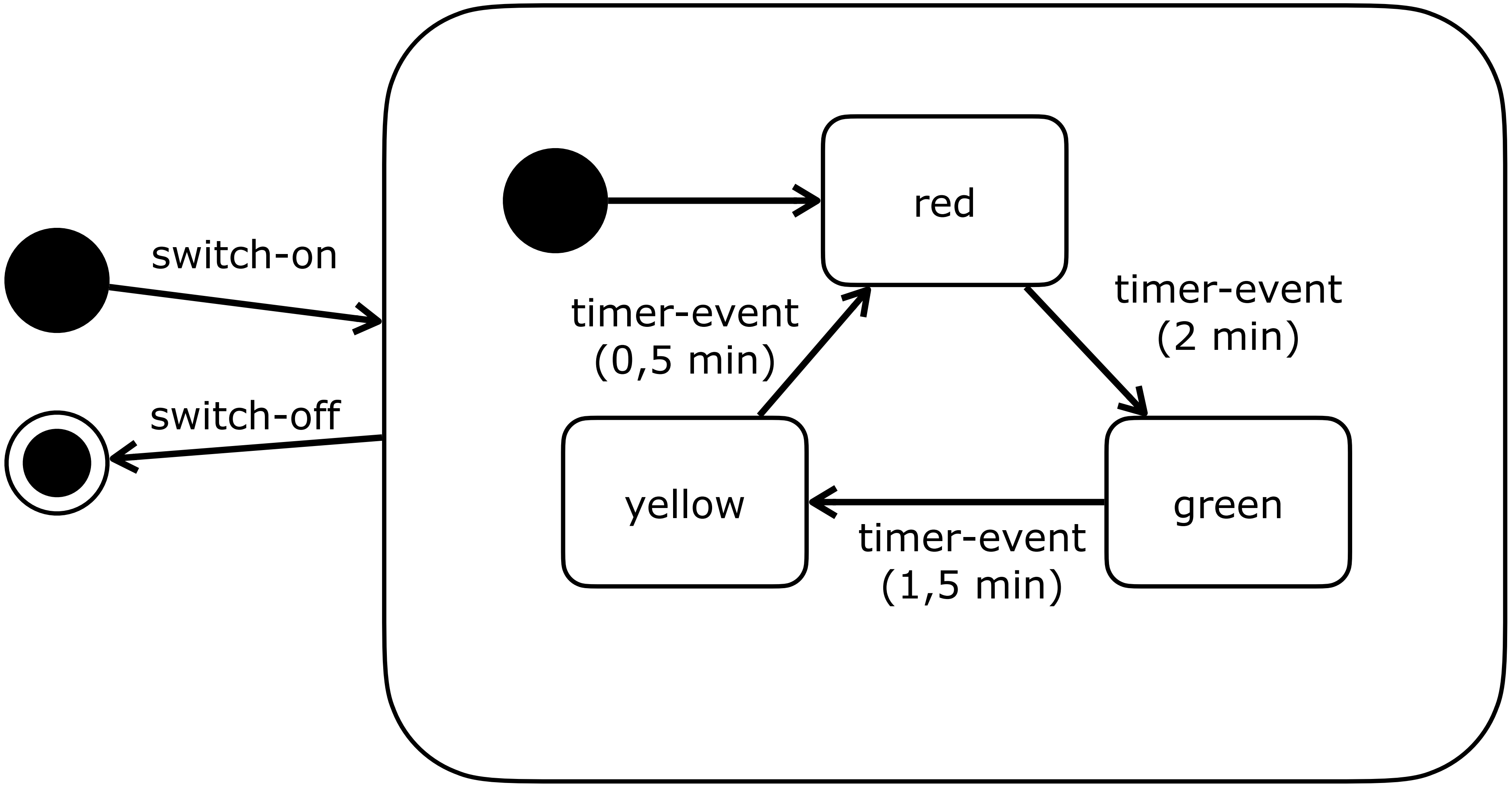
State-based Specifications Revisited

State-based Specifications

- Are particularly suitable for specifying “acceptable” message sequences
  + unify effect of all possible scenarios on one class in one statechart
  + “unacceptable” implies precondition
  + state change implies postconditions
    > Design by Contract

- Specify acceptable message sequences as paths through a graph
    > cover all paths
    > Path Testing

Example (advanced): Traffic Light (1/2)

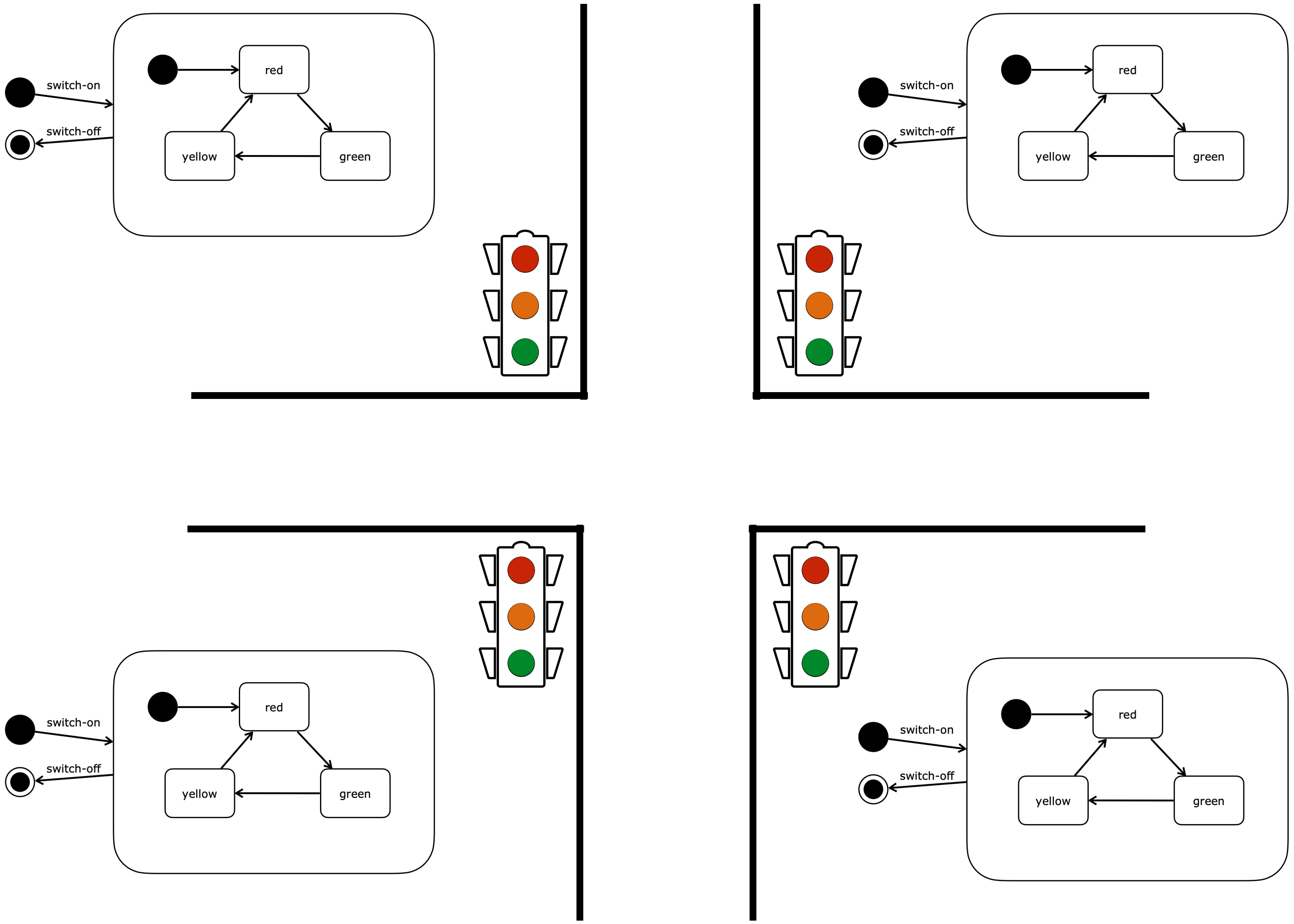


What is the starting state of this statechart?
Is this what you want?

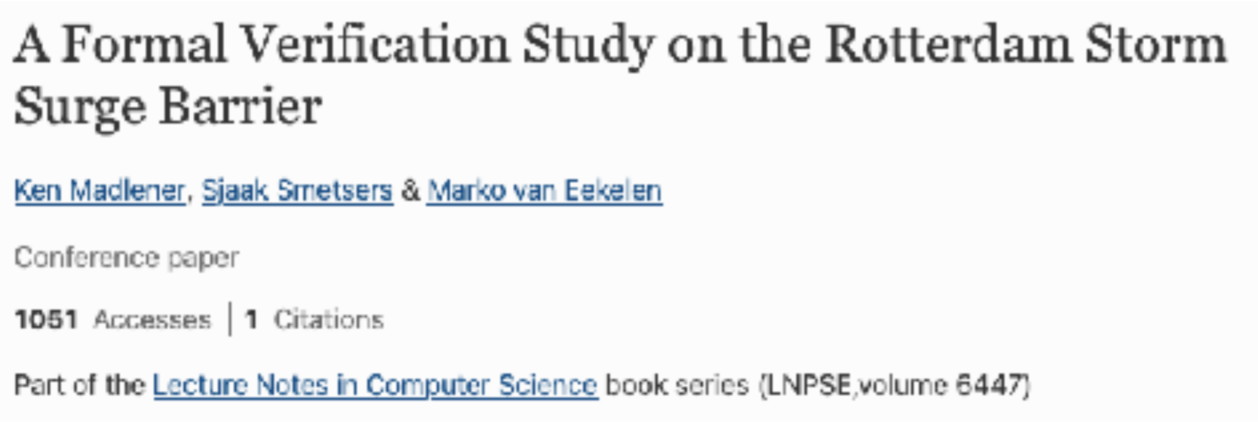
Example (advanced): Traffic Light (2/2)

- safety: “something bad never happens”
- liveness: “something good eventually happens”
- fairness: “if something may happen frequently, it will happen”

} formal verification
+ simulation
+ testing



Formal Verification in Practice (1/2)



A Formal Verification Study on the Rotterdam Storm Surge Barrier

Ken Madlener, Sjaak Smetsers & Marko van Eekelen

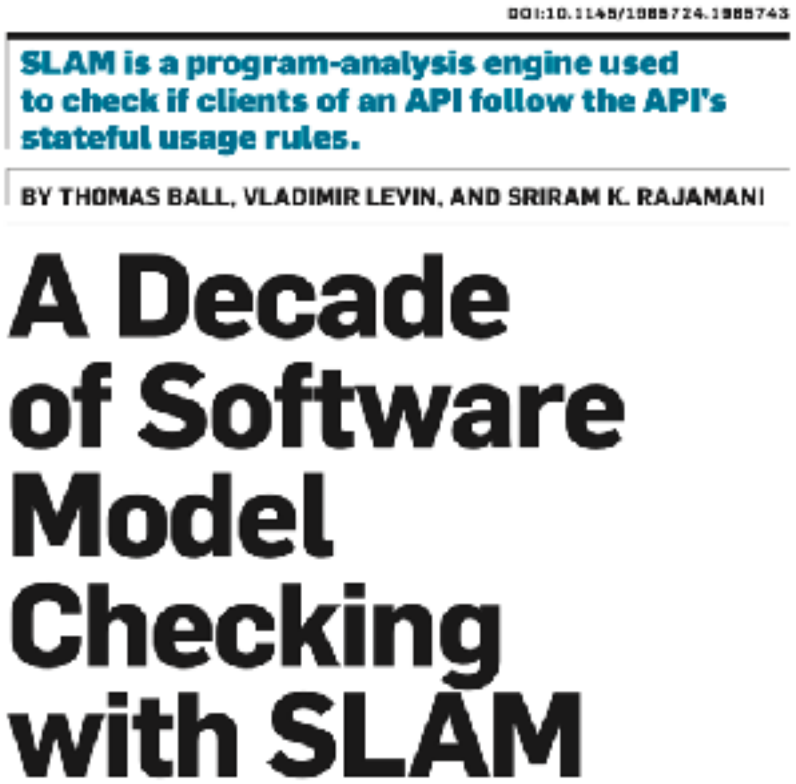
Conference paper

1051 Accesses | 1 Citations

Part of the Lecture Notes in Computer Science book series (LNPSE,volume 6447)

A lightweight model of the C++ code and the Z specification of the component was manually developed in the theorem prover PVS. As a result, some essential mismatches between specification and code were identified.

Windows 7



DOI:10.1145/1965724.1965743

SLAM is a program-analysis engine used to check if clients of an API follow the API's stateful usage rules.

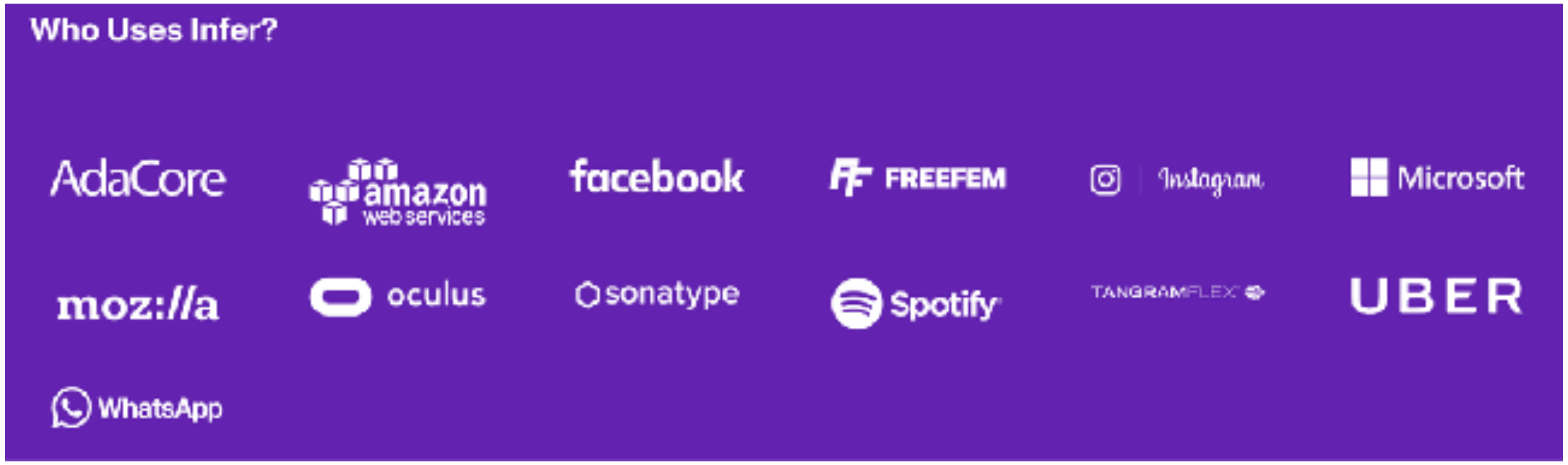
BY THOMAS BALL, VLADIMIR LEVIN, AND SRIRAM K. RAJAMANI

A Decade of Software Model Checking with SLAM

SDV was applied later in the cycle after all other tools, yet found 270 real bugs in 140 WDM and WDF drivers.

Formal Verification in Practice (2/2)

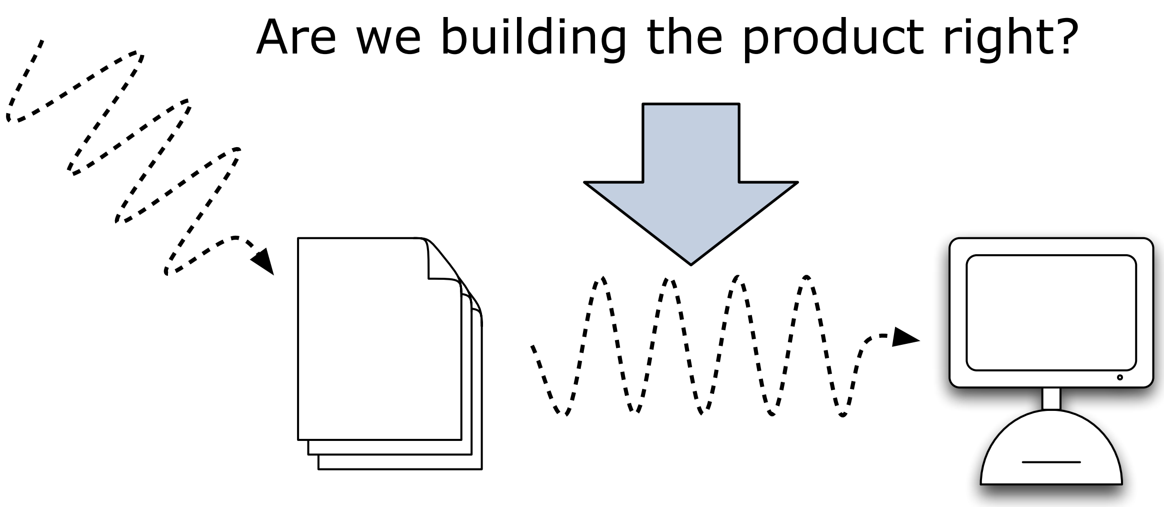
A tool to detect bugs in Java and C/C++/Objective-C code before it ships

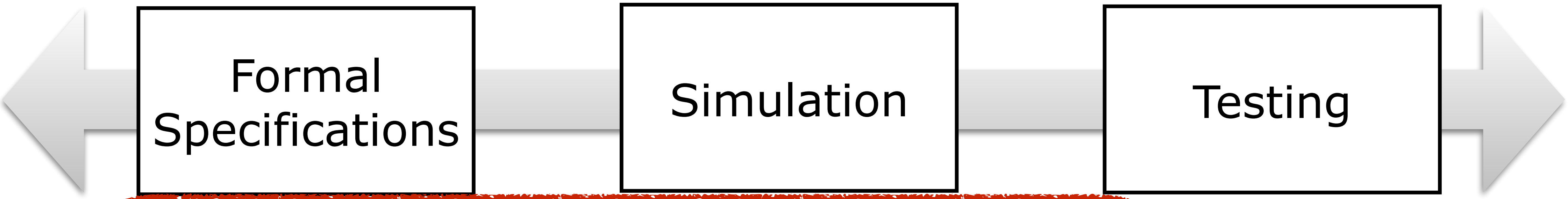


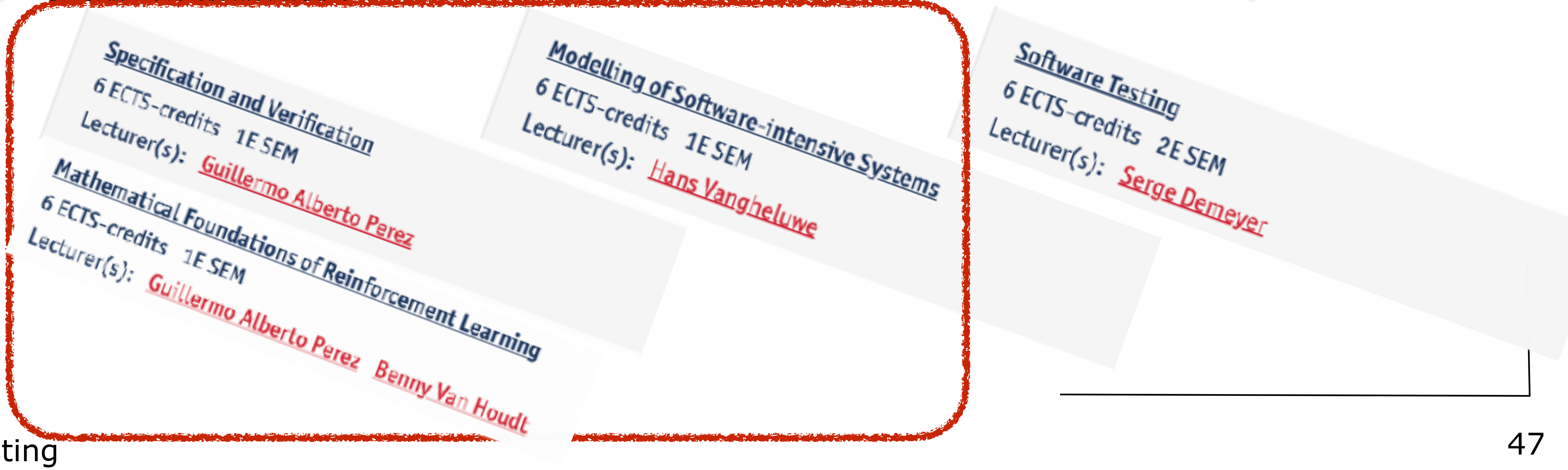
We are committed to helping you achieve the highest levels of security in the cloud. We've developed automated reasoning tools that use mathematical logic to answer critical questions about your infrastructure to detect misconfigurations that could potentially expose your data. We call this provable security because it provides higher assurance in the security of the cloud and in the cloud.

The Verification Landscape

Copied
06.Software Testing







Specification and Verification
6 ECTS-credits 1E SEM
Lecturer(s): Guillermo Alberto Perez

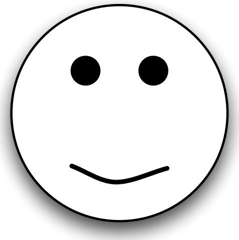
Mathematical Foundations of Reinforcement Learning
6 ECTS-credits 1E SEM
Lecturer(s): Guillermo Alberto Perez Benny Van Houdt

Modelling of Software-intensive Systems
6 ECTS-credits 1E SEM
Lecturer(s): Hans Vangheluwe

Software Testing
6 ECTS-credits 2E SEM
Lecturer(s): Serge Demeyer

Correctness & Traceability

- Correctness
  + Are we building the system right?
    - Formal specifications allow to verify presence of desired properties
      * Mathematical proof
      * Semi-automatic generation of test-cases
    - Faults (omissions!) in the specification are still possible



  + Are we building the right system?
    - (Some) formal specifications can be simulated / animated
      * May play the role of a prototype
      * Counterexamples to illustrate corner case behaviour

- Traceability
  + Requirements ⇔ System?
    - Formal specification is an intermediate representation
      * Traceability depends on usage and discipline

Summary(i)

You should know the answers to these questions

- Why is an UML class diagram a semi-formal specification?
- What is an automated theorem prover?
- What is the distinction between “partially correct” and “totally correct”?
- Give the mathematical definition for the *weakest precondition* of Hoare triple {P} S {Q}
- Why is it necessary to complement sequence diagrams with statecharts?
- What is the notation for the start and termination state on a state-chart? What is the notation for a guard expression on an event?
- What does it mean for a statechart to be
  (a) consistent, (b) complete, and (c) unambiguous?
- How does a formal specification contribute to the correctness of a given system?

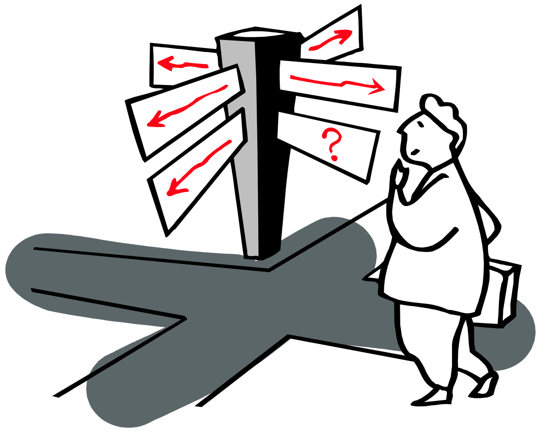
You should be able to complete the following tasks

- Use a theorem prover (Daphny) to prove that a given piece of code is correct.
- Create a statechart specification for a given problem.
- Given a statechart specification, derive a test model using path testing.

Summary (ii)

- Can you answer the following questions?
  - + How does domain modeling help to validate and analyze the requirements?
  - + What’s the problem with “god classes”?
  - + Why are many responsibilities, many collaborators and deep inheritance hierarchies suspicious?
  - + Can you explain how role-playing works? Do you think it helps in creative thinking?
  - + Can you compare Use Cases and CRC Cards in terms of the requirements specification process?
  - + Do CRC cards yield the best possible class design? Why not?
  - + Why are CRC cards maintained with paper and pencil instead of electronically?
  - + What would be the main benefits for thinking in terms of “system families” instead of “one-of-a-kind development? What would be the main disadvantages?
  - + Can you apply scrum to develop a product line? Argue your case.

CHAPTER 8 – Domain Modelling



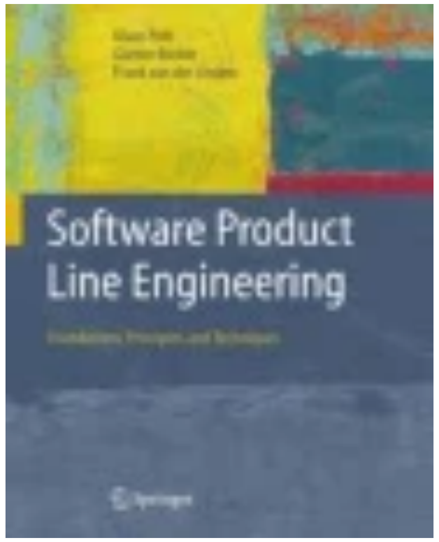
- Introduction
  - + When, Why, How, What
- CRC-Cards
  - + Problem Decomposition
    - Functional vs. Object-Oriented
    - Classes, Responsibilities & Collaborations, Hierarchies
  - + Group work
    - Creative thinking
    - Brainstorming & Role-playing
- Product Lines
  - + Commonalities and Variations
  - + Feature Diagrams
  - + Linux as a product line
  - + “clone and own”
    - Benefits drawbacks
    - Github
- Conclusion
  - + Correctness & Traceability

Literature (1/2)

- Books
  - [Ghez02], [Somm05], [Pres00]
    - Chapters on Specification / (OO)Analysis/ Requirements + Validation

- CRC Cards
  - [Booc94] Object-oriented analysis and design: with applications, Grady Booch, Addison-Wesley, 1994
    - > A landmark book on what object-oriented decomposition is about.
  - [Bell97] The CRC Card Book, David Bellin and Susan Suchman Simone, Addison-Wesley, 1997.
    - An easy to read and practical guide on how apply CRC cards in brainstorm sessions with end users.

- Product Lines
  - [Pohl2005] Software Product Line Engineering: Foundations, Principles and Techniques, 2005. Klaus Pohl, Günter Böckle, Frank J. van der Linden
    - An overview of all aspects of product line engineering (from domain modelling over testing to organisational aspects)



Literature (2/2)

- [Travis2019] “How the Boeing 737 Max Disaster Looks to a Software Developer" Gregory Travis. IEEE Spectrum, April 2019.
  + A sad example on how various political forces around a product ultimately leads to disastrous consequences

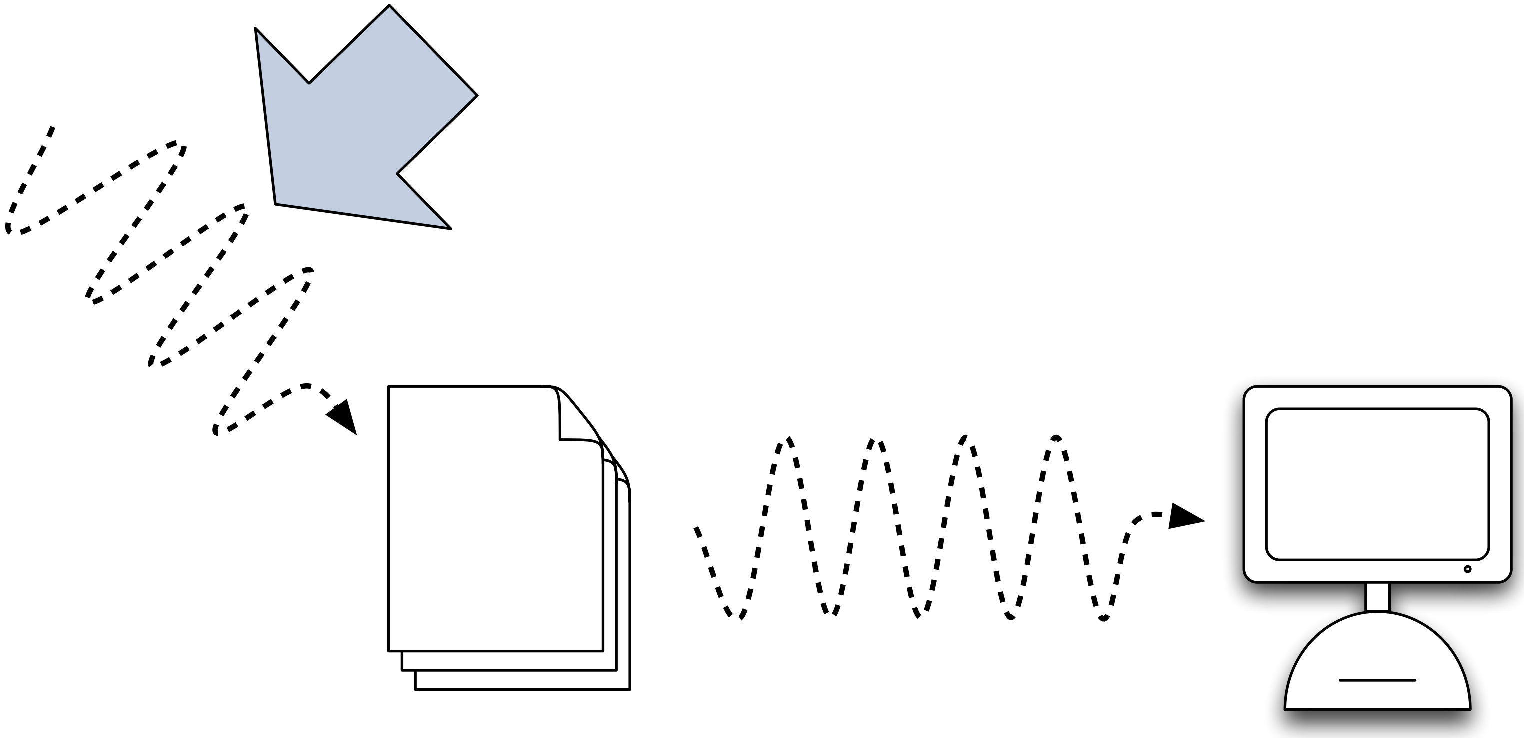
Product Lines

- “Product Line Hall of Fame”
  + http://splc.net/fame.html

- “Software Product Lines Online Tools”
  + http://www.splot-research.org/

- [She10] She, Steven; Lotufo, Rafael; Berger, Thorsten; Wasowski, Andrzej; Czarnecki, Krzysztof. The Variability Model of The Linux Kernel. Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010)
  + Illustrating large scale variability

When Domain Modeling?



A requirements specification must be *validated*

- Are we building the right system?

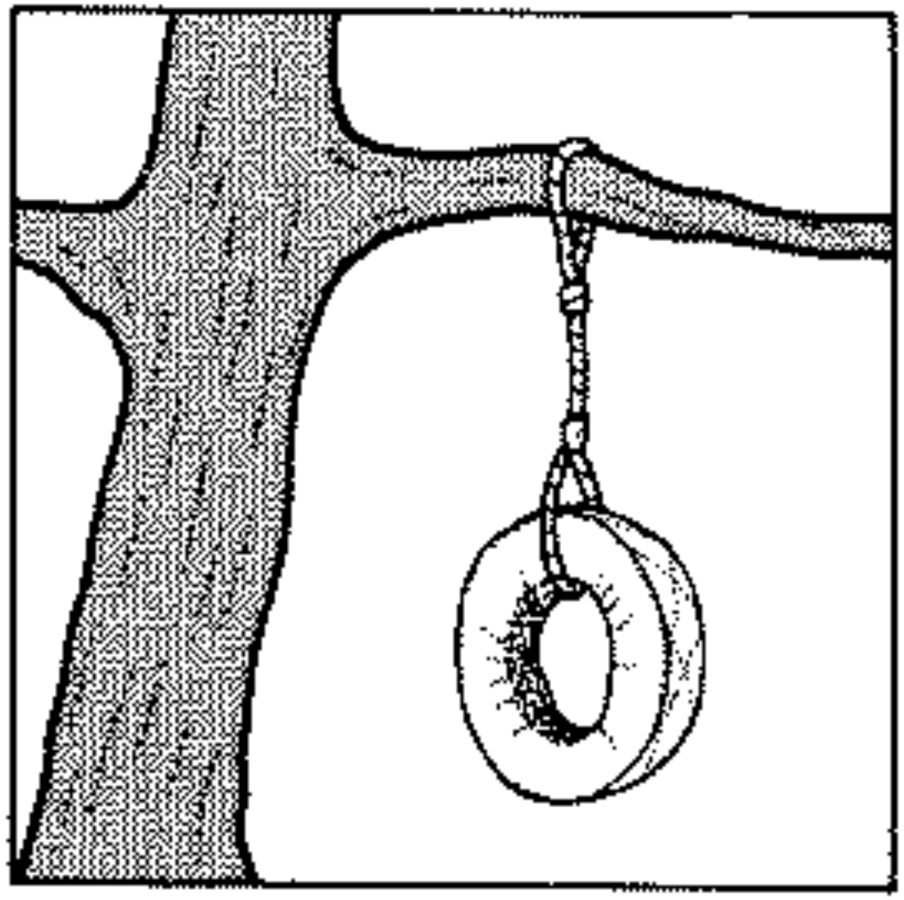
A requirements specification must be *analyzed*

- Did we understand the problem correctly?
  = Are we modeling the problem domain adequately?

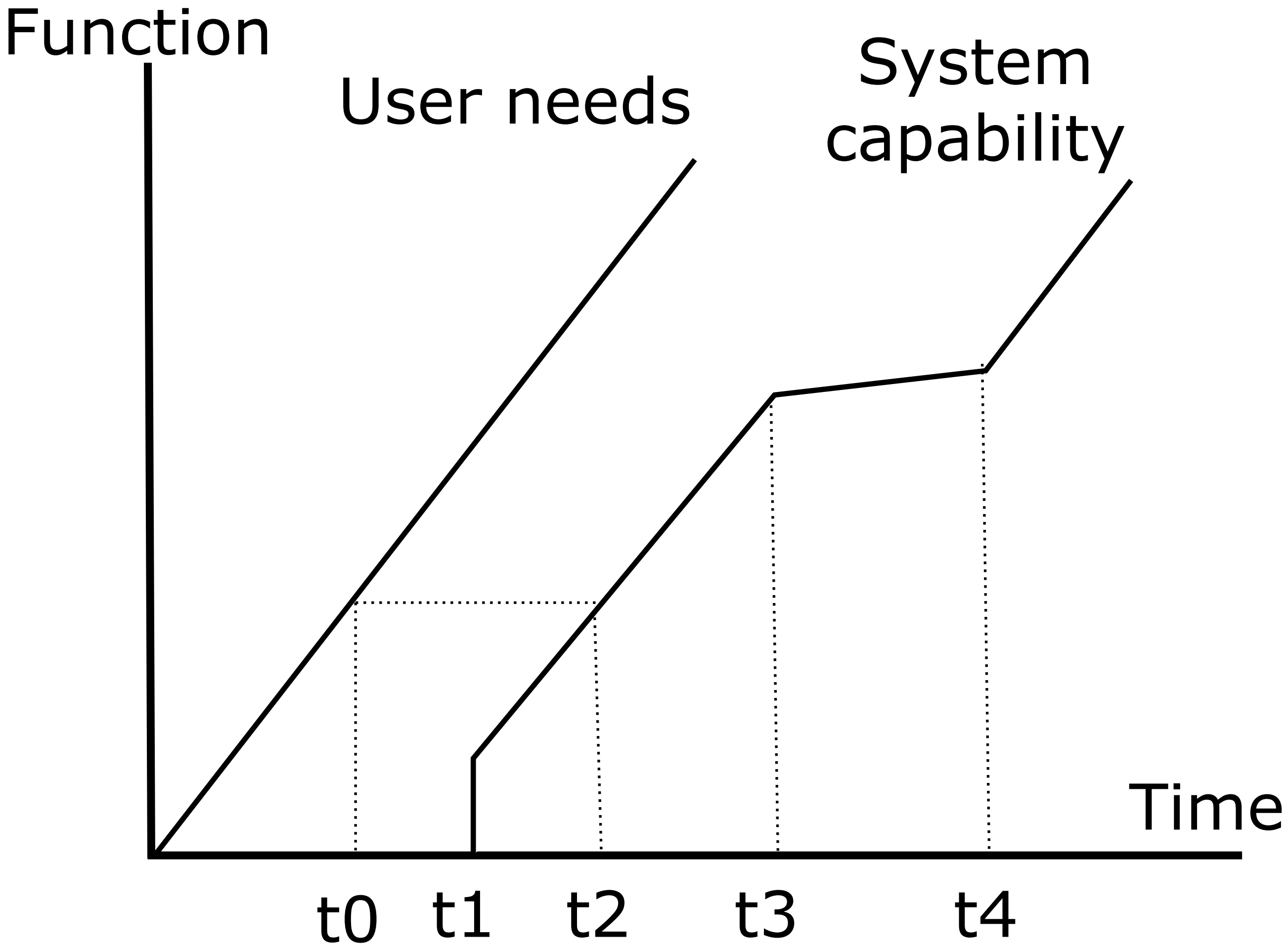
Why Domain Modeling?

The 30++ years of software development taught us one fundamental lesson...

- The customers don’t know what they want!
- And if they do, they will certainly change their mind.

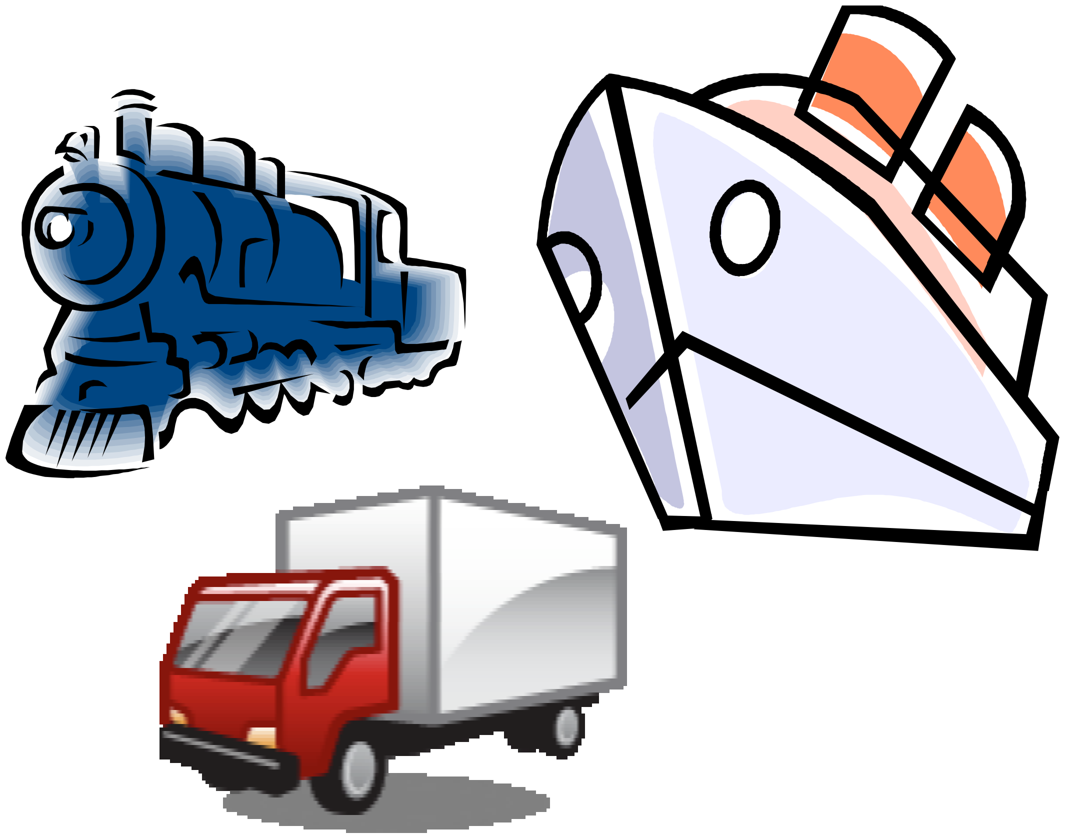


What the customer actually wanted



Why Use Cases are not Sufficient?

- Develop an information system for a transportation company in 1860.
  + “Pony Express” Use Cases
    - refresh horse
    - replace whip
    - clean pistol

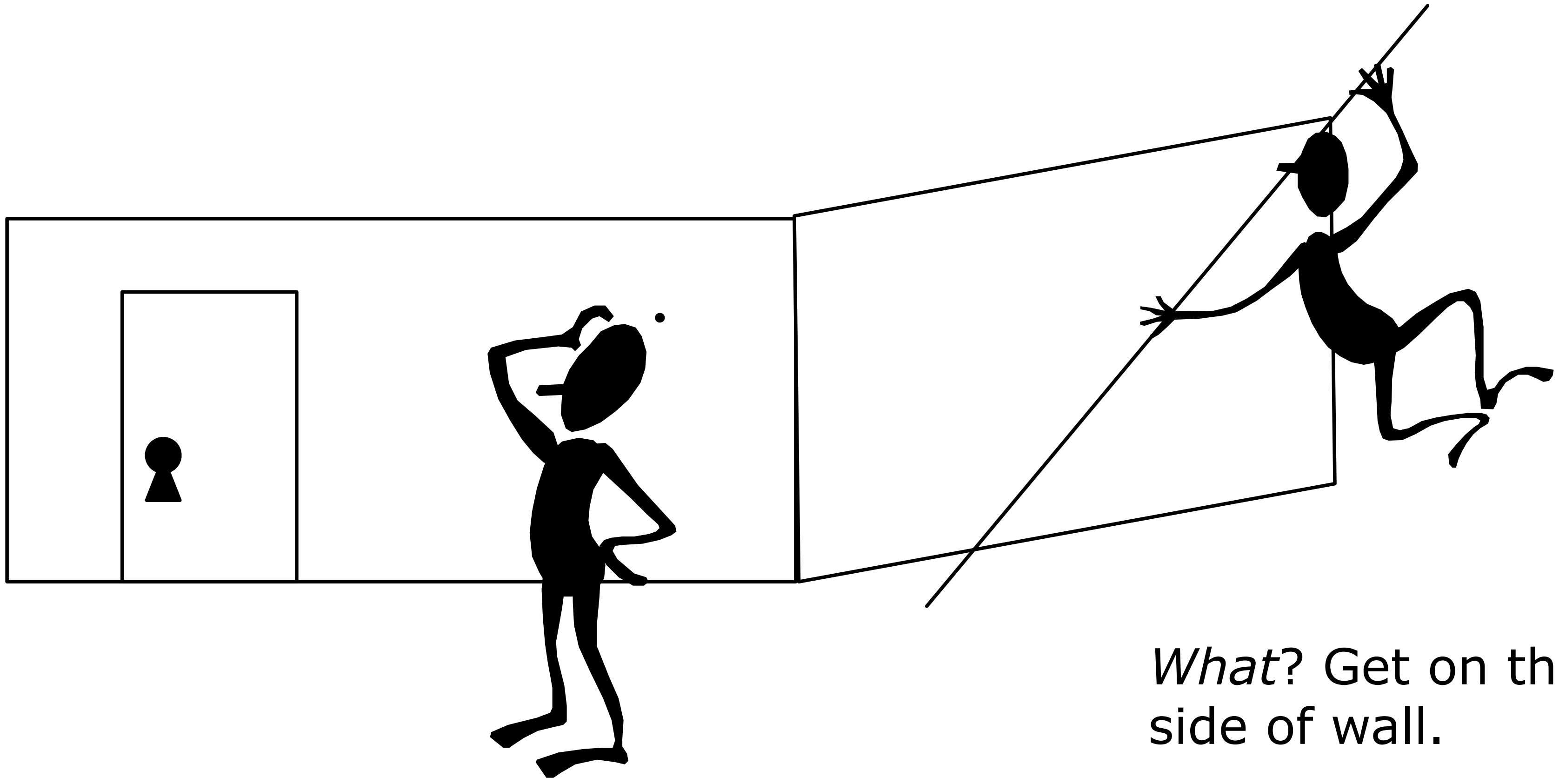


- 100 years later, “Pony Express” is still operating in the transportation business …
  + How about the Use Cases?
    - refresh horse
      ⇒ add fuel
    - replace whip
      ⇒ perform repair
    - clean pistol
      ⇒ include protection

How Domain Modeling?

Domain Models help to anticipate changes, are more robust.

- Focus on the what (goal), not on the how (procedure)!



What? Get on the other side of wall.

How? Open door, break lock, …
… jump over the wall

What is Domain Modeling?

Model of Problem Domain • *Requirements* Model + Focus on WHAT	vs. Model of Solution Domain • *Design* Model + Focus on HOW

- Examples
  + CRC Cards
    - Model the concepts in the problem domain in object-oriented terms.
      > Classes and Inheritance
  + Feature Diagrams
    - Model the requirements of a *family* of systems
      > Commonalities and variations

What are CRC Cards?

CRC = Class-Responsibility-Collaborations

<table>
<tr><td colspan="2">Class: Name</td></tr>
<tr><td colspan="2">superclass: list of superclasses
subclass: list of subclasses</td></tr>
<tr><td>responsibility 1
responsibility 2
…</td><td>collaborations required to achieve responsibility1
collaborations required to achieve responsibility2
…</td></tr>
</table>

+ a short description of the purpose of the class on the back of the card

- CRC Cards
  + compact, easy to manipulate, easy to modify or discard!
  + easy to arrange, reorganize
  + easy to retrieve discarded classes
- Usually CRC cards are not maintained electronically
  + May be used by computer illiterates

Problem Decomposition (1/2)

Object-Oriented Decomposition	Functional Decomposition
Decompose according to the objects a system must manipulate. ⇒ several coupled “is-a” hierarchies	Decompose according to the functions a system must perform. ⇒ single “subfunction-of” hierarchy

Example: Order-processing software for mail-order company	
Order - place - price - cancel Customer - name - address LoyalCustomer - reduction	OrderProcessing - OrderMangement • placeOrder • computePrice • cancelOrder - CustomerMangement • add/delete/update

Problem Decomposition (2/2)

Object-Oriented Decomposition	Functional Decomposition
⇒ distributed responsibilities	⇒ centralized responsibilities

Example: Order-processing software for mail-order company

```
Order::price(): Amount
  {sum := 0
  FORALL this.items do
    {sum := sum + item. price}
  sum:=sum-(sum*customer.reduction)
  RETURN sum
  }
```

```
Customer::reduction(): Amount
  { RETURN 0%}
LoyalCustomer::reduction(): Amount
  { RETURN 5%}
```

```
computeprice(): Amount
  {sum := 0
  FORALL this.items do
    sum := sum + item. price
  IF customer isLoyalCustomer THEN
    sum := sum - (sum * 5%)
  RETURN sum
  }
```

Quizz

Which one do you prefer? Why?

Object-Oriented Decomposition	Functional Decomposition
Example: Order-processing software for mail-order company	

Order::price(): Amount
{sum := 0
FORALL this.items do
{sum := sum + item. price}
sum:=sum-(sum*customer.reduction)
RETURN sum
}
	computeprice(): Amount
	{sum := 0
	FORALL this.items do
	sum := sum + item. price
	IF customer isLoyalCustomer THEN
	sum := sum - (sum * 5%)
	RETURN sum
	}

Customer::reduction(): Amount
{ RETURN 0%}
LoyalCustomer::reduction(): Amount
{ RETURN 5%}	

Functional vs. Object-Oriented

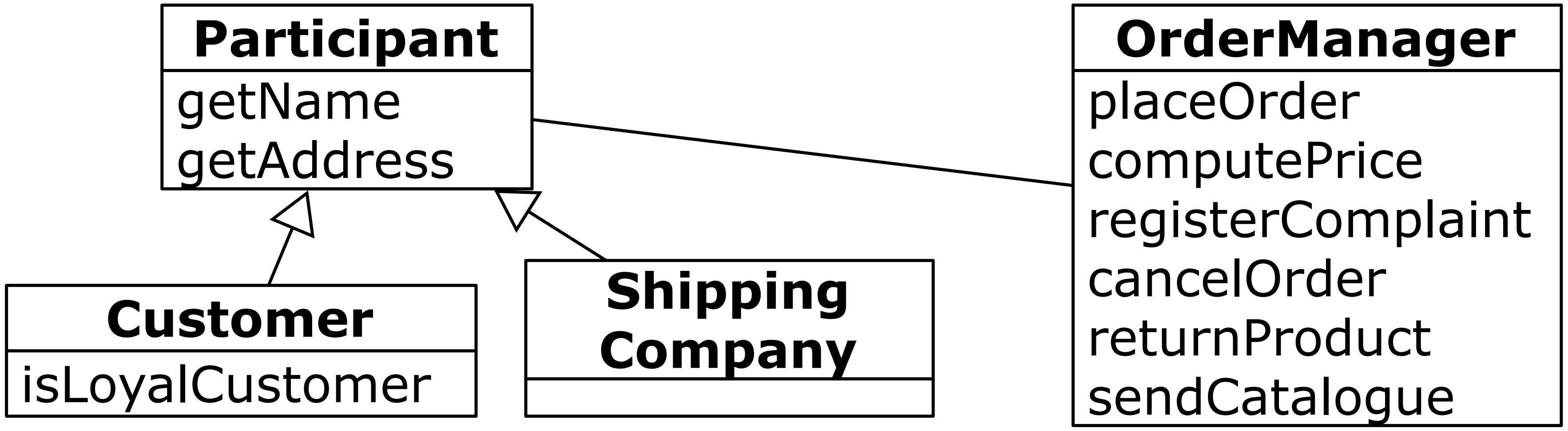
- Functional Decomposition
  + Good with stable requirements or single function (i.e., “waterfall”)
  + Clear problem decomposition strategy
  + However
    - Naive: Modern systems perform more than one function
      > What about “produceQuarterlyTaxForm”?
    - Maintainability: system functions evolve ⇒ cross-cuts whole system
      > How to transform telephone ordering into web order-processing?
    - Interoperability: interfacing with other system is difficult
      > How to merge two systems maintaining customer addresses?

How to find the objects?

- Object-Oriented Decomposition
  + Better for complex and evolving systems
  + Encapsulation provides robustness against typical changes

God Classes

- ... or how to do functional decomposition with an object-oriented syntax



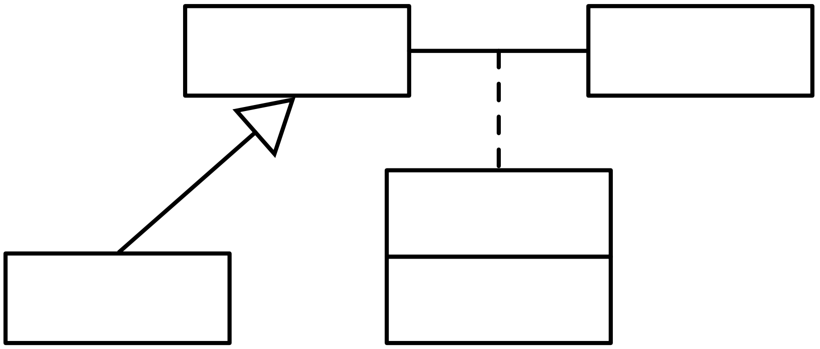
- Symptoms
  - + Lots of tiny “provider” classes, mainly providing accessor operations
    - most of operations have prefix “get”, “set”
  - + Inheritance hierarchy is geared towards data and code-reuse
    - “Top-heavy” inheritance hierarchies
  - + Few large “god” classes doing the bulk of the work
    - suffix “System”, “Subsystem”, “Manager”, “Driver”, “Controller”

Responsibility - driven Design in a Nutshell

- Responsibility-driven design is the analysis method using CRC Cards.

- How do you find objects and their responsibilities?
  + Use nouns & verbs in requirements as clues.
    - Noun phrases lead to objects
    - Verb phrases lead to responsibilities
  + Determine how objects collaborate to fulfill their responsibilities.
    - To collaborate objects will play certain roles

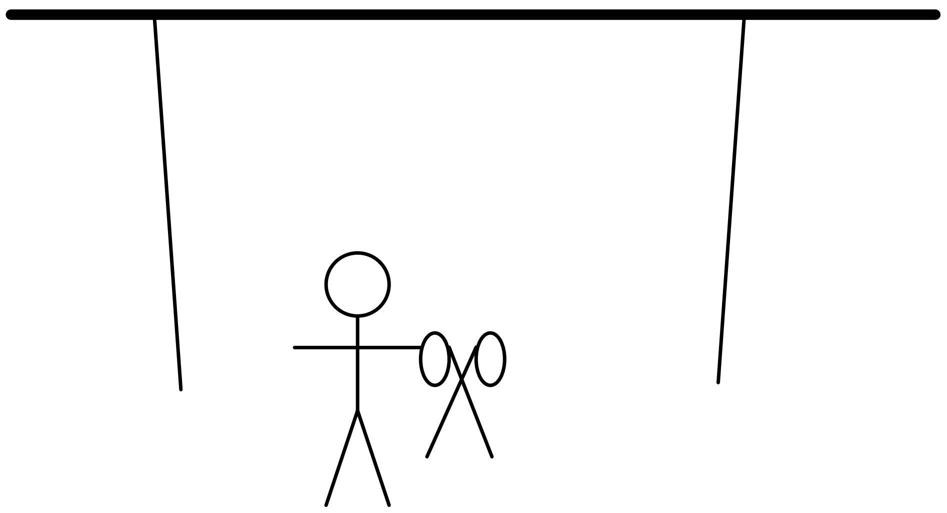
  + Why is this important?
    - Objects lead to classes
    - Responsibilities lead to operations
    - Collaborations & Roles lead to associations



  + Is it that simple?
    - No requires creative thinking!

Creative Thinking

Good problem decomposition requires creative thinking.



2 string puzzle

Within one large empty room, there are two long ropes are hanging from the ceiling. The ropes are too far away to reach the one while holding the other. A woman comes in holding a pair of scissors and she ties the ropes together.
How did she achieve this?

See Communications of the ACM, Vol. 43(7), July 2000, p. 113

Identifying Objects

+ Start with requirements specification/scope description/....
+ 1. Look for noun phrases:
  - separate into obvious classes, uncertain candidates, and nonsense
+ 2. Refine to a list of candidate classes. Some guidelines are:
  - Model physical objects — e.g. disks, printers
  - Model conceptual entities — e.g. windows, files
  - Choose one word for one concept —
    what does it mean within the domain?
  - Be wary of adjectives — does it really signal a separate class?
  - Be wary of missing or misleading subjects — rephrase in active voice
  - Model categories of classes — delay modeling of inheritance
  - Model interfaces to the system — e.g., user interface, program interfaces
  - Model attribute values, not attributes —
    e.g., Customer vs. Customer Address

Identifying Objects: Example (1/2)

“We are developing order-processing **software** for a **mail-order company** called National Widgets, which is a **reseller** of **products** purchased from various **suppliers**.

- Twice a year the **company** publishes a **catalogue of products**, which is mailed to **customers** and other **interested people**.
- **Customers** purchase **products** by submitting a **list of products** with **payment** to National Widgets. National Widgets fills the **order** and ships the **products** to the **customer’s address**.
- The order-processing **software** will track the **order** from the **time it is received** until the **product is shipped**.
- National Widgets will provide **quick service**. They should be able to ship a **customer’s order** by the fastest, most efficient means possible.”

Identifying Objects: Candidate Classes (2/2)

Nouns & Synonyms	Candidate Class Name
software	-: don’t model the system
mail-order company, company, reseller	Company (?: model ourselves)
products	Product (+: core concept)
suppliers	Supplier (+: core concept)
catalogue of products	Catalogue (+: core concept)
customers, interested people	Customer (+: core concept)
list of products, order, customer’s order	Order (+: core concept)
payment	Payment (+: core concept)
customer’s address	Address (?: customer’s attribute)
time it is received	-: attribute of Order
time product is shipped	-: attribute of Order
quick service	-: attribute of Company

*** Expect the list to evolve as analysis proceeds.

- Record why you decided to include/reject candidates
- Candidate Class list follows configuration management & version control

Responsibilities/Collaborations

- What are responsibilities?
  - + The public services an object may provide to other objects,
    - - the knowledge an object maintains and provides
    - - the actions it can perform
  - + ... not the way in which those services may be implemented
    - - specify what an object does, not how it does it
    - - don’t describe the interface yet, only conceptual responsibilities

- What are collaborations?
  - + other objects necessary to fulfill a responsibility
    - - when collaborating these other objects play a role
    - - to play this role, other objects must have certain responsibilities
  - + empty collaborations are possible
    - - can you argue this responsibility in terms of the class description?

Identifying Responsibilities

- To identify responsibilities (and the associated collaborations):
  - + Scenarios and Role Play.
    - - Perform scenario walk-throughs of the system where different persons “play” the classes, thinking aloud about how they will delegate to other objects.
  - + Verb phrase identification.
    - - Similar to noun phrase identification, except verb phrases are candidate responsibilities.
  - + Class Enumeration.
    - - Enumerate all candidate classes and come up with an initial set of responsibilities.
  - + Hierarchy Enumeration.
    - - Enumerate all classes in a hierarchy and compare how they fulfill responsibilities.

- Design guideline(s)
  - + *** Distribute responsibilities uniformly over classes
    (Classes with more than 12 responsibilities are suspicious)
  - + *** A class should have few collaborators
    (Classes with more than 8 collaborators are suspicious)

Hierarchies

+ Look for “kind-of” relationships
    - Liskov Substitution principle:
      You may substitute an instance of a subclass for any of its superclasses.
    - Does the statement “every subclass is a superclass” make sense
      > “Every Rectangle is a Square” vs. “Every Square is a Rectangle”
+ Factor out common responsibilities
    - Classes with similar responsibilities may have a common superclass
+ “kind-of” hierarchies are different from “part-of” relationships
    - Often, the whole will share responsibilities with its part
      (suggesting “kind-of” instead of “part-of”)
+ Name key abstractions
    - Not finding a proper name for the root is a symptom for an improper “kind-of” hierarchy Design guideline(s)

- Design guideline(s)
    + *** Avoid deep and narrow hierarchies
      Classes with more than 6 superclasses are suspicious

Brainstorming

- Team
  - + Keep small: five to six persons
  - + Heterogeneous:
    - 2 domain-experts (involved in day-to-day work; not management)
    - 2 analysts (build connections, abstractions and metaphors)
    - 1 experienced OO-designer (programmer ⇒ involvement)
    - 1 facilitator (chairs the meeting)

- Tips
  - + All ideas are potentially good (i.e., may trigger the creative thinking)
  - + No censorship (even on yourself), no rejection
  - + Think fast, ponder later
  - + Produce as many ideas as possible
  - + Give every voice a turn
  - + round-robin (with an optional “pass” policy)

- Design Guideline(s)

  ** Use white-boards and paper CRC Cards for smooth communication.

Role-playing

- Role-playing is a way to achieve common understanding between all parties involved (domain experts, analysts, ...)

- Basic Steps
  + 1. Create list of scenarios
  + 2. Assign Roles
  + 3. Each member receives a number of CRC Cards
  + 4. Repeat
  + 4.1 Rehearse Scenarios
    - Script = Responsibilities on CRC Cards
  + 4.2 Correct CRC Cards and revise scenarios
    - Rehearsals will make clear which parts are confusing
  + 4.3 Until scenarios are clear
  + 5. Perform final scenario

- Guideline(s)

*** For tips and techniques concerning role-play, see [Bell97]

Role-playing: Example

USE CASE 5	Place Order
Goal in Context	Customer issues request by phone to National Widgets; expects goods shipped and to be billed.

Class: Customer	
provideInfo	CustomerRep

Class: CustomerRep	
acceptCall	Customer

Class: Catalogue	

Class: Product	

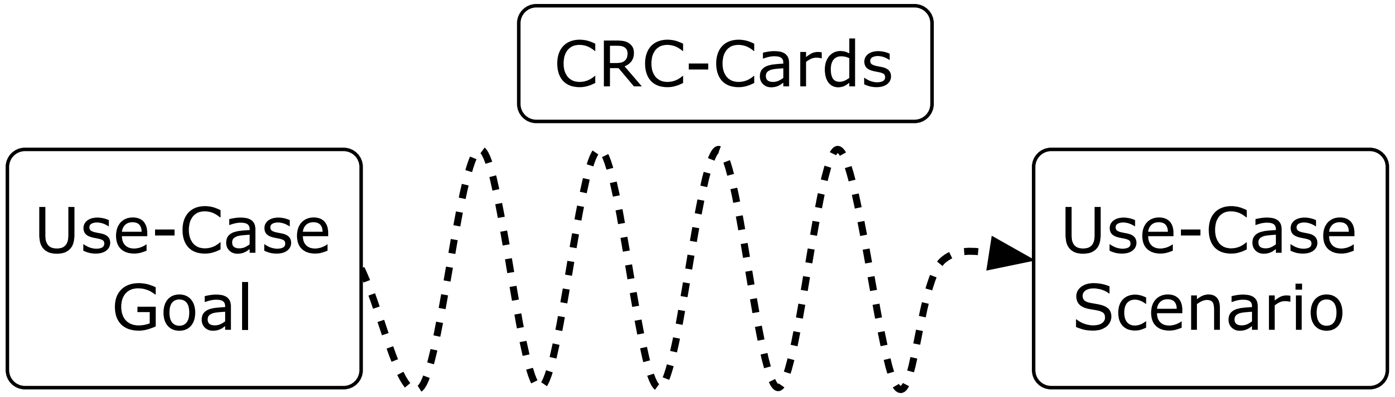
Class:	

Class:	

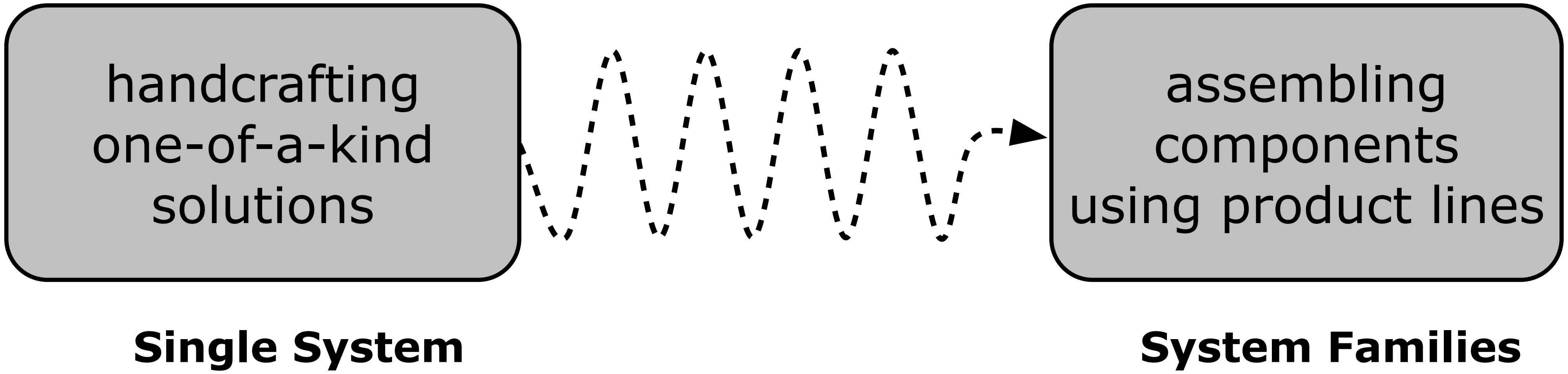
Use cases versus CRC-Cards

- Use cases are a requirements *specification* technique.
- CRC Cards are a requirements *validation* technique.

Use cases & CRC cards complement each other!



Families of Systems


handcrafting one-of-a-kind solutions
assembling components using product lines
Single System
System Families

Examples from “Product Line Hall of Fame” (http://splc.net/fame.html)

- Mobile phones (Nokia) [1 new phone every day!]
- Television sets, medical systems (Philips)
- Gasoline Systems Engine Control (Bosch)
- Telephone switches (Philips, Lucent)
- …

Feature Models

- Feature
  + a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system

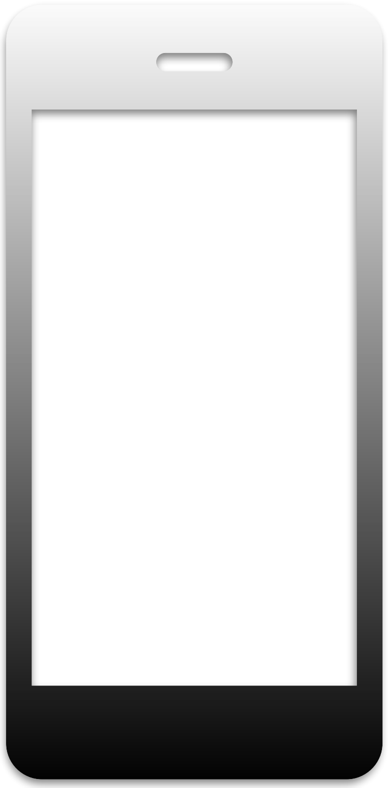
- Feature Models:
  + define a set of reusable and configurable requirements for specifying the systems in a domain
  + = a model that defines features and their dependencies, typically in the form of a *feature diagram*
  + = defines the commonalities and variations between the members of a *software product line*

- Feature Diagram
  + = a visual notation of a feature model, which is basically an and-or tree. (Other extensions exist: cardinalities, feature cloning, feature attributes, …)

- Product Line
  + = a family of products designed to take advantage of their common aspects and predicted variabilities

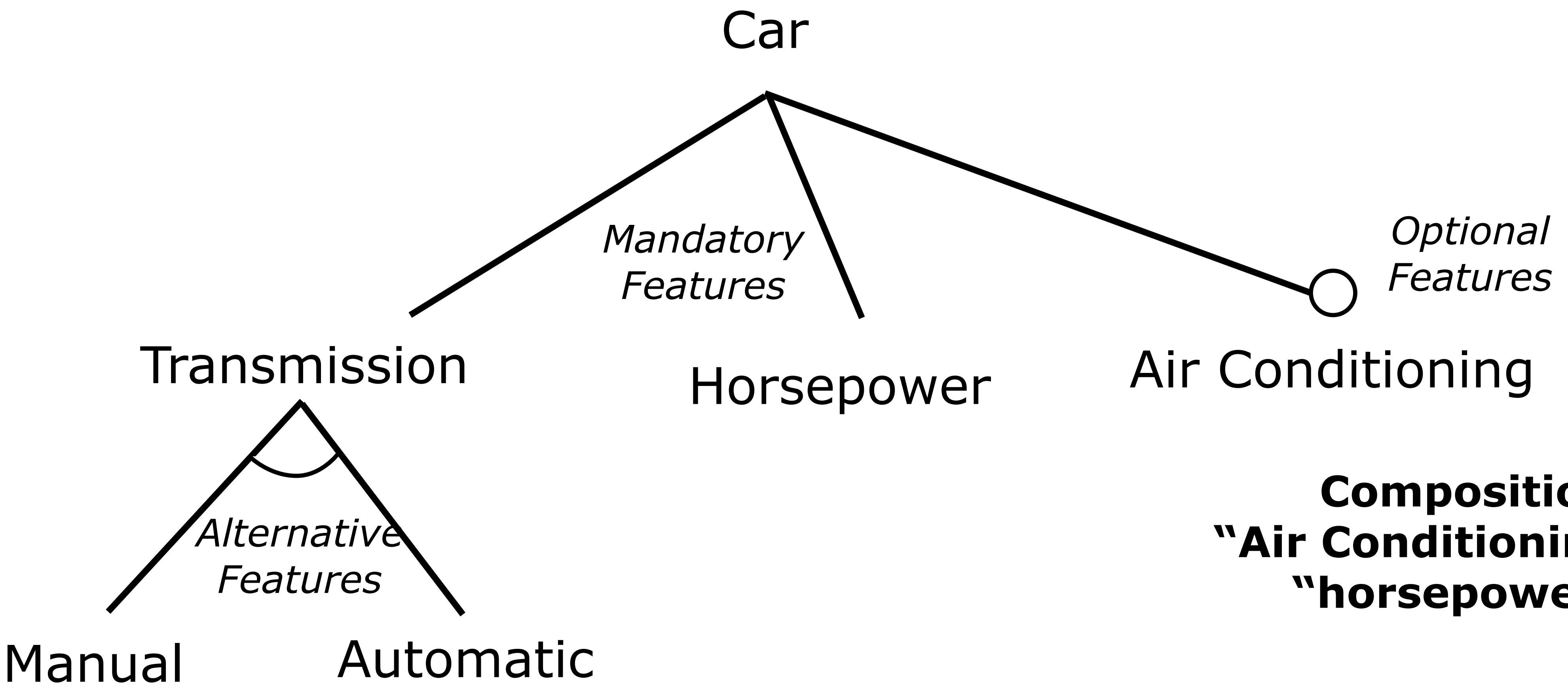
What is a Feature?

- Feature
  + a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system



List the 3 most important features of your phone.

Feature Diagram: Example

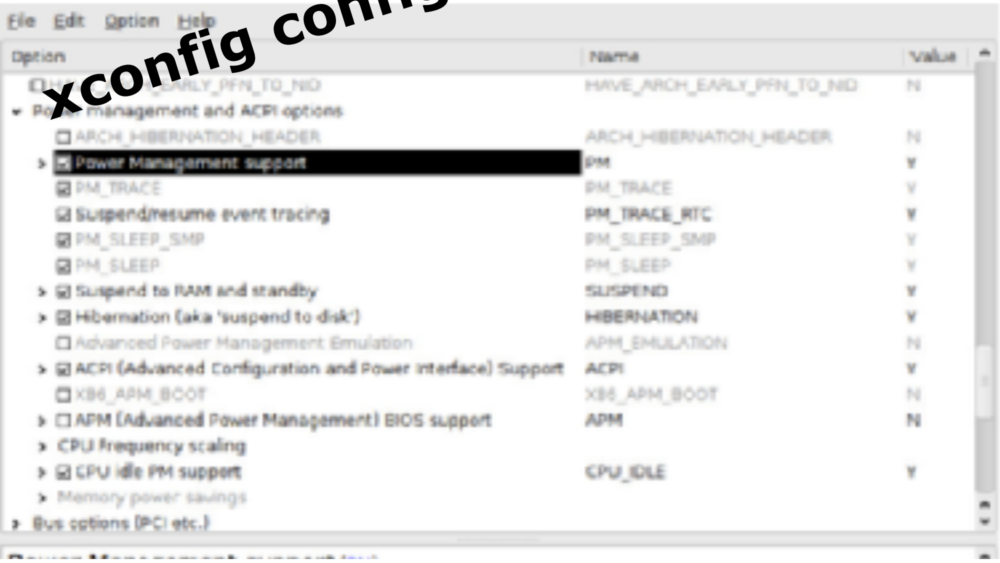


Composition Rule:
“Air Conditioning” requires
“horsepower” > 100

Rationale:
“Manual” more fuel efficient.

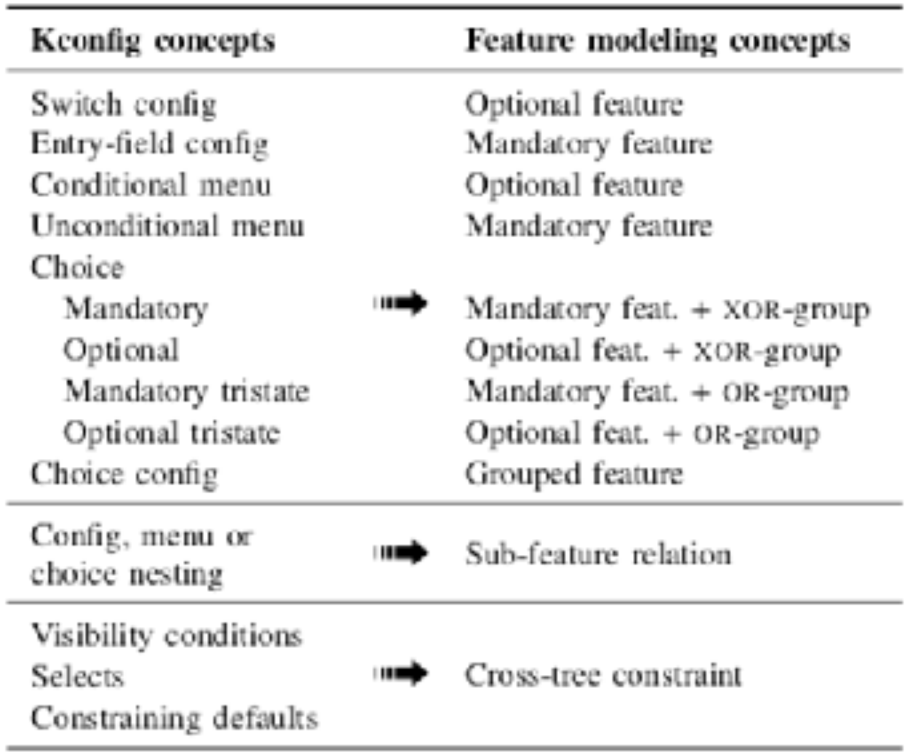
Linux as a Product Line

xconfig configurator

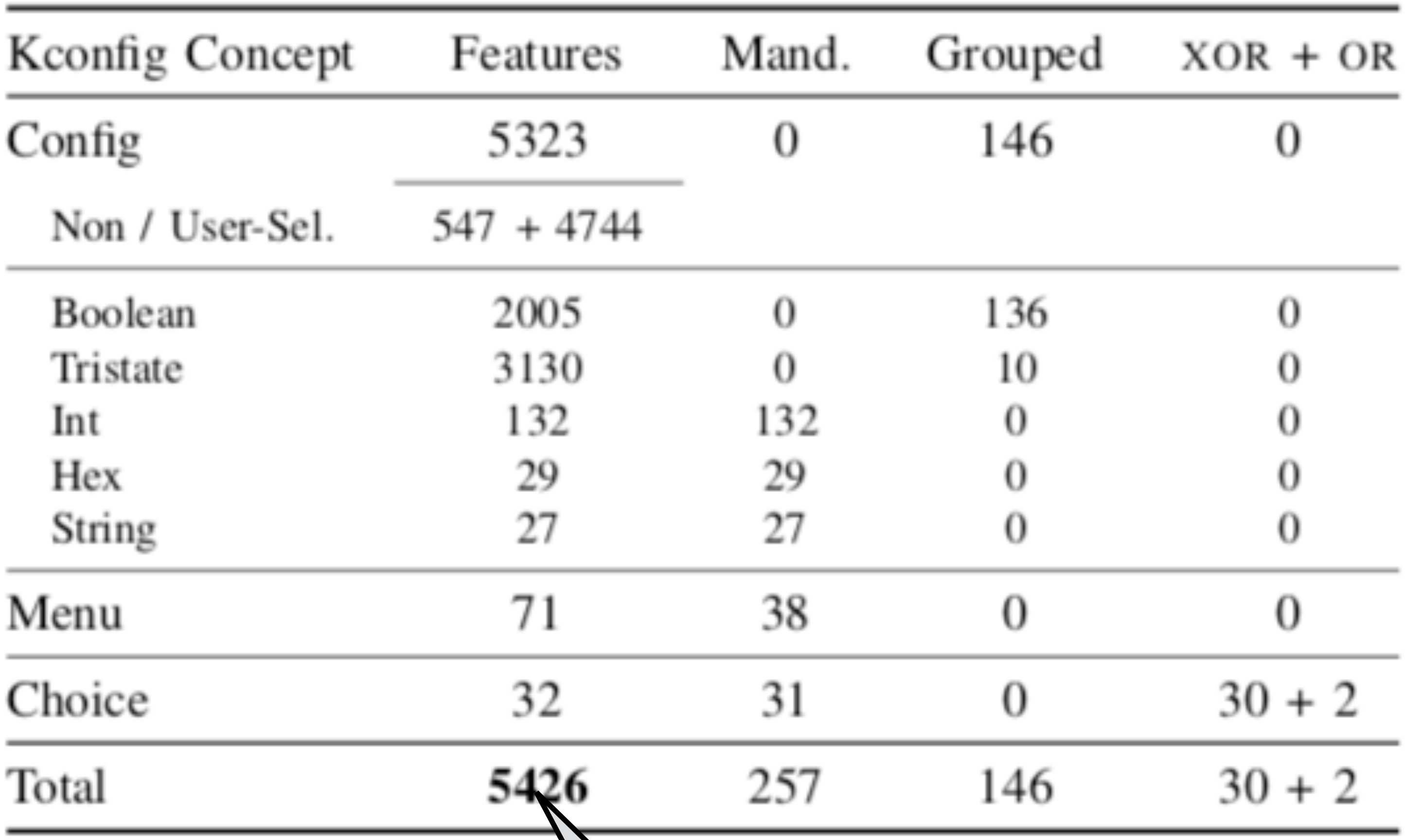


File Edit Option Help

Option	Name	Value
☐ HAVE_ARCH_EARLY_PFN_TO_NID	HAVE_ARCH_EARLY_PFN_TO_NID	N
▾ Power management and ACPI options		
☐ ARCH_HIBERNATION_HEADER	ARCH_HIBERNATION_HEADER	N
› ☑ Power Management support	PM	Y
☑ PM_TRACE	PM_TRACE	Y
☑ Suspend/resume event tracing	PM_TRACE_RTC	Y
☑ PM_SLEEP_SMP	PM_SLEEP_SMP	Y
☑ PM_SLEEP	PM_SLEEP	Y
› ☑ Suspend to RAM and standby	SUSPEND	Y
› ☑ Hibernation (aka 'suspend to disk')	HIBERNATION	Y
☐ Advanced Power Management Emulation	APM_EMULATION	N
› ☑ ACPI (Advanced Configuration and Power Interface) Support	ACPI	Y
☐ X86_APM_BOOT	X86_APM_BOOT	N
› ☐ APM (Advanced Power Management) BIOS support	APM	N
› CPU Frequency scaling		
› ☑ CPU idle PM support	CPU_IDLE	Y
› Memory power savings		
› Bus options (PCI etc.)		



Kconfig concepts		Feature modeling concepts
Switch config		Optional feature
Entry-field config		Mandatory feature
Conditional menu		Optional feature
Unconditional menu		Mandatory feature
Choice		
Mandatory	➡	Mandatory feat. + XOR-group
Optional		Optional feat. + XOR-group
Mandatory tristate		Mandatory feat. + OR-group
Optional tristate		Optional feat. + OR-group
Choice config		Grouped feature
Config, menu or choice nesting	➡	Sub-feature relation
Visibility conditions Selects Constraining defaults	➡	Cross-tree constraint



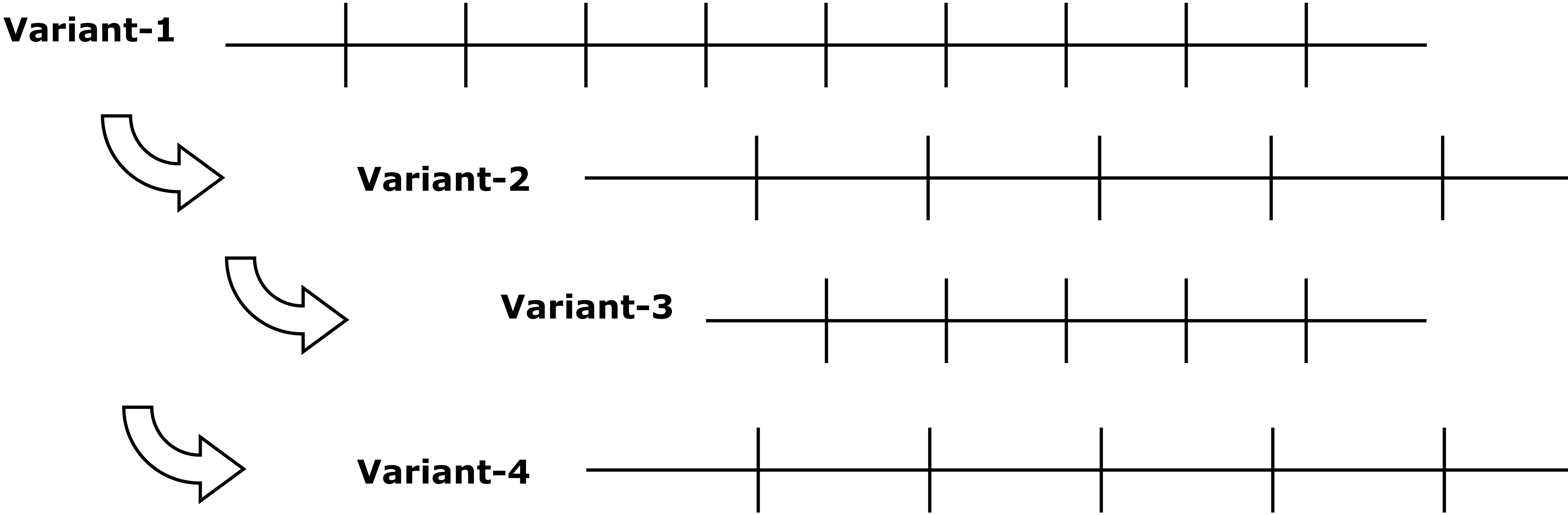
Kconfig Concept	Features	Mand.	Grouped	XOR + OR
Config	5323	0	146	0
Non / User-Sel.	547 + 4744			
Boolean	2005	0	136	0
Tristate	3130	0	10	0
Int	132	132	0	0
Hex	29	29	0	0
String	27	27	0	0
Menu	71	38	0	0
Choice	32	31	0	30 + 2
Total	**5426**	257	146	30 + 2

The 2.6.28.6 version of the kernel contains more than 5000 features!

© [She10] She, Steven; Lotufo, Rafael; Berger, Thorsten; Wasowski, Andrzej; Czarnecki, Krzysztof. The Variability Model of The Linux Kernel. Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010)

“clone and own”

Cloning an existing product variant, then modifying it to add and/or remove some functionalities, in order to obtain a new product variant.



Variant-1

Variant-2

Variant-3

Variant-4

The good, the bad and the ugly

Advantages

Efficiency
- Saves time and reduces costs
- Provides independence
- Readily available

Disadvantages

Overhead
- Propagating changes
- Adapting the clone is difficult
- Repetitive tasks are common
    - Bug fixing
- Which variant to clone from?

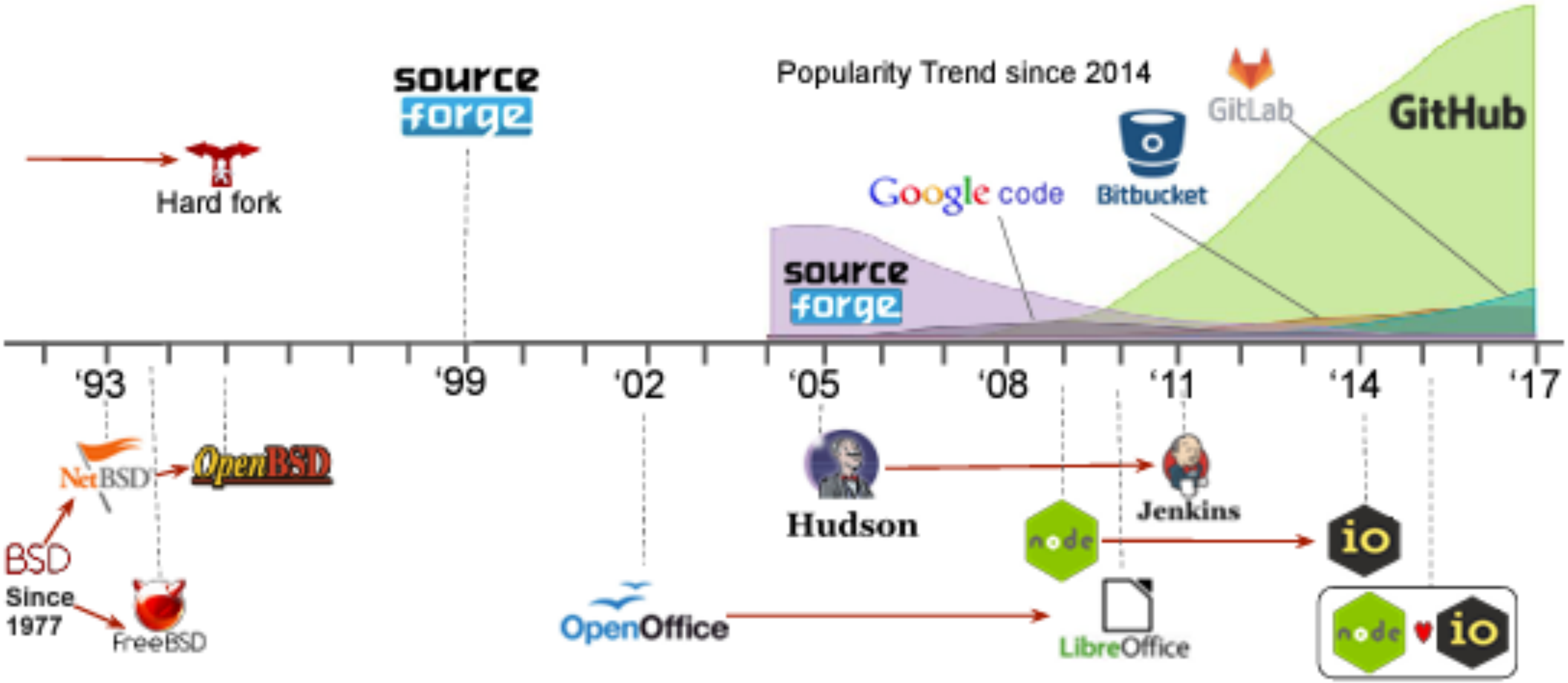
Barriers

Short-Term Thinking
- Lack of Planning
- Lack of Resources
- Unawareness

(Lack of) Governance
- Lack of reuse tracking
- Lack of organizational roles and processes
- Lack of measurement

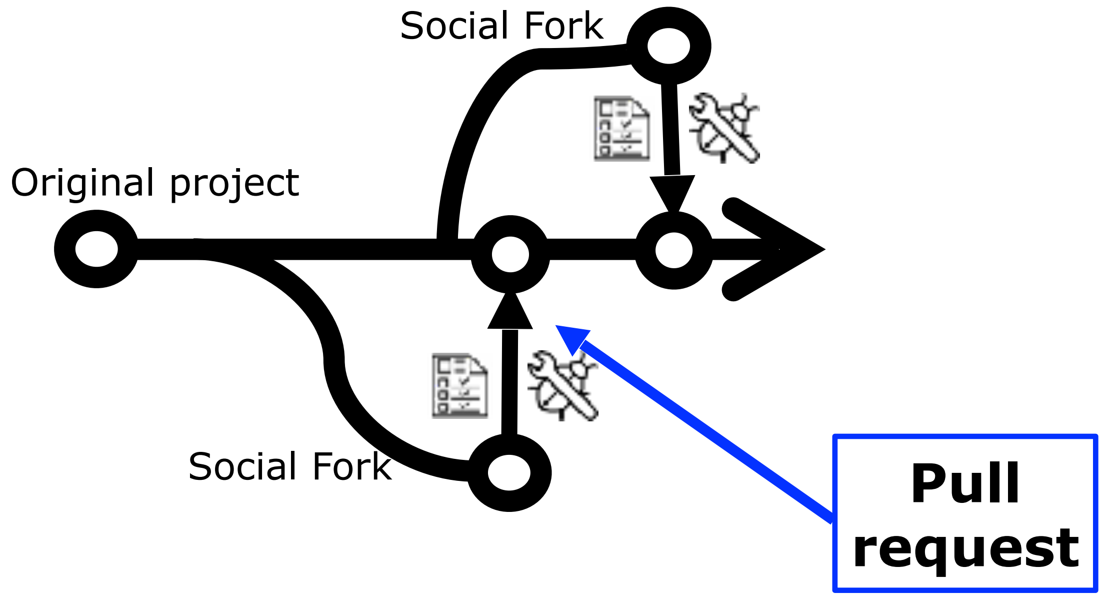
[Dubi13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13). IEEE Computer Society, Washington, DC, USA, 25-34.

Social Coding Platforms (git based)



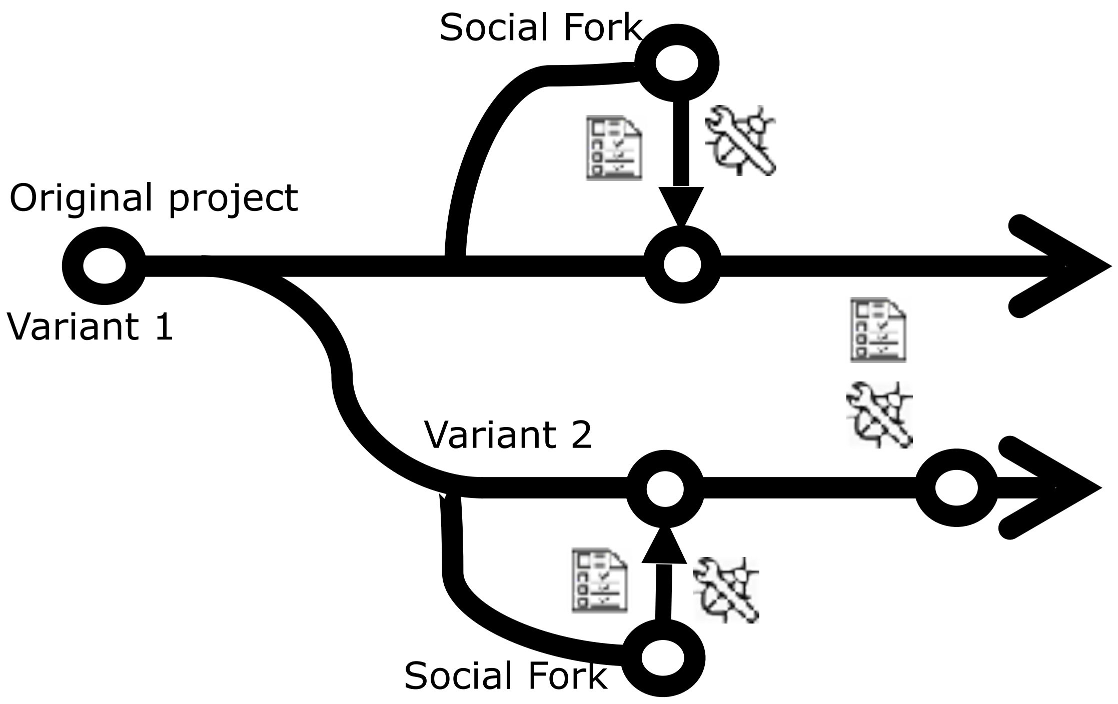
Social Forks versus Variant Forks

Social Forks



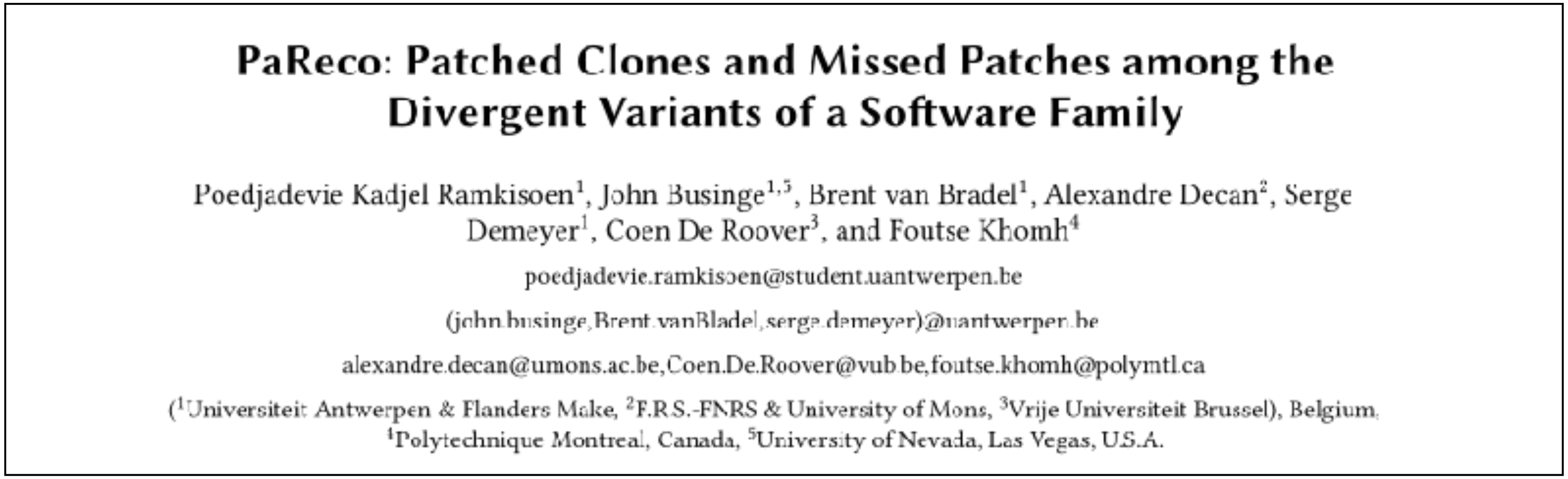
1. Fork is created
2. Modifications (branching)
   + Bug fix, security patch, new features
3. Pull request (merge in main branch)
4. Social fork ceases to be maintained

Variant Forks



1. Fork(s) are created
2. Maintained separately
   + Via social forks
3. Duplicated effort
   + bug fixes, security patches

Master Student Work



PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family

Poedjadevie Kadjel Ramkisoen[1], John Businge[1,5], Brent van Bradel[1], Alexandre Decan[2], Serge Demeyer[1], Coen De Roover[3], and Foutse Khomh[4]

poedjadevie.ramkisoen@student.uantwerpen.be

(john.businge,Brent.vanBladel,serge.demeyer)@uantwerpen.be

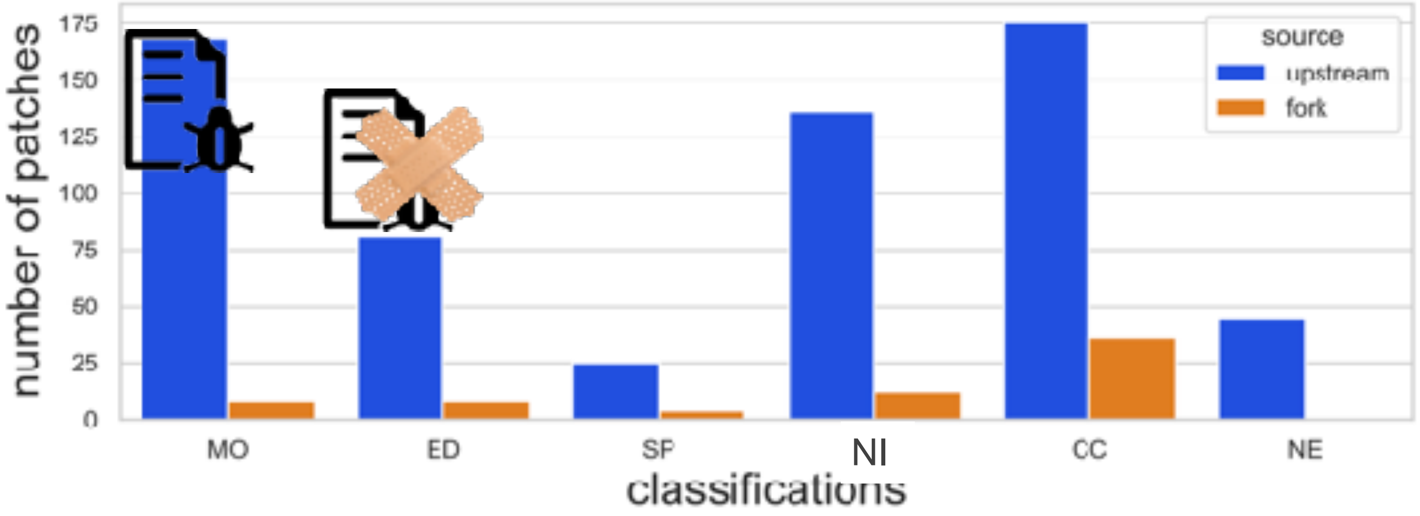
alexandre.decan@umons.ac.be,Coen.De.Roover@vub.be,foutse.khomh@polymtl.ca

([1]Universiteit Antwerpen & Flanders Make, [2]F.R.S.-FNRS & University of Mons, [3]Vrije Universiteit Brussel), Belgium,
[4]Polytechnique Montreal, Canada, [5]University of Nevada, Las Vegas, U.S.A.

Mined 8,323 patches from 364 source variants on GitHub

- Classify into
  + Missed Opportunity
    - fix applied in one variant but not the other
  + Effort Duplication
    - fix applied in both variants
  + …

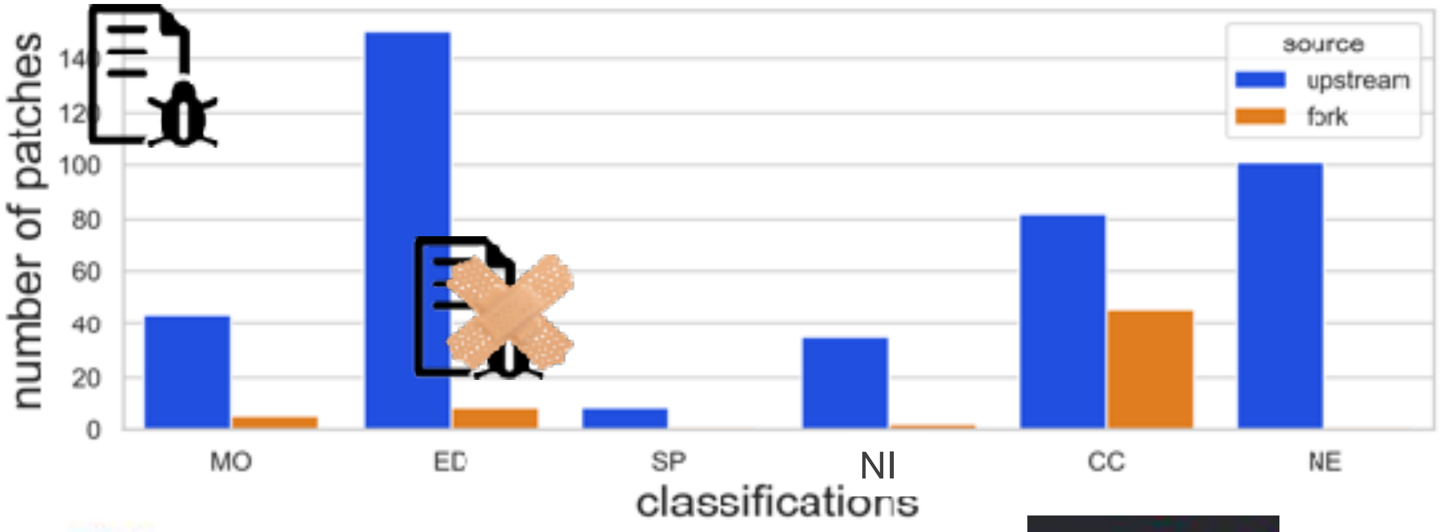
2 examples



- MO – Missed opportunity
- ED – Effort duplication
- SP – Split Patch (Both buggy and patched lines)
- NI – Non Interesting
- CC – Unhandled programming language
- NE – Missing file in target
- EE – Error

apache/kafka (upstream)

linkedin/kafka (fork)



microsoft/azure-tools-for-java (upstream)

JetBrains/azure-tools-for-intellij (fork)

Correctness & Traceability

- Correctness
  + Are we building the system right?
    - Good maintainability via a robust model of the problem domain.
      > Specifying the “what” not the “how”

- Are we building the right system?
  + Model the problem domain from the customer perspective
  + Role-playing scenarios helps to *validate* use cases
    - Paper CRC Cards are easy to reorganize
  + Feature Diagrams focus on commonalities/variations
    - Makes product differences (and choices) explicit

- Traceability
  + Requirements ⇔ System?
    - Via proper naming conventions
    - Especially names of classes and operations

Summary (i)

- You should know the answers to these questions
  + Why is it necessary to validate and analyze the requirements?
  + What’s the decomposition principle for functional and object-oriented decomposition?
  + Can you give the advantages and disadvantages for functional decomposition? What about object-oriented decomposition?
  + How can you recognize “god classes”?
  + What is a responsibility? What is a collaboration?
  + Name 3 techniques to identify responsibilities.

  + What do feature models define?
  + Give two advantages and disadvantages of a “clone and own” approach
  + Explain the main difference between a social fork and a variant fork

  + How does domain modeling help to achieve correctness? Traceability?

- You should be able to complete the following tasks
  + Apply noun identification & verb identification to (a part of) a requirements specification.
  + Create a feature model for a series of mobile phones.

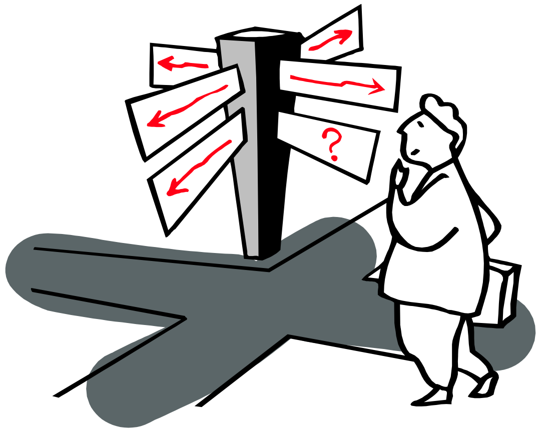
Summary(ii)

Can you answer the following questions?

- (Based on the article “A Formal Approach to Constructing Secure Air Vehicle Software”.)
  + What is according to you the most effective means to achieve “provably secure against cyberattacks”?
- Why is it likely that you will encounter formal specifications?
- Explain why we need both the loop variant and the loop invariant for proving total correctness of a loop?
- What do you think happened with the bug report on the broken Java.utils.Collection.sort ()? *Why* do you think this happened?
- Explain the relationship between “Design By Contract” on the one hand “State based specifications” on the other hand.
- Explain the relationship between “Testing” on the one hand and “State based specifications” on the other hand.
- You are part of a team build a fleet management system for drones transporting medical goods between hospitals. You must secure the system against cyber-attacks. Your boss asks you to look into formal specs; which ones would you advise and why?

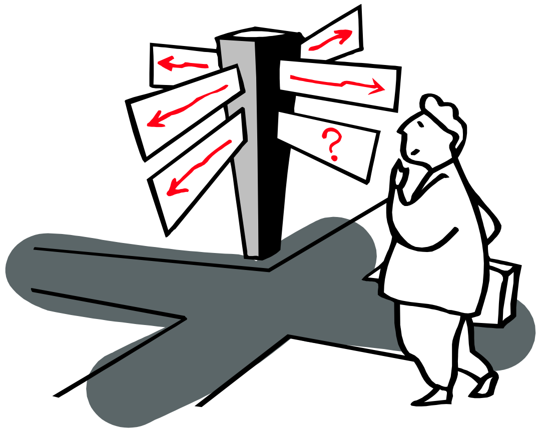
Universiteit restaurant: Kwaliteit of Niet?

CHAPTER 9 – Software Quality

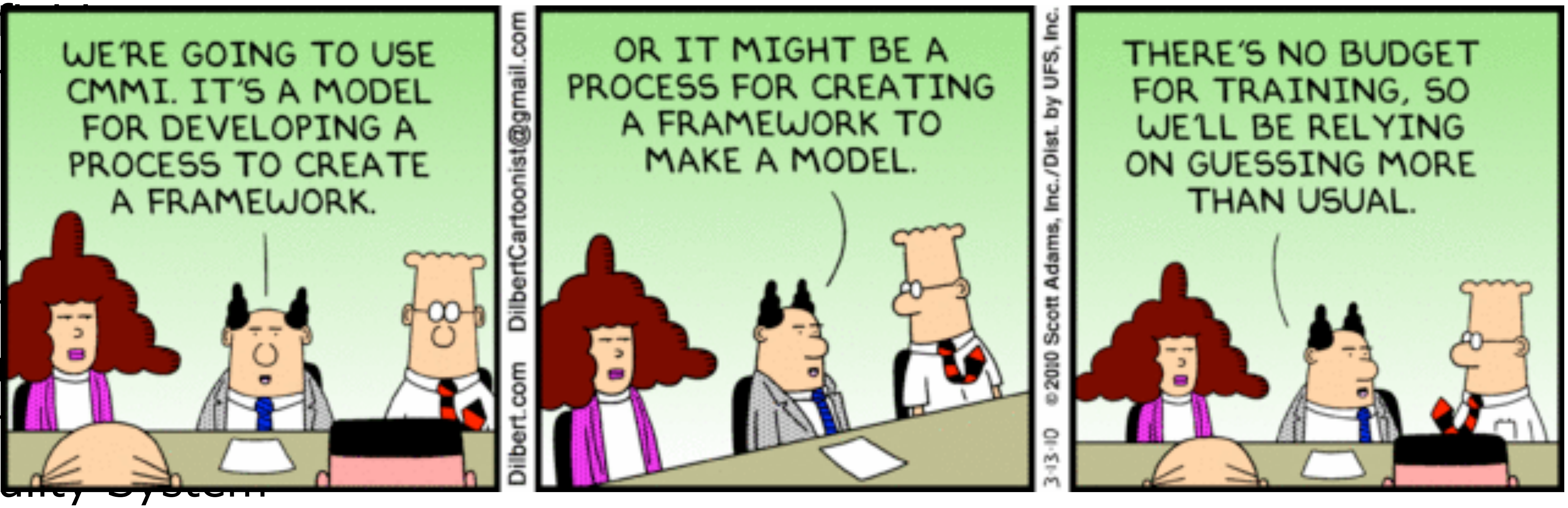


- Introduction
  + When, Why and What?
  + Product & Process Attributes
  + Internal & External Attributes
- Typical Quality Attributes
  + Overview
  + Definitions
- Quality Control
  + Quality Control Assumption
  + Quality Plan
  + Reviews & Inspections
- Quality in Scrum
  + Continuous Improvement
- Quality Standards
  + Quality System
  + ISO 9000, CMM, CMMI
- Conclusion

CHAPTER 9 – Quality Control



- Introduction
  - + When, Why and What?
  - + Product & Process Attributes
  - + Internal & External Attributes
- Typical Quality Attributes
  - + Overview
  - + Def
- Quali
  - + Qu
  - + Qu
  - + Rev
- Quali
  - + Co
- Quali
  - + Qu
  - + ISO 9000, CMM, CMMI
- Conclusion


WE'RE GOING TO USE CMMI. IT'S A MODEL FOR DEVELOPING A PROCESS TO CREATE A FRAMEWORK.
Dilbert.com DilbertCartoonist@gmail.com
OR IT MIGHT BE A PROCESS FOR CREATING A FRAMEWORK TO MAKE A MODEL.
3-13-10 ©2010 Scott Adams, Inc./Dist. by UFS, Inc.
THERE'S NO BUDGET FOR TRAINING, SO WE'LL BE RELYING ON GUESSING MORE THAN USUAL.

© dilbert.com

Literature

+ [Ghez02] In particular, chapter “Software: Its Nature and Qualities”
+ [Pres00] In particular, chapter “Software Quality Assurance”
+ [Somm05] In particular, chapters “Quality Management” & “Process Improvement”

- Web-Resources
    + ISO [http://www.iso.org/]
        - ISO 9000 family
            > [https://www.iso.org/iso-9001-quality-management.html]
    + CMM (Capability Maturity Model)
        - Paulk, Mark C.; Weber, Charles V; Curtis, Bill; Chrissis, Mary Beth (February 1993). "Capability Maturity Model for Software (Version 1.1)" (PDF). Technical Report. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. CMU/SEI-93-TR-024 ESC-TR-93-177.
    + CMMI® (Capability Maturity Model Integration)
        - [https://cmmiinstitute.com]

Famous Quality Incidents

Tacoma Narrows Bridge

Mature engineering disciplines learn from their mistakes

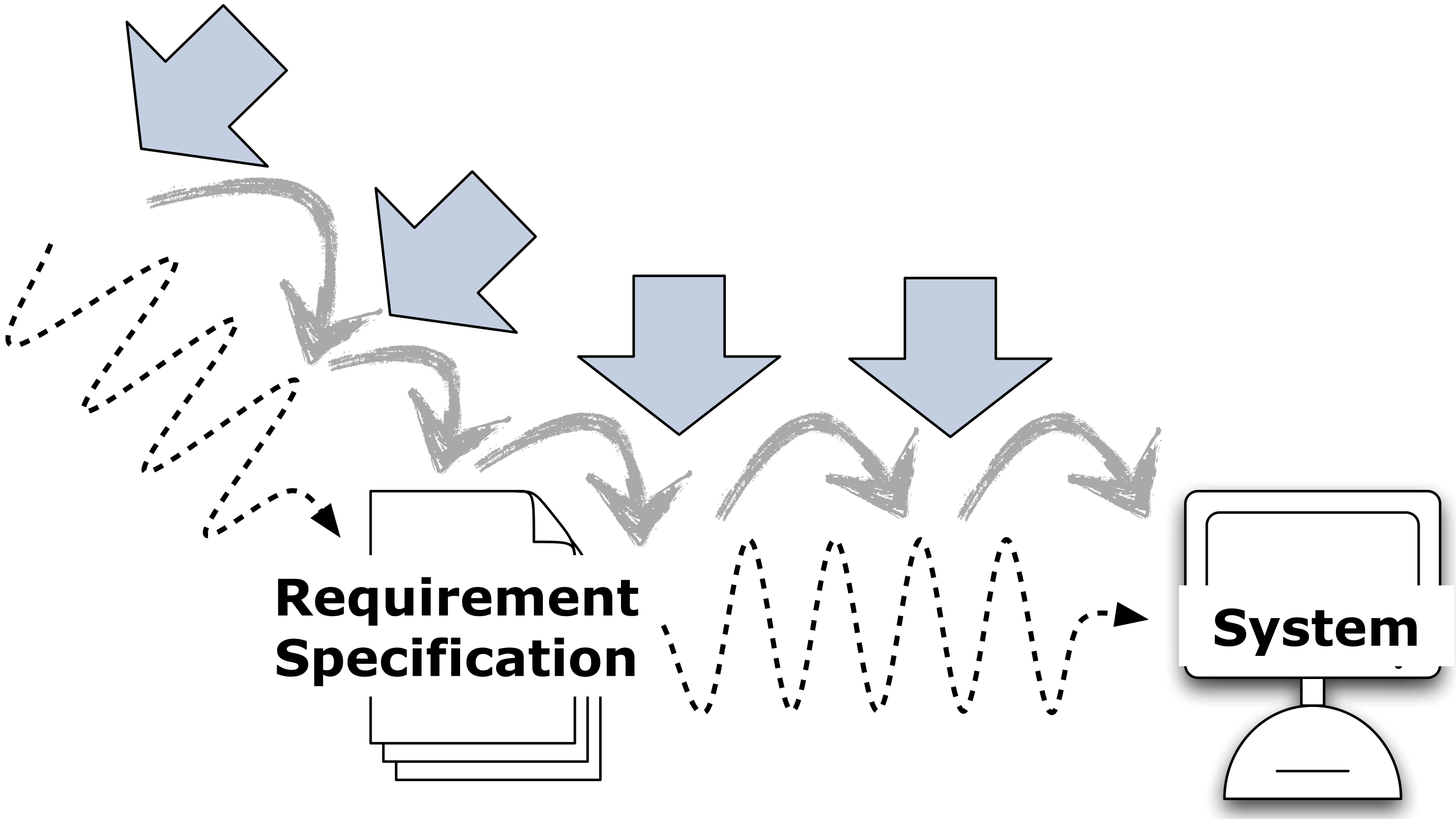
Denver International Airport Baggage Handling System

Ariane 5

UNITED STATES DEPARTMENT OF JUSTICE
GLOBAL INFORMATION SHARING
SENTINEL
TO PROTECT AND DEFEND THE NATION
FEDERAL BUREAU OF INVESTIGATION
DEPARTMENT OF JUSTICE
FEDERAL BUREAU OF INVESTIGATION

FBI Sentinel Project

When Quality Control?



quality in the final system ⇔ control quality of all intermediate steps.

However ... Quality control ⇎ High-quality system

Quality control tries to *eliminate coincidence*
⇒ Quality control makes achieving quality *repeatable*

Why Quality Control?

Lives are at stake
(e.g., automatic pilot)

Huge amounts of money
are at stake
(e.g., Ariane V crash)

Software became Ubiquitous
Our society is vulnerable!

Corporate success or failure is at stake
(e.g., telephone billing,
VTM launching 2nd channel)

Your personal future is
at stake (e.g., Y2K
lawsuits)

Quality vs. Requirements

“Simplistic” Definition: Software Quality =

- Deliver
  + (a) what’s required
  + (b) on time
  + (c) within budget

> Cover quality in the “non-functional” requirements

Acoustics

Earthquake resistant

Quality ≠ Requirements

Consider requirements/implementation table

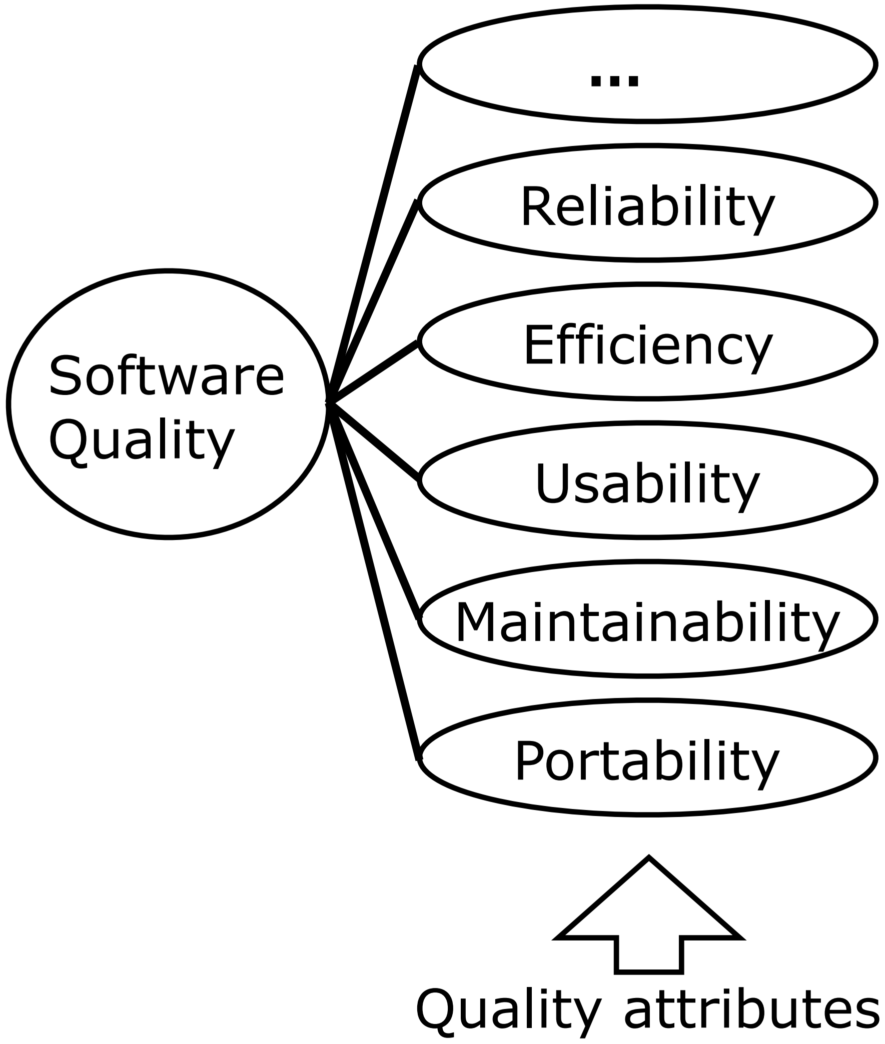
• Requirement	a car, quite cheap	a car, price unimportant
• Implementation		

> Both adhere to their specifications ...
> ... but do they have the same quality?

- Covering quality in the "non-functional" requirements is too simplistic
  + How to assess the quality of the requirements?
    - "Are we building the right product"
      vs. "Are we building the product right"
  + Development team has (implicit) requirements too
    - Maintainability etc. are usually not specified

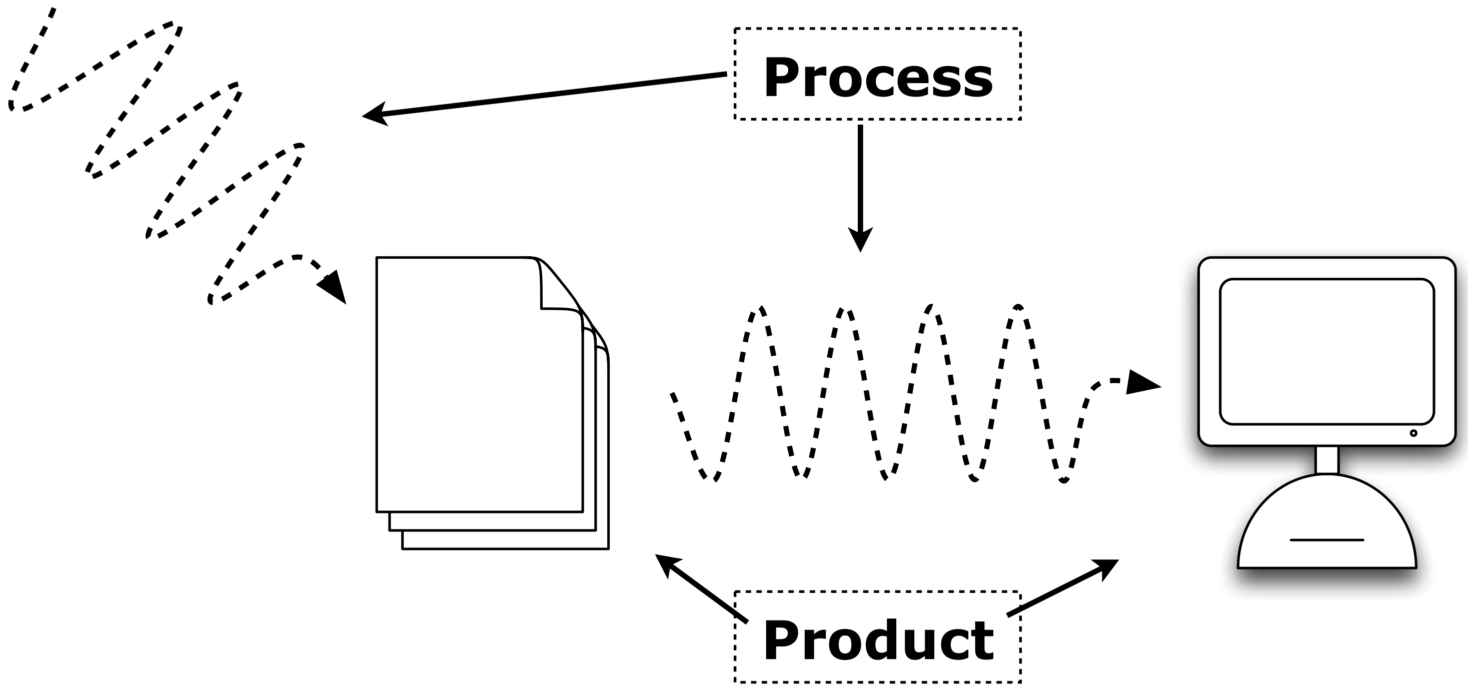
Hierarchical Quality Model

- To side step the quality vs. requirements discussion
  + Define quality via hierarchical quality model, i.e. set of quality attributes (a.k.a. quality factors, quality aspects, ...)
  + Choose quality attributes (and weights) depending on the project context
    > Nevertheless: variation of simplistic quality = requirements



- choose your own set of quality attributes
- may be further refined into subattributes, ...

Product vs. Process Attribute



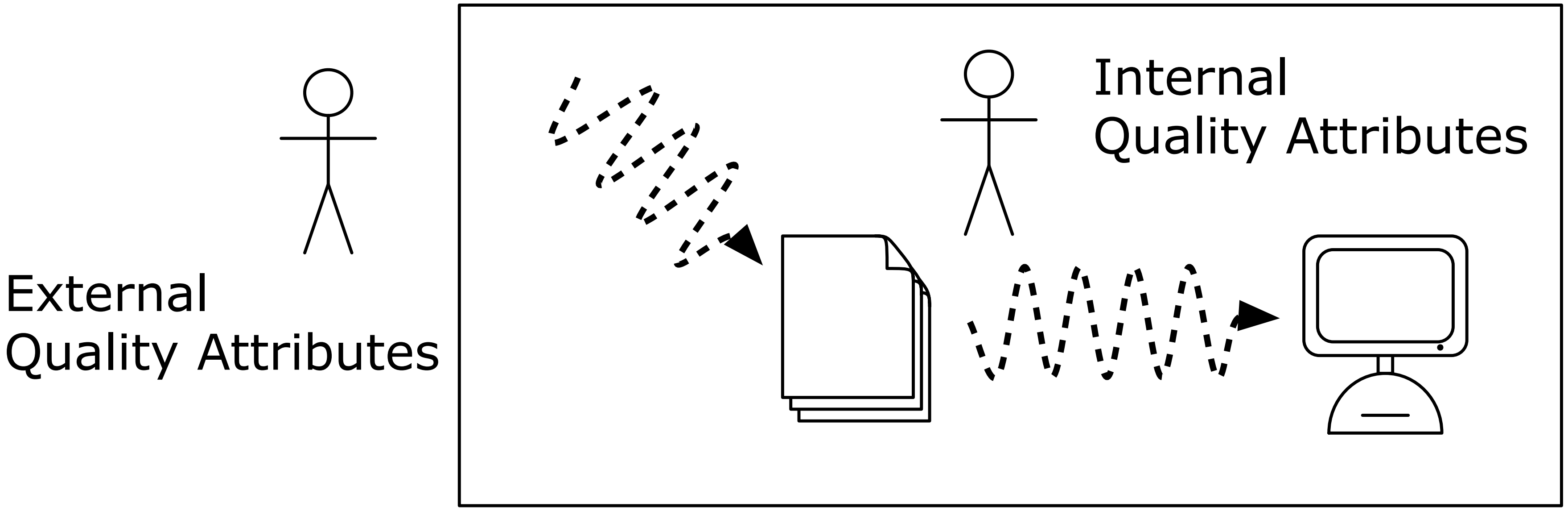
Quality attributes apply both to the product and the process.

- *product*: delivered to the customer
- *process*: produces the software product

Underlying assumption: a quality process leads to a quality product
(cf. metaphor of manufacturing lines)

External vs. Internal Attributes

The distinction between the two is not as sharp as it seems!



External
Quality Attributes

Internal
Quality Attributes

Quality attributes can be external or internal.

- *External*: Derived from relation between environment and system/process.
  > To derive, the system or process must have *run to completion*.
- *Internal*: Derived immediately from the product or process description.
  > To derive, it is sufficient to have the *description*.

Underlying assumption: internal quality leads to external quality
(cf. metaphor of manufacturing lines)

Quality Attributes Overview

See [Ghez02], section 2.2 Representative Qualities

		Product	Process	External	Internal
Product / External					
	Correctness, Reliability, Robustness	x		x	
	Efficiency	x	(x)	x	
	Usability	x	(x)	x	
	Maintainability	x		x	
	• Repairability	x		x	
	• Evolvability	x	(x)	x	
	• Portability	x		x	
Product / Internal					
	Verifiability	x		(x)	x
	Understandability	x		(x)	x
Process					
	Productivity		x	x	
	Timeliness		x	x	
	Visibility		x	(x)	x

Correctness, Reliability, Robustness

3 external product attributes

Correctness

- A system is correct if it behaves according to its specification
  + An absolute property (i.e., a system cannot be “almost correct”)
  + ... in theory and practice undecidable

Reliability

- The user may rely on the system behaving properly
- The probability that the system will operate as expected over a specified interval
  + A relative property (a system has a mean time between failure of 3 weeks)

Robustness

- A system is robust if it behaves reasonably even in circumstances that were not specified
  + A vague property (once you specify the abnormal circumstances they become part of the requirements)

(This slide is a copy from Chapter 5 — Testing)

Efficiency, Usability

2 external attributes, mainly product - sometimes also process

Efficiency (Performance)

- Use of resources such as computing time, memory
  - + Affects user-friendliness and scalability
  - + Hardware technology changes fast!
    - - (Remember: First do it, then do it right, then do it fast)
- For process, resources are man-power, time and money
  - + relates to the “productivity” of a process

Usability (User Friendliness, Human Factors, Human Engineering)

- The degree to which the human users find the system (process) easy to use
  - + Depends a lot on the target audience (novices vs. experts)
  - + Often a system has various kinds of users (end-users, operators, installers)
  - + Typically expressed in “amount of time to learn the system”

Maintainability

external product attributes (evolvability also applies to process)

Maintainability

- How easy it is to change a system after its initial release
  - + software entropy
    - > maintainability gradually decreases over time

Often refined in ...

Repairability

- How much work is needed to correct a defect (= corrective maintenance)

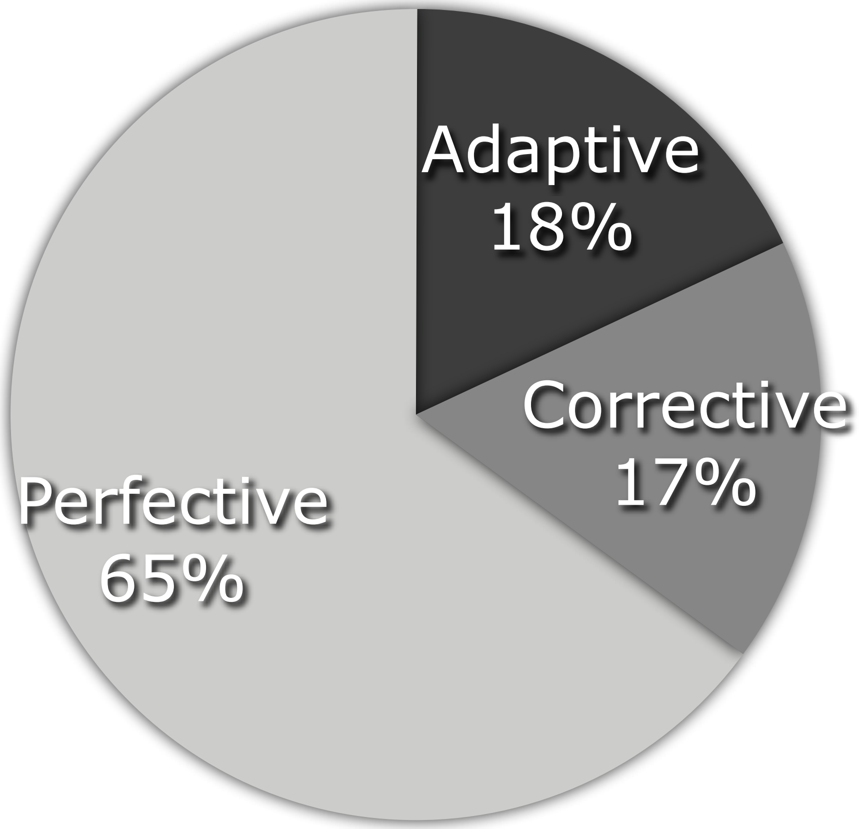
Adaptability (Evolvability)

- How much work is needed to adapt to changing requirements (= perfective maintenance)
  - > both system and process

Portability

- How much work is needed to port to new environment or platforms (= adaptive maintenance)

Maintenance costs



Verifiability, Understandability

internal (and external) product attribute

Verifiability

- How easy it is to verify whether desired attributes are there?
  + internally: e.g., verify requirements, code inspections
  + externally: e.g., testing, efficiency

Understandability

- How easy it is to understand the system
  + internally: contributes to maintainability
  + externally: contributes to usability

Productivity, Timeliness, Visibility

external process attribute (visibility also internal)

Productivity

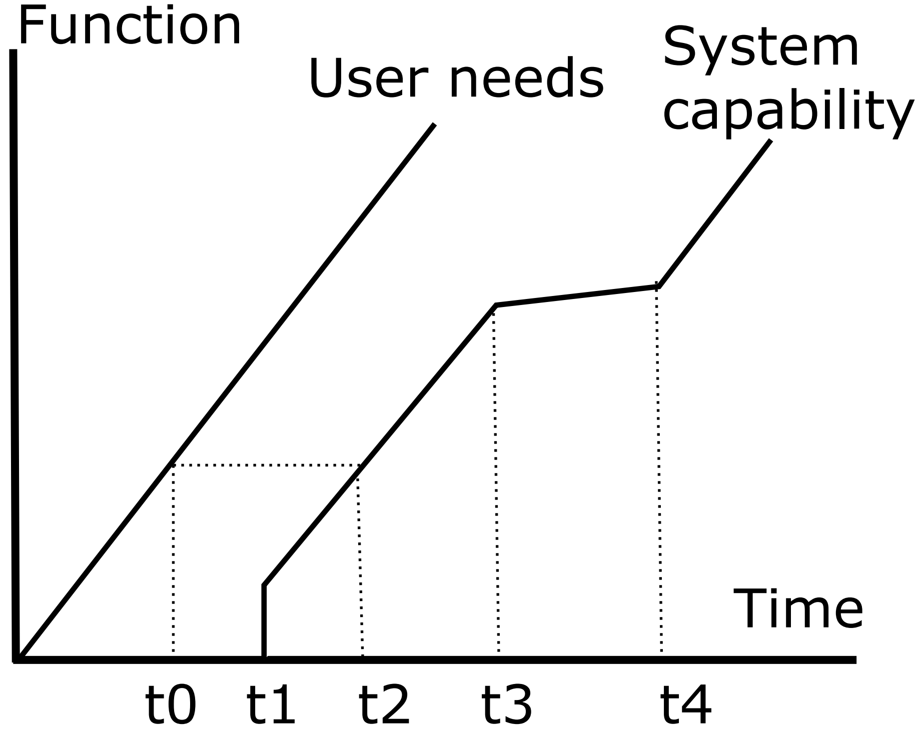
- Amount of product produced by a process for a given number of resources
  - + productivity among individuals varies a lot
  - + often: productivity (∑ individuals) < ∑ productivity (individuals)

Timeliness

- Ability to deliver the product on time
  - + important for marketing ("short time to market")
  - + often a reason to sacrifice other quality attributes
  - + incremental development may provide an answer

Visibility (Transparency, Glasnost)

- Current process steps and project status is accessible
  - + important for management; also deal with staff turn-over



Productivity, Timeliness, Visibility

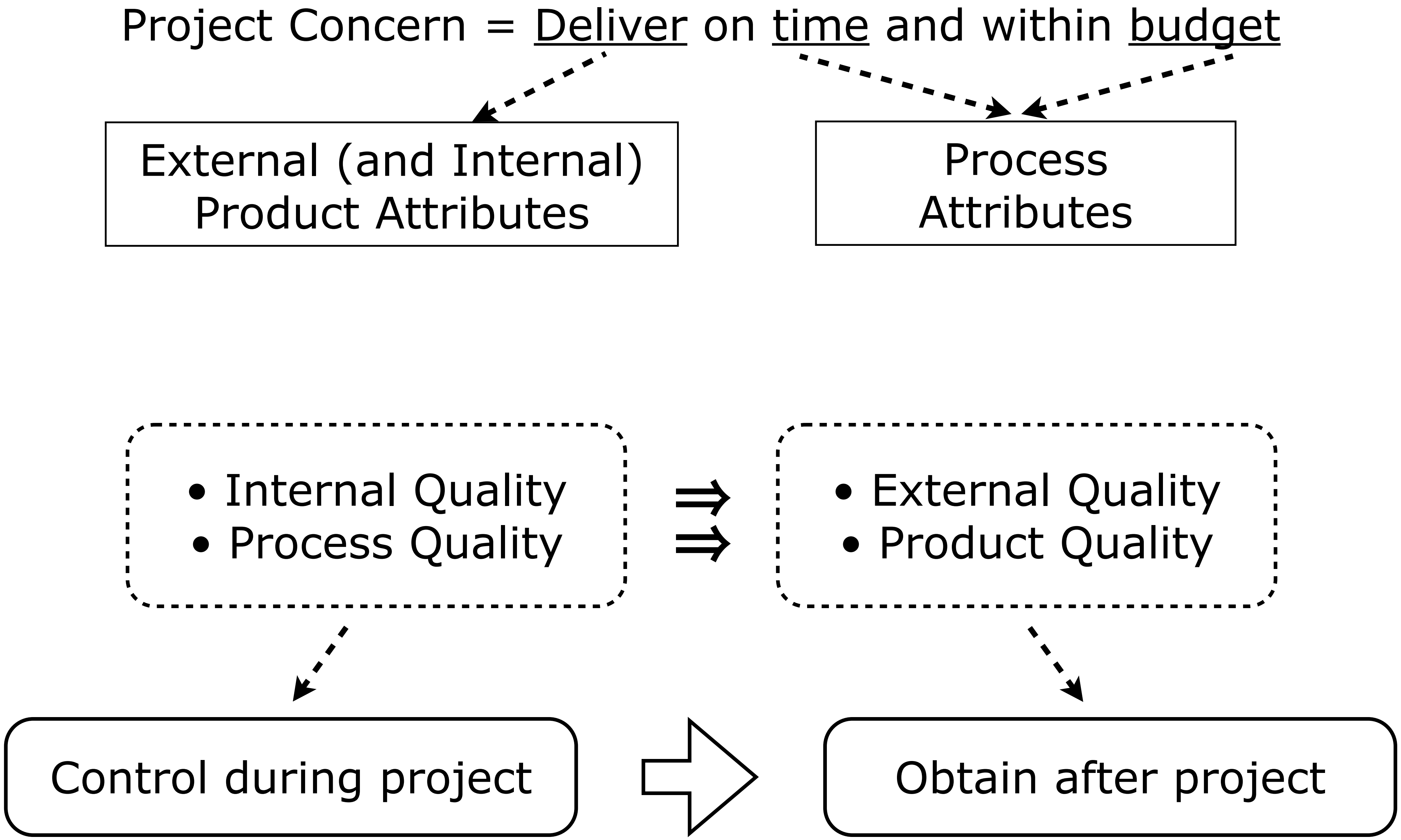
What is meant with "short time to market"?

- Time to market = The time that is needed between requirements specification (feature, user story, use case) agreed upon and deploying said requirement in production
- Short time to market = we can release the requested features fast

Can you name 3 related quality attributes and provide definitions for each of them?

- Productivity
  + Amount of product produced by a process for a given number of resources
- Timeliness
  + Ability to deliver the product on time
- Visibility (Transparency, Glasnost)
  + Current process steps and project status is accessible

Quality Control Assumption



Project Concern = Deliver on time and within budget

External (and Internal) Product Attributes

Process Attributes

- Internal Quality
- Process Quality

⇒
⇒

- External Quality
- Product Quality

Control during project

Obtain after project

Otherwise, quality is mere coincidence!

Quality Plan

Project Concern = Deliver on time and within budget

Project Plan
schedule: plan time budget: plan money quality plan: plan quality

A quality plan should:

- set out *desired product qualities* and how these are assessed
  + define the *most significant* quality attributes
  (cf. Quality Attributes Overview)
- set out which *organizational standards* should be applied
  + typically by means of *check-lists* and *standards*
- define the quality *assessment* process
  + typically done via *quality reviews* after internal release

Types of Quality Reviews

Review type	Principal purpose
Formal Technical Reviews (a.k.a. design or program inspections)	Driven by *checklist* • detect detailed errors in any product • mismatches between requirements and product • check whether standards have been followed.
Progress reviews	Driven by *budgets, plans and schedules* • check whether project runs according to plan • requires precise milestones • both a process and a product review

- Reviews should be recorded and records maintained
  + Software or documents may be “signed off” at a review
  + Progress to the next development stage is thereby approved

Review Meetings and Minutes

- (See [Pres00])

Review meetings should:

- typically involve 3-5 people
- require a maximum of 2 hours advance *preparation*
  + reviewers use checklists to evaluate products
- last less than 2 hours

The review minutes should summarize:

- 1. What was reviewed
- 2. Who reviewed it?
- 3. What were the findings and conclusions?
- 4. Decision
  + *Accepted* without modification
  + *Provisionally accepted*, subject to corrections (no follow-up review)
  + *Rejected*, subject to corrections and follow-up review

Review Guidelines

- 1. Review the product, not the producer
    - > Quality is a team responsibility
- 2. Set an agenda and maintain it
- 3. Limit debate and rebuttal
    - > Consensus is not required
- 4. Identify problem areas, but don’t attempt to solve every problem noted
- 5. Take written notes
    - > Blackboard or electronic white-boards for group awareness
- 6. Limit the number of participants and insist upon advance preparation
- 7. Develop a checklist for each product that is likely to be reviewed
- 8. Allocate resources and time schedule for reviews
    - > Including time for the modifications after the review
- 9. Conduct meaningful training for all reviewers
- 10. Review your early reviews
    - > Customise the review process by learning from your early attempts

Sample Review Checklists (i)

Project Plan

- 1. Is software scope unambiguously defined and bounded?
- 2. Are resources adequate for scope?
- 3. Have risks in all important categories been defined?
- 4. Are tasks properly defined and sequenced?
- 5. Is the basis for cost estimation reasonable?
- 6. Have historical productivity and quality data been used?
- 7. Is the schedule consistent?
- ...

Requirements Specification

- 1. Is information domain analysis complete, consistent and accurate?
- 2. Does the data model properly reflect data objects, attributes and relationships?
- 3. Are all requirements traceable to system level?
- 4. Has prototyping been conducted for the user/customer?
- 5. Are requirements consistent with schedule, resources and budget?
- ...

Sample Review Checklists (ii)

Design

- 1. Has modularity been achieved?
- 2. Are interfaces defined for modules and external system elements?
- 3. Are the data structures consistent with the information domain?
- 4. Are the data structures consistent with the requirements?
- 5. Has maintainability been considered?
- …

Code

- 1. Does the code reflect the design documentation?
- 2. Has proper use of language conventions been made?
- 3. Have coding standards been observed?
- 4. Are there incorrect or ambiguous comments?
- …

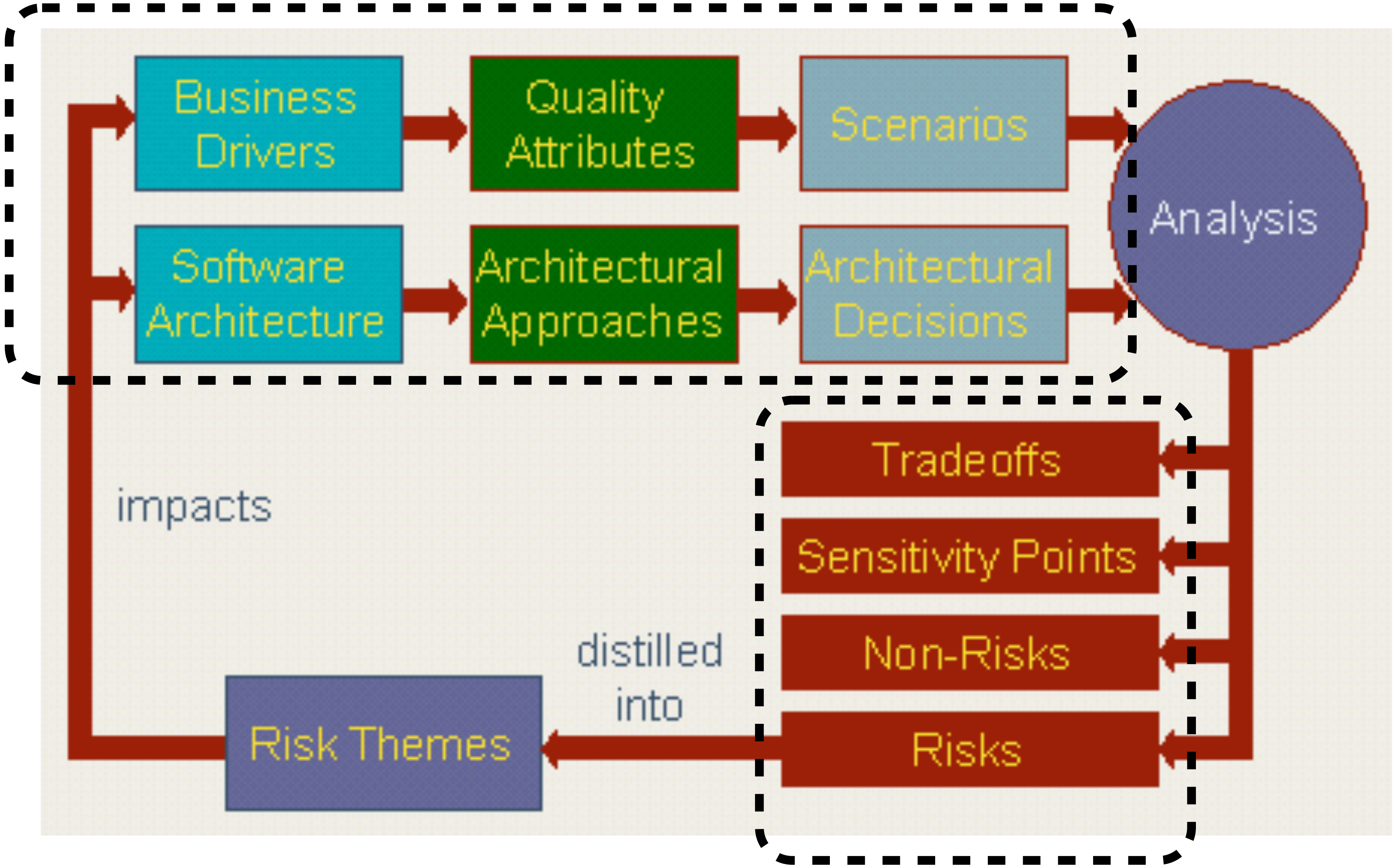
Testing

- 1. Have test resources and tools been identified and acquired?
- 2. Have both white and black box tests been specified?
- 3. Have all the independent logic paths been tested?
- 4. Have test cases been identified and listed with expected results?
- 5. Are timing and performance to be tested?
- ...

Sample Review Process: ATAM

ATAM = Architecture Tradeoff Analysis Method (see chapter “Software Architecture”)

Review these …



… to arrive at these!

Product and Process Standards

Product standards define characteristics that all components should exhibit.
Process standards define how the software process should be conducted.

Product standards	Process standards
Design review form	Design review conduct
Document naming standards (++)	Configuration management (++)
Procedure header format	Version release process
Coding conventions standard (++)	Project plan approval process
Project plan format	Change control process (++)
Change request form (+)	Test recording process (+)

Problems

- Not always seen as relevant and up-to-date by software engineers
- May involve too much bureaucratic form filling
- May require tedious manual work if unsupported by software tools

Sample Java Code Conventions

https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

4.2 Wrapping Lines

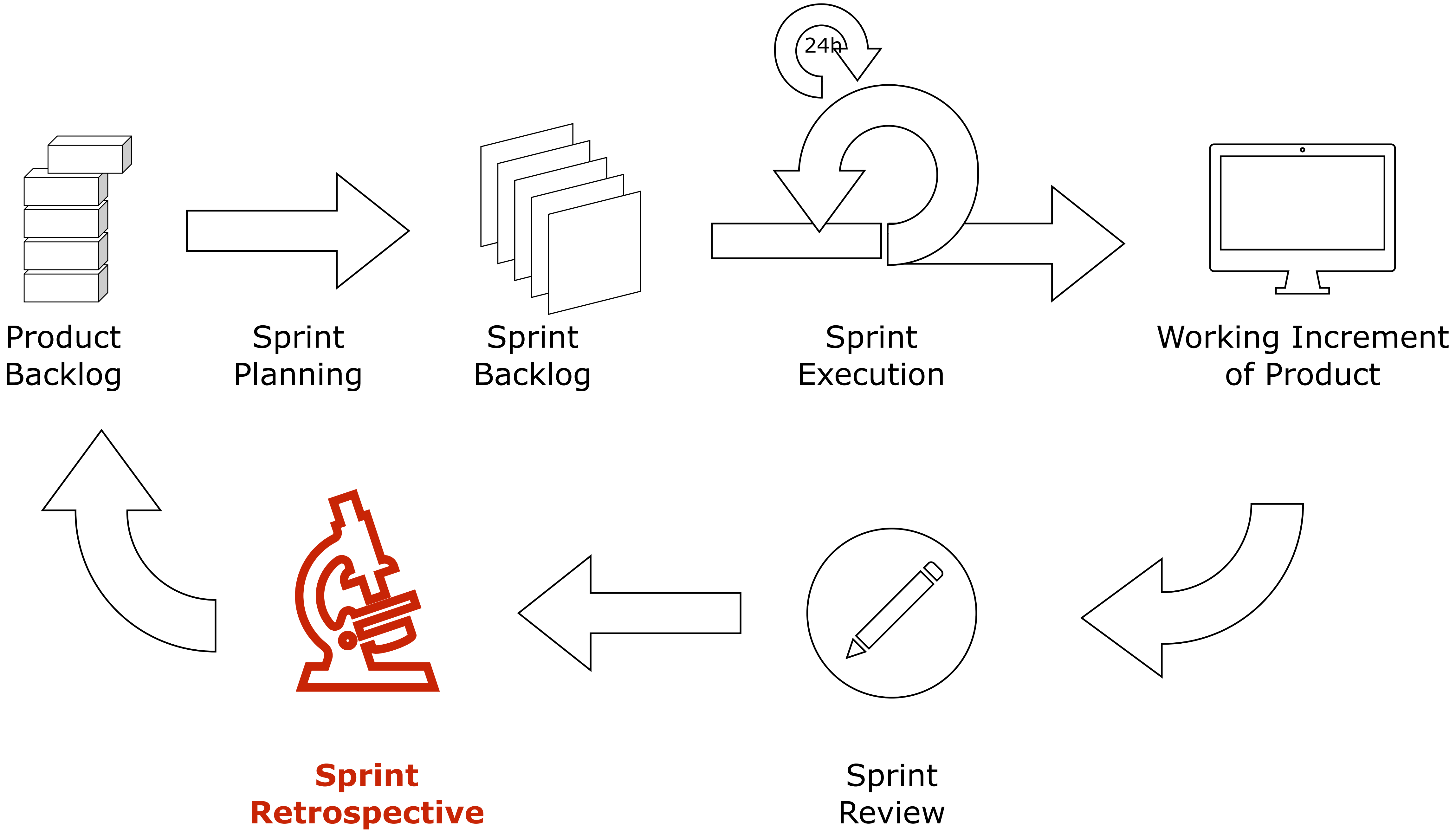
- When an expression will not fit on a single line, break it according to these general principles:
  + Break after a comma.
  + Break before an operator.
  + Prefer higher-level breaks to lower-level breaks.
  + Align the new line with the beginning of the expression at the same level on the previous line.
  + If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.

…

10.3 Constants

- Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Quality Culture: Continuous Improvement

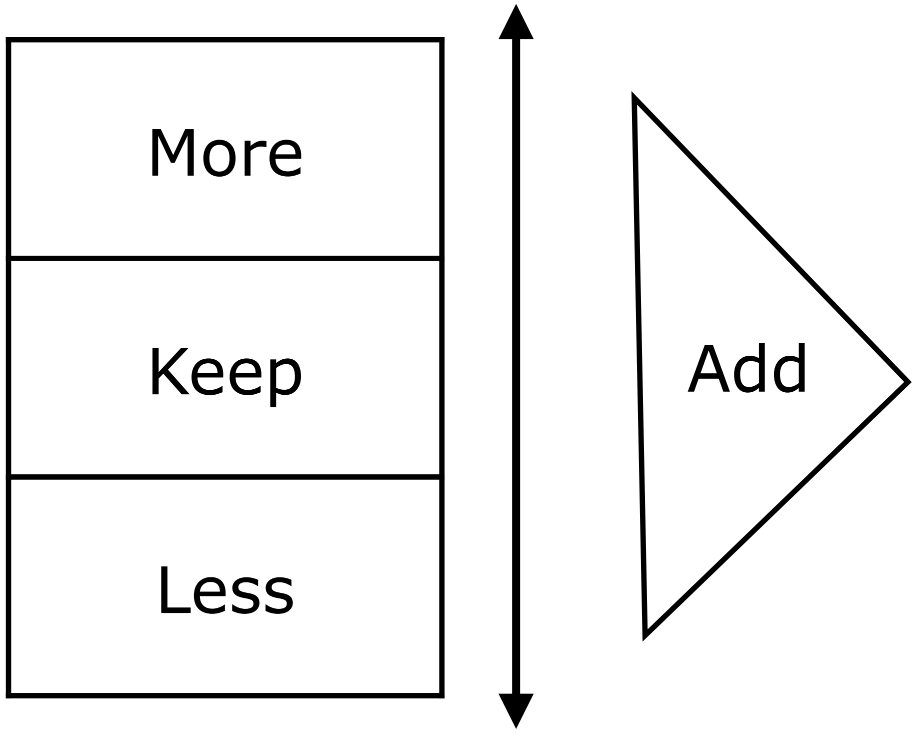


- What worked well this sprint that we want to continue doing?
- What didn’t work well this sprint that we should stop doing?
- What should we start doing or improve?

FUN RETROSPECTIVES *Activities and ideas for making agile retrospectives more engaging*

CAPSTONE PROJECT

KALM (Keep - Add - Less - More)



More

Keep

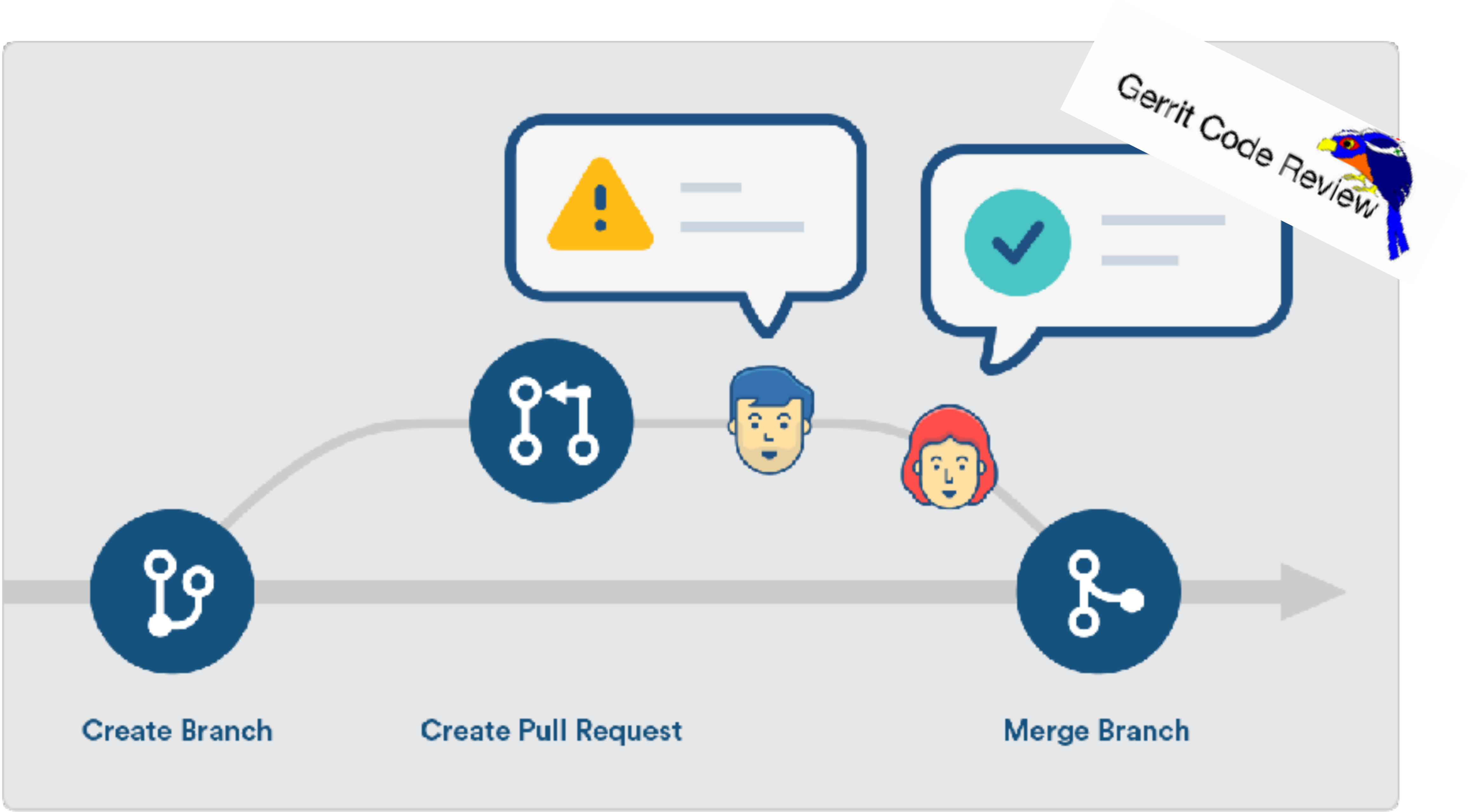
Less

Add

DAKI (Drop - Add - Keep - Improve)

Drop	Add
Keep	Improve

Code Review on Pull Requests



© Amber Frauenholtz - Bitbucket — 5 elements of a perfect pull request

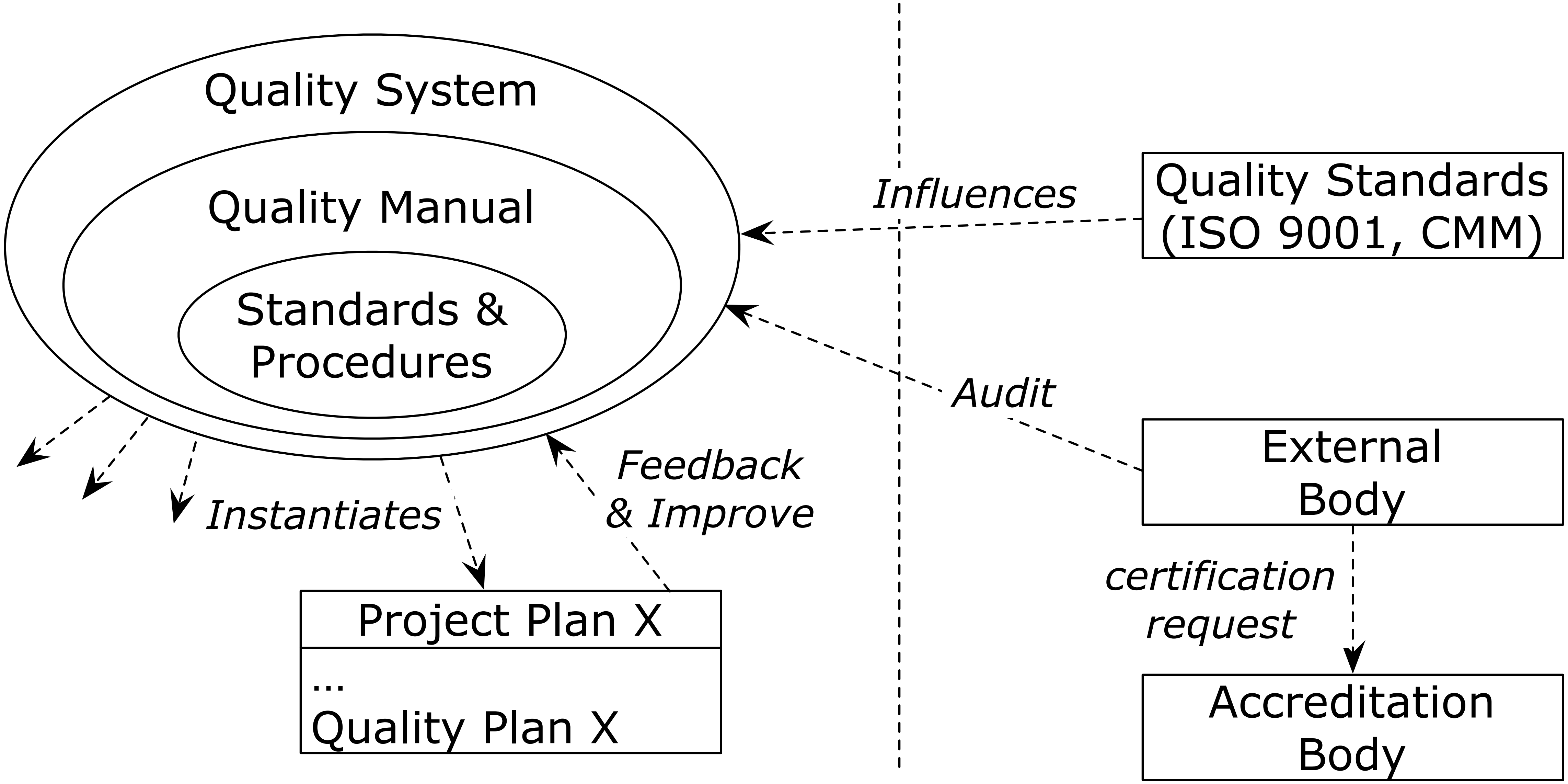
Quality System & Certification

When starting a project, the project will include a Quality Plan

- Ideally, such plan is an instance of the organization’s *Quality System*

Certain customers require an externally reviewed quality system

- An organization may request to *certify* its quality system



ISO 9000

ISO 9000

- is an international set of standards for quality management applicable to a range of organisations from manufacturing to service industries.

ISO 9001

- is a generic model of the quality process, applicable to organisations whose business processes range all the way from design and development, to production, installation and servicing;

- ISO 9001 must be instantiated for each organisation
- ISO 9000-3 interprets ISO 9001 for the software developer

ISO = International Organisation for Standardization

- ISO main site: ISO [http://www.iso.org/]
- ISO 9000 family
  + [https://www.iso.org/iso-9001-quality-management.html]

ISO 9001

- Describes quality standards and procedures for developing products of any kind:

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging & delivery	Purchasing
Purchaser-supplied products	Product identification & traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

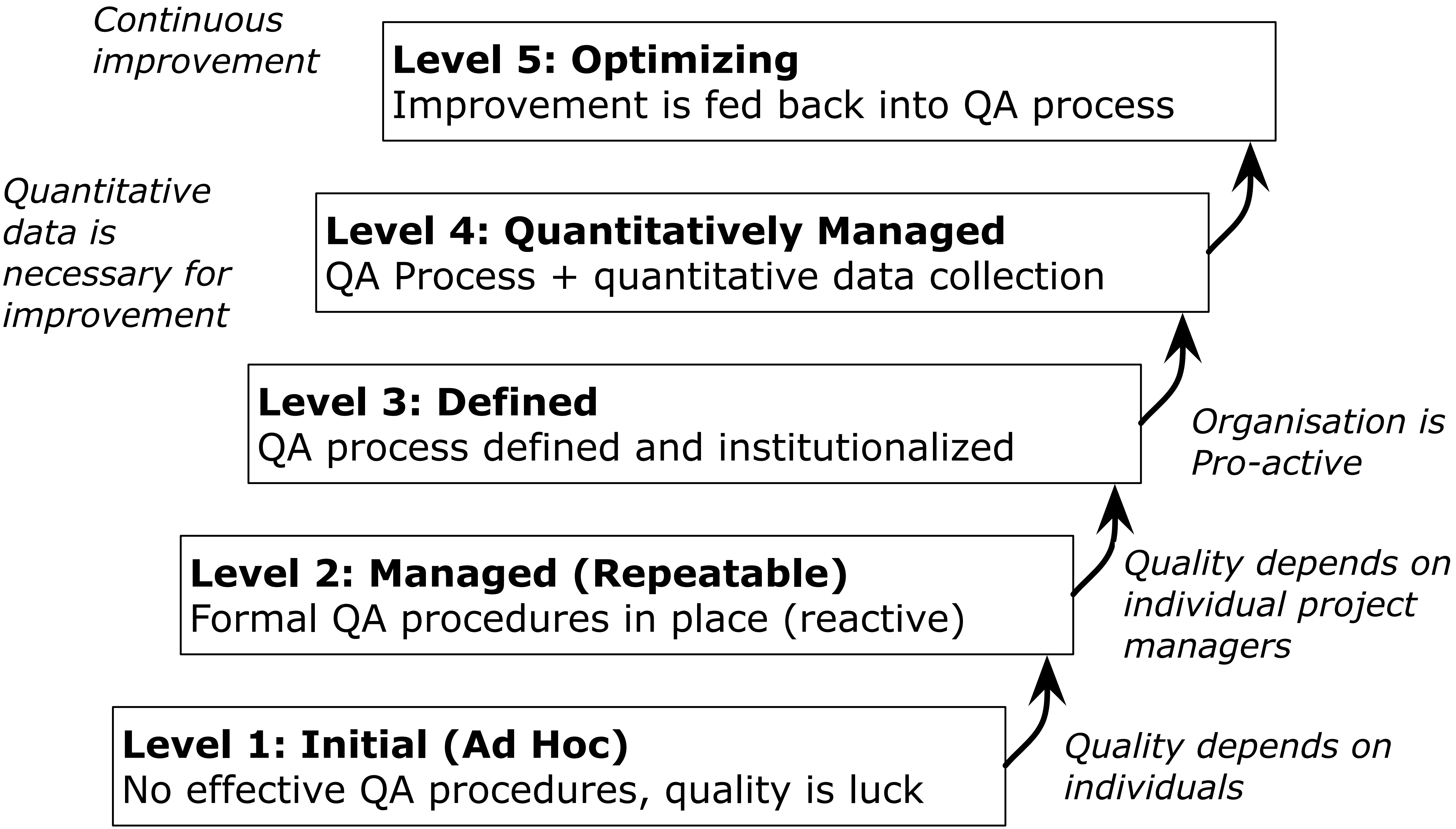
Capability Maturity Model

- Process maturity model from SEI (Software Engineering institute)
  + Initiated in 1991
  + Version 1.1 completed in January 1993
- Tool to evaluate the ability of government contractors to perform a contracted software project
  + assess how well contractors manage software processes
  + says little on individual projects
  + not necessarily applicable to commercial-off-the-shelf (COTS)

- CMM is now superseded by CMMI
  + **CMMI** = Capability Maturity Model Integration
  + Integrate software quality standard with standards other disciplines
  + Version 1.1 in 2002 – Version 1.2 in August 2006

- Essentially CMMI consists of
  + 5 maturity levels
    - Separate standards for development / services / acquisition
  + Core Process Area
    - identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important

CMMI: Overview



Continuous improvement

Level 5: Optimizing
Improvement is fed back into QA process

Quantitative data is necessary for improvement

Level 4: Quantitatively Managed
QA Process + quantitative data collection

Level 3: Defined
QA process defined and institutionalized

Organisation is Pro-active

Level 2: Managed (Repeatable)
Formal QA procedures in place (reactive)

Quality depends on individual project managers

Level 1: Initial (Ad Hoc)
No effective QA procedures, quality is luck

Quality depends on individuals

Core Process Areas

Abbr.	Name	Area	Level
CM	Configuration Management	Support	2
MA	Measurement and Analysis	Support	2
PMC	Project Monitoring and Control	Project Management	2
PP	Project Planning	Project Management	2
PPQA	Process and Product Quality Assurance	Support	2
REQM	Requirements Management	Project Management	2
DAR	Decision Analysis and Resolution	Support	3
IPM	Integrated Project Management	Project Management	3
OPD	Organizational Process Definition	Process Management	3
OPF	Organizational Process Focus	Process Management	3
OT	Organizational Training	Process Management	3
RSKM	Risk Management	Project Management	3
OPP	Organizational Process Performance	Process Management	4
QPM	Quantitative Project Management	Project Management	4
CAR	Causal Analysis and Resolution	Support	5
OPM	Organizational Performance Management	Process Management	5

Core Process Areas

Abbr.	Name	Area	Level
CM	Configuration Management	Support	2
MA	Measurement and Analysis	Support	2
PMC	Project Monitoring and Control	Project Management	2
PP	Project Planning	Project Management	2
PPQA	Process and Product Quality Assurance	Support	2
REQM	Requirements Management	Project Management	2
DAR	Decision Analysis and Resolution	Support	3
IPM	Integrated Project Management	Project Management	3
OPD	Organizational Process Definition	Process Management	3
OPF	Organizational Process Focus	Process Management	3
OT	Organizational Training	Process Management	3
RSKM	Risk Management	Project Management	3
OPP	Organizational Process Performance	Process Management	4
QPM	Quantitative Project Management	Project Management	4
CAR	Causal Analysis and Resolution	Support	5
OPM	Organizational Performance Management	Process Management	5

testing??

use-cases??

Conclusion: Reviews

Reviews and Inspections

- Low on ceremony, high on external product quality
- Side-effect: team ownership

- Very effective
  + Empirical evidence shows that reviews find more errors than tests
  (+ reviews usually indicate a solution)

- Very cost effective
  + Empirical evidence shows that reviews find errors more cheaply than tests

- However: tests find other errors than reviews
  > Reviews must supplement testing

Conclusion: Quality Standards

Quality Standards (ISO9000 and CMMI)

- No guarantee for external *product* quality
  + There is NO empirical evidence that ISO, CMMI actually improve quality

- Adequate for process *quality*
  + ... on time and within budget

- Does not scale down
  + developing a quality system is an overhead
  + difficult for small enterprises (where most software development is done)

- Eliminate *coincidence*
  + ... eliminates creativity (to some degree)
  + often obstructed by people doing the work
- Tendency towards high ceremony
  + difficult for rapidly changing contexts (e-commerce)

Is a means, not a goal
Illustrates that quality is an important issue
Certification is a driving force

Summary (i)

You should know the answers to these questions

- Why is software quality more important than it was a decade ago?
- Can a correctly functioning piece of software still have poor quality? Why?
- If quality control can't guarantee results, why do we bother?
- What's the difference between an external and an internal quality attribute? And between a product and a process attribute?
- What's the distinction between correctness, reliability and robustness?
- How can you express the "user friendliness" of a system?
- Can you name three distinct refinements of "maintainability"? What do each of these names mean?
- What is meant with "short time to market"? Can you name 3 related quality attributes and provide definitions for each of them?
- Name four things which should be recorded in the review minutes.
- Explain briefly the three items that should be included in a quality plan.
- What's the relationship between ISO9001, CMMI standards and an organization's quality system? How do you get certified?
- Can you name and define the 5 levels of CMMI?
- Where would "use-cases" as defined in chapter 3 fit in the table of core process areas (p. 32)? Motivate your answer shortly.

Summary (ii)

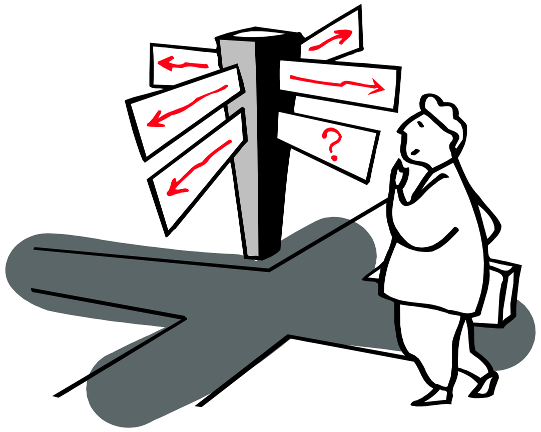
You should be able to complete the following tasks

- Given a piece of code and a coding standard, review the code to verify whether the standard has been adhered to.

Can you answer the following questions?

- Given the Quality Attributes Overview table, argue why the crosses and blanks occur at the given positions.
- Why do quality standards focus on process and internal attributes instead of the desired external product attributes?
- Why do you need a quality plan? Which topics should be covered in such a plan?
- How should you organize and run a review meeting?
- Why are coding standards important?
- What would you include in a documentation review checklist?
- How often should reviews by scheduled?
- Could you create a review check-list for ATAM?
- Would you trust software from an ISO 9000 certified company? And if it were CMMI?
- You are supposed to develop a quality system for your organization. What would you include?
- Where would “testing” fit in the table of core process areas (p. 32). Does it cover a single row or not? Argue why (not)?

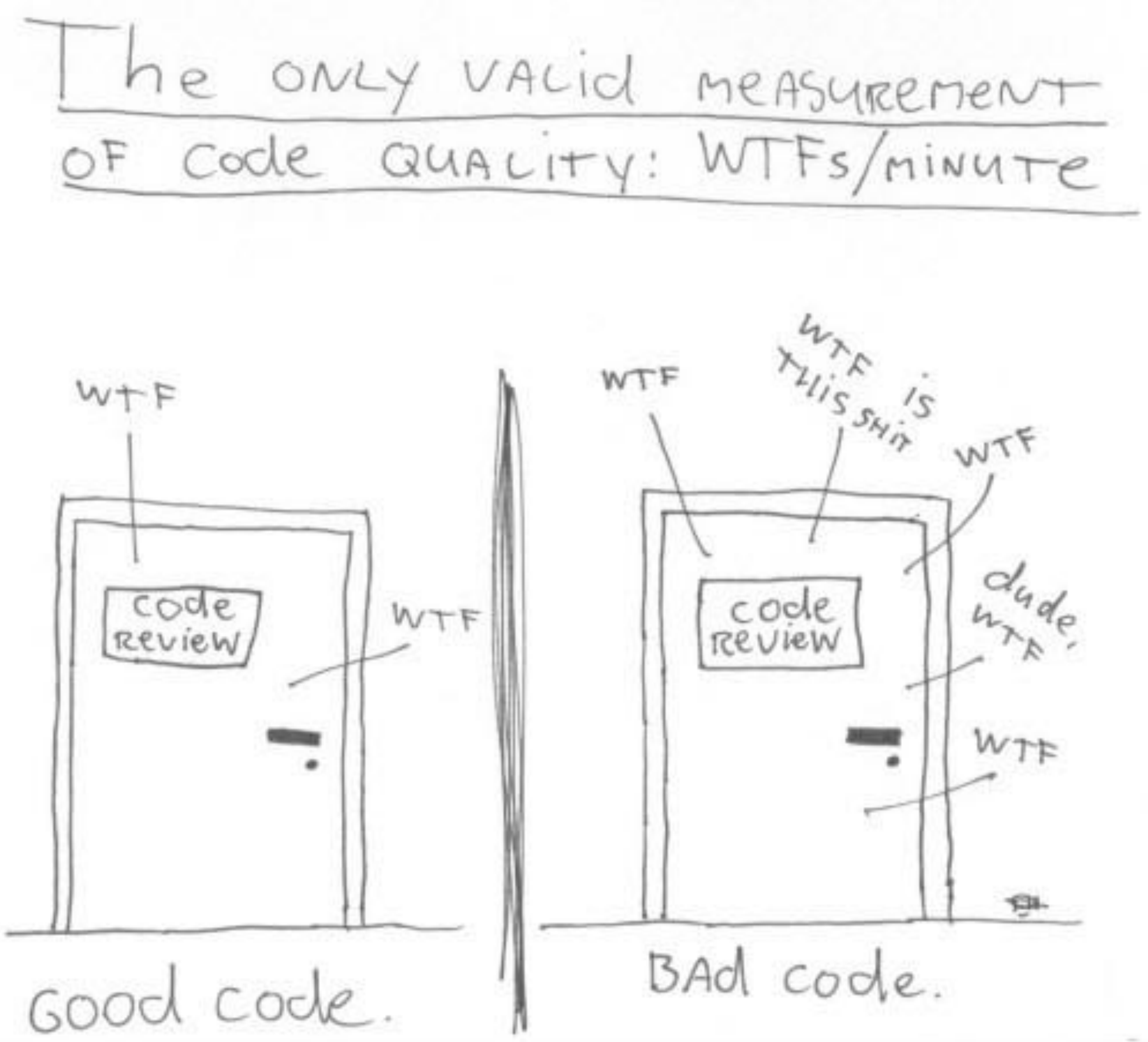
CHAPTER 10 – Software Metrics



- Introduction
  + When, Why and What?
    - Measurement Theory
    - GQM Paradigm
- Effort Estimation
  + Algorithmic Cost Modeling
  + COCOMO
  + Putnam’s model (SLIM)
  + Size Measures
    - Lines of Code, Function Points
    - Use case points
    - Story Points
- Quality Control
  + Quantitative Quality Model
  + Sample Quality Metrics
- Conclusion
  + Metrics for Effort Estimation & Quality Control

Does this make sense?

New Slide
(Quizz)



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Literature

- [Ghez02] In particular, section 6.9 “Verifying Other Software Properties” and 8.2 “Project Planning”
- [Pres00] In particular, chapters “Software Process and Project Metrics” and “Software Project Planning‟
- [Somm05] In particular, chapters “Software Cost Estimation” and “Process Improvement”

Other

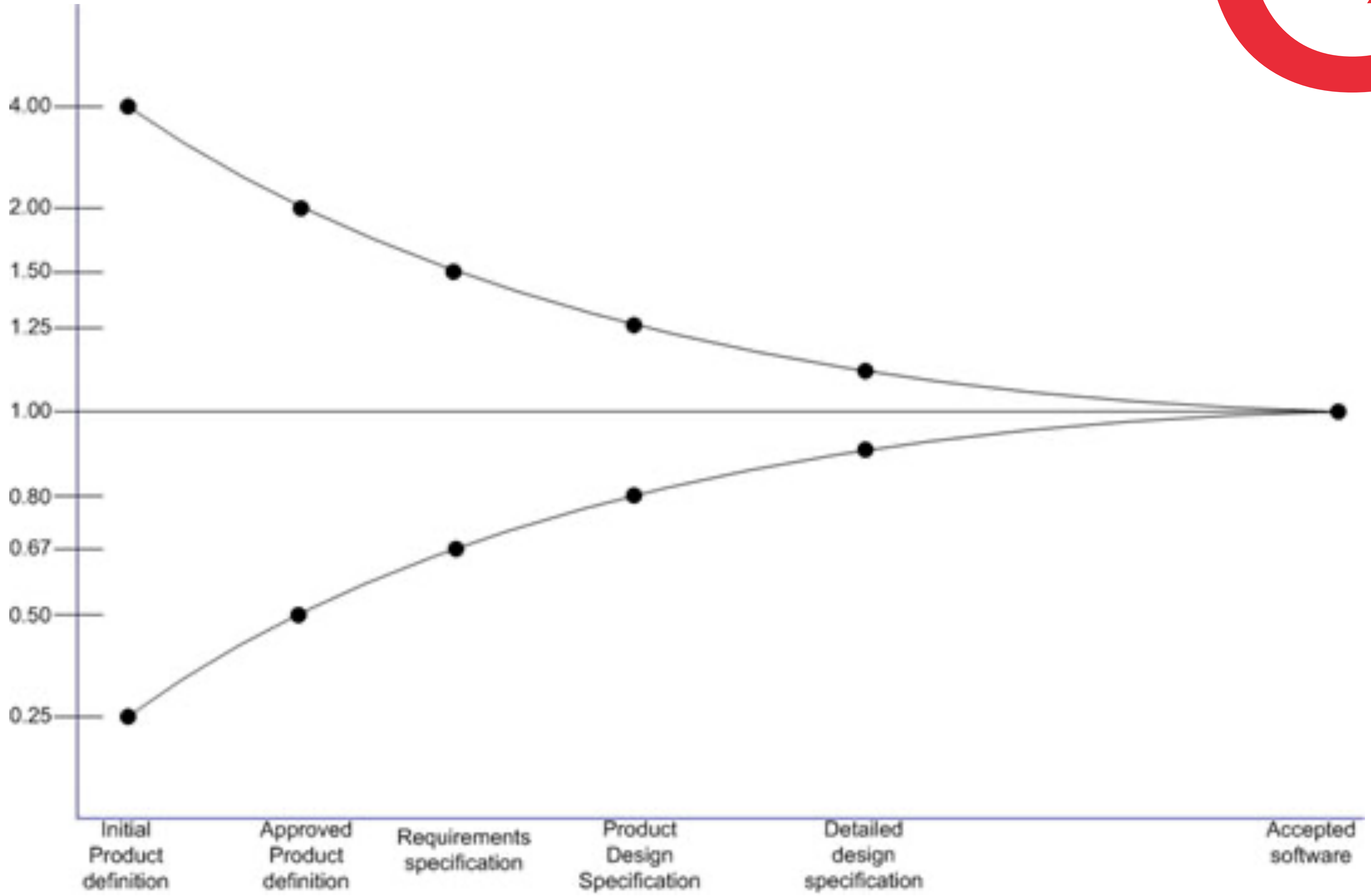
- [Fent96] Norman E. Fenton, Shari l. Pfleeger, “Software Metrics: A rigorous & Practical Approach”, Thompson Computer Press, 1996.
  + Thorough treatment of measurement theory with lots of advice to make it digestible.
- [Putn03] Lawrence H. Putnam and Ware Myers, “Five Core Metrics - The intelligence behind Successful Software Management”, Dorset House Publishing, 2003.
  + Software estimation: Time and Effort are dependent variables !
- [Schn98] Applying Use Cases - a Practical Guide, Geri Schneider, Jason, P. Winters, Addison-Wesley, 1998.
  + Chapter 8 describes early work on estimation based on use cases.

Literature (bis)

New Slide
(Quizz)

Fingers in the air: a Gentle Introduction to Software Estimation

estimate vs. target vs. commitment
accuracy vs. precision
cone of uncertainty



Why Metrics?

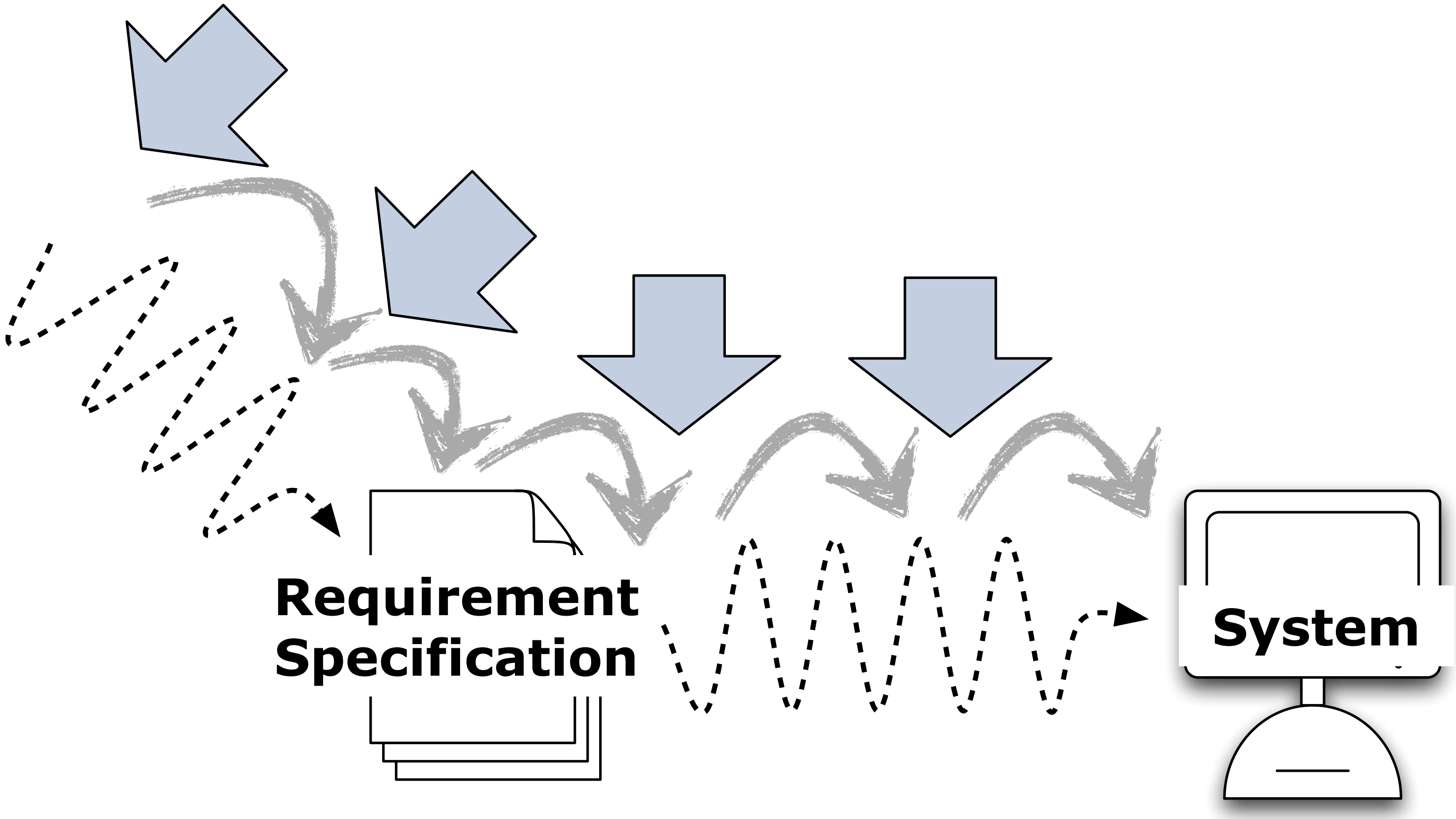
U.S. National Archives

The Weather on D-Day

When Metrics?

Effort (and Cost) Estimation

- Measure early in the life-cycle to deduce later production efforts



Quality Assessment and Improvement

- Control software quality attributes during development
- Compare (and improve) software production processes

Why (Software) Metrics?

You cannot control what you cannot measure [De Marco]

What is not measurable, make measurable [Galileo Galilei, 1564-1642]

- Measurement quantifies concepts
  + understand, control and improve
- Example:
  + historical advances in temperature measurement

Time	Measurement	Comment
2000 BC	Rankings “hotter than”	By touching objects, people could compare temperature
1600 AD	Thermometer “hotter than”	A separate device is able to compare temperature
1720 AD	Fahrenheit scale	Quantification allows to log temperature, study trends, predict phenomena (weather forecasting), ...
1742 AD	Celsius scale	
1854 AD	Kelvin scale	Absolute zero allows for more precise descriptions of physical phenomena

What are Software Metrics?

Software metrics

- Any type of measurement which relates to a software system, process or related documentation
  + Lines of code in a program
  + the Fog index (calculates readability of a piece of documentation)
    - 0.4 *(# words / # sentences) + (percentage of words >= 3 syllables)
  + number of person-days required to implement a use-case
- According to measurement theory, Metric is an incorrect name for Measure
  + a Metric m is a function measuring distance between two objects such that m(x,x) = 0; m(x,y) = m(y,x); m(x,z) <= m(x,y) + m(y,z)

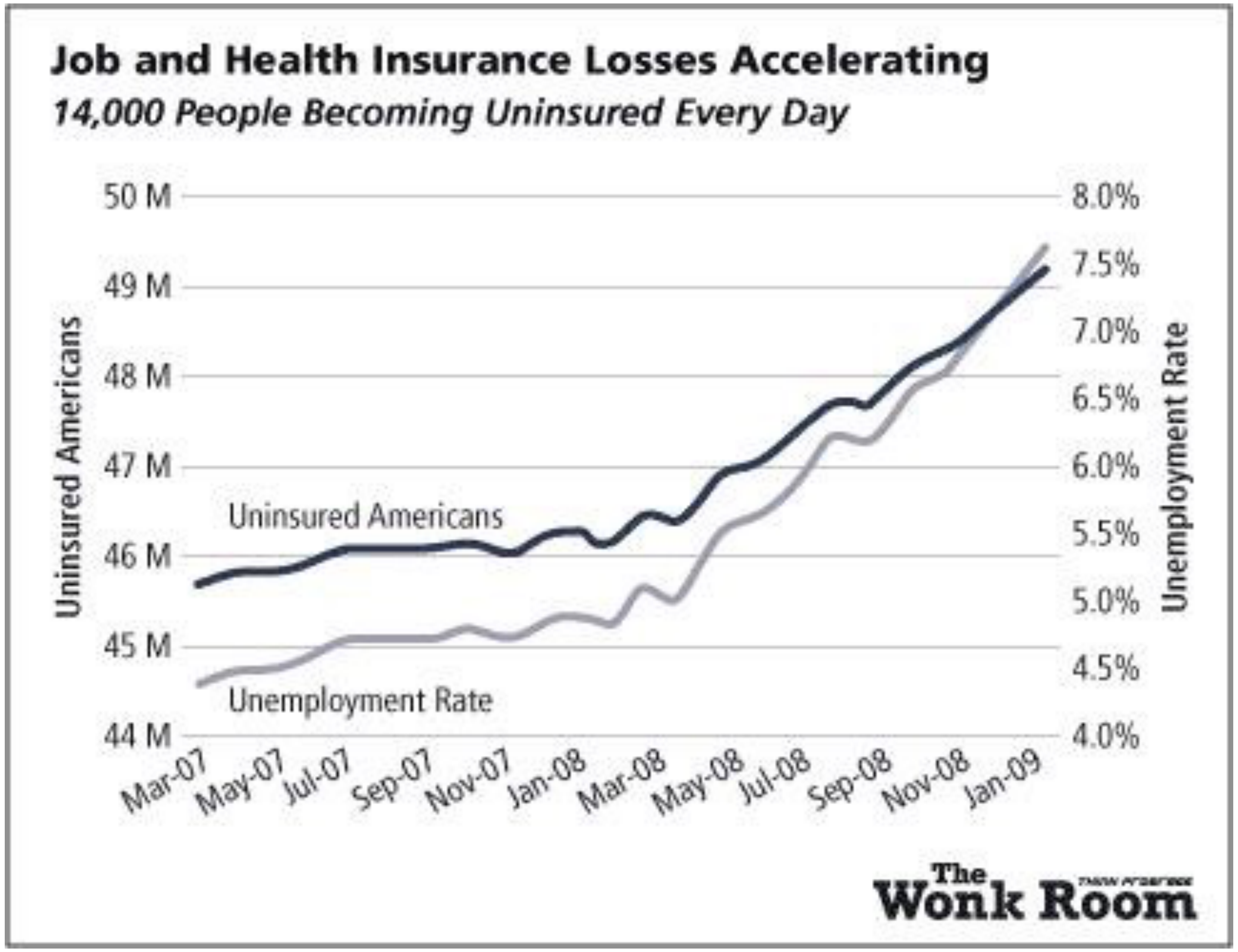
Direct Measures

- Measured directly in terms of the observed attribute (usually by counting)
  + Length of source-code
  + Duration of process
  + Number of defects discovered

Indirect Measures

- Calculated from other direct and indirect measures
  + Module Defect Density = Number of defects discovered / Length of source
  + Temperature is usually derived from the length of a liquid or metal

How to Lie with Statistics



A Misleading Graph Creating An Extreme Trend Where There is Only a Small Increase

© Will Koehrsen, “Lessons on How to Lie with Statistics”, Jul 28, 2019
[https://towardsdatascience.com/lessons-from-how-to-lie-with-statistics-57060c0d2f19]

Possible Problems

Example:
Compare productivity of programmers in lines of code per time unit.

- Preciseness (a): Do we use the same units to compare?
  + What is a “line of code”? What exactly is a “time unit”?
- Preciseness (b): Is the context the same?
  + Were programmers familiar with the language?
- Representation Condition: Is “code size” really what we want to have?
  + What about code quality?
- Scale and Scale Types: How do we want to interpret results?
  + Average productivity of a programmer?
  + Programmer X is more productive than Y?
  + Programmer X is twice as productive as Y?
- GQM-paradigm: What do we want to do with the results?
  + Do you reward “productive” programmers?
  + Do you compare productivity of software processes?

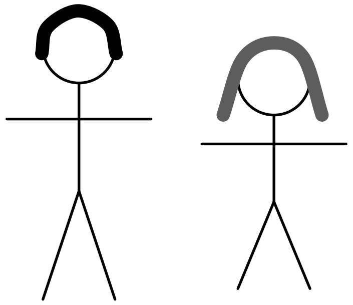
Measurement theory will help us to answer these questions...

Empirical Relations

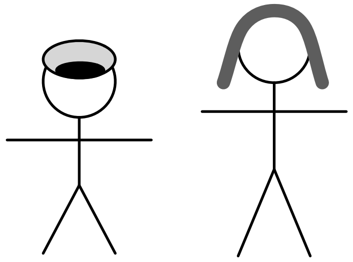
Observe true/false relationships between (attributes of) real world entities
Empirical relations are *complete*, i.e. defined for all possible combinations

Example: empirical relationships between height attributes of persons

“is taller than” binary relationship

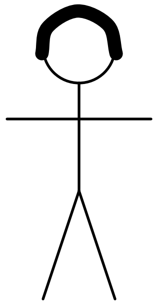


Frank “is taller than” Laura

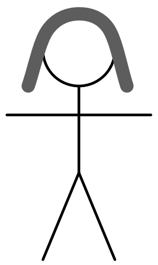


Joe “is not taller than” Laura

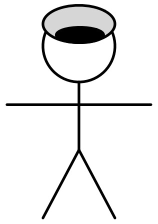
“is tall” unary relationship



Frank “is tall”

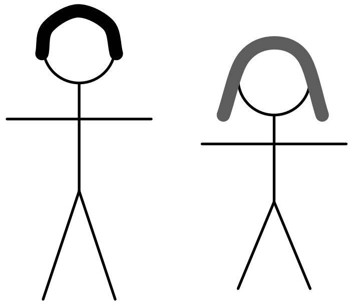


Laura “is tall”

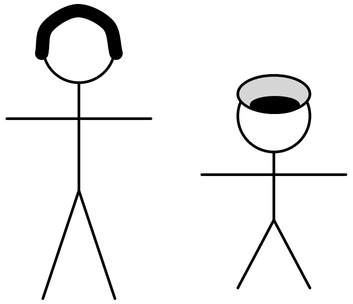


Joe “is not tall”

“is much taller than” binary relationship

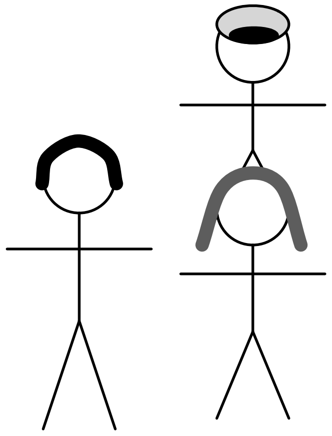


Frank “is not much taller than” Laura



Frank “is much taller than” Joe

“... is higher than ... + ...” ternary relationship

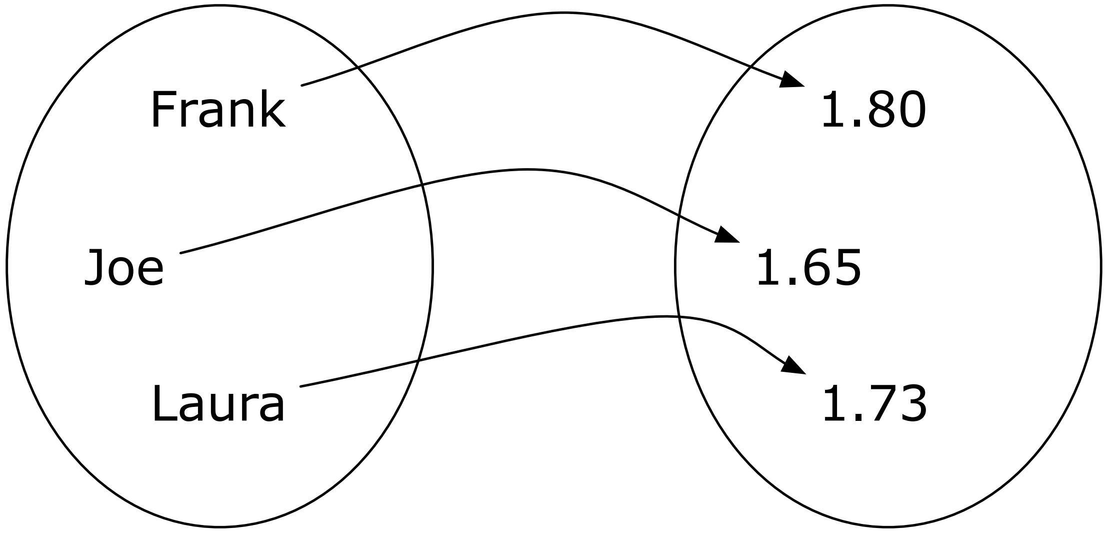


Frank “is not higher than” Joe on Laura’s shoulders

Measure & Measurement

- A *measure* is a function mapping
  - an attribute of a real world entity (= the domain)

  + onto
  - a symbol in a set with known mathematical relations (= the range).
- A *measurement* is then the symbol assigned to the real world attribute by the measure.
- A *metric* is a measure with as range the real numbers and which satisfies
  - m(x,x) = 0
  - m(x,y) = m(y,x)
  - m(x,z) <= m(x,y) + m(y,z)



Example: measure mapping "height" attribute of person on a number representing "height in meters".

Purpose

- Manipulate symbol(s) in the range
  ⇒ draw conclusions about attribute(s) in the domain

Preciseness

- To be *precise*, the definition of the measure must specify
  + domain: do we measure people's height or width?
  + range: do we measure height in centimeters or inches?
  + mapping rules: do we allow shoes to be worn?

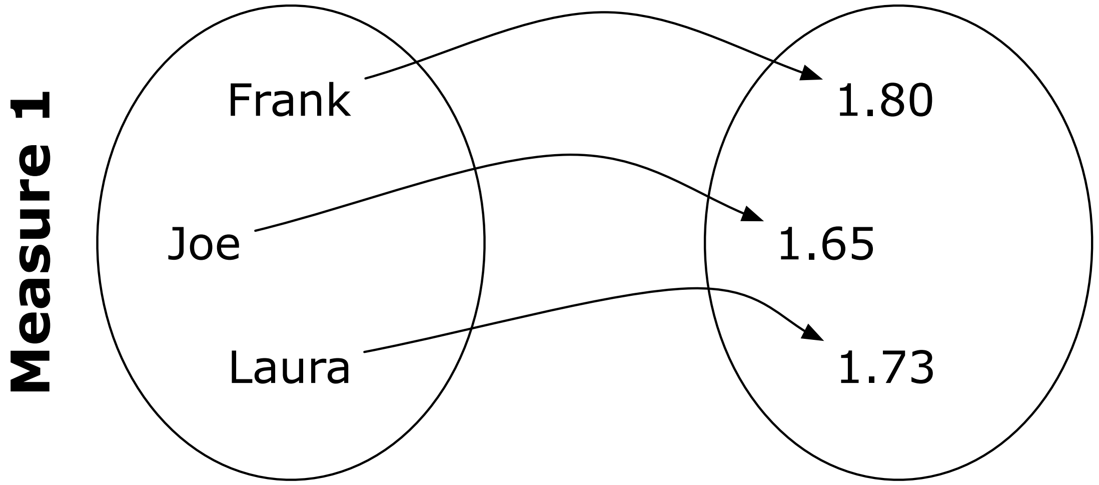
Representation Condition

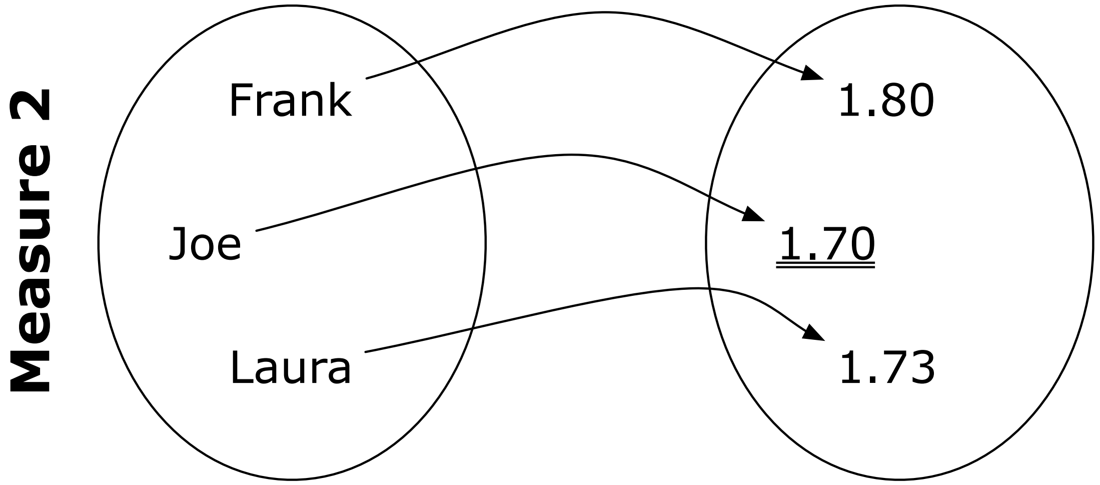
To be valid …

- a measure must satisfy the *representation condition*
  + empirical relations (in domain) ⇔ mathematical relations (in range)

In general

- the more empirical relations, the more difficult it is to find a valid measure.





Empirical Relation		**Measure 1**		**Measure 2**	
is-taller-than		x > y ???		x > y ???	
Frank, Laura	TRUE	1.80 > 1.73	TRUE	1.80 > 1.73	TRUE
Joe, Laura	FALSE	1.65 > 1.73	FALSE	1.70 > 1.73	FALSE
is-much-taller-than		x > y + 0.10		x > y + 0.10	
Frank, Laura	FALSE	1.80 > 1.73 + 0.10	FALSE	1.80 > 1.73 + 0.10	FALSE
Frank, Joe	TRUE	1.80 > 1.65 + 0.10	TRUE	1.80 > 1.70 + 0.10	FALSE

Scale

Scale

- = the symbols in the range of a measure + the permitted manipulations
  + When choosing among valid measures, we prefer a richer scale (i.e., one where we can apply more manipulations)
  + Classify scales according to permitted manipulations ⇒ Scale Type

Typical Manipulations on Scales

- Mapping:
  + Transform each symbol in one set into a symbol in another set
    > {false, true} ➤ {0, 1}
- Arithmetic:
  + Add, Subtract, Multiply, Divide
    > It will take us twice as long to implement use-case X than use-case Y
- Statistics:
  + Averages, Standard Deviation, ...
    > The average air temperature in Antwerp this winter was 8°C

Scale Types

Name	Characteristics / Permitted Manipulations	Example / Forbidden Manipulations
Nominal	- n different symbols - no ordering	{true, false} {design error, implementation error}
	- all one-to-one transformations	- no magnitude, no ordering - no median, no percentile
Ordinal	- n different symbols - ordering is implied	{trivial, simple, moderate, complex} {superior, equal, inferior}
	- order preserving transformations - median, percentile	- no arithmetic - no average, no deviation
Interval	Difference between any pair is preserved by measure	Degrees in Celsius or Fahrenheit
	- Addition (+), Subtraction (-) - Averages, Standard Deviation - Mapping have the form M = aM’ + b	- no Multiplication (*) nor Division (/) (“20$_o$C is twice as hot as 10$_o$C” is forbidden as expression)
Ratio	Difference and ratios between any pair is preserved by measure. There is an absolute zero.	Degrees in Kelvin Length, size, …
	- all arithmetic - Mappings have the form M = aM’	nihil

Scales

- What kind of measurement scale would you need to say
  + “A specification error is worse than a design error”?

- And what if we want to say
  + “A specification error is *twice* as bad as a design error?”

GQM

Goal - Question - Metrics approach

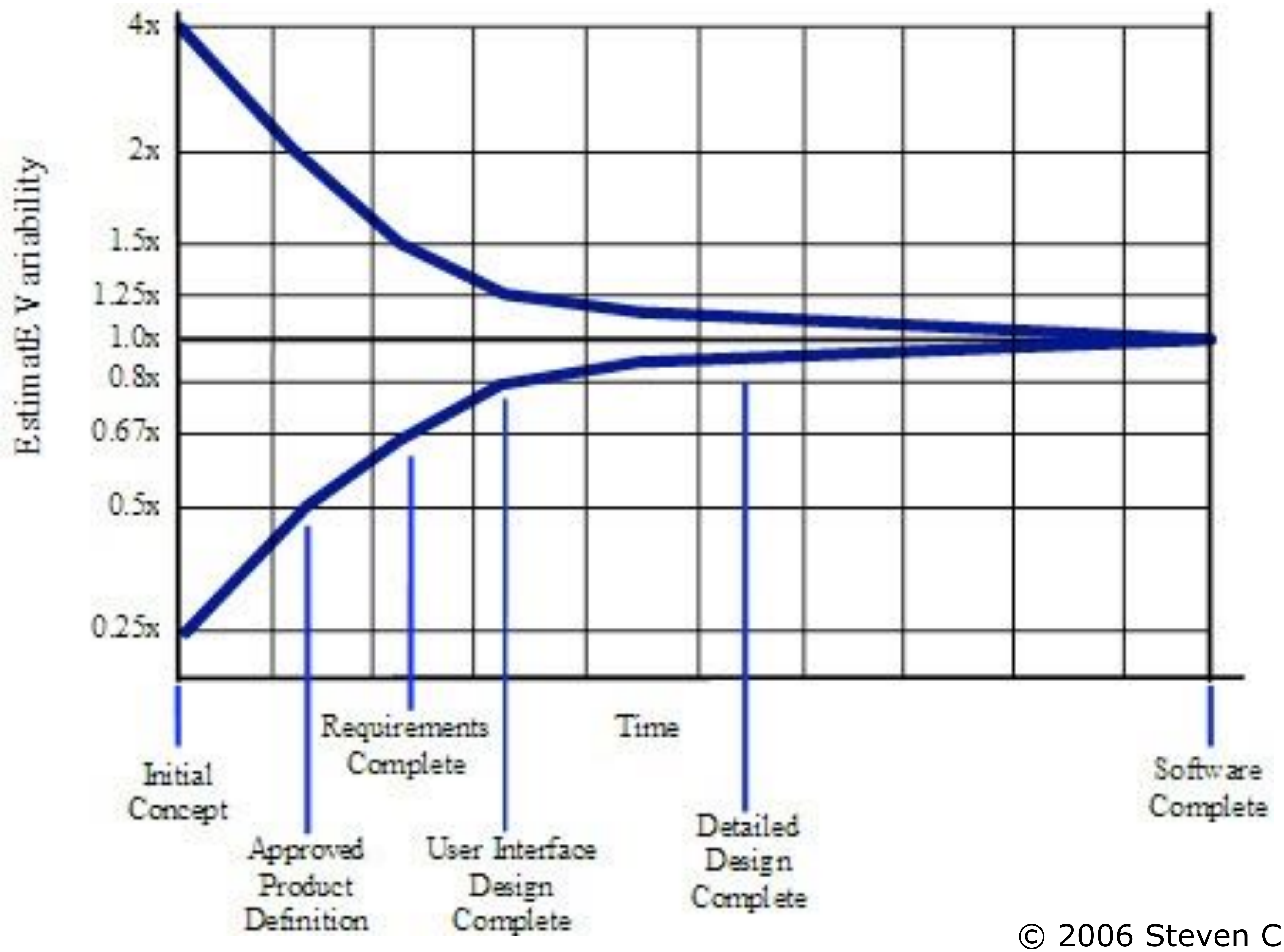
- V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," in IEEE Transactions on Software Engineering, vol. SE-10, no. 6, pp. 728-738, Nov. 1984, doi: 10.1109/TSE.1984.5010301.

- Define Goal
  + e.g., “How effective is the coding standard XYZ?”

- Break down into Questions
  + “Who is using XYZ?”
  + “What is productivity/quality with/without XYZ?”

- Pick suitable Metrics
  + Proportion of developers using XYZ
  + Their experience with XYZ ...
  + Resulting code size, complexity, robustness ...

Effort Estimation — Cone of Uncertainty



© 2006 Steven C. McConnell

Estimation techniques

Estimation Strategies

- Expert judgement: Consult experts and compare estimates
  > Cheap and very accurate, but unpredictable
- Estimation by analogy: Compare with other projects in the same domain
  > Cheap and quite accurate, but limited applicability
- Parkinson's Law: Work expands to fill the time available
  > pessimistic management strategy
- Pricing to win: You do what you can with the budget available
  > requires trust between parties
- Empirical Estimation: You estimate based on an empirical data

Empirical Estimation

- ("Decomposition" and "Algorithmic cost modeling" are used in combination)
- Decomposition: Estimate costs for components + integrating costs ...
  > top-down or bottom-up estimation
- Algorithmic cost modeling: Exploit database of historical facts
  to map component size on costs
  > requires correlation data

Algorithmic Cost Modeling

1) Choose system model

- Formula consisting of product and process attributes + parameters
  - + product attributes
    - requirements specification size:
      typically some form of word count
    - code size: typically in Lines of Code or Function Points
  - + process attribute
    - number of persons available
    - complexity of project

2) Calibrate system model

- Choose values for parameters based on historical costing data

3) Measure (or estimate) attributes

- Some attributes are fixed, others may vary
  > choose to fit project needs

4) Calculate Effort

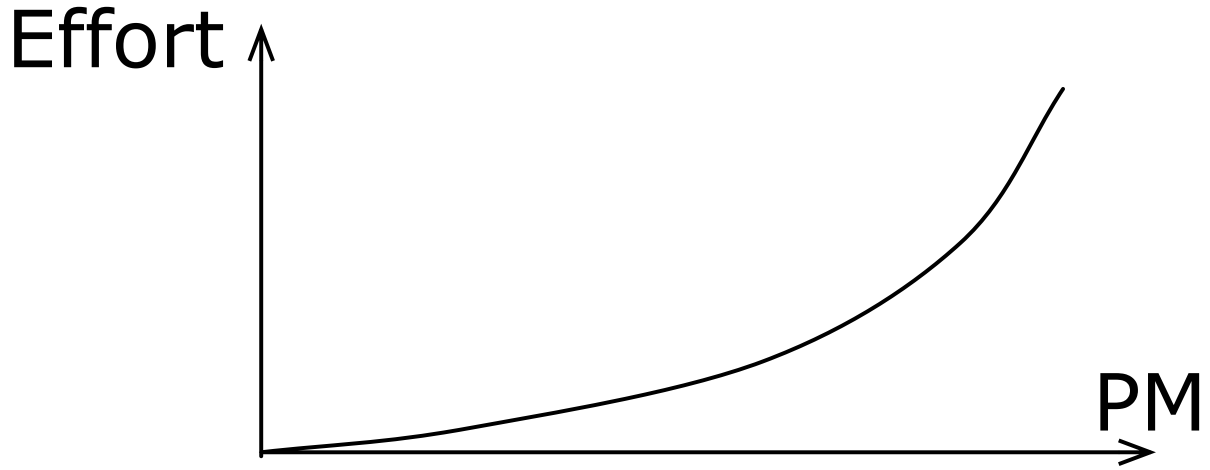
- ... and iterate until satisfied

Examples

- COCOMO (Constructive Cost Model)
- Putnam’s model; the SLIM tool (Software Lifecycle Management)

COCOMO Model (before calibration)

Model: Effort = C x PM^{S}



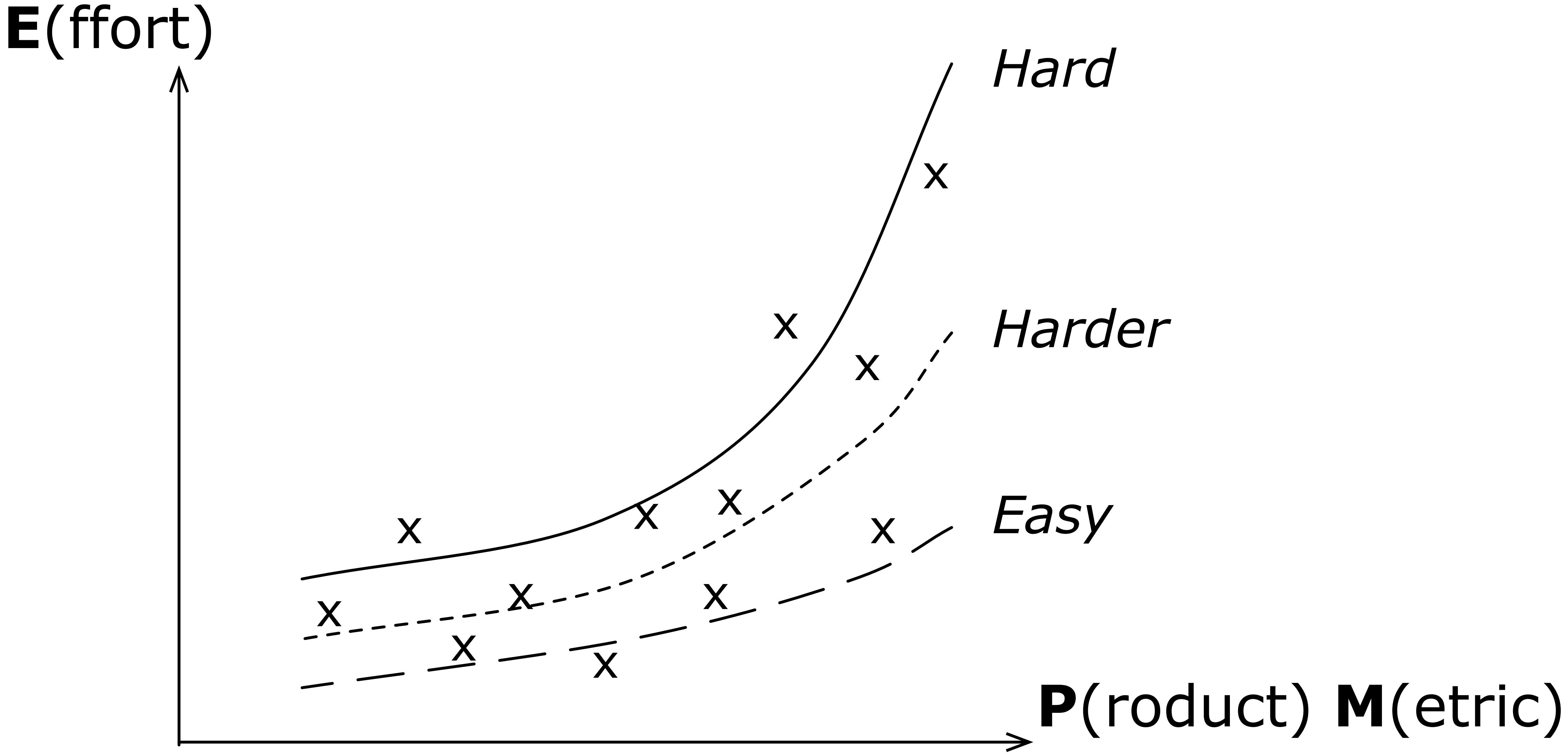
- C is a complexity factor
- PM is a product size metric
  + size (lines of code)
  + functionality (function points)
- exponent S is close to 1, but increasing for difficult projects

Values for C and S?

- regression analysis against database of more than 60 projects

COCOMO Regression analysis

- Gather “time required” (E) and “number of source code instructions” (PM) for 60 projects
- Projects were classified as EASY, HARDER and HARD
- Afterwards regression analysis to find values for C and S in E = C x PM^{S}



COCOMO Model (with calibration)

Organic mode

- Small teams, familiar environment, well-understood applications, no difficult non-functional requirements (EASY)

  > Effort = 2.4 (KDSI) $^{1.05}$ x M

  [KDSI = Kilo Delivered Source Instructions]

Semi-detached mode.

- Project team may have experience mixture, system may have more significant non-functional constraints, organization may have less familiarity with application (HARDER)

  > Effort = 3 (KDSI) $^{1.12}$ x M

Embedded Hardware/software systems.

- Tight constraints, unusual for team to have deep application experience (HARD)

  > Effort = 3.6 (KDSI) $^{1.2}$ x M

M (between 0.7-1.66) is calibration factor for fine-tuning

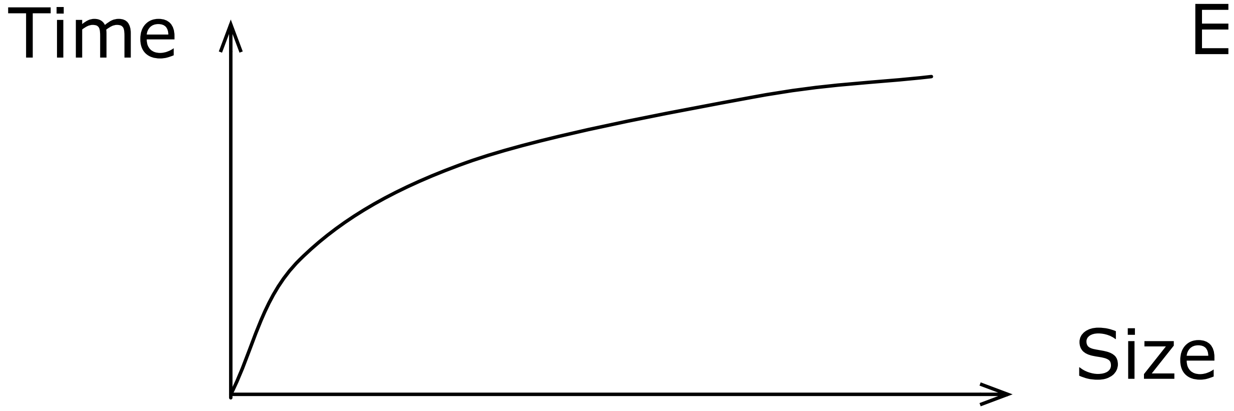
- taking into account quality attributes (reliability, performance)
- and project constraints (tool usage, fast to market)

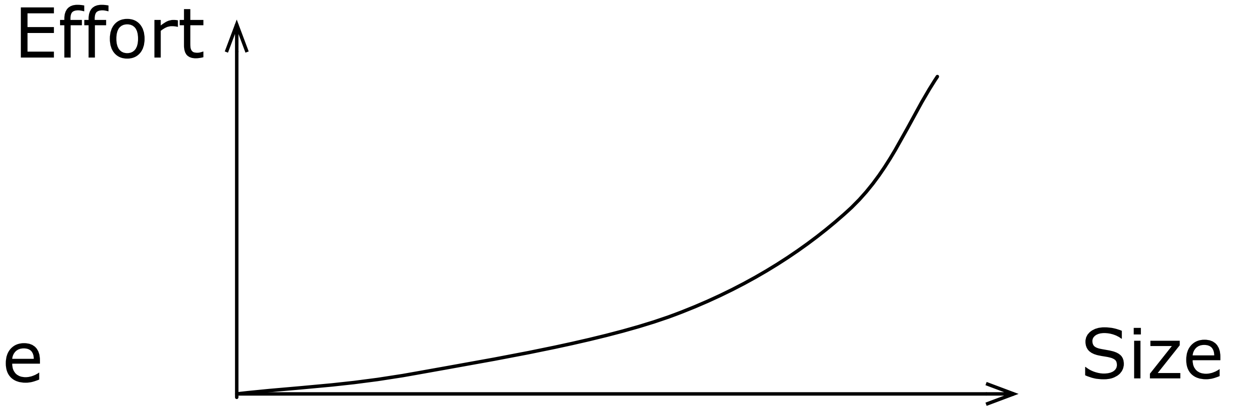
Putnam’s Model

- Based on + 7.200 projects !

$$Size = ProcessProductivity \times \sqrt[3]{\frac{Effort}{\beta}} \times \sqrt[3]{Time^4}$$

- *Size*: quantity of function; typically size (lines of code; function points)
  - a product at a given defect rate (reliability is implicitly implied)
- *Process Productivity*: amount of functionality for time and effort expended
- *Effort*: the amount of work expended (in person-months)
- *β*: A calibration factor, close to 1.
  - > 1: for large, complex projects with large teams
  - < 1: for small, simple projects with small teams
- *Time*: the duration of the project (in calendar months)
  (in the right-hand side of the equation, this is the **scheduled** time for the project)





Time & Effort are interdependent

- If you want to finish earlier (= decrease scheduled time), you should
  > increase / decrease
- the effort
  > a lot / a little .

increase	a lot
decrease	a little
don’t know	don’t know

Putnam’s Model: Deriving Productivity

Productivity is normally defined as Size / Effort

$$Size = \mathrm{Pr}ocess\mathrm{Pr}oductivity \times \sqrt[3]{\frac{Effort}{\beta}} \times \sqrt[3]{Time^4}$$

$$Size^3 = \mathrm{Pr}ocess\mathrm{Pr}oductivity^3 \times \frac{\mathrm{Effort}}{\beta} \times Time^4$$

$$\mathrm{Pr}ocess\mathrm{Pr}oductivity^3 = \frac{Size^3}{\frac{\mathrm{Effort}}{\beta} \times Time^4}$$

$$\mathrm{Pr}ocess\mathrm{Pr}oductivity^3 \times Time^4 = \frac{Size^3}{\left(\frac{\mathrm{Effort}}{\beta}\right)}$$

$$\frac{\left(\mathrm{Pr}ocess\mathrm{Pr}oductivity^3 \times Time^4\right)}{\beta} = \frac{Size^3}{Effort}$$

$$\boxed{\frac{\left(\mathrm{Pr}ocess\mathrm{Pr}oductivity^3 \times Time^4\right)}{\mathrm{Size}^2 \times \beta} = \frac{Size}{Effort}}$$

Putnam’s Model: Productivity

Productivity is normally defined as Size / Effort

$$\boxed{\frac{\left(ProcessProductivity^{3} \times Time^{4}\right)}{\mathrm{Size}^{2} \times \beta} = \frac{Size}{Effort}}$$

Conventional productivity (Size / Effort)
is dependent on (scheduled) Time !

- Time: is raised to the fourth power
  + increase scheduled time a little
    > will *increase* productivity a lot !
  + decrease scheduled time a little
    > will *decrease* productivity a lot !

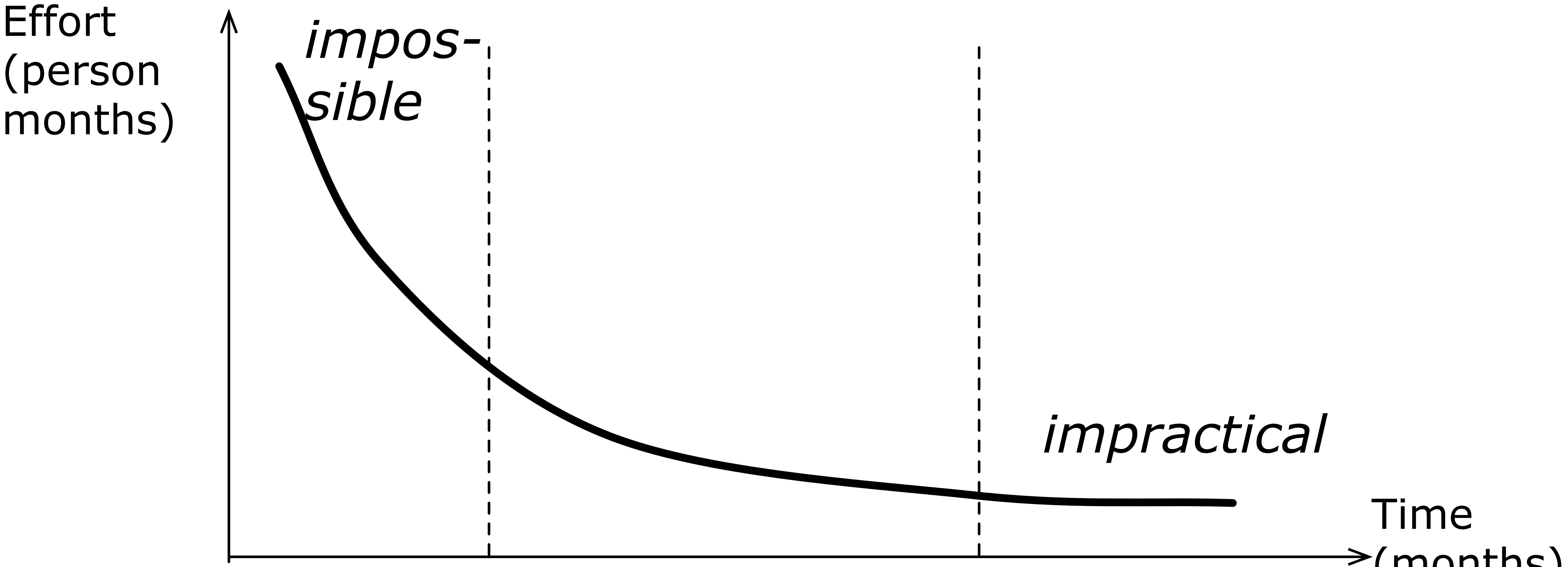
- Process Productivity: is raised to the 3rd power
  - having good people with good tools and process has a lot of impact

- Size: is raised to the 2nd power in denominator
  - smaller projects have better productivity

Time & Effort are interdependent

$$\sqrt[3]{\frac{Effort}{\beta}} \times \sqrt[3]{Time^4} = \frac{Size}{ProcessProductivity}$$

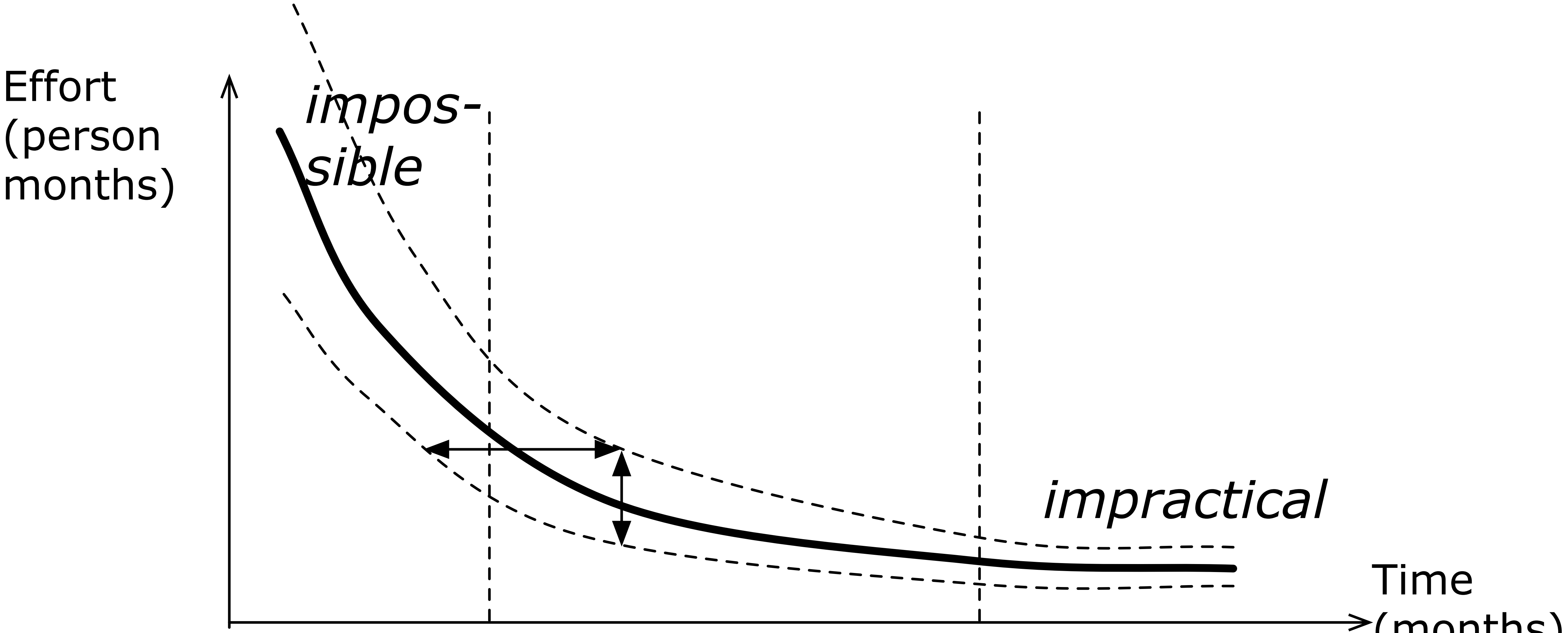
- Assume that the size and process productivity are given (i.e. specification is complete; tools & process is defined)
- Time is raised to the power (4/3)
  - + To finish earlier, you must invest *MANY* more man months
  - + To decrease the cost, you must spend *A LOT* more time
    - If you don't: reliability (implicitly implied in Size) will adapt



Time & Effort are interdependent

$$\sqrt[3]{\frac{Effort}{\beta}} \times \sqrt[3]{Time^4} = \frac{Size}{Process Productivity}$$

- size and process productivity are *estimated*
  > degree of uncertainty (inherent in calibration factor β)
- Time is raised to the power (4/3)
  + Project bidding with reduced time: uncertainty has larger effect
  + Close to the “Impossible” region: risk of entering into it



Size: Lines of code

Lines of Code (LOC) as a measure of system size?

- Counter intuitive for effort estimation
  + Once you know the lines of code, you have done the effort
  + Typically dealt with by "estimating" the lines of code needed
- Easy to measure; but not well-defined for modern languages
  + What's a line of code?
  + What modules should be counted as part of the system?
- Assumes linear relationship between system size and volume of documentation
  + Documentation is part of the product too!
- A poor indicator of productivity
  + Ignores software reuse, code duplication, benefits of redesign
  + The lower level the language, the more productive the programmer
  + The more verbose the programmer, the higher the productivity

Yet, lines of code is the size metric that is used most often ...
because it is very *tangible* (representation condition)

Size: Function points

Function Points (FP)

- Based on a combination of program characteristics:
  - + external inputs (e.g., screens, files) and outputs (e.g., reports)
  - + user interactions (inquiries)
  - + external interfaces (e.g., API)
  - + files used by the system (logical files, database tables, ...)
- A weight is associated with each of these depending on complexity
- Function point count is sum, multiplied with complexity

	Weighting Factor			
Item	**Simple**	**Average**	**Complex**	
External Inputs	... x 3 = ...	... x 4 = ...	... x 6 = ...	sum(left)
External Outputs	... x 4 = ...	... x 5 = ...	... x 7 = ...	sum(left)
Inquiries	... x 3 = ...	... x 4 = ...	... x 6 = ...	sum(left)
External Interfaces	... x 5 = ...	... x 7 = ...	... x 10 = ...	sum(left)
Logical Files	... x 7 = ...	... x 10 = ...	... x 15 = ...	sum(left)
Unadjusted Function Points				sum(above)
Adjusted Function Points	x Complexity factor (0.65...1.35)		...	...

Function Points: Trade-offs

Points in Favor

- Can be measured after design
  + not after implementation
- Independent of implementation language
- Measure functionality
  + customers willing to pay
- Works well for data-processing

Counter argument

- Requires fully specified design
  + not in the early life cycle
- Dependent on specification method
- Counterintuitive
  + 2000 FP is meaningless
- Other domains less accepted

Points Against

- Requires subjective expert judgement
- Cannnot be calculated automatically

Counter argument

- International Function Point Users Group
  + publishes rule books
- Backfire LOC in FP via table of average FP for a given implementation language

Conclusion

- To compare productivity, defect density, ...
  + Function Points are preferable over Lines of Code
- To estimate effort, Function Points come quite late in the life-cycle

Size: Use Case Points

- (see [Schn98]; Chapter 8: Use Cases and the Project Plan)

Use CasePoints (UCP)

- Based on a combination of use case characteristics (actors & use cases)
- A weight is associated with each of these depending on complexity
  - + Actors:
    - API = simple; command line or protocol = average; GUI = complex
  - + use cases
    - number of transactions: <= 3 = simple; <= 7 average; > 7 complex
    - or number of CRC-cards: <= 5 = simple; <= 10 average; > 10 complex
- sum = Unadjusted Use Case Points

	Weighting Factor			
Item	Simple	Average	Complex	Total
Actors	... x 1 = ...	... x 2 = ...	... x 3 = ...	sum(left)
Use Cases	... x 5 = ...	... x 10 = ...	... x 15 = ...	sum(left)
Unadjusted Use Case Points				*sum(above)*
Adjusted Use Case Points	x Technical Complexity factor [0.6 … 1.3] x Environmental Complexity Factor [1.4 … 2.75]			**...**

Use Case Points: Technical Complexity factor

Calculation of technical complexity factor.
Rate every complexity factor on a scale from 0 (irrelevant) to 5 (essential).

Complexity factor	Rating (0 .. 5)	Weight	Total
Distributed system	…	x 2 = …	…
Performance objectives	…	x 1 = …	…
End-use efficiency	…	x 1 = …	…
Complex internal processing	…	x 1 = …	…
Code must be reusable	…	x 1 = …	…
Easy to install	…	x 0.5 = …	…
Easy to use	…	x 0.5 = …	…
Portable	…	x 2= …	…
Easy to change	…	x 1 = …	…
Concurrent	…	x 1 = …	…
Special security	…	x 1 = …	…
Direct access for 3rd parties	…	x 1 = …	…
Special user training	…	x 1 = …	…
Total Technical Complexity	= sum(above)		
Technical Complexity factor	TCF = 0.60 + (Total Technical Complexity x 0.01)		

Maximum possible is 70

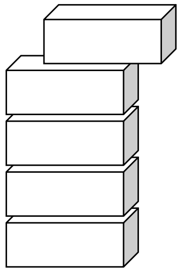
Use Case Points: Environmental Complexity factor

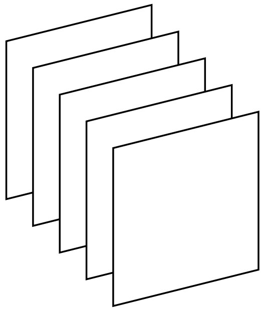
Calculation of environmental complexity factor.
Rate every factor on a scale from 0 (no experience) to 5 (expert).

Complexity factor	Rating (0 .. 5)	Weight	Total
Familiarity with development process	…	x 1.5 = …	…
Application experience	…	x 0.5 = …	…
Object-oriented experience of team	…	x 1 = …	…
Lead analyst capability	…	x 0.5 = …	…
Motivation of the team	…	x 1 = …	…
Stability of requirements	…	x 2 = …	…
Part-time staff	…	x -1 = …	…
Difficult programming language	…	x -1= …	…
Total Environmental Complexity	= sum(above)		
Environmental Complexity factor	ECF = 1.4 + (Total Environmental Complexity x -0.03)		

Maximum possible is 22,5

Story Points (Planning Poker)





CAPSTONE PROJECT

1/2	1	2	3	5	8	13	20	40	100	∞

- Choose one representative item of size 1.
- Compare items against representative.
- Put in the corresponding bin.
- Bins are classified according to the fibonacci series.
- Group game with physical cards.

Public Domain

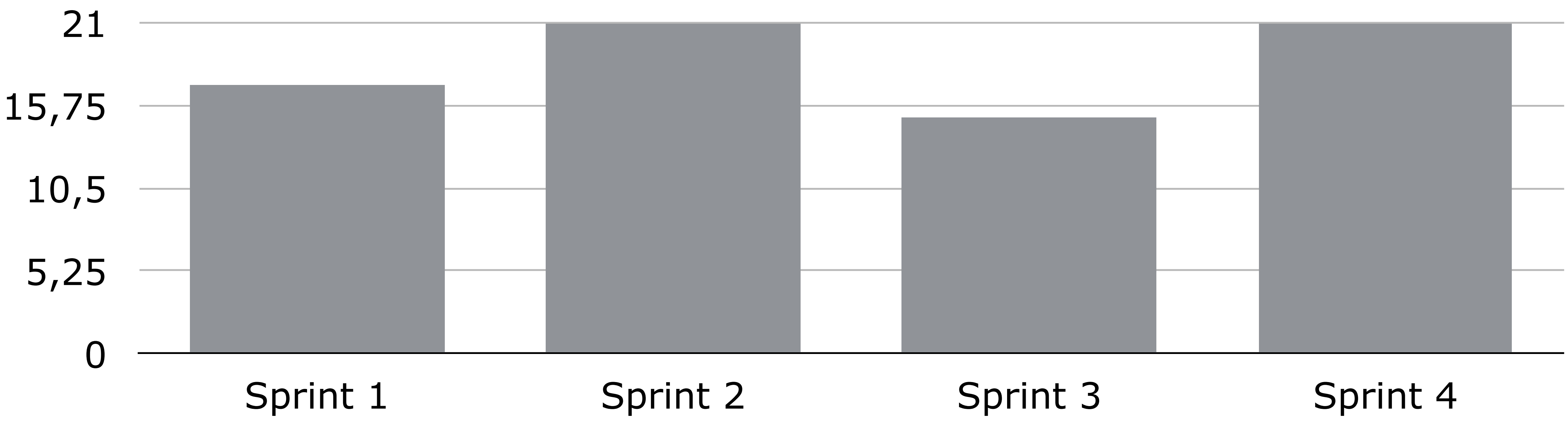
Velocity

= Amount of business value created for a given time interval

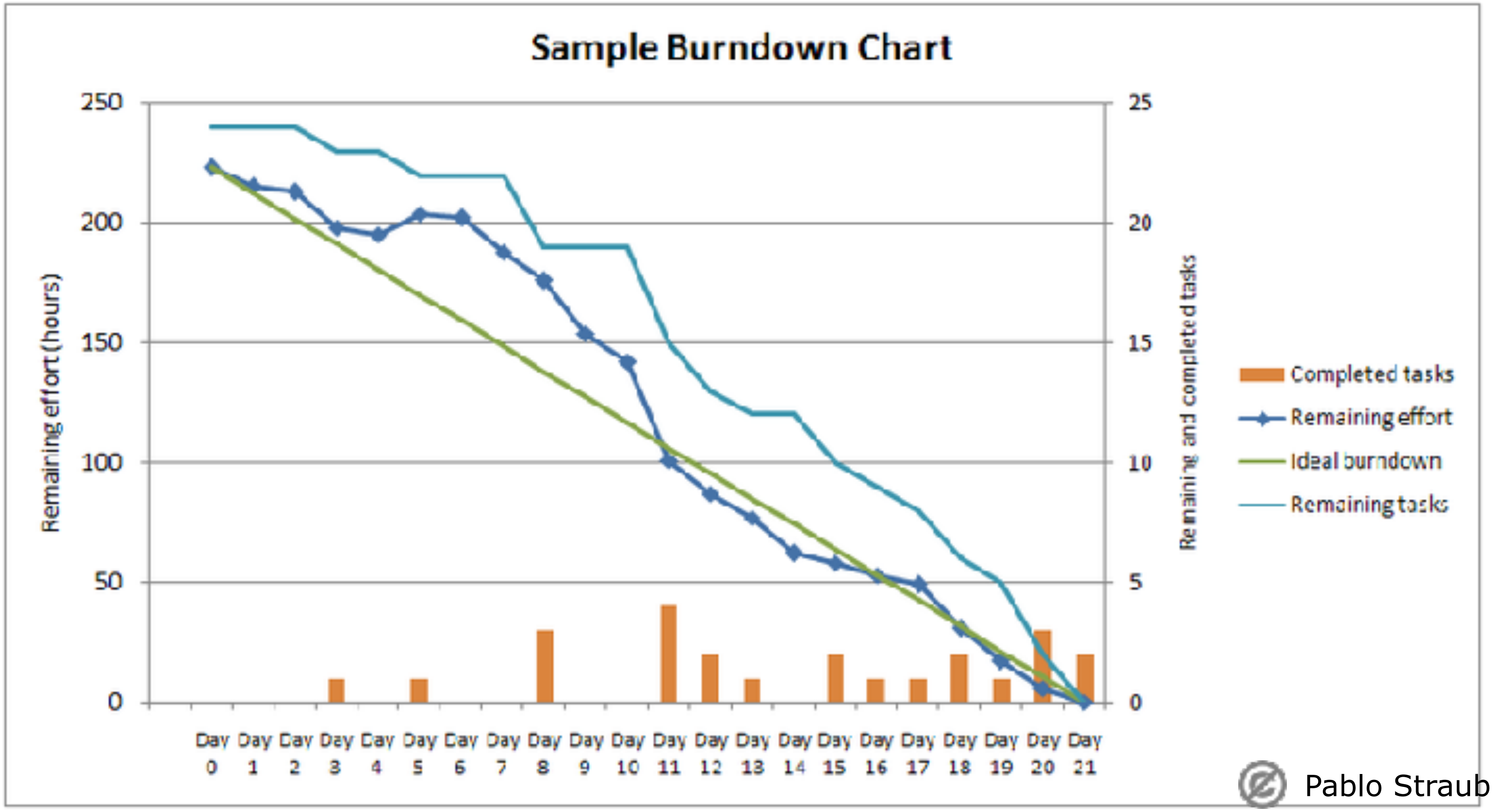
Typically: amount of story points for a sprint

Relative measure

- Absolute values are meaningless but trends are important
- Do not compare across teams!



BurnDown Charts



Pablo Straub Wikipedia

Steps (Repeat every day)

- 1. Calculate remaining effort
  > # days + # work hours per day
- 2. Track daily progress
  > # story points *completed* per work day
- 3. Calculate remaining effort
  > # unfinished story points * velocity per work hour

Quizz

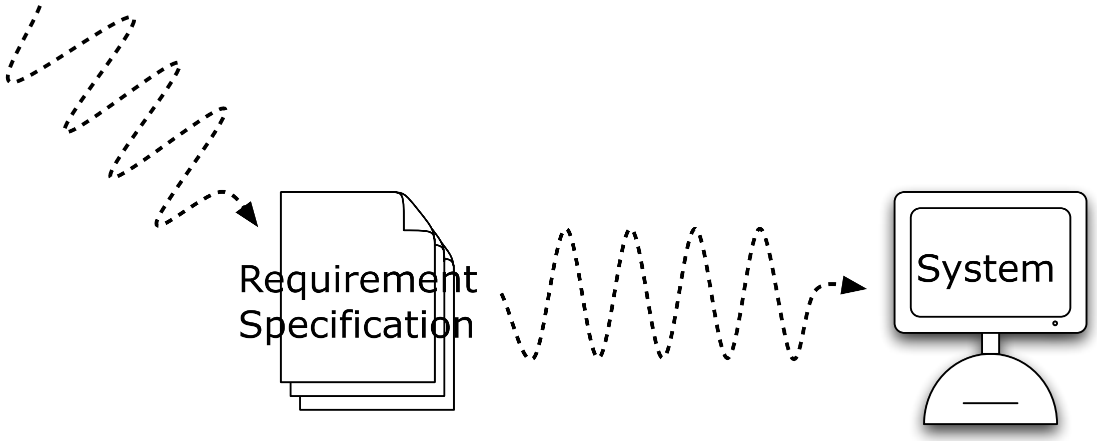
Give three metrics for measuring size of a software product.

Product =System

- Lines of code

Product = Requirements

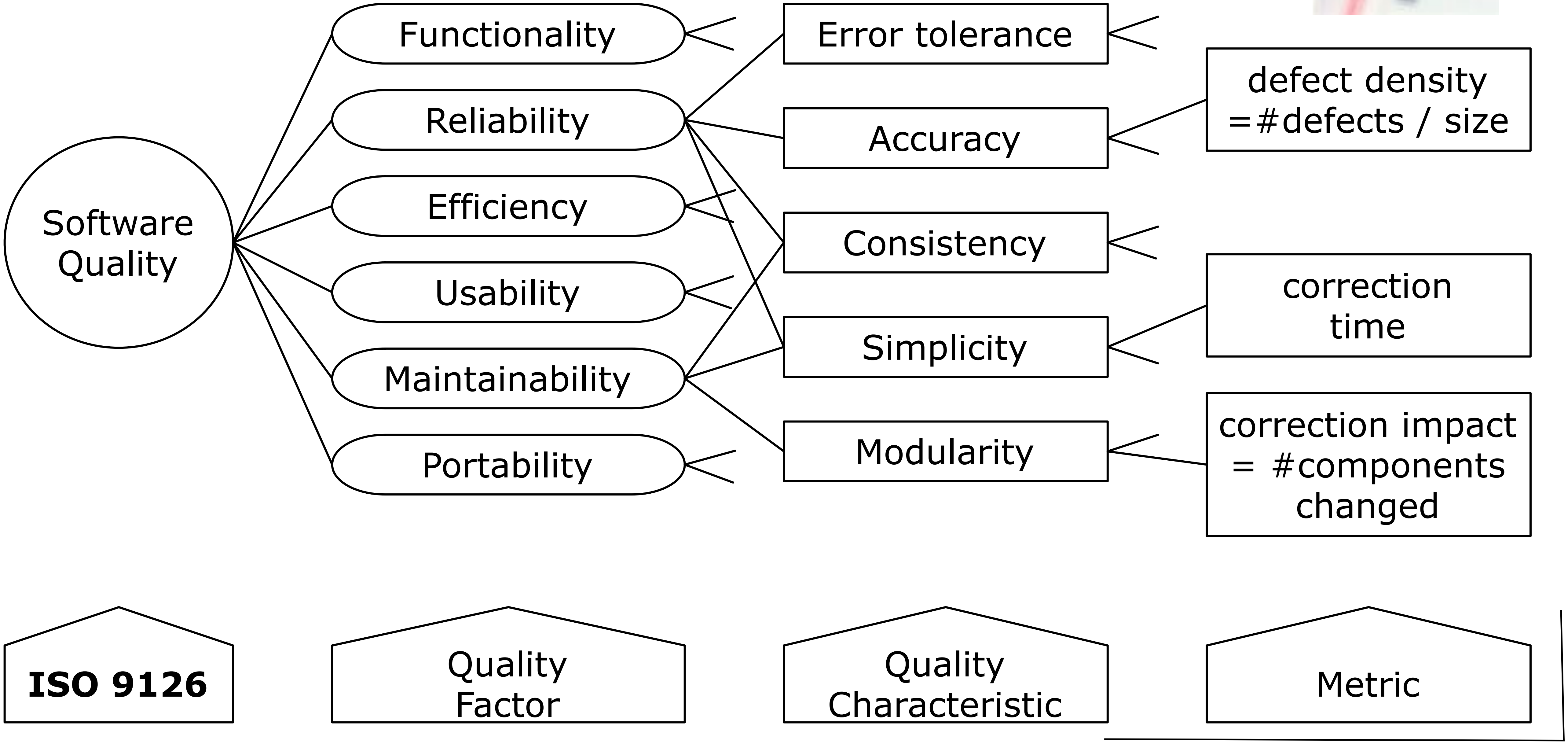
- Function Points
- Use case points
- Story points



Quantitative Quality Model

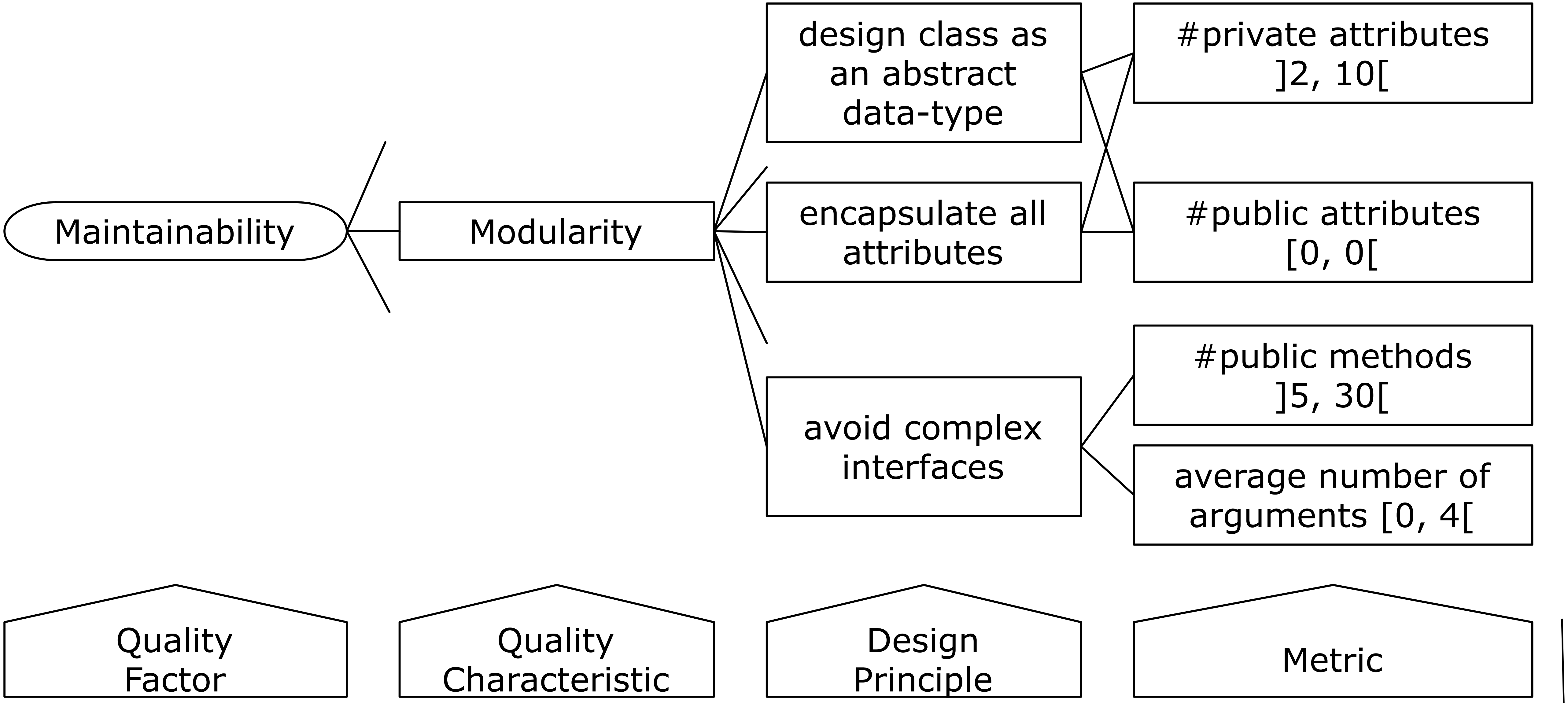
Quality according to ISO 9126 standard

- Divide-and conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes



“Define your own” Quality Model

- Define the quality model with the development team
  + Team chooses the characteristics, design principles, metrics...
  + ... and the *thresholds*



Maintainability
Modularity
design class as an abstract data-type
encapsulate all attributes
avoid complex interfaces
#private attributes]2, 10[
#public attributes [0, 0[
#public methods]5, 30[
average number of arguments [0, 4[
Quality Factor
Quality Characteristic
Design Principle
Metric

Sample Size Metrics

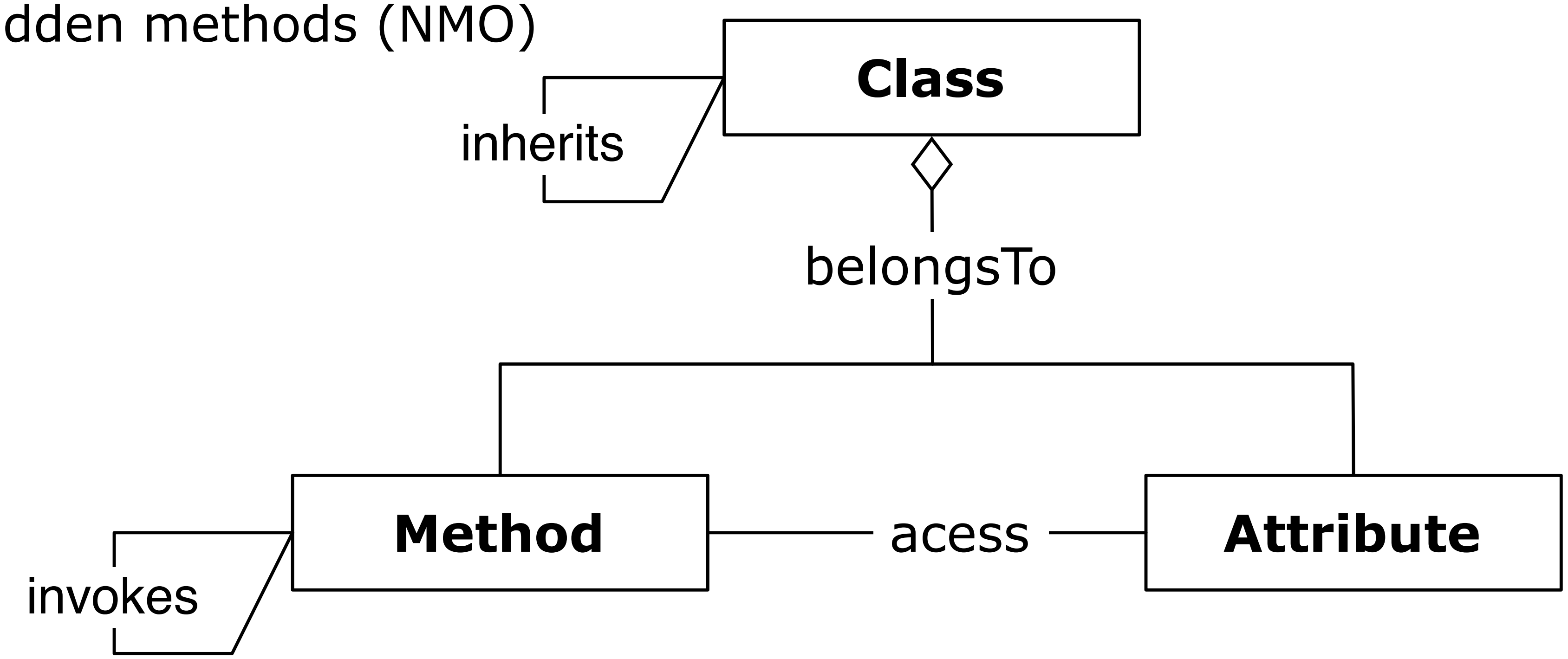
These are Internal Product Metrics

Inheritance Metrics

- hierarchy nesting level (HNL)
- # immediate children (NOC)
- # inherited methods, unmodified (NMI)
- #overridden methods (NMO)

Class Size Metrics

- # methods (NOM)
- # attributes, instance/class (NIA, NCA)
- # Σ of method size (WMC)



Class

inherits

belongsTo

Method

acess

Attribute

invokes

Method Size Metrics

- # invocations (NOI)
- # statements (NOS)
- # lines of code (LOC)
- # arguments (NOA)

Sample Coupling & Cohesion Metrics

These are Internal Product Metrics

+ Following definitions stem from
  - S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, June 1994, doi: 10.1109/32.295895.

Coupling Between Objects (CBO)

- CBO = number of other class to which given class is coupled
  + Interpret as “number of other classes required to compile”

Lack of Cohesion in Methods (LCOM)

- collect local methods not accessing same attribute
- LCOM = number of disjoint sets

Beware

- Disagreement whether coupling/cohesion metrics satisfy the representation condition
  + Classes that are observed to be cohesive may have a high LCOM value
    - due to accessor methods
  + Classes that are not much coupled may have high CBO value
    - no distinction between data, method or inheritance coupling

Sample External Quality Metrics

Correctness (Product Metric)

- a system is correct or not, so one cannot measure correctness
- Proxy metric: **defect density** = # known defects / product size
  + product size in LOC or FP
  + # known defects is a time based count!
- do NOT compare across projects unless you're data collection is sound!

Maintainability (Product Metric)

- #time to repair certain categories of changes
  + "average time to repair"
- beware for the units
  + categories of changes is subjective
  + measuring time precisely is difficult
    - problem recognition time + administrative delay time + problem analysis time + change time + testing & reviewing time

Productivity (Process Metric)

- functionality / time
- functionality in LOC or FP; time in hours, weeks, months
- be careful to compare: the same unit does not always represent the same concept
- Does not take into account the quality of the functionality!

Developer Productivity: Multi-Faceted

Myths

1. Productivity is all about developer activity
2. Productivity is only about individual performance
3. One productivity metric can tell us everything
4. Productivity Measures ARE useful only for managers
5. Productivity is only about engineering systems and developer tools

Dimensions

1. Satisfaction
2. Performance
3. Activity
4. Communication and collaboration
5. Efficiency and flow

Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of developer productivity. Commun. ACM 64, 6 (June 2021), 46–53. https://doi.org/10.1145/3453928

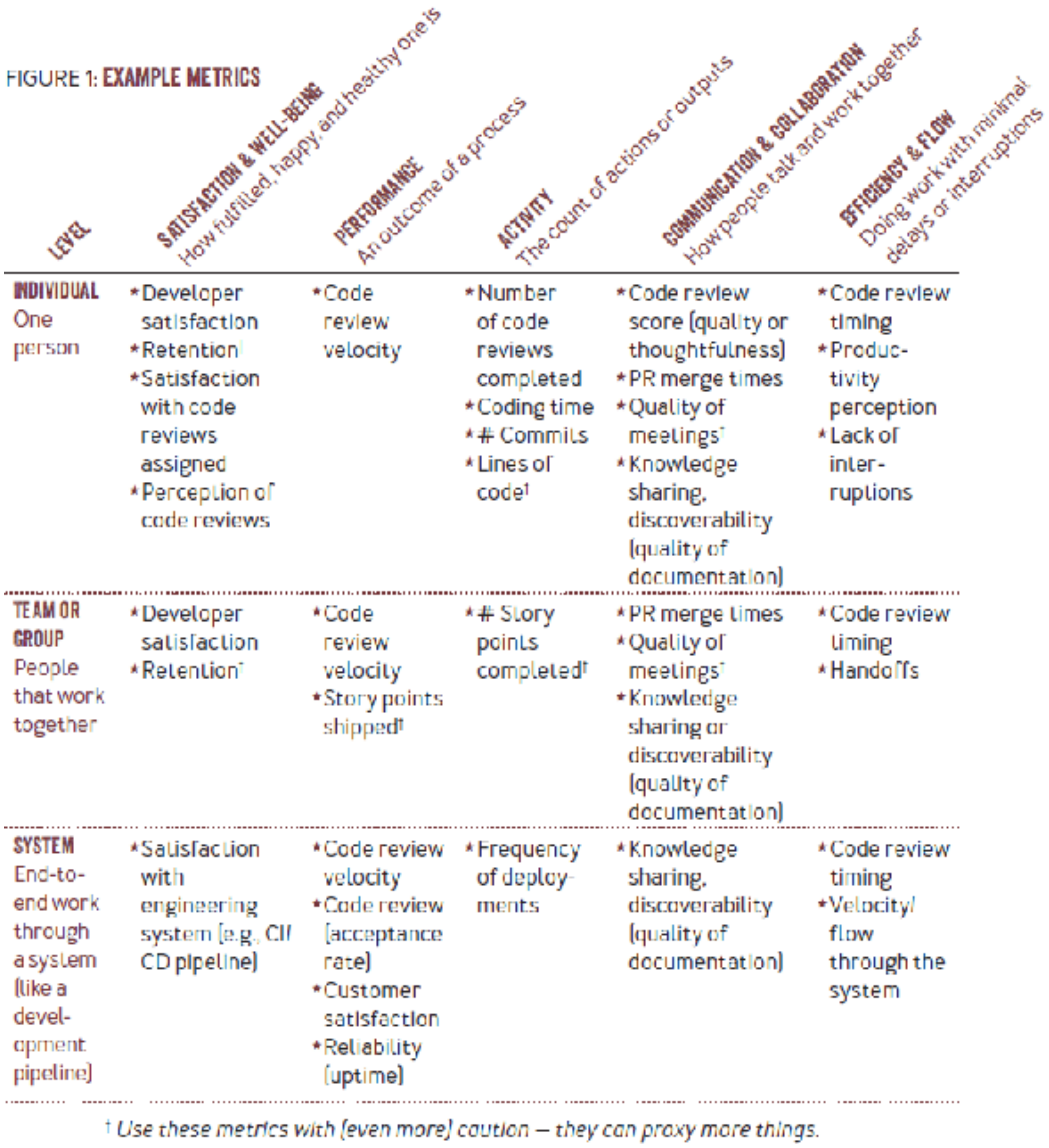


FIGURE 1: **EXAMPLE METRICS**

LEVEL	SATISFACTION & WELL-BEING How fulfilled, happy, and healthy one is	PERFORMANCE An outcome of a process	ACTIVITY The count of actions or outputs	COMMUNICATION & COLLABORATION How people talk and work together	EFFICIENCY & FLOW Doing work with minimal delays or interruptions
INDIVIDUAL One person	* Developer satisfaction * Retention† * Satisfaction with code reviews assigned * Perception of code reviews	* Code review velocity	* Number of code reviews completed * Coding time * # Commits * Lines of code†	* Code review score (quality or thoughtfulness) * PR merge times * Quality of meetings† * Knowledge sharing, discoverability (quality of documentation)	* Code review timing * Productivity perception * Lack of interruptions
TEAM OR GROUP People that work together	* Developer satisfaction * Retention†	* Code review velocity * Story points shipped†	* # Story points completed†	* PR merge times * Quality of meetings† * Knowledge sharing or discoverability (quality of documentation)	* Code review timing * Handoffs
SYSTEM End-to-end work through a system (like a development pipeline)	* Satisfaction with engineering system (e.g., CI/CD pipeline)	* Code review velocity * Code review (acceptance rate) * Customer satisfaction * Reliability (uptime)	* Frequency of deployments	* Knowledge sharing, discoverability (quality of documentation)	* Code review timing * Velocity/flow through the system

† Use these metrics with (even more) caution — they can proxy more things.

Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of developer productivity. Commun. ACM 64, 6 (June 2021), 46–53. https://doi.org/10.1145/3453928

Conclusion: Metrics for Effort Estimation

Question:

- Can metrics be used for effort estimation?

Yes, but…

- Come a bit too late in the life-cycle
  + Require a quite complete “Requirements Specification”
- Requires database of historical facts about projects
  + small numbers statistics is required if you do it yourself
  + or hire external estimation consultants (which have such database)
- Can never be the sole basis for estimating
  + models allow “trial and error” estimation
  + complement with “Expert Judgement” or “Estimate by Analogy”

However…

- Collecting historical data is a good idea anyway
  + Provides a basis for Quantitative analysis of processes
  + “Levels 4 & 5” of CMM

Conclusion: Metrics for Quality Assurance (i)

Question:

- Can internal product metrics reveal which components have good/poor quality?

Yes, but...

- Not reliable
  + false positives: “bad” measurements, yet good quality
  + false negatives: “good” measurements, yet poor quality
- Heavy weight approach
  + Requires team to develop/customize a quantitative quality model
  + Requires definition of thresholds (trial and error)
- Difficult to interpret
  + Requires complex combinations of simple metrics

However...

- Cheap once you have the quality model and the thresholds
- Good focus (± 20% of components are selected for further inspection)
  + Note: focus on the most complex components first

Conclusion: Metrics for Quality Assurance (ii)

Question:

- Can external product/process metrics reveal quality?

Yes, ...

- More reliably then internal product metrics

However...

- Requires a finished product or process
- It is hard to achieve preciseness
  + even if measured in same units
  + beware to compare results from one project to another

Summary (i)

You should know the answers to these questions

- Can you give three possible problems of metrics usage in software engineering? How does the measurement theory address them?
- What's the distinction between a measure and a metric?
- Can you give an example of a direct and an indirect measure?
- What kind of measurement scale would you need to say "A specification error is worse than a design error"? And what if we want to say "A specification error is twice as bad as a design error?"
- Explain the need for a calibration factor in Putnam's model.
- Fill in the blanks in the following sentence. Explain briefly, based on the Putnam's model.
  + If you want to finish earlier (= decrease scheduled time), you should ... the effort ... .
- Give three metrics for measuring size of a software product.
- Discuss the main advantages and disadvantages of Function Points.
- What does it mean for a coupling metric not to satisfy the representation condition?
- Can you give 3 examples of impreciseness in Lines of Code measurements?

You should be able to complete the following tasks

- Given a set of use cases (i.e. your project) calculate the use case points.
- Given a set of user stories, perform a poker planning session.

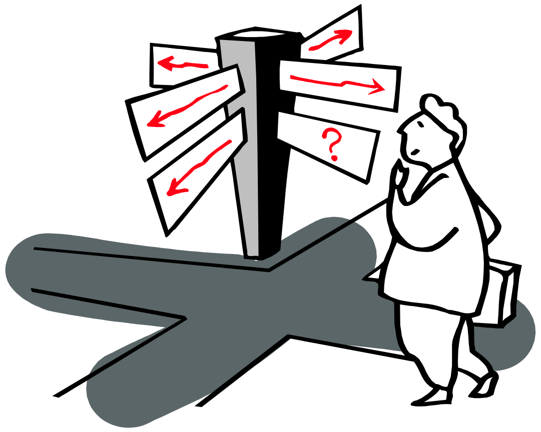
CAPSTONE PROJECT

Summary (ii)

Can you answer the following questions?

- During which phases in a software project would you use metrics?
- Why is it so important to have “good” product size metrics?
- Can you explain the two levels of calibration in COCOMO (i.e. C & S vs. M)? How can you derive actual values for these parameters?
- Can you motivate why in software engineering, productivity depends on the scheduled time? Do you have an explanation for it?
- Can you explain the cone of uncertainty? And why is it so relevant to cost estimation in software projects?
- How can you decrease the uncertainty of a project bid using Putnam’s model?
- Why do we prefer measuring Internal Product Attributes instead of External Product Attributes during Quality Control? What is the main disadvantage of doing that?
- You are a project manager and you want to convince your project team to apply algorithmic cost modeling. How would you explain the technique?
- Where would you fit coupling/cohesion metrics in a hierarchical quality model like ISO 9126?
- Why are coupling/cohesion metrics important? Why then are they so rarely used?
- Do you believe that “defect density” says something about the correctness of a program? Motivate your answer?

CHAPTER 11 – Refactoring

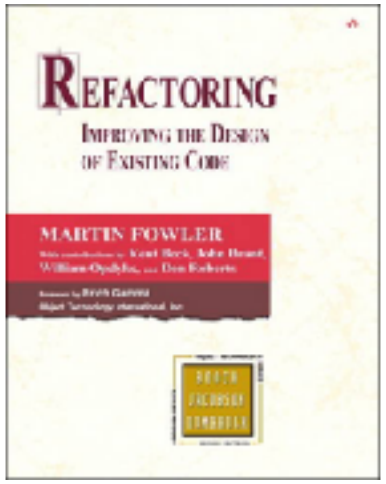


- Introduction
  + When, Why, What?
  + Which Refactoring Tools?
- Demonstration: Internet Banking
  + Iterative Development Life-cycle
  + Prototype
  + Consolidation: design review
  + Expansion: concurrent access
  + Consolidation: more reuse
- Miscellaneous
  + Tool Support
  + Code Smells
  + Refactoring God Class
    - An empirical study
  + Scrum: Technical Debt
- Conclusion
  + Correctness & Traceability

Literature

- [Somm05]: Chapter “Software Evolution”
- [Pres00], [Ghez02]: Chapters on Reengineering / Legacy Software

- [Fowl99] Refactoring, Improving the Design of Existing Code by Martin Fowler, Addison-Wesley, 1999.
  + A practical book explaining when and how to use refactorings to cure typical code-smells.

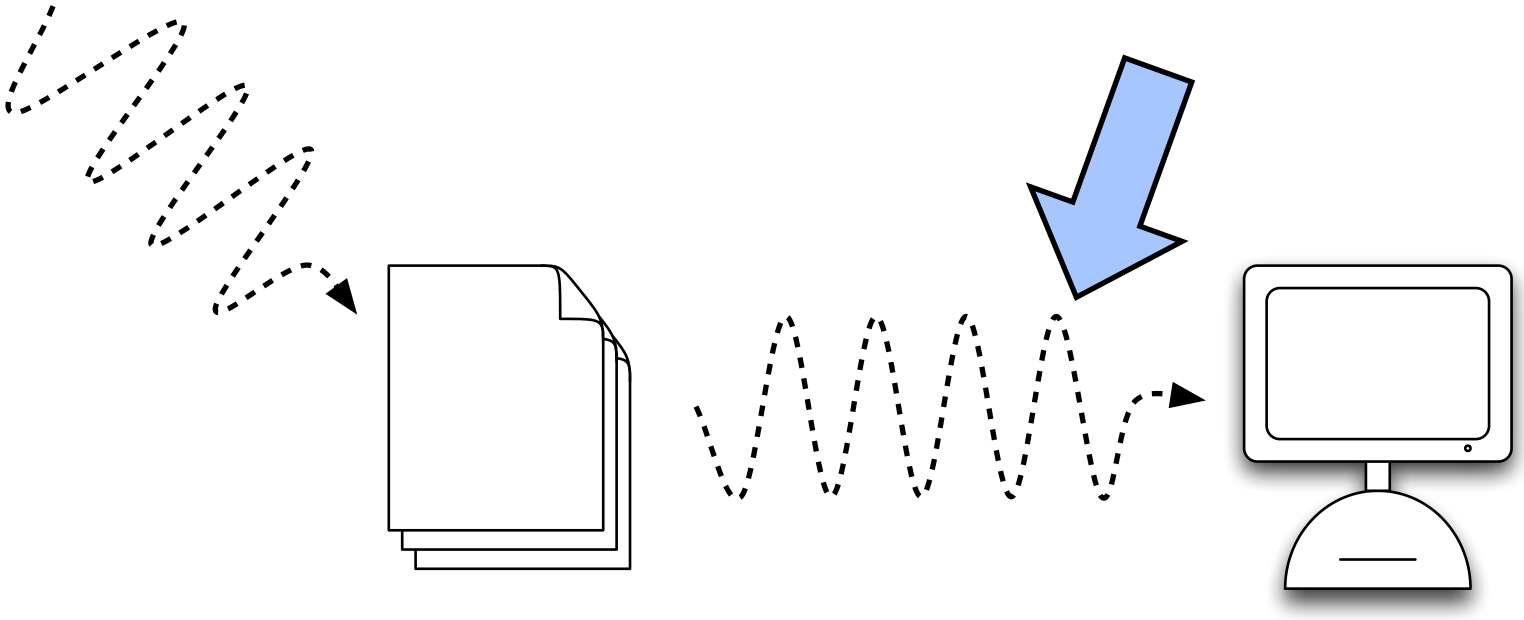


- [Deme02] Object-Oriented Reengineering Patterns by Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, Morgan Kaufmann, 2002.
  + A book describing how one can reengineer object-oriented legacy systems.

Web-Resources

- Following web-site lists a number of relevant code smells (= symptoms in code where refactoring is probably worthwhile) https://wiki.c2.com/?CodeSmell

When Refactoring?



Any software system must be maintained

- The worst that can happen with a software system is that the people actually *use* it.
  - + >> Users will request changes ...
  - + >> Intangible nature of software
    - > … makes it hard for users to understand the impact of changes

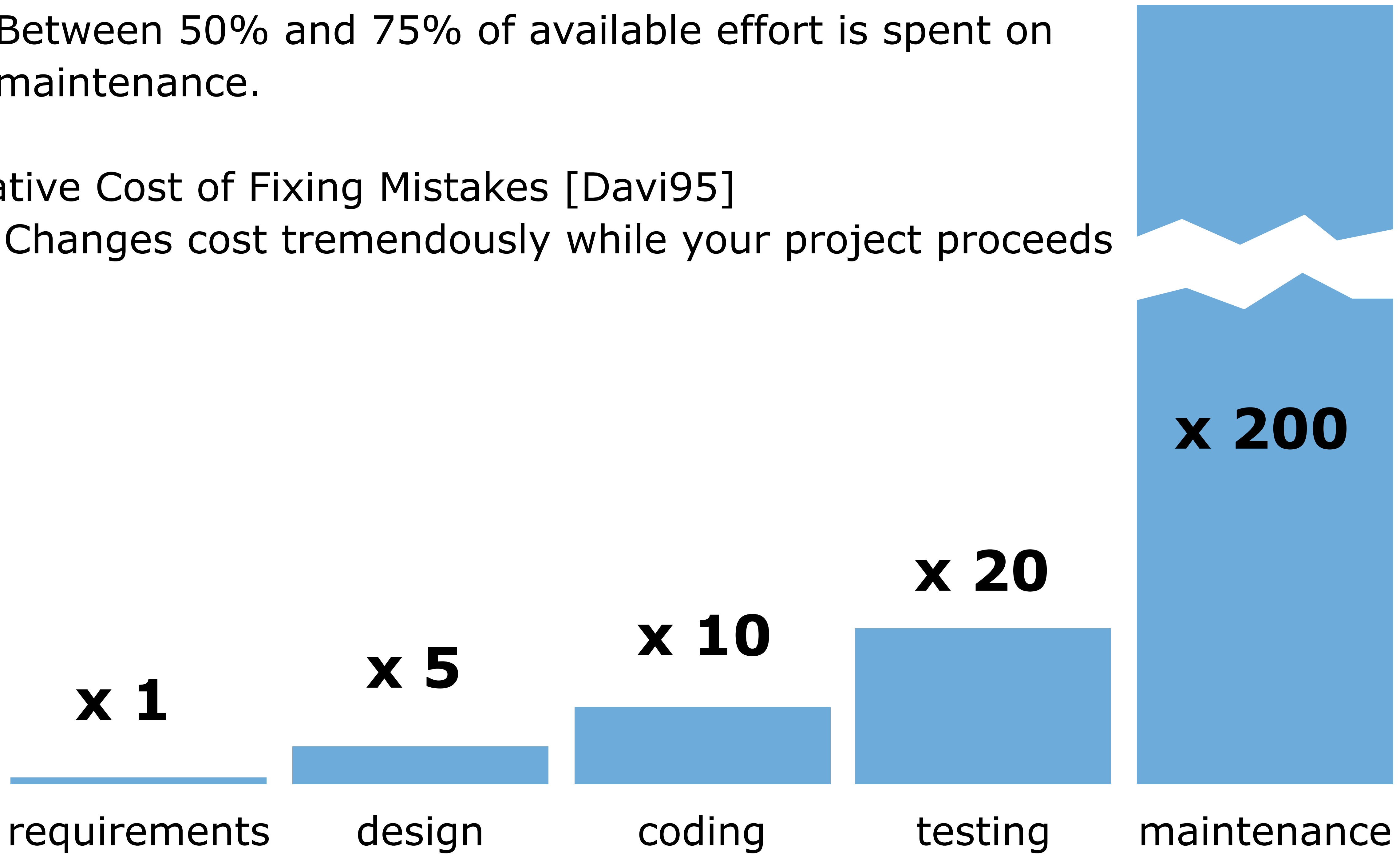
Why Refactoring? (1/2)

Relative Effort of Maintenance [Lien80]

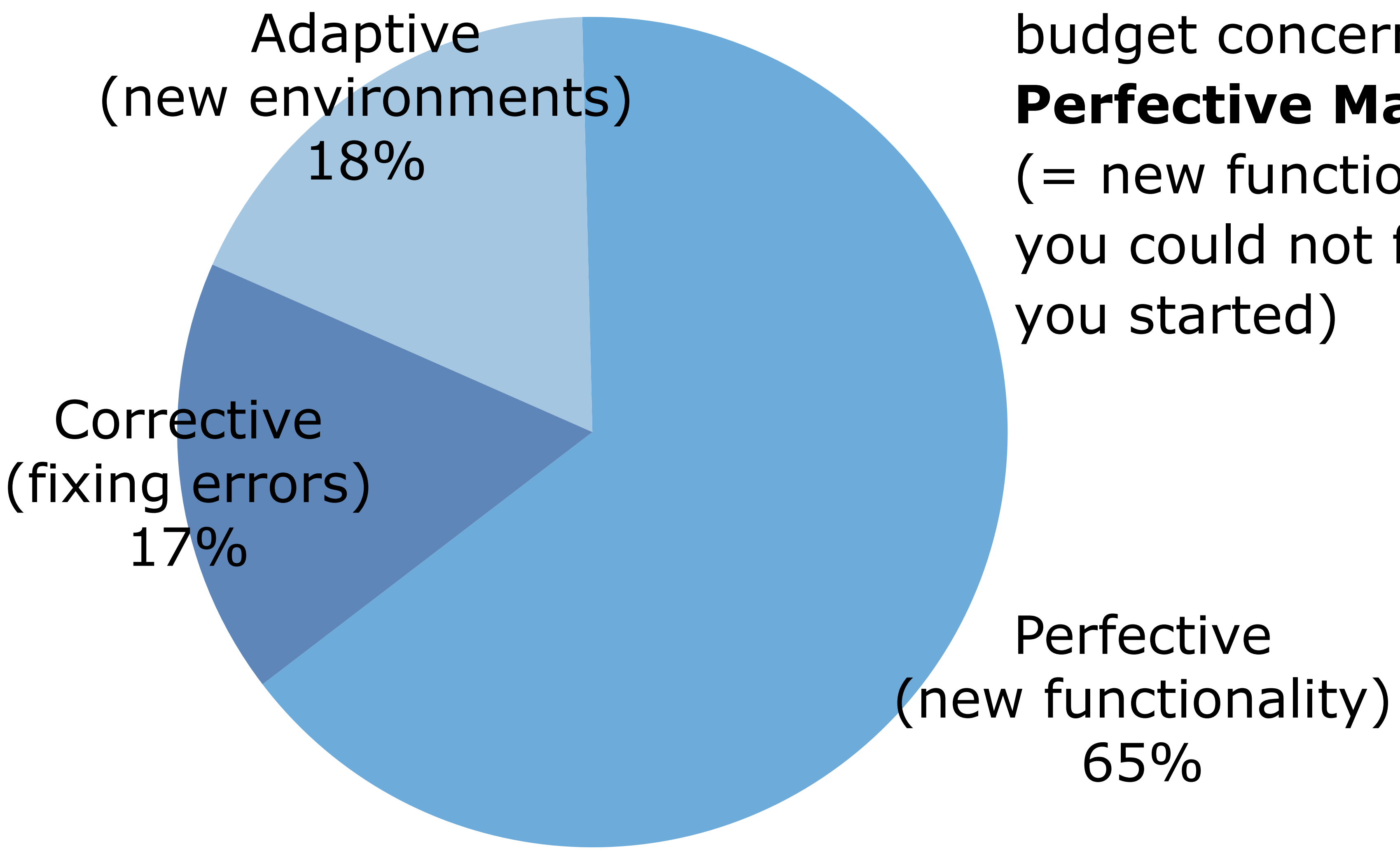
- Between 50% and 75% of available effort is spent on maintenance.

Relative Cost of Fixing Mistakes [Davi95]

⇒ Changes cost tremendously while your project proceeds



Why Refactoring? (2/2)



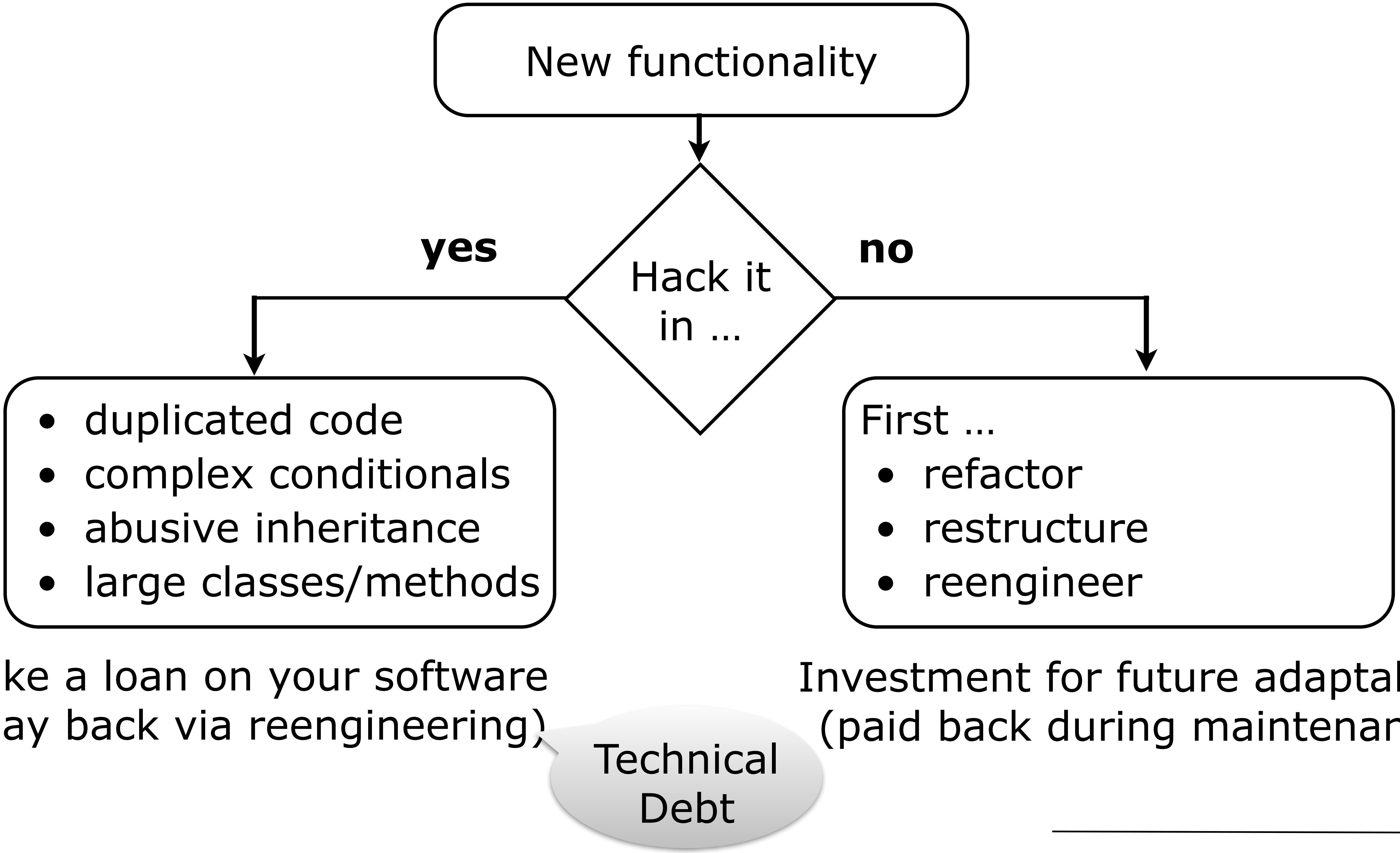
50-75% of maintenance budget concerns **Perfective Maintenance** (= new functionality, which you could not foresee when you started)

⇒New category of maintenance

Preventive Maintenance

Why Refactoring in OO?

New or changing requirements will gradually degrade original design, ...
… unless extra development effort is spent to adapt the structure.



New functionality

Hack it in …

yes

- duplicated code
- complex conditionals
- abusive inheritance
- large classes/methods

Take a loan on your software
(pay back via reengineering)

no

First …

- refactor
- restructure
- reengineer

Investment for future adaptability
(paid back during maintenance)

Technical Debt

What is Refactoring?

Two Definitions

- *VERB*: The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure [Fowl99]
- *NOUN*: A behaviour-preserving source-to-source program transformation [Robe98]

> *Primitive* refactorings vs. *Composite* refactorings

Typical Primitive Refactorings

Class Refactorings	Method Refactorings	Attrute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	pull up	pull up
	push down	push down
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Quizz

Can you explain why

+ add class
+ add method
+ add attribute

… are behaviour preserving?

Tool support

Change Efficient

Refactoring

- Source-to-source program transformation
- Behaviour preserving

⇒ improve the program structure

Programming Environment

- Fast edit-compile-run cycles
- Support small-scale reverse engineering activities

⇒ convenient for “local” ameliorations

Failure Proof

Regression Testing

- Repeating past tests
- Tests require no user interaction
- Tests are deterministic
  (Answer per test is yes / no)

⇒ improvements do not break anything

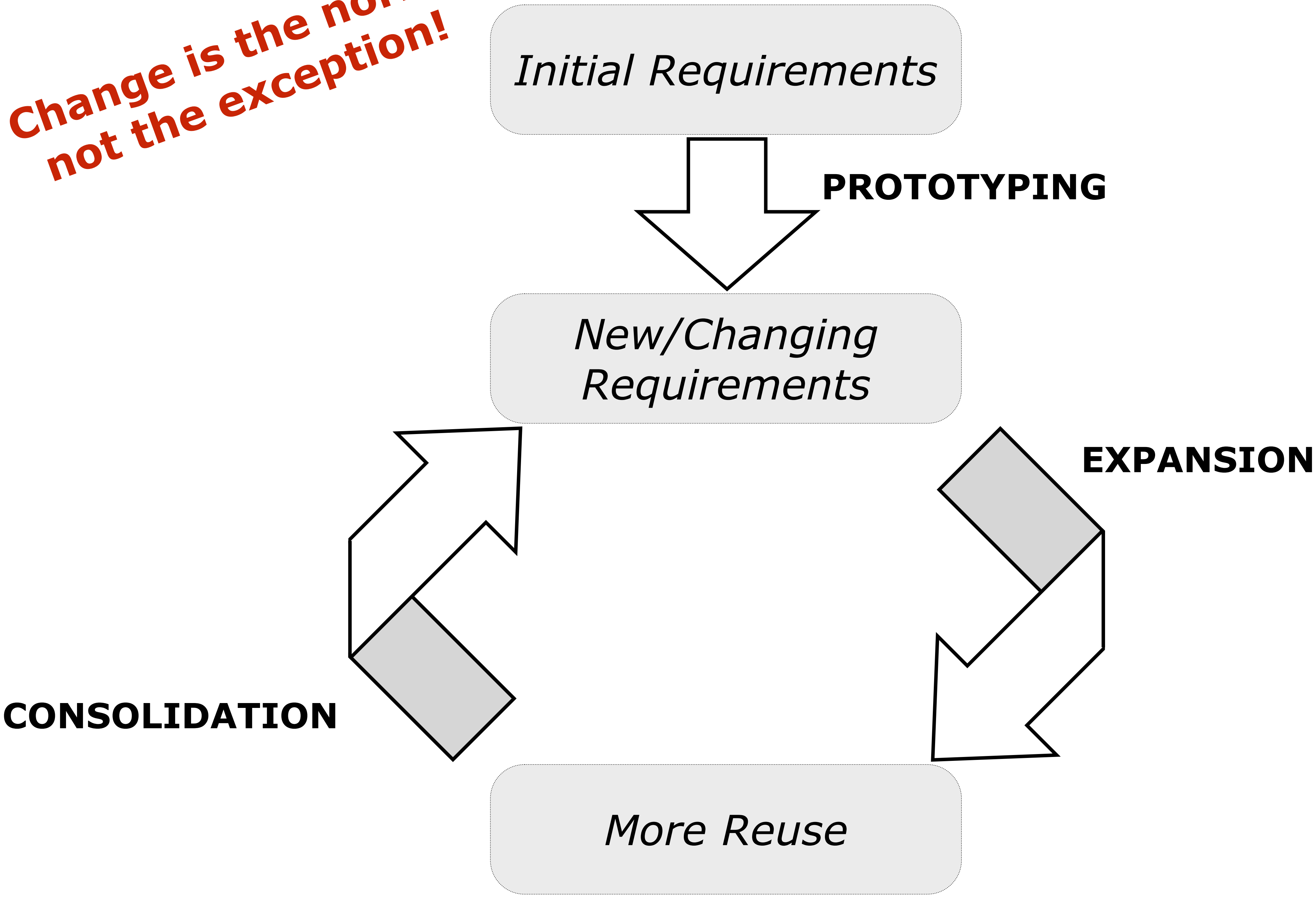
Configuration & Version Management

- keep track of versions that represent project milestones

⇒ go back to previous version

Iterative Development Life-cycle

Change is the norm, not the exception!

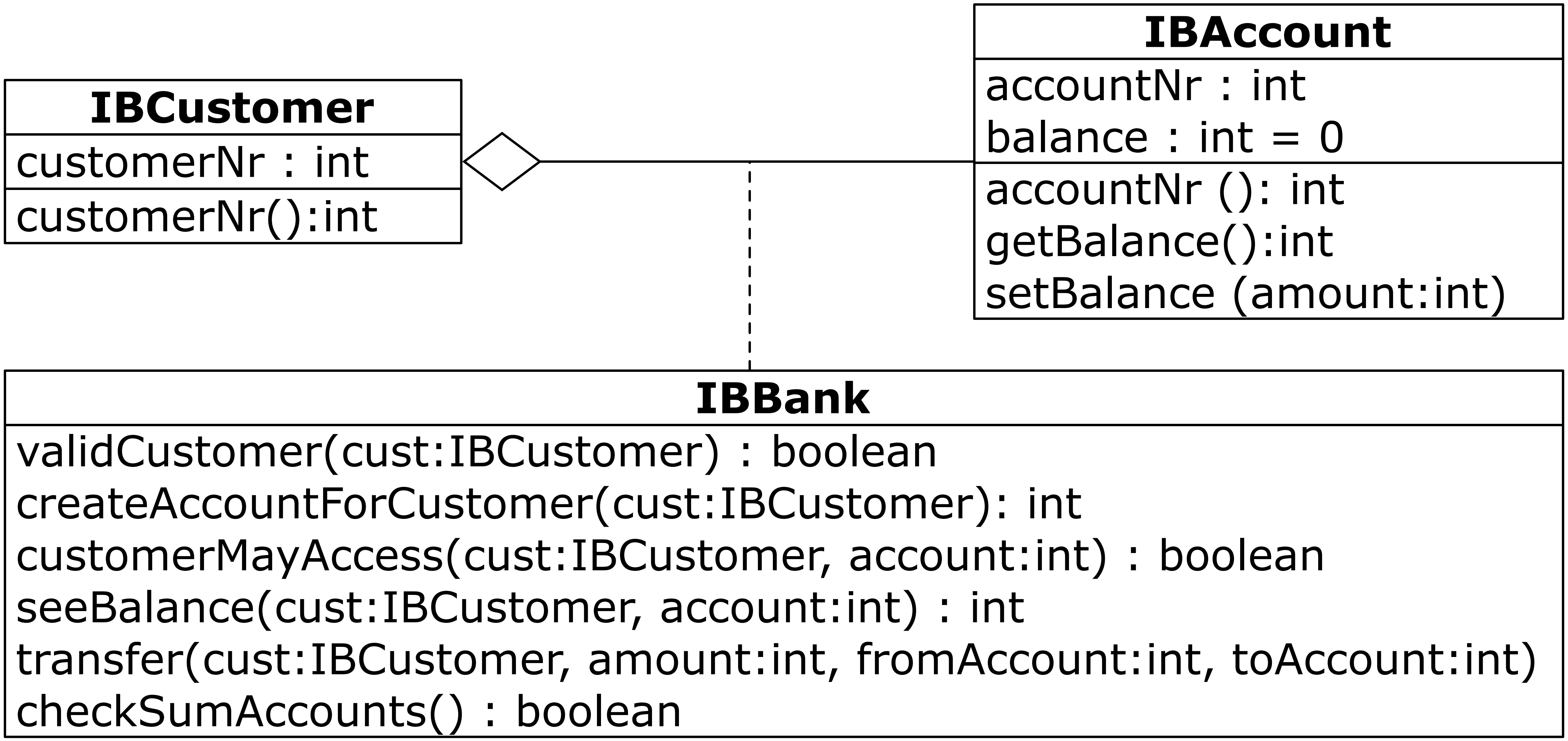


Example: Banking - Requirements

+ a bank has customers
+ customers own account(s) within a bank
+ with the accounts they own, customers may
  - deposit / withdraw money
  - transfer money
  - see the balance

- Non-functional requirements
  + *secure*: only authorised users may access an account
  + *reliable*: all transactions must maintain consistent state

Example: Banking - Class Diagram



IBCustomer

customerNr : int

customerNr():int

IBAccount

accountNr : int
balance : int = 0

accountNr (): int
getBalance():int
setBalance (amount:int)

IBBank

validCustomer(cust:IBCustomer) : boolean
createAccountForCustomer(cust:IBCustomer): int
customerMayAccess(cust:IBCustomer, account:int) : boolean
seeBalance(cust:IBCustomer, account:int) : int
transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
checkSumAccounts() : boolean

Example: Banking - Contracts

Ensure the “secure” and “reliable” requirements.

```
IBBank
    invariant: checkSumAccounts()


IBBank::createAccountForCustomer(cust:IBCustomer): int
    precondition: validCustomer(cust)
    postcondition: customerMayAccess(cust, <<result>>)


IBBank::seeBalance(cust:IBCustomer, account:int) : int
    precondition: (validCustomer(cust)) AND
        (customerMayAccess(cust, account))
    postcondition: true


IBBank::transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
    precondition: (validCustomer(cust))
        AND (customerMayAccess(cust, fromAccount))
        AND (customerMayAccess(cust, toAccount))
    postcondition: true
```

Example: Banking - CheckSum

Bookkeeping systems always maintain two extra accounts, “incoming” and “outgoing”

- ⇒ the sum of the amounts of all transactions is always 0 ⇒ consistency check

Incoming

date	amount
1/1/2000	-100
1/2/2000	-200

MyAccount

date	amount
1/1/2000	+100
1/2/2000	+200
1/3/2000	-250

OutGoing

date	amount
1/3/2000	+250

Prototype Consolidation

Design Review (i.e., apply refactorings AND RUN THE TESTS!)

- Rename attribute
  + rename "_balance" into "_amountOfMoney" (run test!)
  + apply "rename attribute" refactoring to the above
    > run test!
  + check the effect on source code
    - comments + getter/setter methods
- Rename method
  + rename "get_balance" into "get_amountOfMoney"
    > run test!
- Change Method Signature
  + change order of arguments for "transfer" (run test!)
- Rename class
  + check all references to "Customer"
  + apply "rename class" refactoring
    > rename into "Client"
    > run test!
  + check the effect on source code
    - file name / makefiles / …
    - CustomerTest >> ClientTest??

What is Refactoring?

Can you give the pre-conditions for
a “rename method” refactoring?

Class Refactorings	Method Refactorings	Attrute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	pull up	pull up
	push down	push down
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Expansion

Additional Requirement

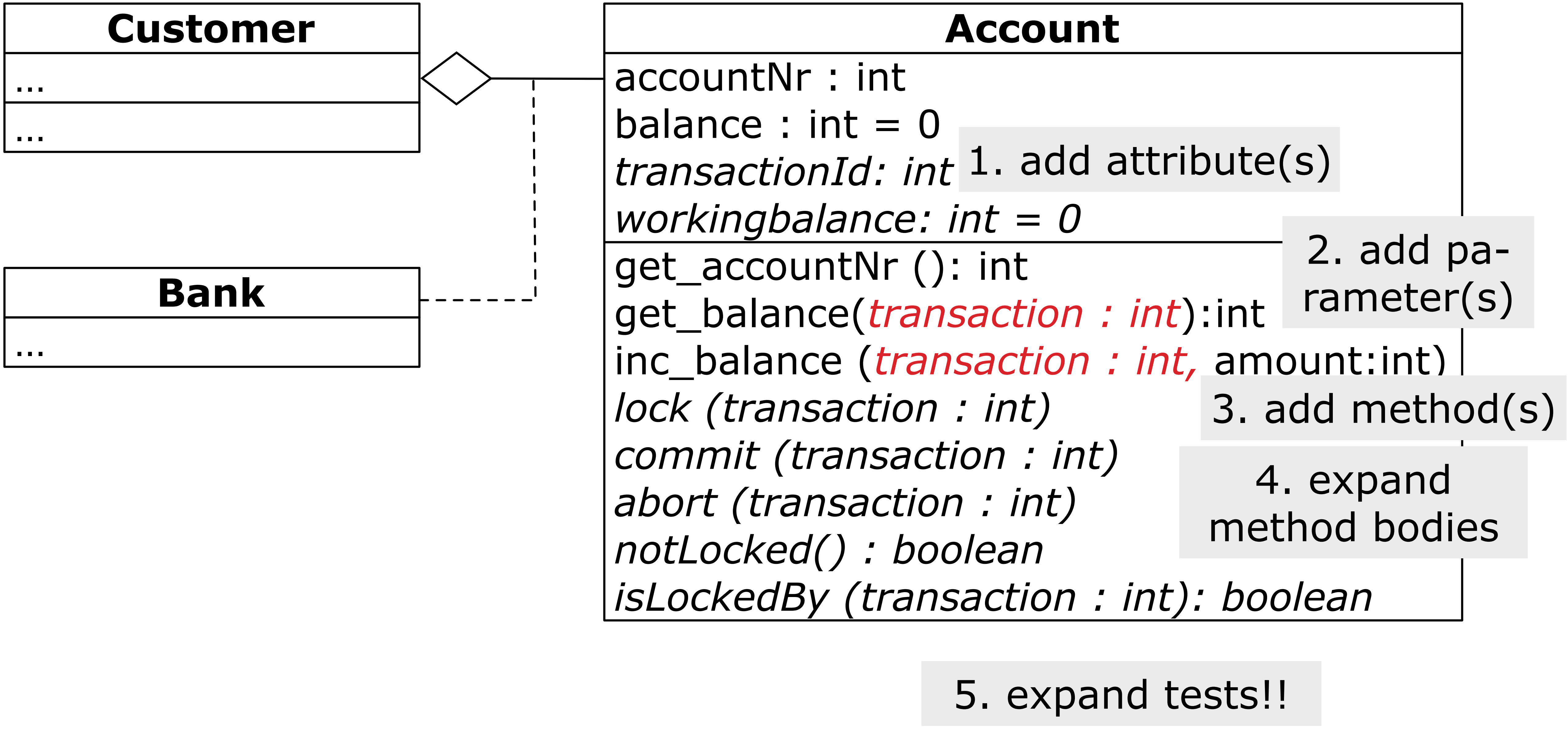
- concurrent access of accounts

Add test case for

- Bank
  - + testConcurrent: Launches 10 processes that simultaneously transfer money between same accounts
    - > test fails!

Can you explain why the test fails?

Expanded Design: Class Diagram



Customer

…

…

Bank

…

Account

accountNr : int

balance : int = 0

transactionId: int

workingbalance: int = 0

get_accountNr (): int

get_balance(*transaction : int*):int

inc_balance (*transaction : int,* amount:int)

lock (transaction : int)

commit (transaction : int)

abort (transaction : int)

notLocked() : boolean

isLockedBy (transaction : int): boolean

1. add attribute(s)

2. add parameter(s)

3. add method(s)

4. expand method bodies

5. expand tests!!

Expanded Design: Contracts

```
Account
    invariant: (isLocked()) OR (NOT isLocked())

   Account::get_balance(transaction:int): int
    precondition: isLockedBy(transaction)
    postcondition: true

   Account::inc_balance(transaction:int, amount: int)
    precondition: isLockedBy(transaction)
        postcondition: peek_balance() = peek_balance() + amount

   Account::lock(transaction:int)
    precondition: notLocked()
    postcondition: isLockedBy(transaction)

   Account::commit(transaction:int)
    precondition: isLockedBy(transaction)
    postcondition: notLocked()

   Account::abort(transaction:int)
    precondition: isLockedBy(transaction)
    postcondition: notLocked()
```

Expanded Implementation

Adapt implementation

- 1. Manually add attributes on Account
  + "transactionId" and "workingBalance"
- 2. apply "change method signature"
  + add "transaction"
  + to "get_balance()" and "inc_balance()"
- 3. apply "add method"
  + lock, commit, abort, isLocked, isLockedBy
- 4. expand method bodies (i.e. careful programming)
  + of "seeBalance()" and "transfer()"
  > load "Banking12"

- 5. expand Tests
  + previous tests for "get_balance()" and "inc_balance()"
  - should now fail
  * adapt tests
  + new contracts, incl. commit and abort
  * new tests

testConcurrent works!

> we can confidently ship a new release

Consolidation: Problem Detection

More Reuse

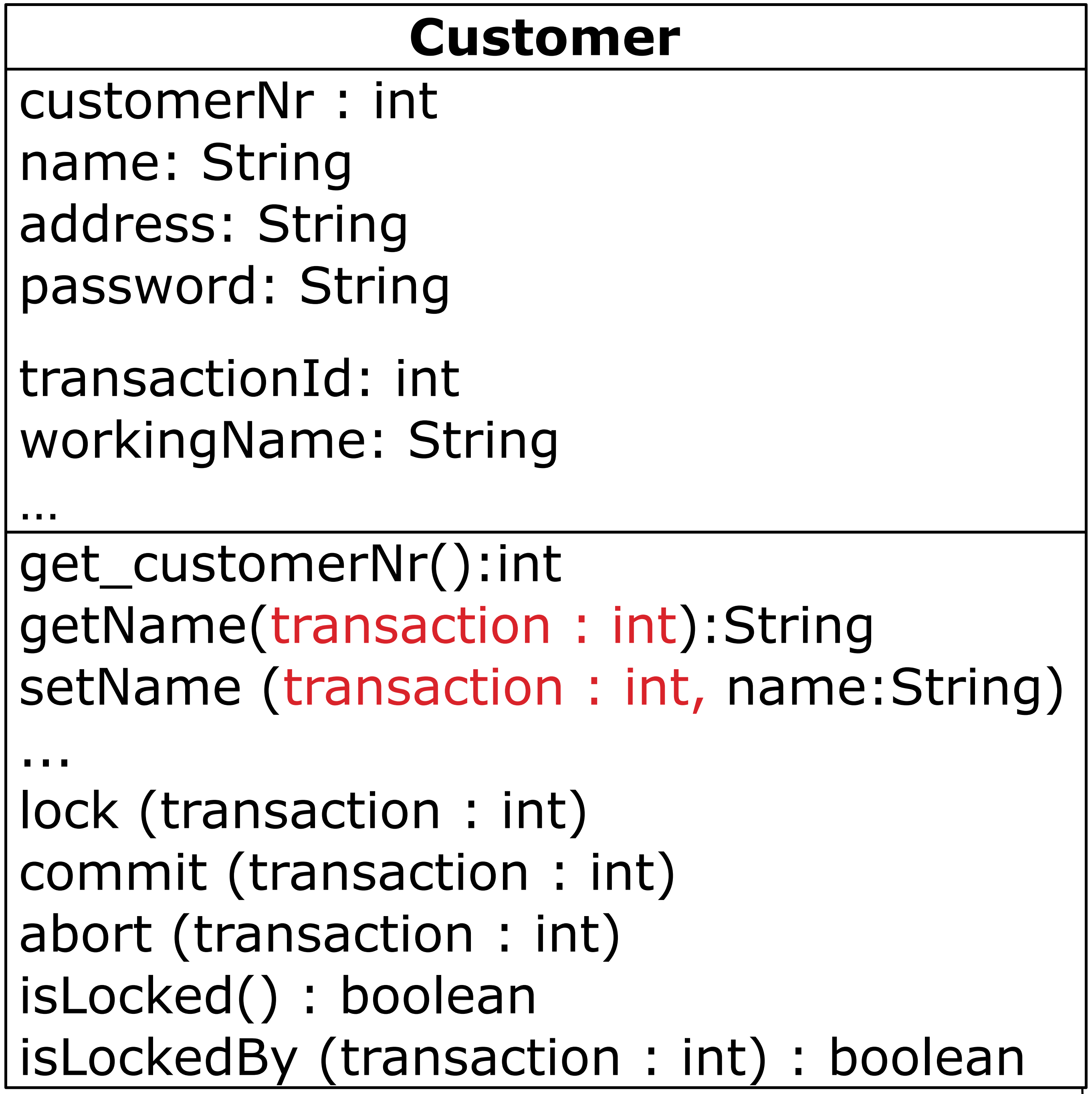
- A design review reveals that this “transaction” stuff is a good idea and is applied to Customer as well.

⇒ **Code Smells**

- duplicated code
  + lock, commit, abort
  + transactionId
- large classes
  + extra methods
  + extra attributes

⇒ **Refactor**

- “Lockable” should become a separate component, to be reused in Customer and Account



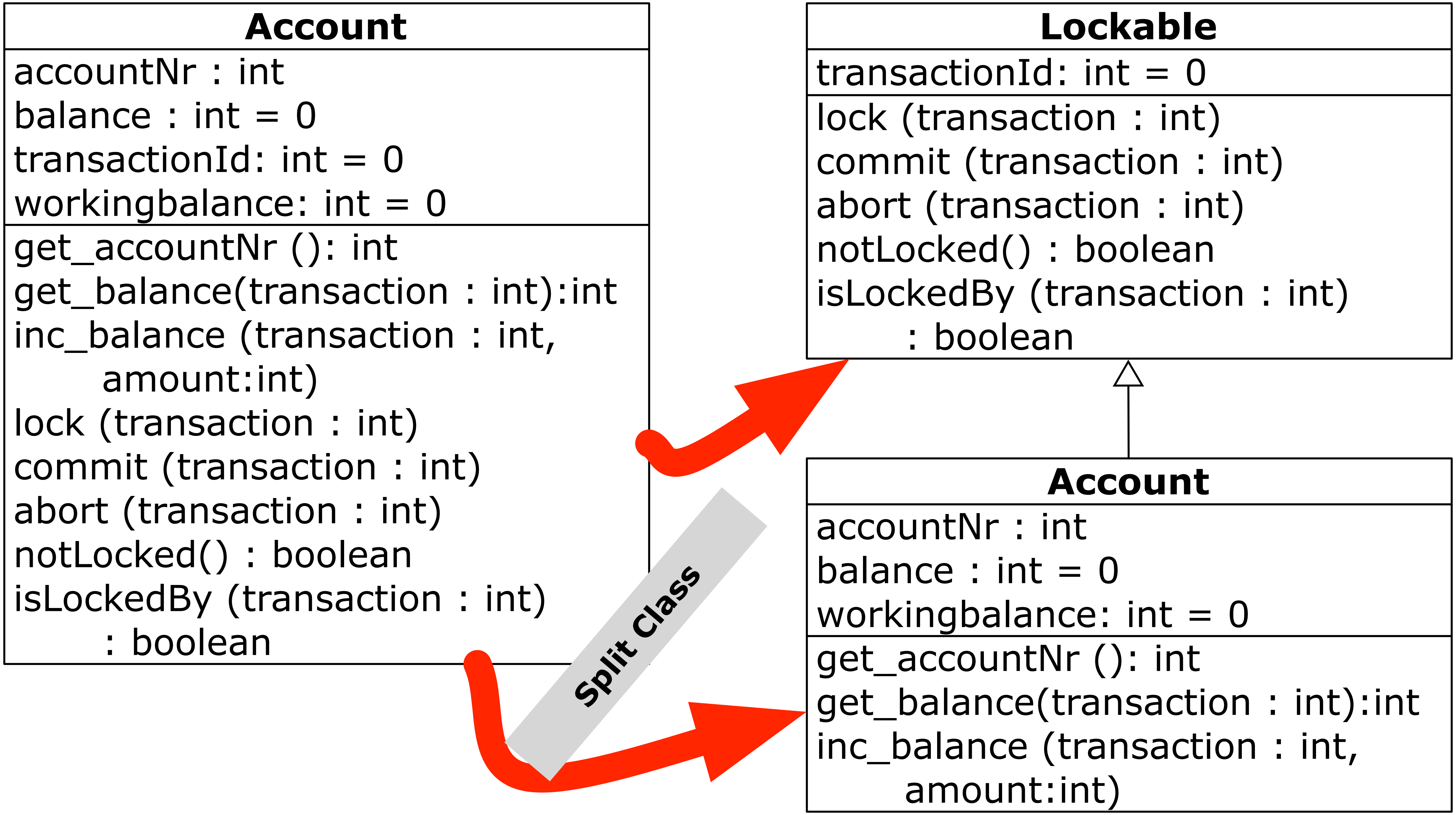
Customer

customerNr : int
name: String
address: String
password: String

transactionId: int
workingName: String
…

get_customerNr():int
getName(transaction : int):String
setName (transaction : int, name:String)
...
lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
isLocked() : boolean
isLockedBy (transaction : int) : boolean

Consolidation: Refactored Class Diagram



Account

accountNr : int
balance : int = 0
transactionId: int = 0
workingbalance: int = 0

get_accountNr (): int
get_balance(transaction : int):int
inc_balance (transaction : int,
amount:int)
lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
notLocked() : boolean
isLockedBy (transaction : int)
: boolean

Lockable

transactionId: int = 0

lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
notLocked() : boolean
isLockedBy (transaction : int)
: boolean

Account

accountNr : int
balance : int = 0
workingbalance: int = 0

get_accountNr (): int
get_balance(transaction : int):int
inc_balance (transaction : int,
amount:int)

Refactoring Sequence: 1/4

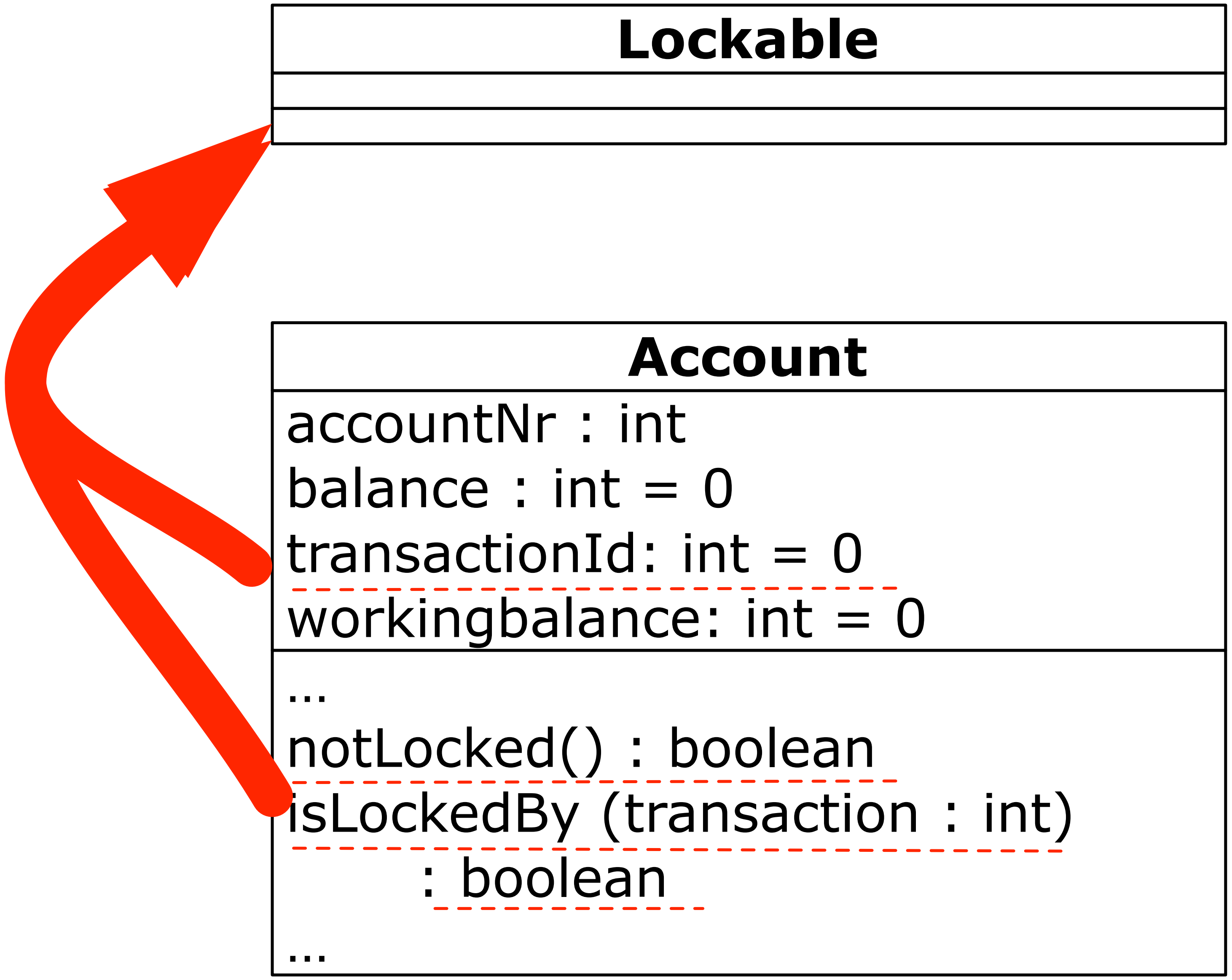
Refactoring: Extract Superclass

- Position on Account

  + superclass name = Lockable

  + members: transactionId + notLocked + isLockedBy

  - action = extract
- verify effect on code
- run the tests!



Lockable

Account

accountNr : int
balance : int = 0
transactionId: int = 0
workingbalance: int = 0

…
notLocked() : boolean
isLockedBy (transaction : int)
: boolean
…

Refactoring Sequence: 2/4

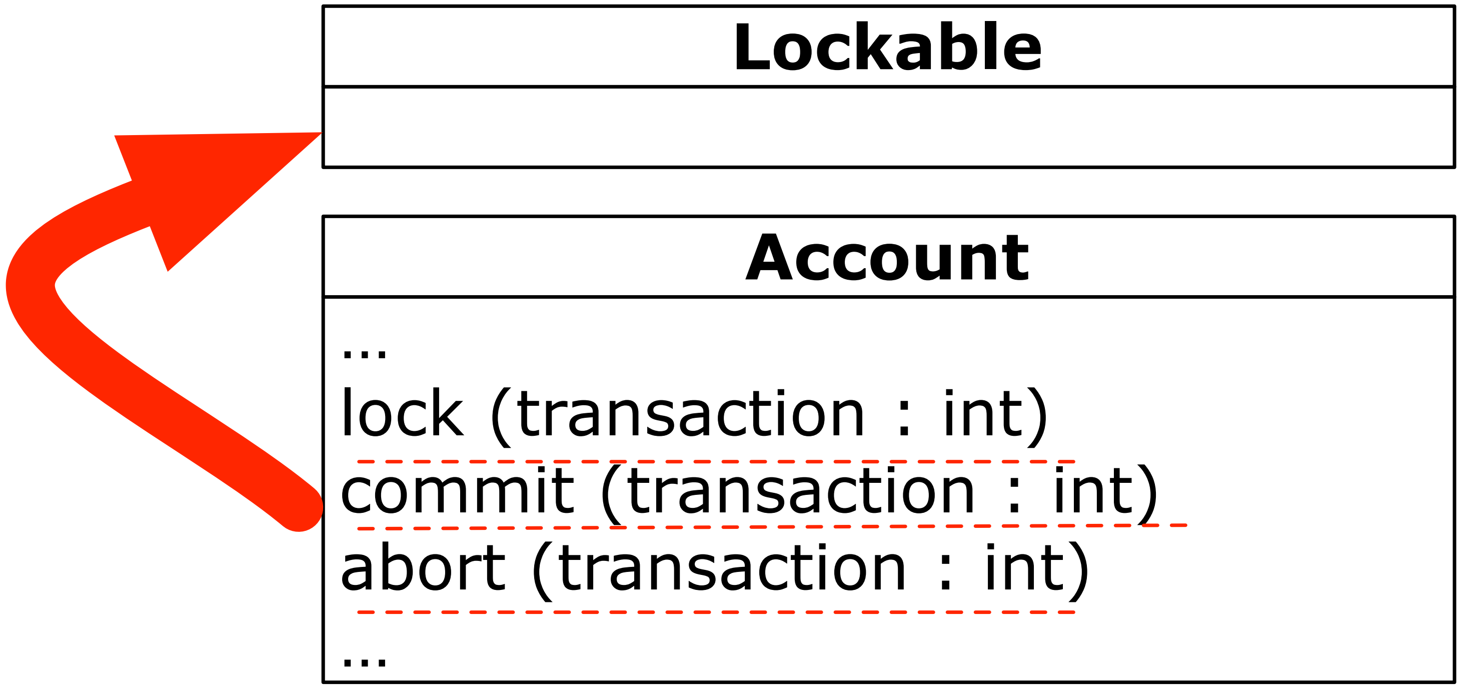
Refactoring: Pull Up

- apply “pull up …” on “Account”

  + to move “lock / commit / transaction” onto lockable

  + apply “pull up” to “abort:”, “commit:”, “lock:”

  > failure: why???



Refactoring Sequence: 3/4

Refactoring: Extract Method

- apply “extract method” on
  + groups of accesses to “balance” and “WorkingBalance”

```
public synchronized void lock(int transaction) {
  this.require(this.notLocked(), "No other transaction ….”);
  this._transactionId = transaction;
  this._workingBalance = this._balance;
  this.ensure(this.isLockedBy(transaction), "Lock must ….”);
  }
```

```
protected void copyToWorkingState() {
  this._workingBalance = this._balance;
}
```

```
public synchronized void lock(int tr
  this.require(this.notLocked(), "No other transaction ….”);
  this._transactionId = transaction;
  copyToWorkingState();
  this.ensure(this.isLockedBy(transaction), "Lock must ….”);
  }
```

- similar for
  + “abort” (⇒ clearWorkingState) & “commit” (⇒ commitWorkingState)

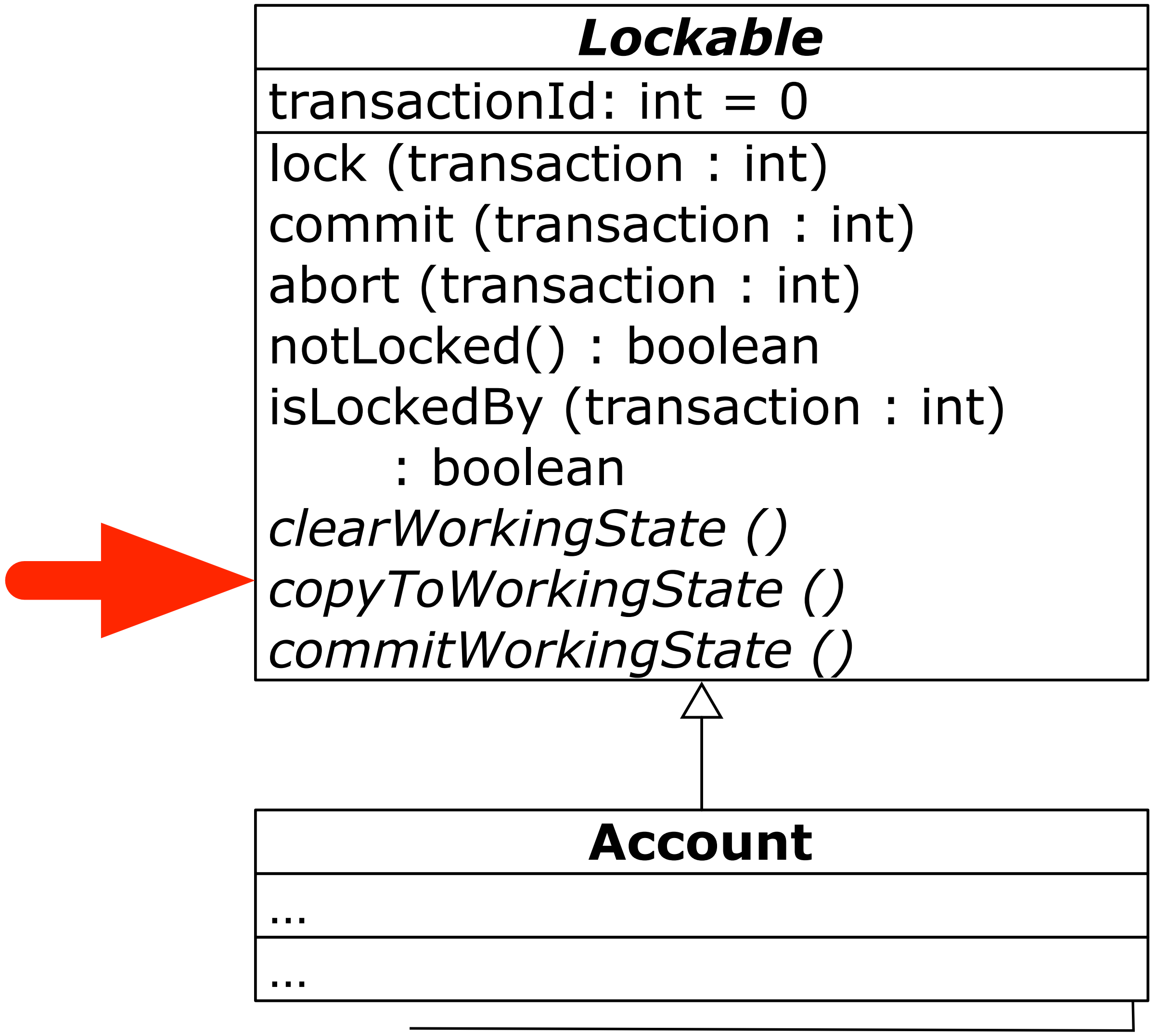
Refactoring Sequence: 4/4

Refactoring "Pull up…" revisited

- apply "pull up …" on "Account"
  - + members clearWorkingState / copyToWorkingState / commitWorkingState
    - - action = declare abstract in destination
  - + members "abort", "commit", "lock"
    - - action = pull up

Are we done?

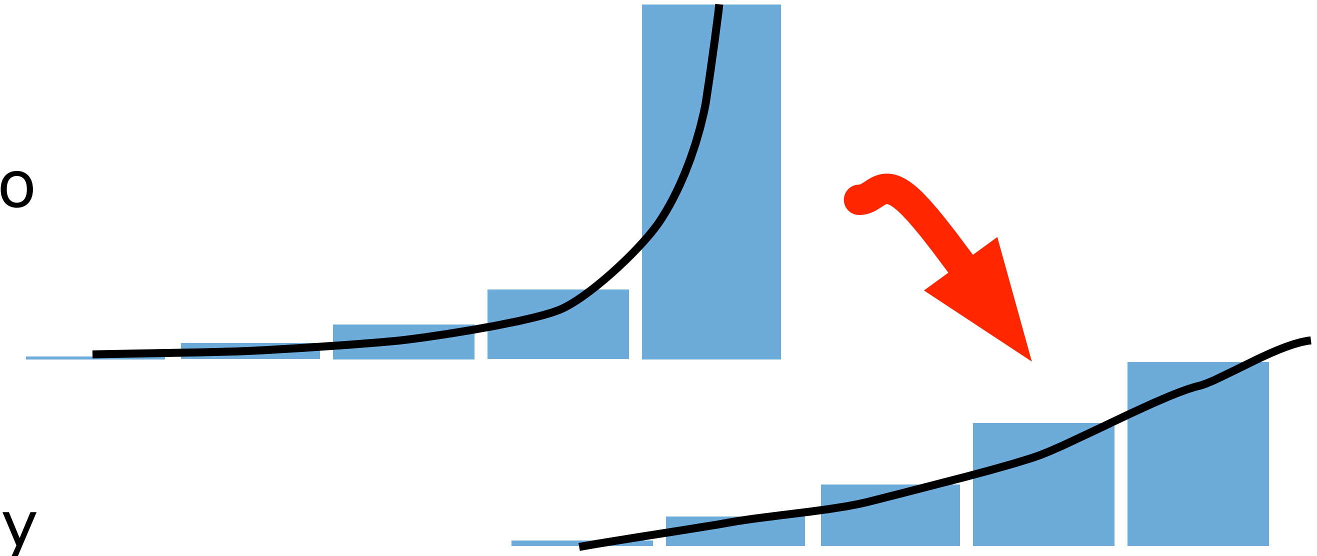
- Run the tests …
- Customer subclass of Lockable
  - + expand functionality to incorporate locking protocol



Tool Support

Refactoring Philosophy

- combine simple refactorings into larger restructuring (and eventually reengineering)
  - > improved design
  - > ready to add functionality
- Do not apply refactoring tools in isolation



	Smalltalk	C++	Java
refactoring	+	- (?)	+
rapid edit-compile-run cycles	+	-	+-
reverse engineering facilities	+-	+-	+-
regression testing	+	+	+
version & configuration management	+	+	+

Code Smells

Know when is as important as know-how

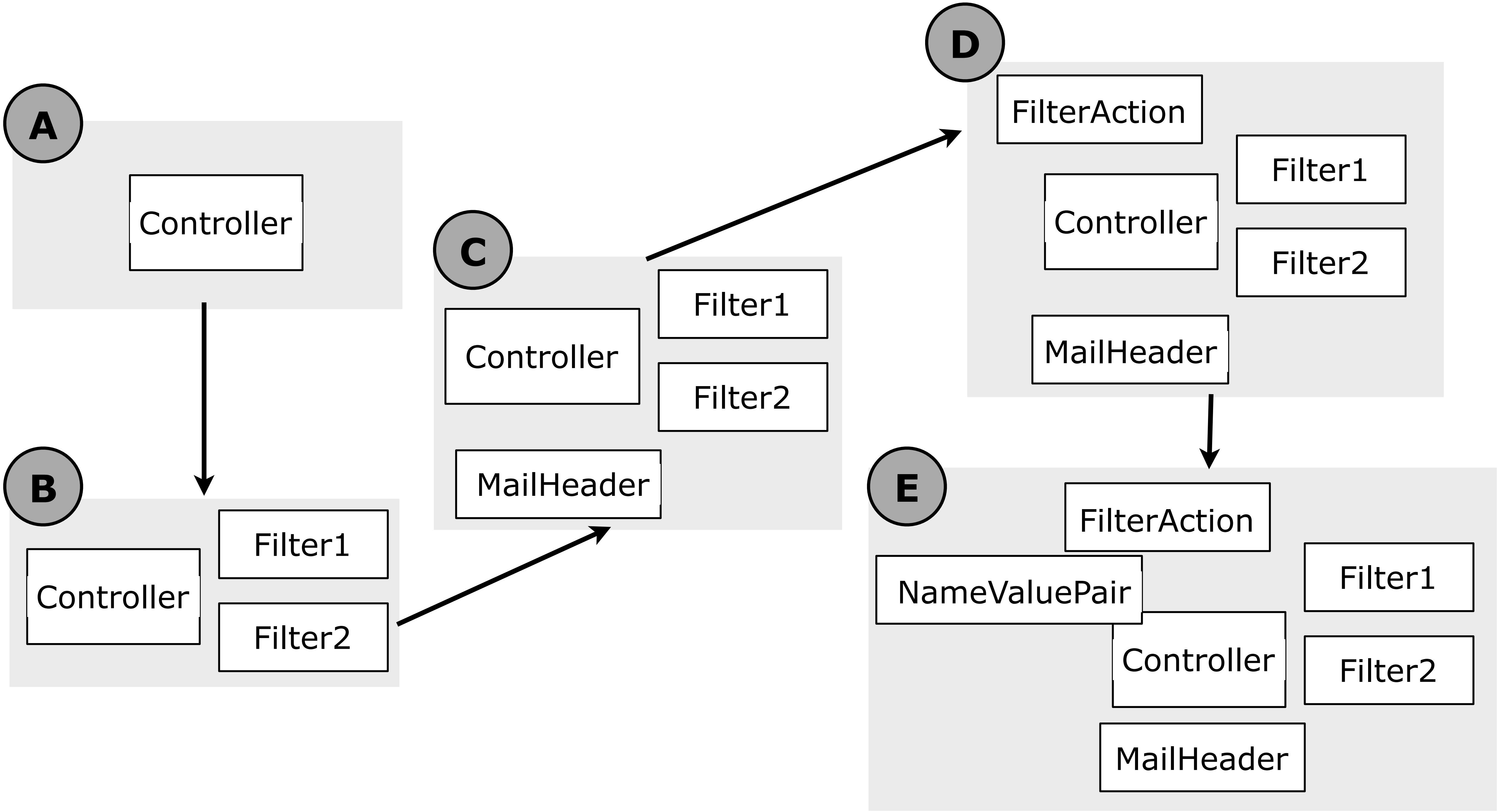
- Refactored designs are more complex
  - > Introduce a lot of extra small classes/methods
- Use “code smells” as symptoms for refactoring opportunities
  - + Duplicated code
  - + Nested conditionals
  - + Large classes/methods
  - + Abusive inheritance

- Rule of the thumb:
  - + All system logic must be stated *Once and Only Once*
    - > a piece of logic stated more than once implies refactoring

More about code smells and refactoring

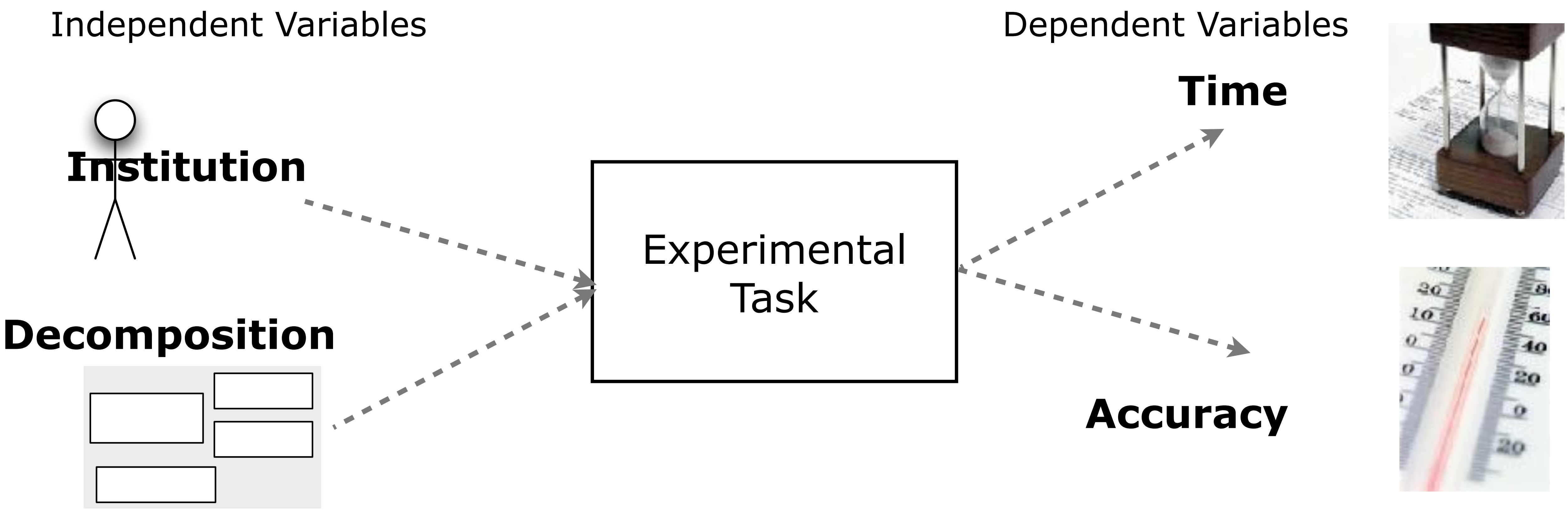
- Wiki-web with discussion on code smells
  - + https://wiki.c2.com/?CodeSmell

Refactoring God Class: Optimal Decomposition?



Empirical Validation

Controlled experiment with 63 last-year master-level students (CS and ICT)



Independent Variables

Institution

Decomposition

Experimental Task

Dependent Variables

Time

Accuracy

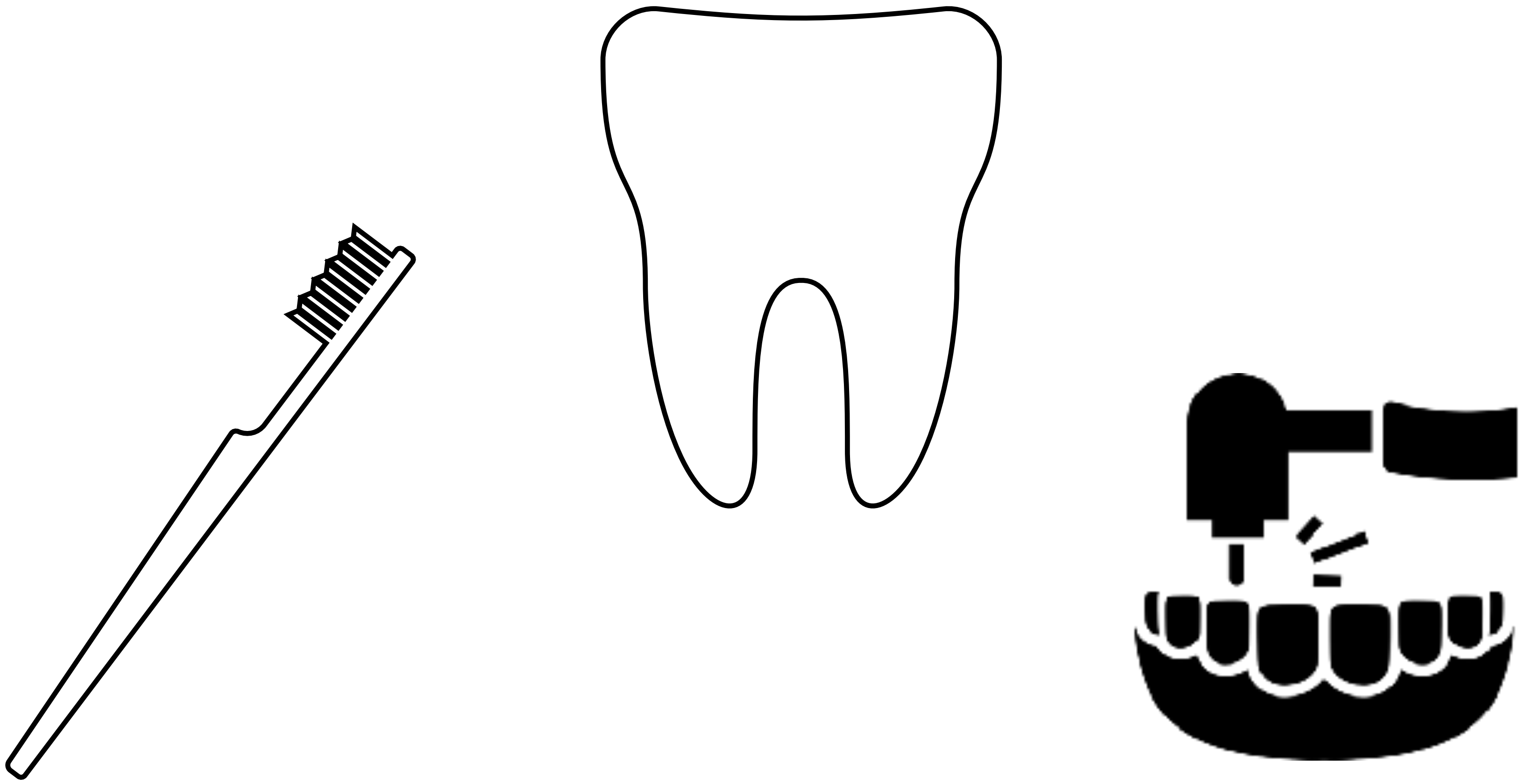
"Optimal decomposition" differs with respect to education

- Computer science: preference towards decentralized designs (C-E)
- ICT-electronics: preference towards centralized designs (A-C)

Advanced OO training can induce preference

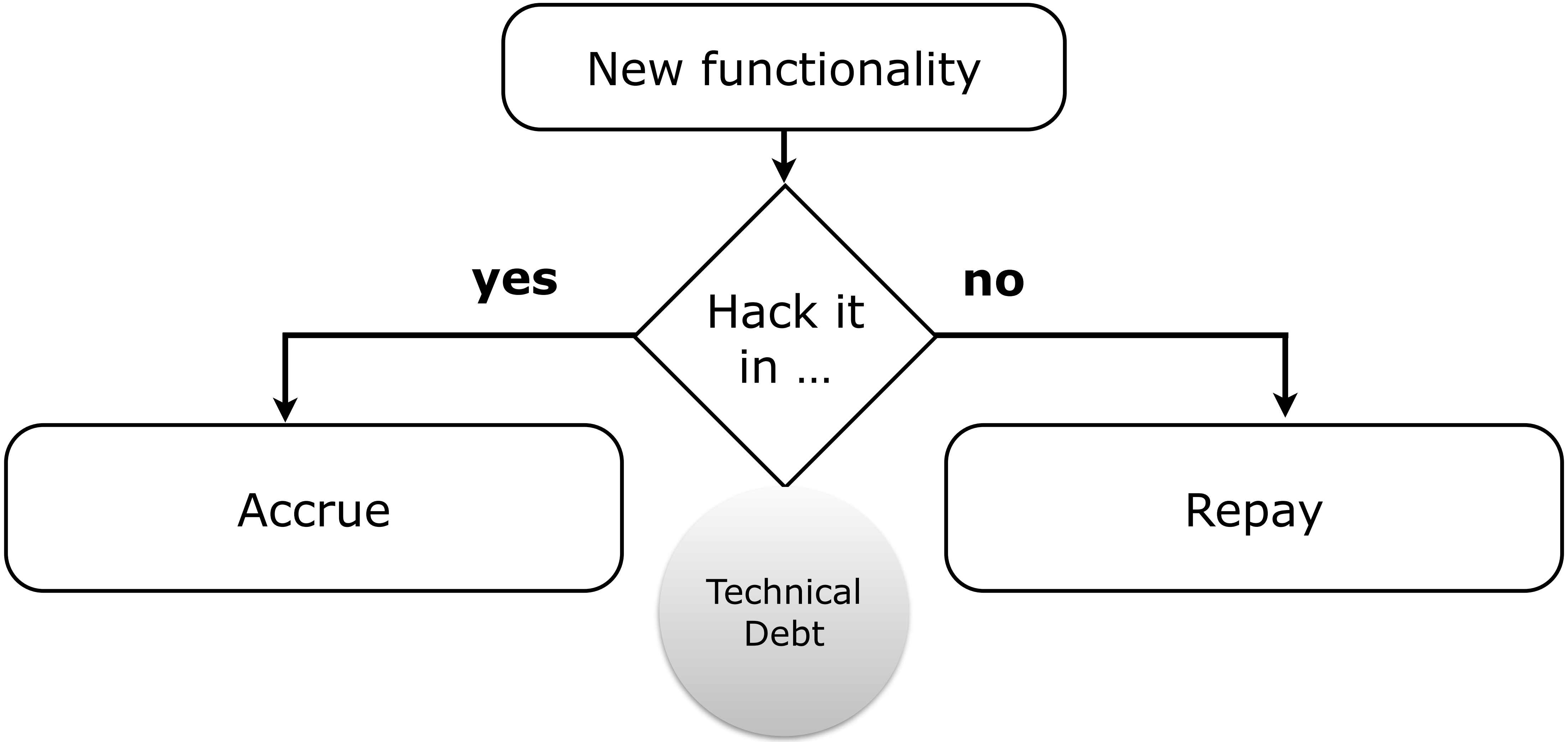
- Consistent with [Arisholm et al. 2004]

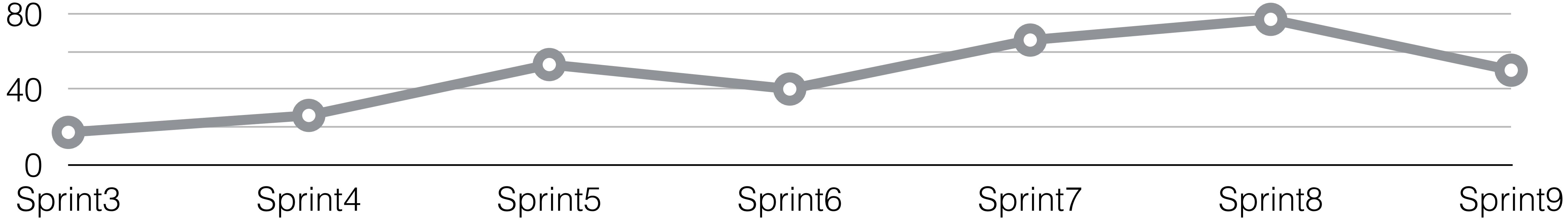
Floss Refactoring vs. Root-Canal Refactoring



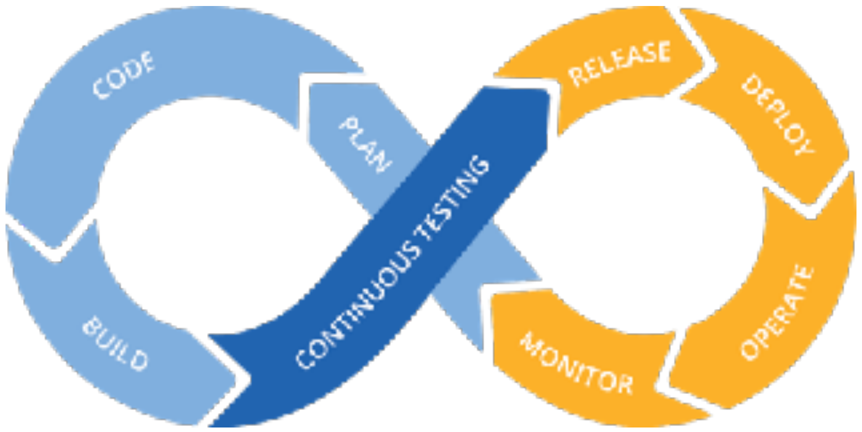
E. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," in IEEE Software, vol. 25, no. 5, pp. 38-44, Sept.-Oct. 2008, doi: 10.1109/MS.2008.123.

Technical Debt

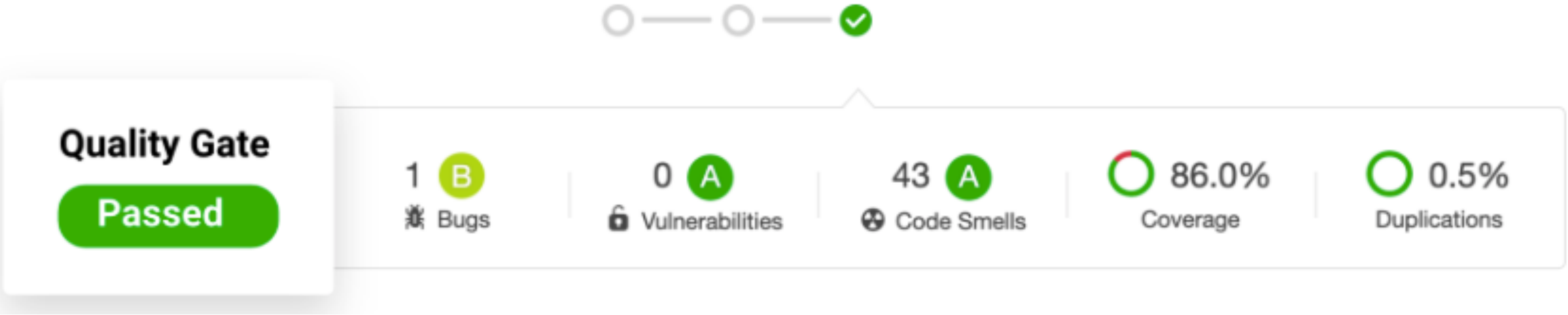




DevOps: Monitor Technical Debt



CAPSTONE PROJECT



Quality Gate

Passed

1 B Bugs

0 A Vulnerabilities

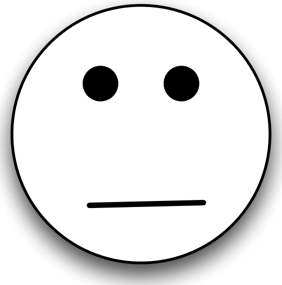
43 A Code Smells

86.0% Coverage

0.5% Duplications

sonarqube

Correctness & Traceability



Correctness

- Are we building the system right?
- Assured via “behaviour preserving” nature & regression testing

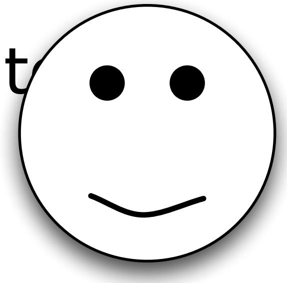
  > We are sure the system remains as “correct” as it was before

- Are we building the right system?

  + By improving the internal design we can cope with mismat

  > First refactor (= consolidate) …

  > then new requirements (= expand)



Traceability

- Requirements <-> System?

  + Requires a lot of discipline ... thus extra effort!

  + But renaming is refactoring too

  > Adjust code to adhere to naming conventions

Summary (i)

You should know the answers to these questions:

- Can you explain how refactoring differs from plain coding?
- Can you tell the difference between Corrective, Adaptive and Perfective maintenance? And how about preventive maintenance?
- Can you name the three phases of the iterative development life-cycle? Which of the three does refactoring support the best? Why do you say so?
- Can you give 4 symptoms for code that can be “cured” via refactoring?
- Can you explain why add class/add method/add attribute are behaviour preserving?
- Can you give the pre-conditions for a “rename method” refactoring?
- Which 4 activities should be supported by tools when refactoring?
- Why can’t we apply a “push up” to a method “x()” which accesses an attribute in the class the method is defined upon (see Refactoring Sequence on page 27–31)?

You should be able to complete the following tasks

- Two classes A & B have a common parent class X. Class A defines a method a() and class B a method b() and there is a large portion of duplicated code between the two methods. Give a sequence of refactorings that moves the duplicated code in a separate method x() defined on the common superclass X.
- What would you do in the above situation if the duplicated code in the methods a() and b() are the same except for the name and type of a third object which they delegate responsibilities too?

- Monitor the technical debt of you bachelor capstone project.

CAPSTONE PROJECT

Summary (ii)

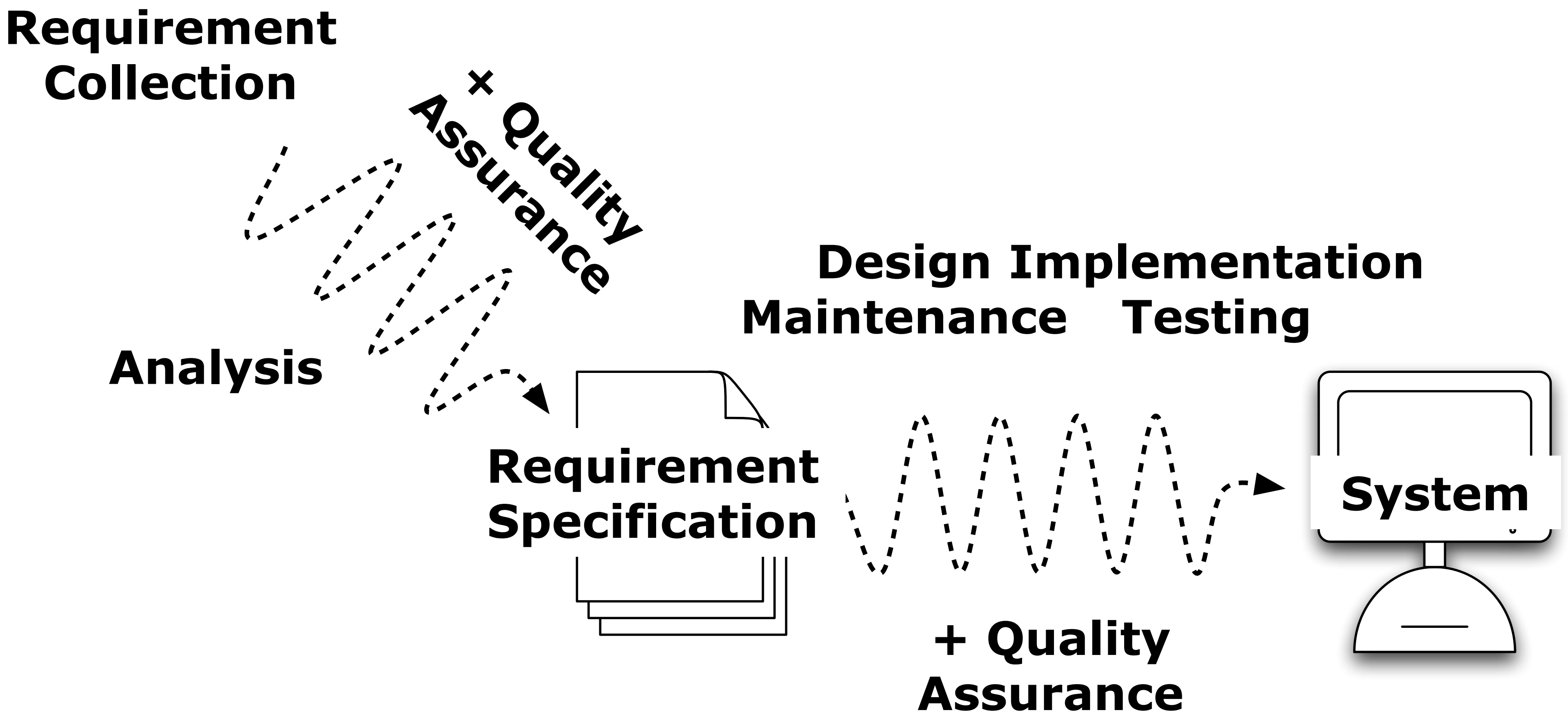
Can you answer the following questions?

- Why would you use refactoring in combination with Design by Contract and Regression Testing?
- Can you give an example of a sequence of refactorings that would improve a piece of code with deeply nested conditionals?
- How would you refactor a large method? And a large class?
- Consider an inheritance relationship between a superclass "Square" and a subclass "Rectangle". How would you refactor these classes to end up with a true "is-a" relationship? Can you generalise this procedure to any abusive inheritance relationship?

CHAPTER 12 – Conclusion

- Overview
  + 1.Introduction
  + 2.Requirements
  + 3.Software Architecture
  + 4.Project Management
  + 5.Design by Contract
  + 6.Testing
  + 7.Formal Specification
  + 8.Domain Modeling
  + 9.Software Quality
  + 10.Software Metrics
  + 11.Refactoring
- Articles
  + The Quest for the Silver Bullet
  + The Case of the Killer Robot
- Professional Ethics
  + Cases
- The future: Software Engineering Tools

Software Product & Process



Requirement Collection

+ Quality Assurance

Analysis

Requirement Specification

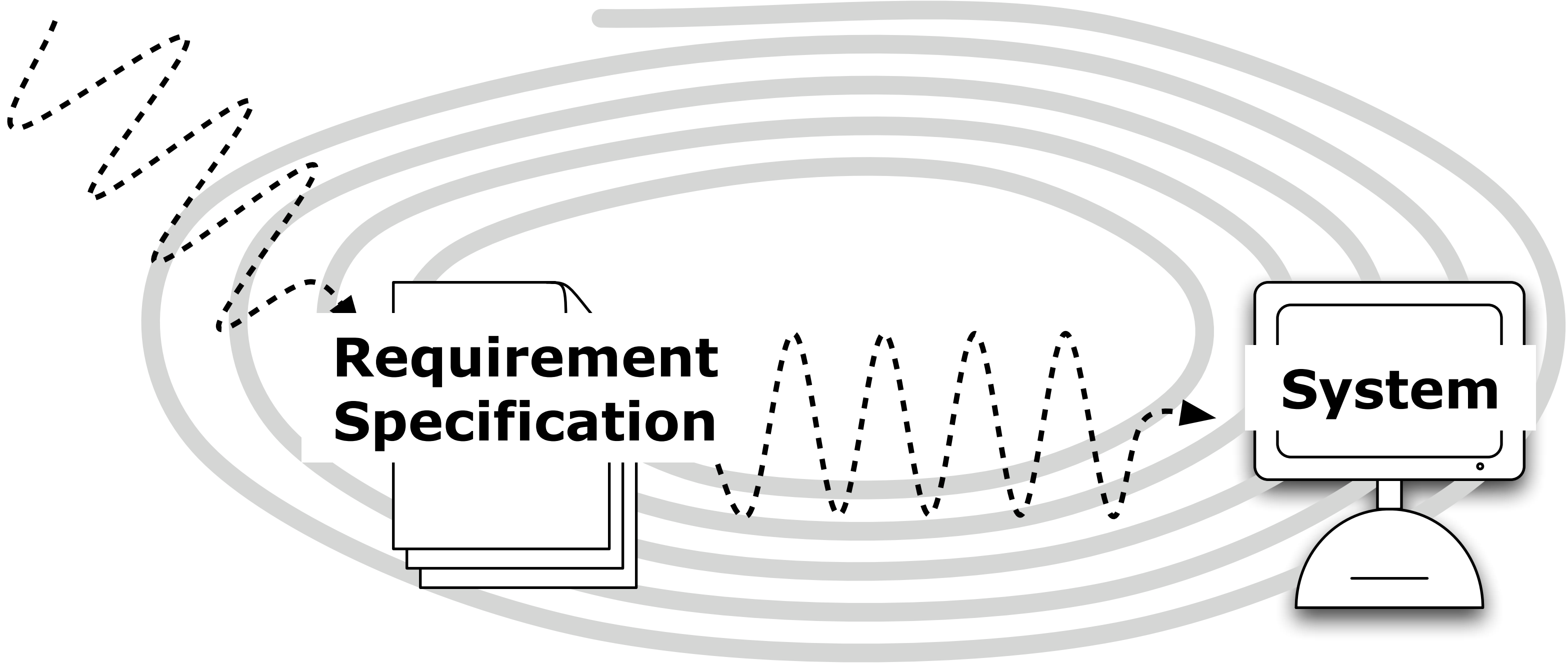
Design Implementation Maintenance Testing

+ Quality Assurance

System

- **Software Process:**
  + Requirements Collection + Analysis + Design + Implementation
  + Testing + Maintenance + Quality Assurance
- **Software Product:**
  + Requirements Specification (= functional & non-functional)
  + System (= executable + code + documentation)

Evaluation Criteria



2 evaluation criteria to assess techniques applied during process

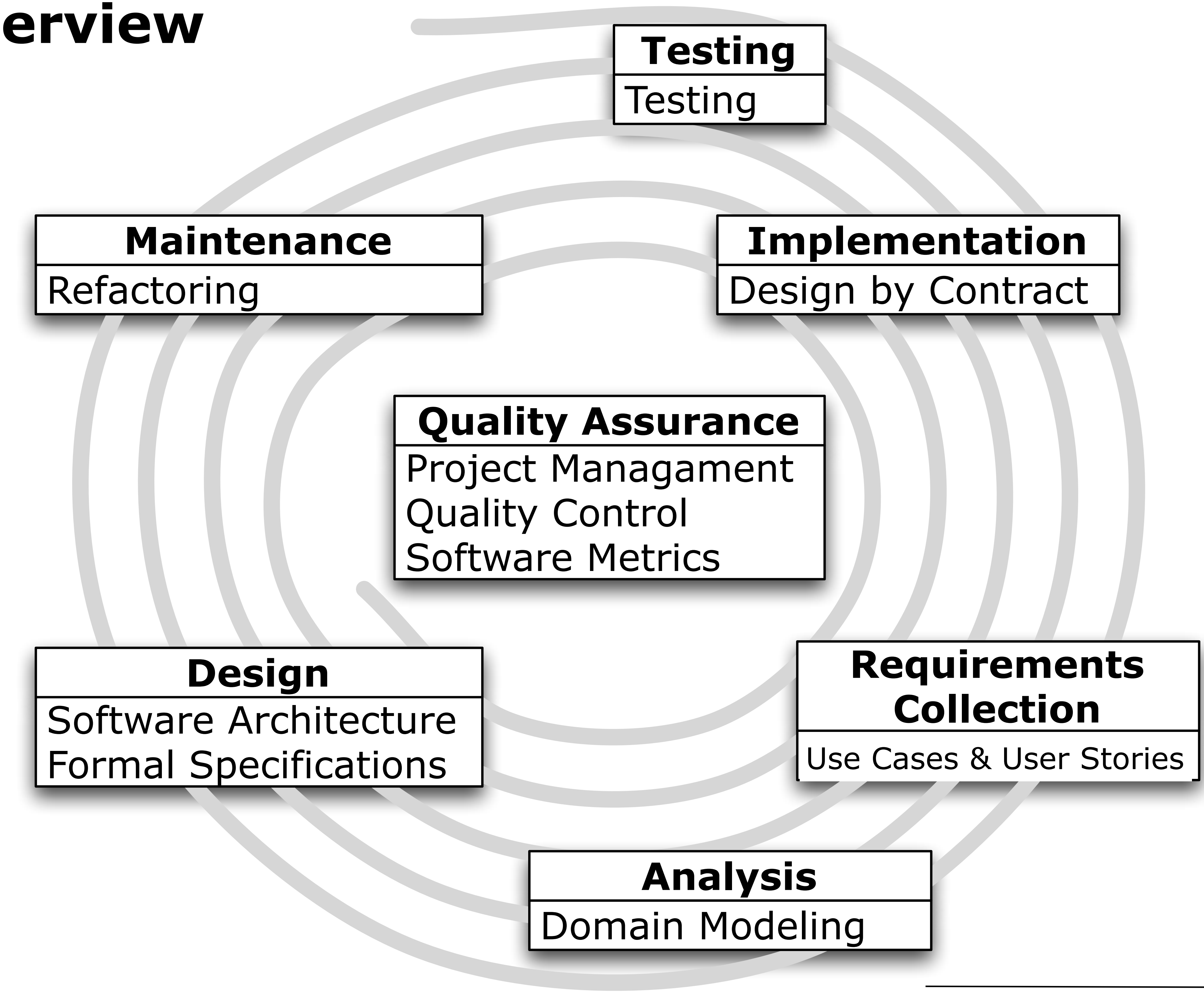
Correctness

- Are we building the right product? = VALIDATION
- Are we building the product right? = VERIFICATION

Traceability

- Can we deduce which product components will be affected by changes?

Overview



Testing

Testing

Implementation

Design by Contract

Requirements Collection

Use Cases & User Stories

Analysis

Domain Modeling

Design

Software Architecture
Formal Specifications

Maintenance

Refactoring

Quality Assurance

Project Managament
Quality Control
Software Metrics

Requirements

- Use Cases
  + = Specify expected system behavior as a set of generic scenarios
- User Stories
  + = Express expected functionality with the behaviour driven template
    - As a <user role> I want to <goal> so that <benefit>.

- Are we building the system right?
  + Well specified scenarios help to verify system against requirements
- Are we building the right system?
  + Validation by means of CRC Cards and role playing.
  + Safety Critical ⇒ Failure Mode and Affect Analysis (FMEA)

- Traceability? Requirements ⇔ System

  + Via proper naming conventions

- Traceability? Requirements ⇔ Project Plan

  + Use cases & User stories form good milestones

Software Architecture

- Software Architecture
  + = Components & Connectors describing high-level view of a system.
  + Decomposition implies trade-offs expressed via *coupling* and *cohesion*.
  + Proven solutions to recurring problems are recorded as *patterns*.
- Architecture Tradeoff Analysis Method (ATAM)
  + Review: identify risks, non-risks, sensitivity points and trade-off points

- Are we building the system right?
  + For the *non-functional* parts of the requirements

- Traceability?
  + Extra level of abstraction may hinder traceability

Project Management

- Project Management
  + = plan the work and work the plan
  + PERT and Gantt charts with various options
  + Critical path analysis and monitoring

- Are we building the system right?
  + Deliver what’s required on time within budget
  + Calculate risk to the schedule via optimistic and pessimistic estimates
  + Monitor the critical path to detect delays early
  + Plan to re-plan to meet the deadline

- Traceability? Project Plan ⇔ Requirements & System

  + The purpose of a plan is to detect deviations as soon as possible
  + Small tasks + Milestones verifiable by customer

Design by Contract

- Contractual Obligations Explicitly recorded in Interface
  + pre-condition = obligation to be satisfied by invoking method
  + post-condition = obligation to be satisfied by method being invoked
  + class invariant = obligation to be satisfied by both parties

- Are we building the system right?
  + Recorded obligations prevent defects
  + and ... remain in effect during changes
- Consumer-driven contract testing
  - Test distributed components in isolation via contractual obligations
- Traceability?
  + Obligations express key requirements in source code

- Liskov Substitution Principle?

	stronger	weaker	equal
{I’} vs. {I}			x
{P’} vs. {P}		x	x
{Q’} vs. {Q}	x		x

Testing

- Automated Regression Testing
  + = Deterministic tests (no user intervention), answering whether the system did regress (red = failing tests) or not (green = all tests pass)

- Are we building the system right?
  + Tests only reveal the presence of defects, not their absence
    yet ... Tests verify whether a system is as right as it was before
- Traceability?
  + Link from requirements specification to system source code

- Test techniques
  + Individual test are white box or black box tests
    - **White box**: exploit knowledge of internal structure
      > e.g., path testing, condition testing
    - **Black box**: exploit knowledge about inputs/outputs
      > e.g., input- and output partitioning + boundary conditions
  + Code *Coverage* to measure the strength of a test suite
    - Line - statement - MC/DC - mutation

Formal Specifications

- Input/Output Specifications

  + = include logic assertions (pre- and postconditions + invariants) in algorithm

  > prove assertions via formal reasoning

- State-Based Specifications

  + = Specify acceptable message sequences by means of state machine

- Are we building the system right?

  + Makes verification easier

  > generation of test cases

  > deduction of contractual obligations

- Traceability?

  + Extra intermediate representation may hinder traceability

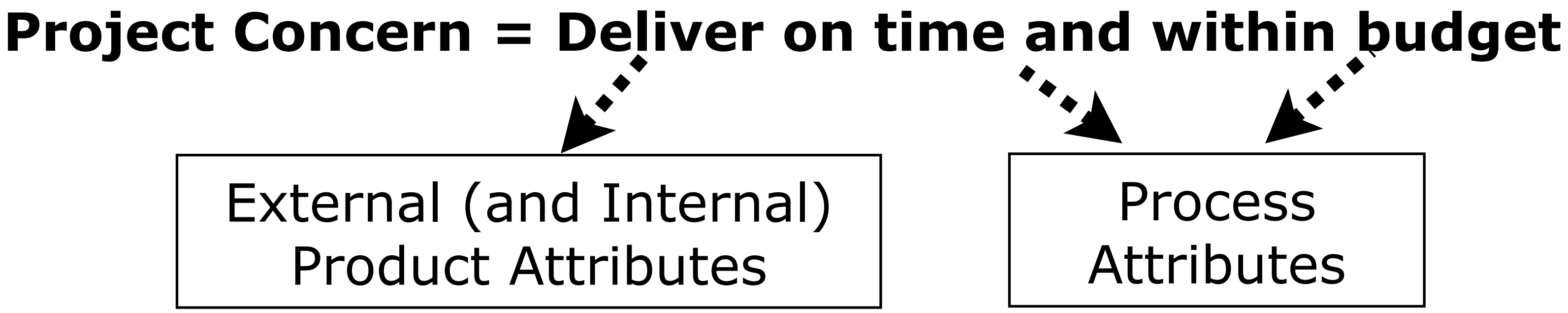
Domain Modeling

- CRC Cards
  + = Analyse system as a set of classes
    - ... each of them having a few responsibilities
    - ... and collaborating with other classes to fulfill these responsibilities
- Feature Model
  + a set of reusable and configurable requirements for specifying system families (a.k.a. product line)

- Are we building the system right?
  + A robust domain model is easier to maintain (= long-term reliability).
- Are we building the right system?
  + CRC Cards and role playing validate use cases.
  + Feature diagrams make product differences (and choices) explicit

- Traceability?
  + Via proper naming conventions

Quality Control



Project Concern = Deliver on time and within budget

External (and Internal) Product Attributes

Process Attributes

- Quality Control
  + = include checkpoints in the process to verify quality attributes
  + Formal technical reviews are very effective and cost effective!
- Quality Standards (ISO9000 and CMM)
  + = Checklists to verify whether a quality system may be certified

- Are we building the system right?
  Are we building the right system?
  + Quality Control eliminates coincidence.
- Traceability?
  + Only when part of the quality plan/system

Software Metrics

- Effort and Cost Estimation
  + = measure early products to estimate costs of later products
  + algorithmic cost modeling, i.e. estimate based on previous experience

- Correctness?
  + Algorithmic cost modeling provides reliable estimates (incl. risk factor)
- Traceability?
  + Quantification of estimates allows for negotiations

- Quality Assurance
  + = quantify the quality model
  + Via internal and external product metrics
- Correctness & Traceability?
  + Software metrics are too premature too assure reliable assessment

Refactoring

- Refactoring Operation
  + = Behaviour-preserving program transformation
  + e.g., rename, move methods and attributes up and down in the hierarchy
- Refactoring Process
  + = Improve internal structure without altering external behaviour
- Code Smell
  + = Symptom of a not so good internal structure
  + e.g, complex conditionals, duplicated code
- Are we building the system right?
  + Behaviour preserving ⇒ as right as it was before (cfr. tests)
- Are we building the right system?
  + Improve internal structure ⇒ cope with requirements mismatches.
- Traceability?
  + Renaming may help to maintain naming conventions
  + Refactoring makes it (too) easy to alter the code without changing the documentation

CHAPTER 12 – Conclusion

- Overview
  - + 1.Introduction
  - + 2.Requirements
  - + 3.Software Architecture
  - + 4.Project Management
  - + 5.Design by Contract
  - + 6.Testing
  - + 7.Formal Specification
  - + 8.Domain Modeling
  - + 9.Software Quality
  - + 10.Software Metrics
  - + 11.Refactoring
- Articles
  - + The Quest for the Silver Bullet
  - + The Case of the Killer Robot
- Professional Ethics
  - + Cases
- The future: Software Engineering Tools

Assignment: Study an Article of your Choice

- Find and read both of the following articles. Pick the one you liked the most, study it carefully and compare the article with the course contents.

- The Quest for the Silver Bullet
  + [Broo87] Frederick P. Brooks, Jr. “No Silver Bullet: Incidents and Accidents in Software Engineering” IEEE Computer, April 1987.
  + See also [Broo95] Frederick P. Brooks, Jr. “The Mythical Man-Month (20th anniversary edition)” Addison-Wesley.
    - The article is more than 15 years old. Yet, it succeeds in explaining why there will never be an easy solution for solving the problems involved in building large and complex software systems.

- The Killer Robot Case
  + [Epst94] Richard G. Epstein, "The use of computer ethics scenarios in software engineering education: the case of the killer robot.", Software Engineering Education: Proceedings of the 7th SEI CSEE Conference
    - The article is a faked series of newspaper articles concerning a robot which killed its operators due to a software fault. The series of articles conveys the different viewpoints one might have concerning the production of quality software.

Software Engineering & Society

Lives are at stake
(e.g., automatic pilot)

Huge amounts of money
are at stake
(e.g., Ariane V crash)

Software became Ubiquitous
Our society is vulnerable!
⇒ Deontology, Licensing, …

Corporate success or failure is at stake
(e.g., telephone billing,
VTM launching 2nd channel)

Your personal future is
at stake (e.g., Y2K lawsuits)

Code of Ethics

- Software Engineering Code of Ethics and Professional Practice
  + ACM-site: http://www.acm.org/serving/se/code.htm
  + IEEE-site: http://computer.org/tab/swecc/code.htm
- Recommended by
  + IEEE-CS (Institute of Electrical and Electronics Engineers - Computer Society)
  + ACM (Association for Computing Machinery)

- "Software Engineering Code of Ethics is Approved", Don Gotterbarn, Keith Miller, Simon Rogerson, Communications of the ACM, October 1999, Vol42, no. 10, pages 102-107.
  + Announces the revised 5.2 version of the Code

- "Using the New ACM Code of Ethics in Decision Making", Ronald E. Anderson, Deborah G. Johnson, Donald Gotterbarn, Judith Perrolle, Communications of the ACM, February 1993, Vol36, no. 2, pages 98-104.
  + Discusses 9 cases of situations you might encounter and how (an older version of) the code address them

Code of Ethics: 8 Principles

  - ACM-site: http://www.acm.org/serving/se/code.htm
  - IEEE-site: http://computer.org/tab/swecc/code.htm

- 1. PUBLIC
  - Software engineers shall act consistently with the public interest.
- 2. CLIENT AND EMPLOYER
  - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- 3. PRODUCT
  - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
- 4. JUDGMENT
  - Software engineers shall maintain integrity and independence in their professional judgment.
- 5. MANAGEMENT
  - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
- 6. PROFESSION
  - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- 7. COLLEAGUES
  - Software engineers shall be fair to and supportive of their colleagues.
- 8. SELF
  - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Case: Privacy - Description

- Case Description
  + You consult a company concerning a database for personnel management.
  + Database will include sensitive data: performance evaluations, medical data.
  + System costs too much and company wants to cut back in security.

- What does the code say?
  + 1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
  + 3.12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

  > Situation is unacceptable.

Case study: Privacy - Solution

- Applicable Clauses
  + 1.02. Moderate the interests of the software engineer, the employer, the client and the users with the public good.
  + 1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.
  + 2.07. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.
  + 6.09. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of ethics, and the subsequent ramifications of such commitment.

- Actions
  + Try to convince management to keep high security standards.
  + Include in contract a clause to cancel contract when against the code of ethics.
  + Alarm other institutions if you later hear that others accepted the contract.

Case study: Privacy - Solution

If you are an independent consultant, how can you ensure that you will not have to act against the code of ethics?

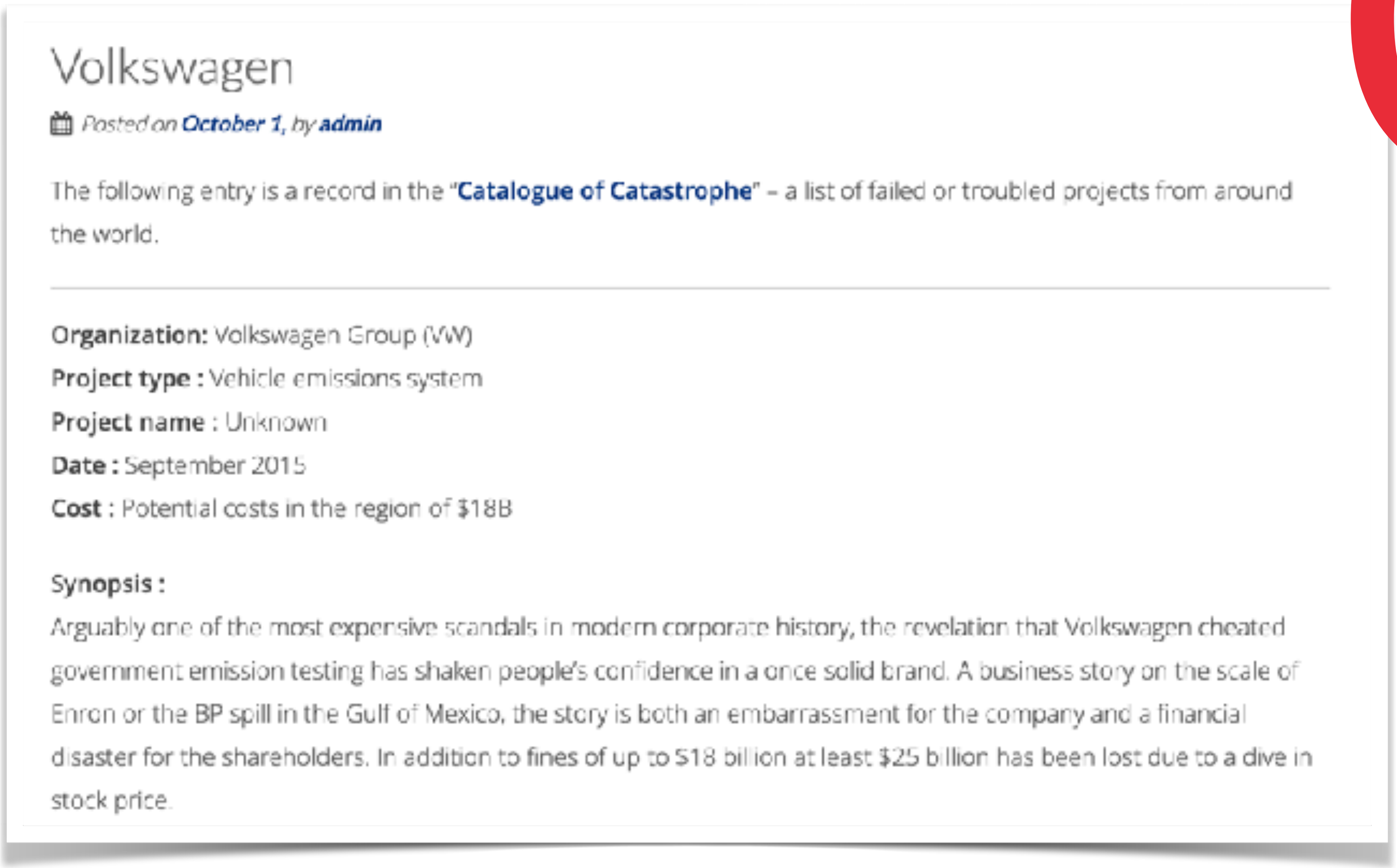
- Actions
  - + …
  - + Include in contract a clause to cancel contract when against the code of ethics.
  - + …

Case: Unreliability

- Case Description
  + You’re the team leader of a team building software for calculating taxes.
  + Your team and your boss are aware that the system contains a lot of defects. Consequently you state that the product can’t be shipped in its current form.
  + Your boss ships the product anyway, with a disclaimer “Company X is not responsible for errors resulting from the use of this program”.

- What does the code say?
  + 1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
  + 5.11. Not ask a software engineer to do anything inconsistent with this Code.
  + 5.12. Not punish anyone for expressing ethical concerns about a project.
    > Disclaimer does not apply: can only be made in “good conscience”.
    > In court you can not be held liable.

VW emissions scandal



Volkswagen

Posted on ***October 1**, by* ***admin***

The following entry is a record in the "**Catalogue of Catastrophe**" – a list of failed or troubled projects from around the world.

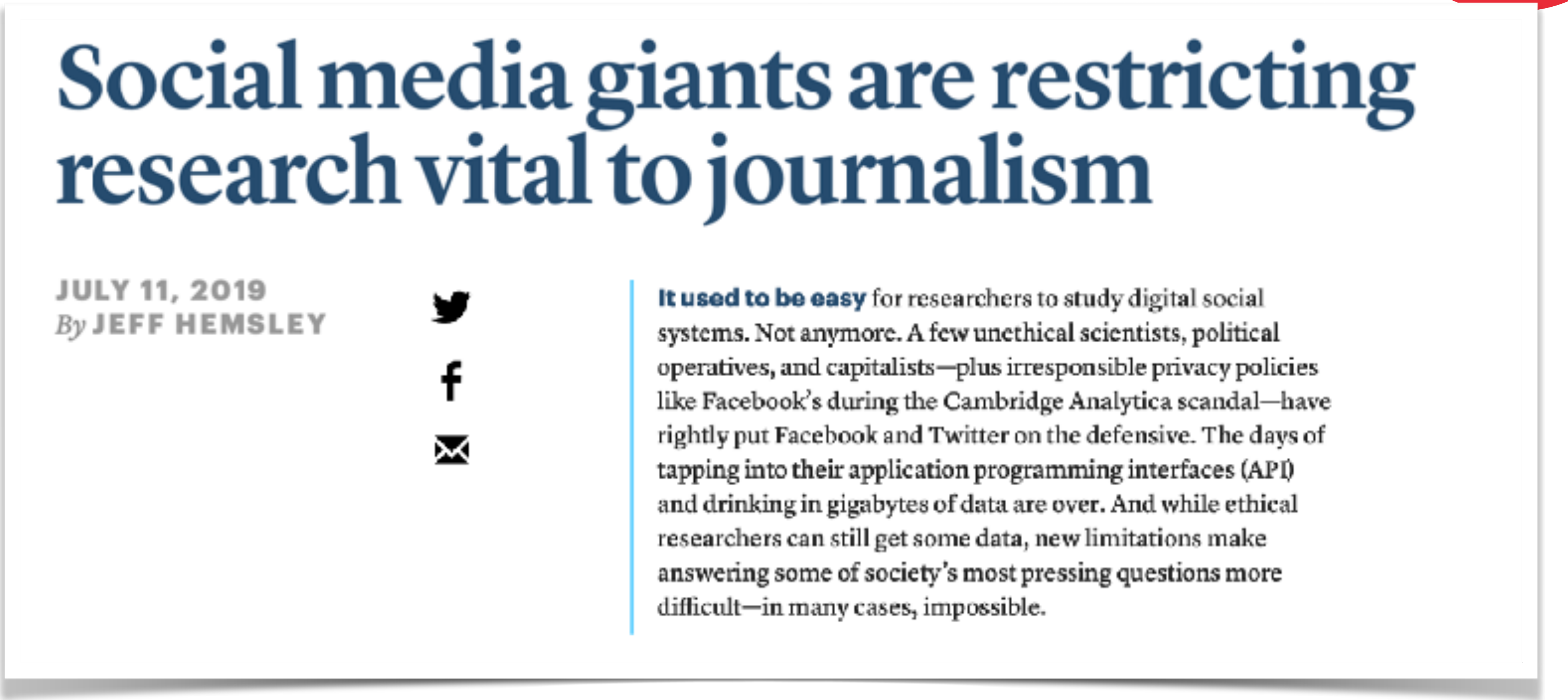
Organization: Volkswagen Group (VW)
Project type : Vehicle emissions system
Project name : Unknown
Date : September 2015
Cost : Potential costs in the region of $18B

Synopsis :
Arguably one of the most expensive scandals in modern corporate history, the revelation that Volkswagen cheated government emission testing has shaken people's confidence in a once solid brand. A business story on the scale of Enron or the BP spill in the Gulf of Mexico, the story is both an embarrassment for the company and a financial disaster for the shareholders. In addition to fines of up to $18 billion at least $25 billion has been lost due to a dive in stock price.

Your mission should you choose to accept.

- You are a software engineer working for volkswagen. Your management asks to install a so called "defeat device" into the car to circumvent emission tests.

Facebook / Twitter API



Social media giants are restricting research vital to journalism

JULY 11, 2019
By JEFF HEMSLEY

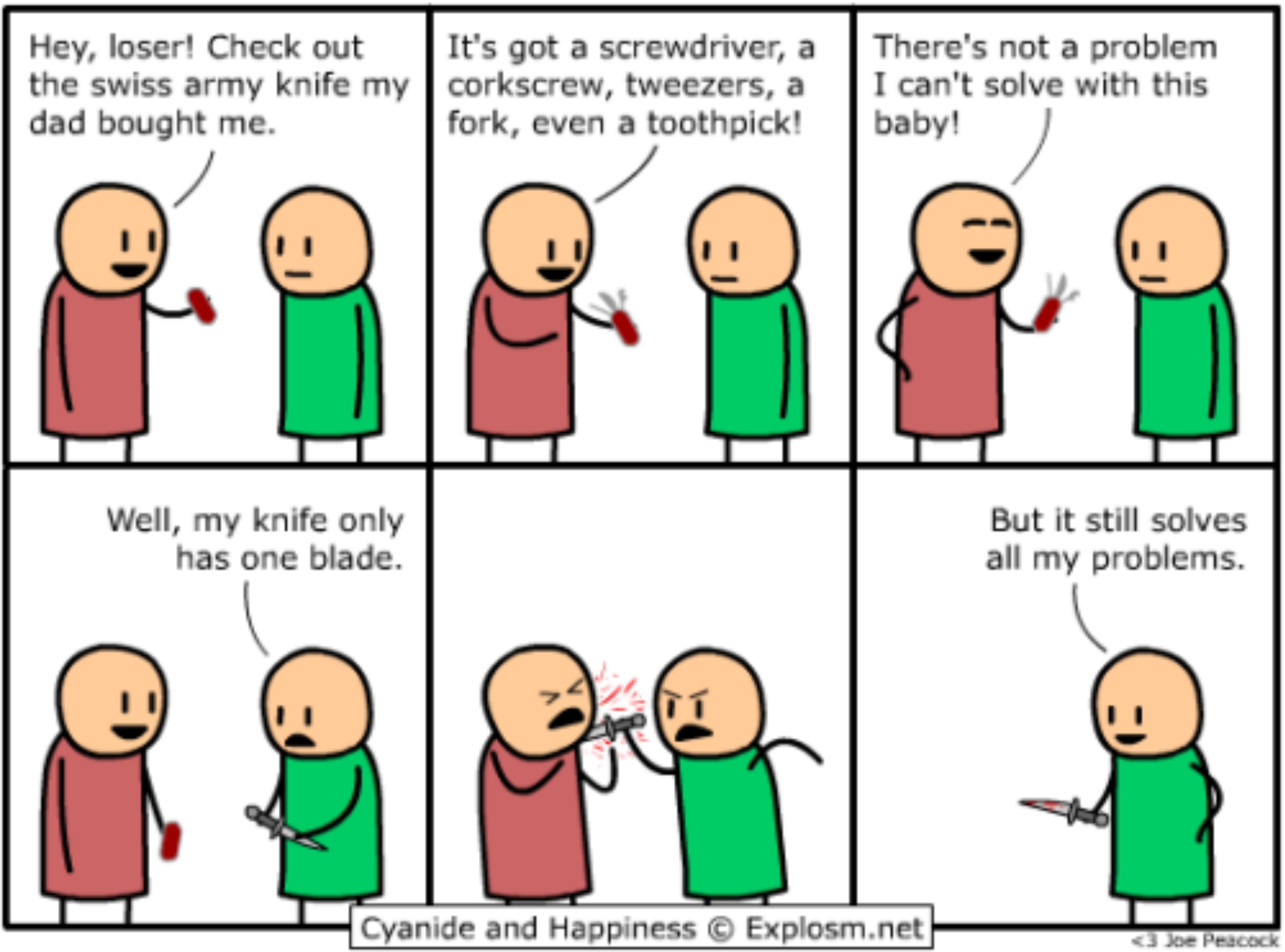
It used to be easy for researchers to study digital social systems. Not anymore. A few unethical scientists, political operatives, and capitalists—plus irresponsible privacy policies like Facebook's during the Cambridge Analytica scandal—have rightly put Facebook and Twitter on the defensive. The days of tapping into their application programming interfaces (API) and drinking in gigabytes of data are over. And while ethical researchers can still get some data, new limitations make answering some of society's most pressing questions more difficult—in many cases, impossible.

Your mission should you choose to accept.

- You are a master thesis student and you are asked to inject "spy software" on the API of big social media for research purposes.

CHAPTER 12 – Conclusion

- Over
  + 1.
  + 2.
  + 3.
  + 4.
  + 5.
  + 6.
  + 7.
  + 8.
  + 9.
  + 10
  + 11
- Artic
  + Th
  + Th
- Profe
  + Ca
- The future: Software Engineering Tools


Hey, loser! Check out the swiss army knife my dad bought me.
It's got a screwdriver, a corkscrew, tweezers, a fork, even a toothpick!
There's not a problem I can't solve with this baby!
Well, my knife only has one blade.
But it still solves all my problems.

Cyanide and Happiness © Explosm.net

<3 Joe Peacock

Innovation

Underlying Technology

1908 — patent on paper filter

1946 — commerical piston espresso machine

2000 — nespresso

2001 — senseo

Business Models

1475 — Kiva Han coffee house
(Constantinople)

(Vienna)
1529 — European coffee house

1971 — Starbucks
(seattle)

Innovation

Underlying Technology

1908 — patent on paper filter

1946 — commerical piston espresso machine

2000 — nespresso

2001 — senseo

Business Models

1475 — Kiva Han coffee house (Constantinople)

1529 — European coffee house (Vienna)

1971 — Starbucks (seattle)

STARBUCKS COFFEE

Technology changes every 20 years

...

Underlying business models rarely change!

Innovation in ICT

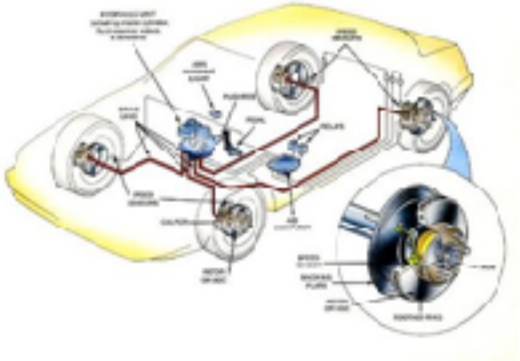
ENIAC, 1945

IBM PC, 1981

NEC ultralite, 1989

iPad, 2010

Underlying Technology



Embedded

Internet

Innovation in ICT

ENIAC, 1945

IBM PC, 1981

NEC ultralite, 1989

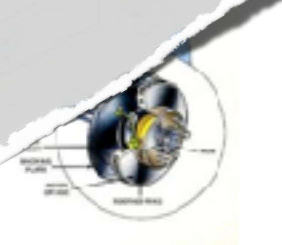
iPad, 2010

Underlying Technology

Technology changes every 5 years

…

Underlying business models change often!



Embedded

Google YouTube Drupal facebook twitter

Internet

Market pressure in ICT

YAHOO!

altavista

Google

Measure of innovation

- # products in portfolio younger than 5 years
  + in ICT usually more than 1/2 the portfolio

Significant investment in R&D

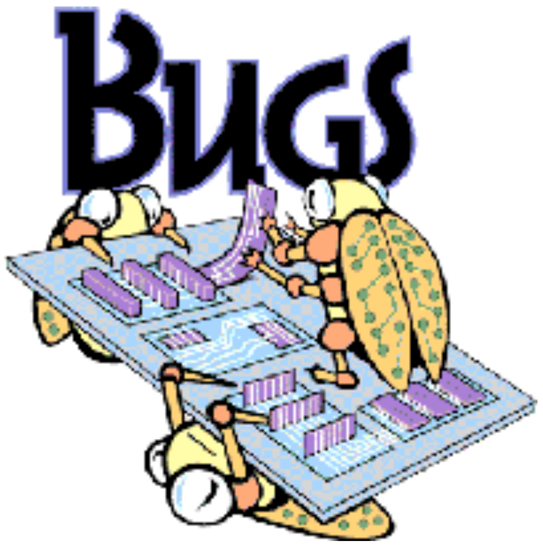
- more products … faster

RELIABILITY ↔ AGILITY

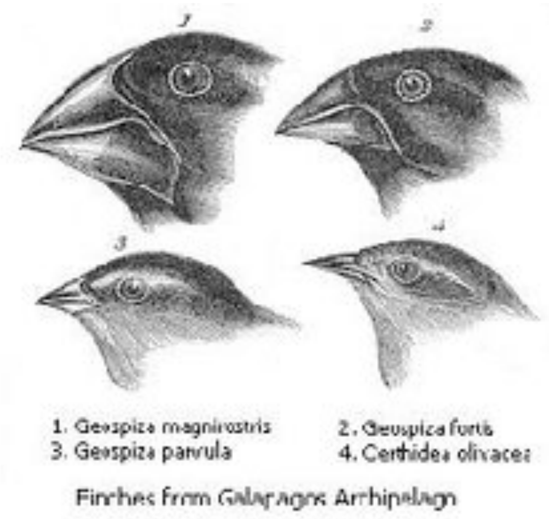
Reliability vs. Agility

Software is vital to our society ⇒ Software must be reliable

Traditional Software Engineering
Reliable = Software without bugs

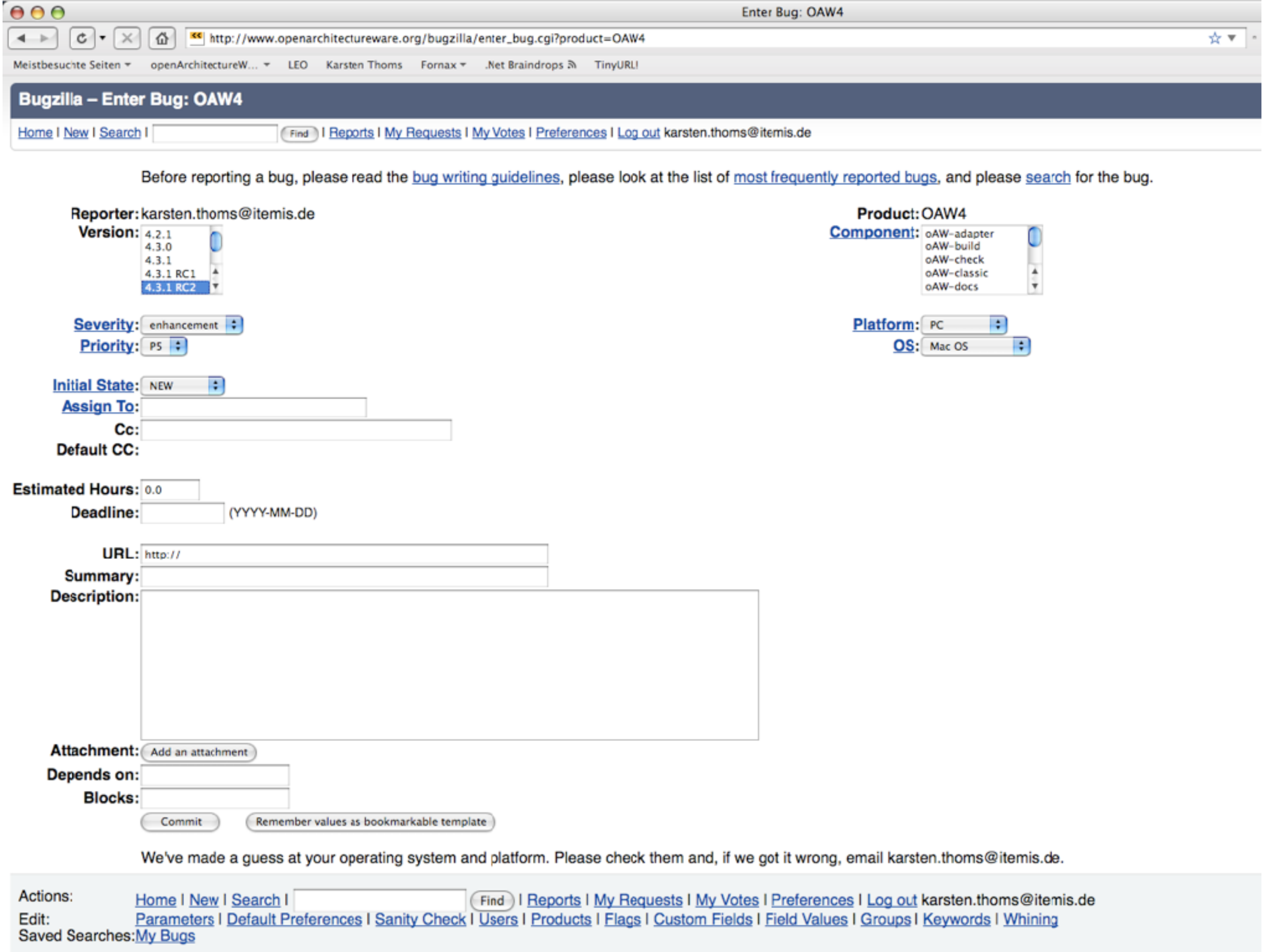


Today's Software Engineering
Reliable = Easy to Adapt



On the Origin of Species

Striving for RELIABILITY (Optimise for *perfection*)	↔	Striving for AGILITY (Optimise for *development speed*)



Enter Bug: OAW4
http://www.openarchitectureware.org/bugzilla/enter_bug.cgi?product=OAW4
Meistbesuchte Seiten
openArchitectureW...
LEO
Karsten Thoms
Fornax
.Net Braindrops
TinyURL!

Bugzilla – Enter Bug: OAW4

Home | New | Search | Find | Reports | My Requests | My Votes | Preferences | Log out karsten.thoms@itemis.de

Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

Reporter: karsten.thoms@itemis.de

Product: OAW4

Version: 4.2.1, 4.3.0, 4.3.1, 4.3.1 RC1, 4.3.1 RC2

Component: oAW-adapter, oAW-build, oAW-check, oAW-classic, oAW-docs

Severity: enhancement

Platform: PC

Priority: P5

OS: Mac OS

Initial State: NEW

Assign To:

Cc:

Default CC:

Estimated Hours: 0.0

Deadline: (YYYY-MM-DD)

URL: http://

Summary:

Description:

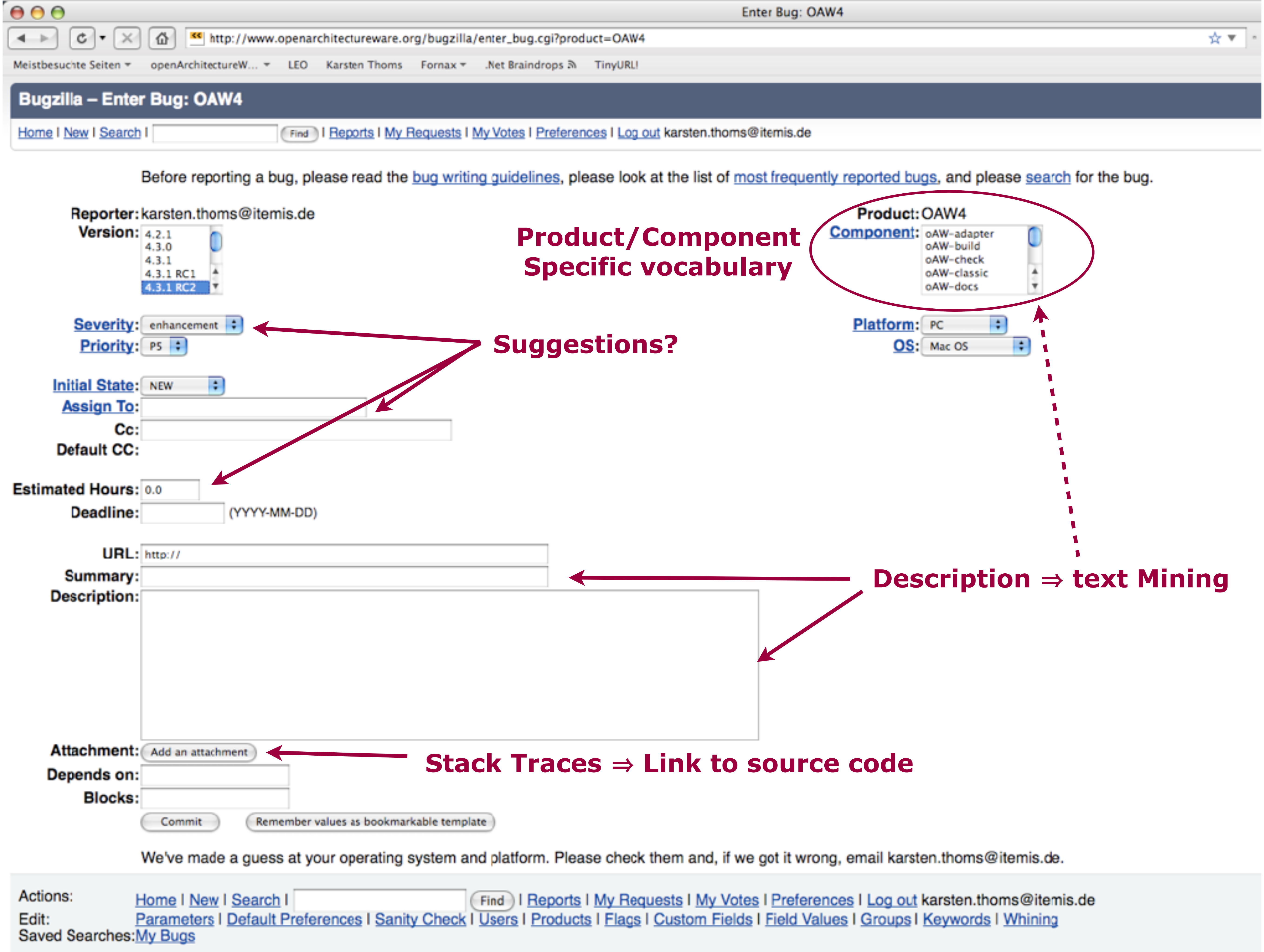
Attachment: Add an attachment

Depends on:

Blocks:

We've made a guess at your operating system and platform. Please check them and, if we got it wrong, email karsten.thoms@itemis.de.

Actions: Home | New | Search | Find | Reports | My Requests | My Votes | Preferences | Log out karsten.thoms@itemis.de
Edit: Parameters | Default Preferences | Sanity Check | Users | Products | Flags | Custom Fields | Field Values | Groups | Keywords | Whining
Saved Searches: My Bugs



Enter Bug: OAW4

http://www.openarchitectureware.org/bugzilla/enter_bug.cgi?product=OAW4

Bugzilla – Enter Bug: OAW4

Home | New | Search | [Find] | Reports | My Requests | My Votes | Preferences | Log out karsten.thoms@itemis.de

Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

Reporter:	karsten.thoms@itemis.de
Version:	4.2.1 4.3.0 4.3.1 4.3.1 RC1 4.3.1 RC2
Severity:	enhancement
Priority:	P5
Initial State:	NEW
Assign To:	
Cc:	
Default CC:	
Estimated Hours:	0.0
Deadline:	(YYYY-MM-DD)
URL:	http://
Summary:	
Description:	
Attachment:	[Add an attachment]
Depends on:	
Blocks:	

Product:	OAW4
Component:	oAW-adapter oAW-build oAW-check oAW-classic oAW-docs
Platform:	PC
OS:	Mac OS

[Commit] [Remember values as bookmarkable template]

We've made a guess at your operating system and platform. Please check them and, if we got it wrong, email karsten.thoms@itemis.de.

Actions: Home | New | Search | [Find] | Reports | My Requests | My Votes | Preferences | Log out karsten.thoms@itemis.de

Edit: Parameters | Default Preferences | Sanity Check | Users | Products | Flags | Custom Fields | Field Values | Groups | Keywords | Whining

Saved Searches: My Bugs

Product/Component Specific vocabulary

Suggestions?

Description ⇒ text Mining

Stack Traces ⇒ Link to source code

Bug Report Triaging

Artificial Intelligence Inside

Question	Cases	Precision	Recall
Who should fix this bug?	Eclipse, Firefox, gcc	eclipse: 57% firefox: 64% gcc: 6%	—
How long will it take to fix this bug?	JBoss	depends on the component many similar reports: off by one hour few similar reports: off by 7 hours	
What is the severity of this bug?	Mozilla, Eclipse, Gnome	mozilla, eclipse:67% - 73% gnome: 75%-82%	mozilla, eclipse:50% - 75% gnome: 68%-84%

Bug Report Triaging

Artificial Intelligence Inside

Question	Cases	Precision	Recall
~~Who should fix this bug?~~	~~E [illegible] x, gcc~~	~~eclipse: 57%~~ ~~firefox: 64%~~ ~~gcc: 6%~~	—
How long will it take to fix this bug?	JBoss	depends on the component many similar reports: off by one hour few similar reports: off by 7 hours	
What is the severity of this bug?	M [illegible] me	mozilla, eclipse:67% - 73% gnome: 75%-82%	mozilla, eclipse:50% - 75% gnome: 68%-84%

Irrelevant for Practitioners

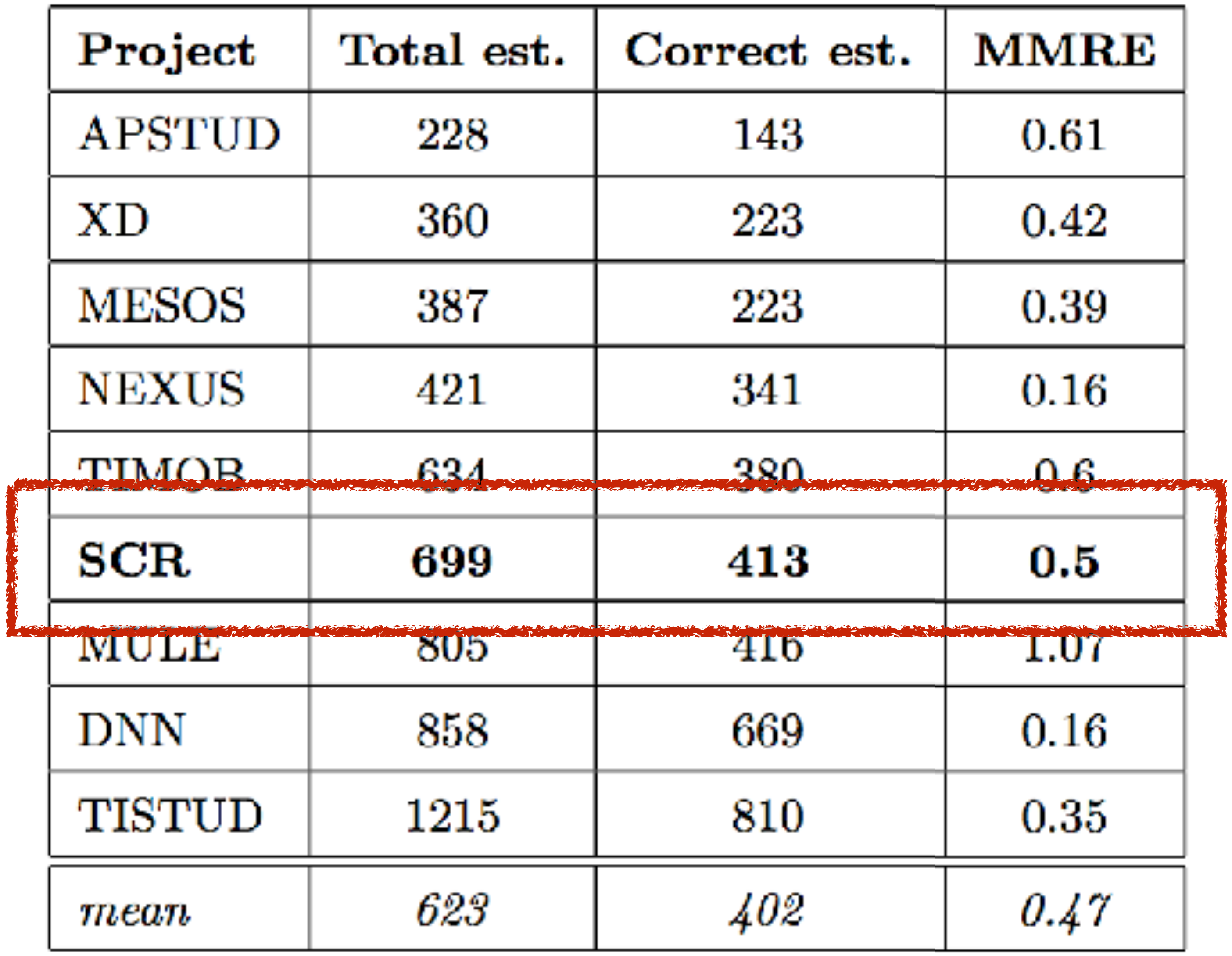
Internal vs. External Bug Reports

Story Points (Planning Poker)

Public Domain

1/2	1	2	3	5	8	13	20	40	100	∞

Results

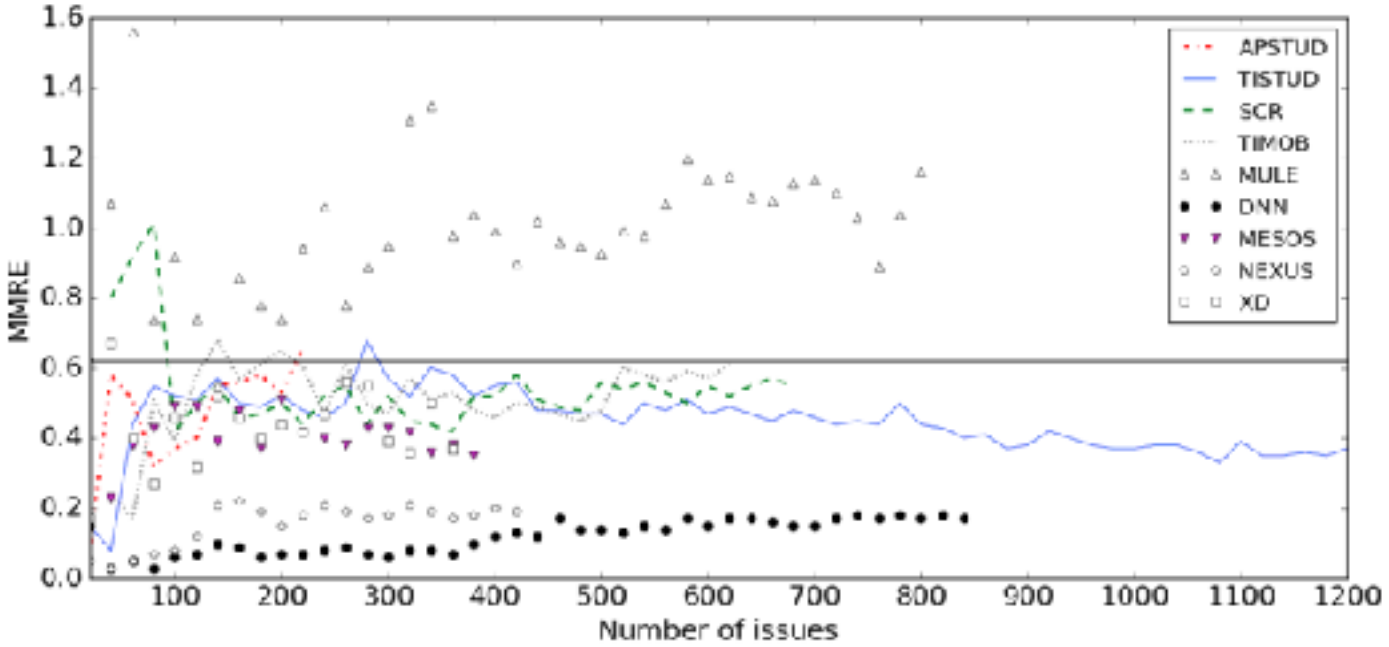


Project	Total est.	Correct est.	MMRE
APSTUD	228	143	0.61
XD	360	223	0.42
MESOS	387	223	0.39
NEXUS	421	341	0.16
TIMOB	634	380	0.6
SCR	**699**	**413**	**0.5**
MULE	805	416	1.07
DNN	858	669	0.16
TISTUD	1215	810	0.35
mean	*623*	*402*	*0.47*

Human
MMRE: **0.48**

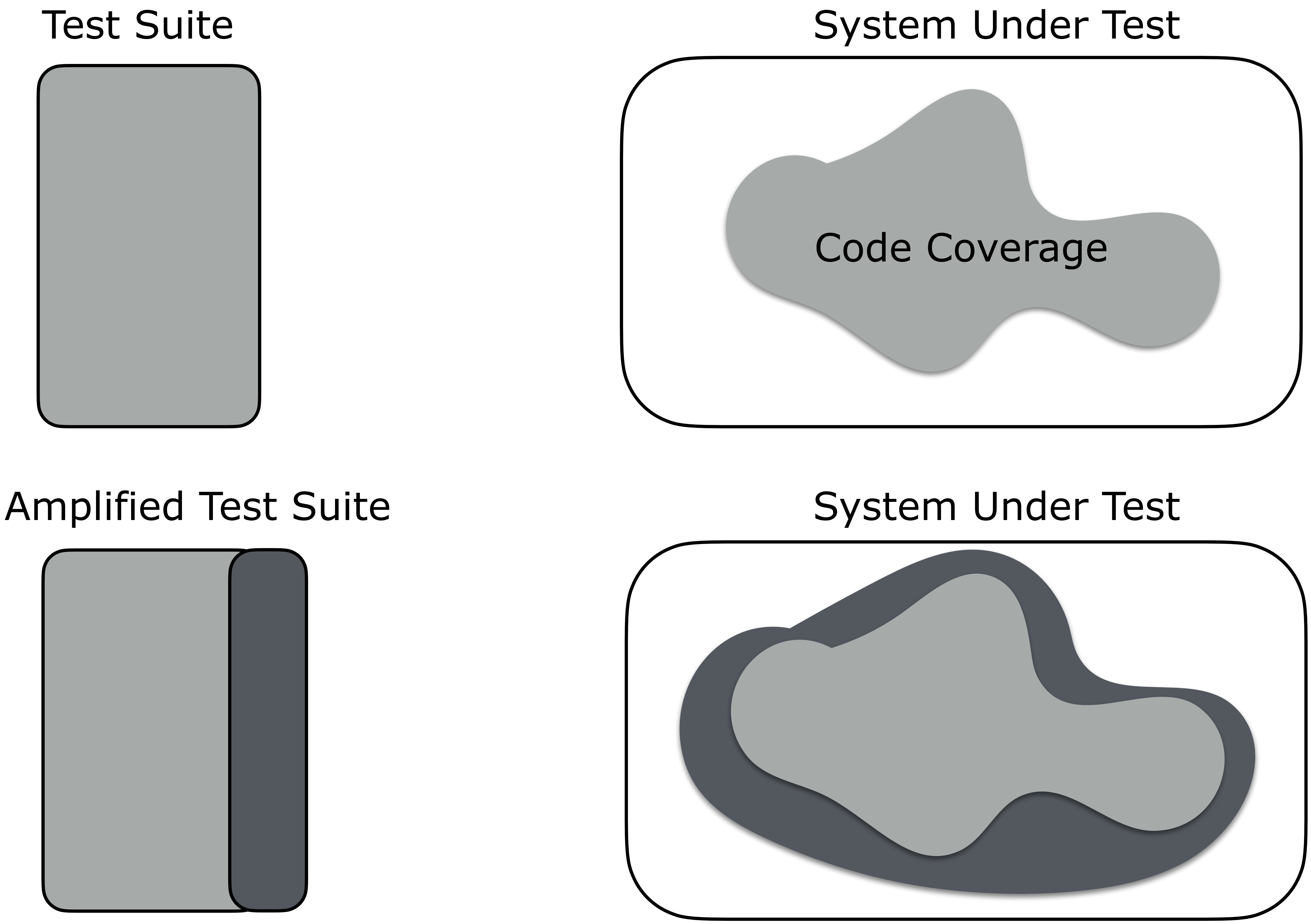
(*) Mean Magnitude
of Relative Error

Learning Curve



“in vivo” Validation

Test Amplification



Test Suite

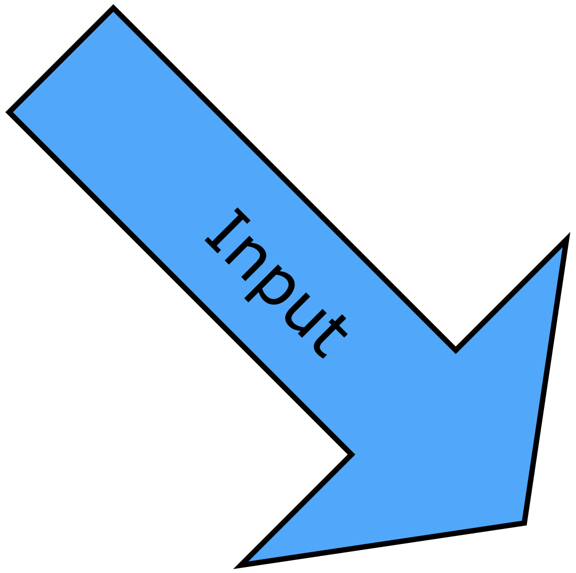
System Under Test

Code Coverage

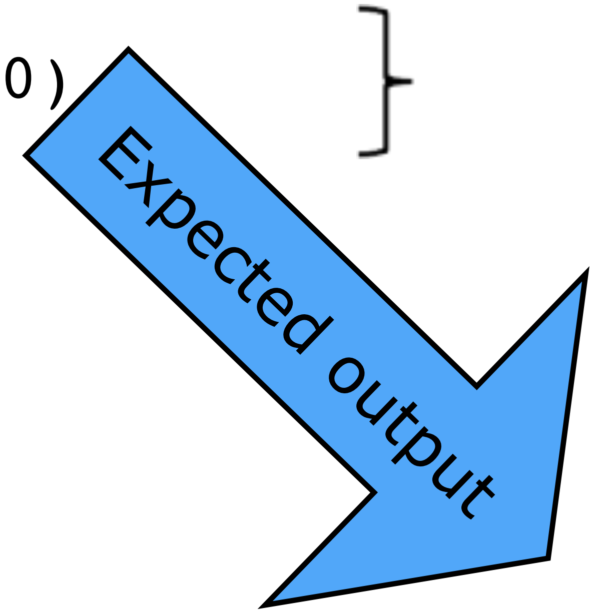
Amplified Test Suite

System Under Test

Example - testDeposit



```
def testDeposit (self) :
  self.b.set_owner('Iwena Kroka')
  self.b.deposit(10)
  self.assertEqual(self.b.get_balance(), 10)
  self.b.deposit(100)
  self.b.deposit(100)
  self.assertEqual(self.b.get_balance() , 210)
```



Example - testDeposit_amplified (1/2)

Assertion Amplification

```
def testDeposit_amplified (self) :
  self.b.set_owner(’Iwena Kroka’)
  self.b.deposit(10)
  self.assertEqual(self.b.
          get_transactions(), [10])
  self.assertFalse(self.b.is_empty () )
  self.assertEqual(self.b.owner, ’Iwena Kroka’)
  self.assertEqual(self.b.get_balance(), 10)
  …
```

Assertion Amplification = (re)generate appropriate assertions to verify the actual state of the object under test by observing the run-time behaviour.

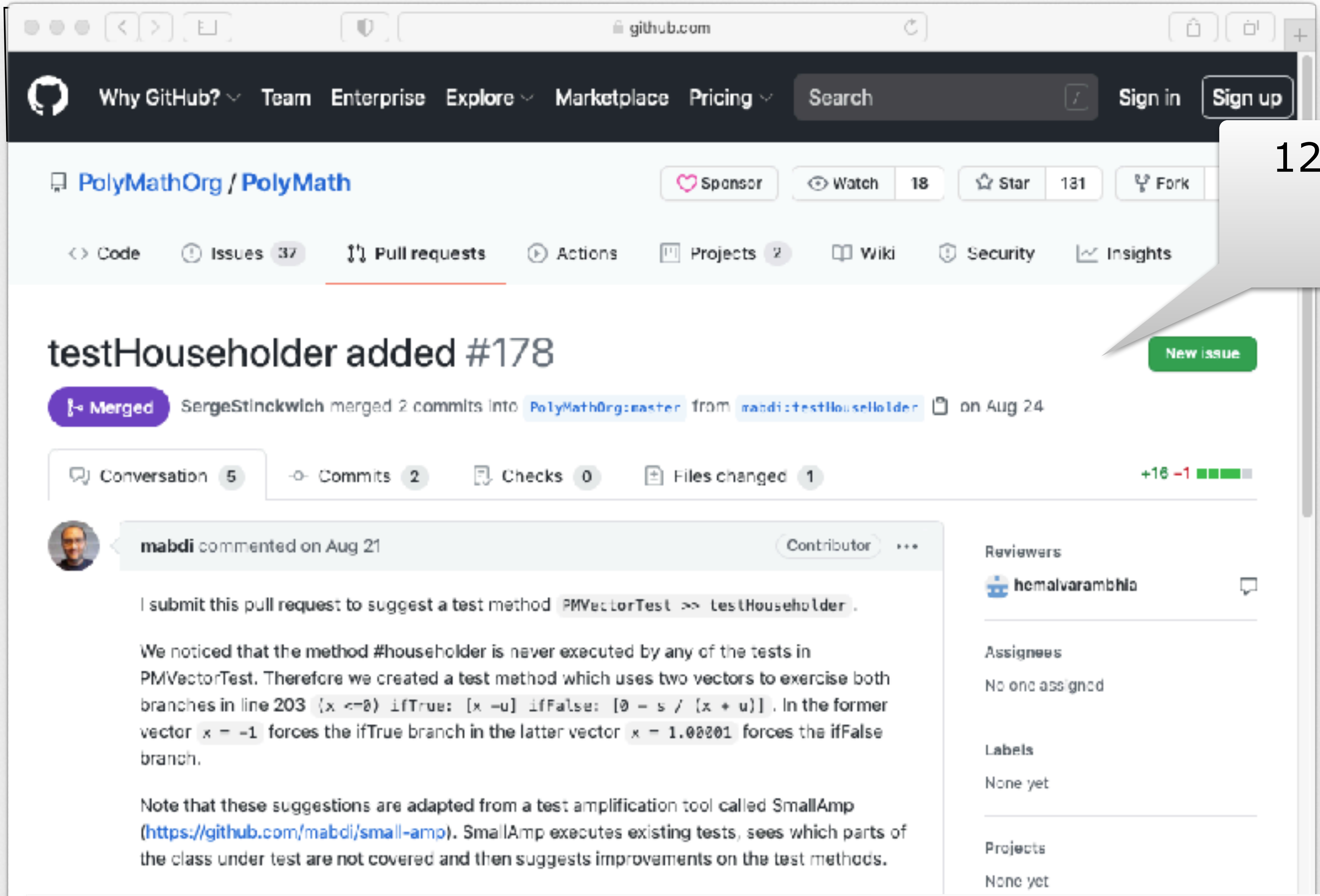
Example - testDeposit_amplified (2/2)

Input Amplification

```
def testDeposit_amplified (self) :
  self.b.set_owner('Iwena Kroka')
  self.b.deposit(10)
  self.assertEqual(self.b.
          get_transactions(), [10])
  self.assertFalse(self.b.is_empty () )
  self.assertEqual(self.b.owner, 'Iwena Kroka')
  self.assertEqual(self.b.get_balance(), 10)
  with self.assertRaises(Exception):
     self.b.deposit(-56313)
  self.b.deposit(100)
  self.b.set_owner('Guido van Rossum')
  self.assertEqual(self.b.
           get_transactions(), [10])
…
```

Input Amplification = Transform the original test method(*); forcing previously untested paths.
(*) Change the set-up of the object under test, providing parameters that represent boundary conditions; inject calls to state-changing methods

⇒ Brute force but optimize via increase in code coverage



testHouseholder added #178

Merged SergeStinckwich merged 2 commits into `PolyMathOrg:master` from `mabdi:testHouseholder` on Aug 24

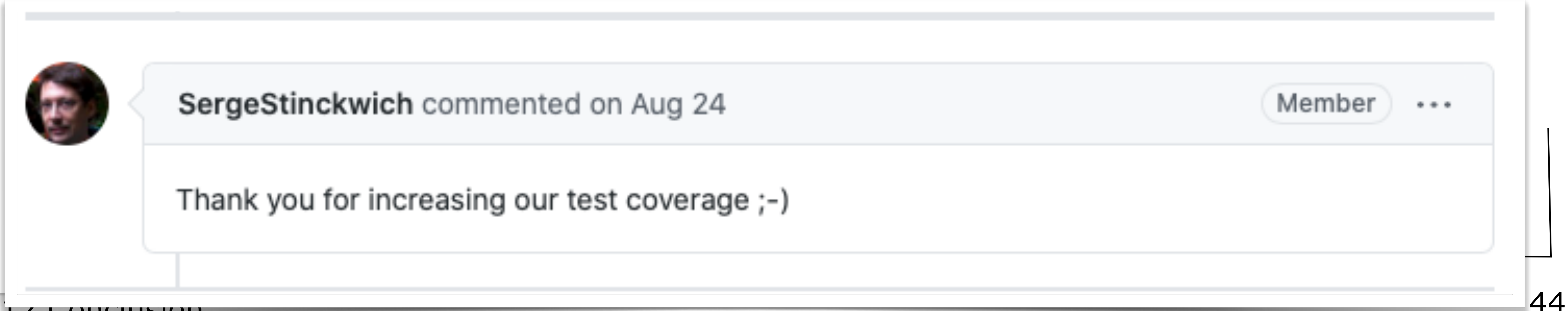
mabdi commented on Aug 21

I submit this pull request to suggest a test method `PMVectorTest >> testHouseholder`.

We noticed that the method #householder is never executed by any of the tests in PMVectorTest. Therefore we created a test method which uses two vectors to exercise both branches in line 203 `(x <=0) ifTrue: [x -u] ifFalse: [0 - s / (x + u)]`. In the former vector `x = -1` forces the ifTrue branch in the latter vector `x = 1.00001` forces the ifFalse branch.

Note that these suggestions are adapted from a test amplification tool called SmallAmp (https://github.com/mabdi/small-amp). SmallAmp executes existing tests, sees which parts of the class under test are not covered and then suggests improvements on the test methods.

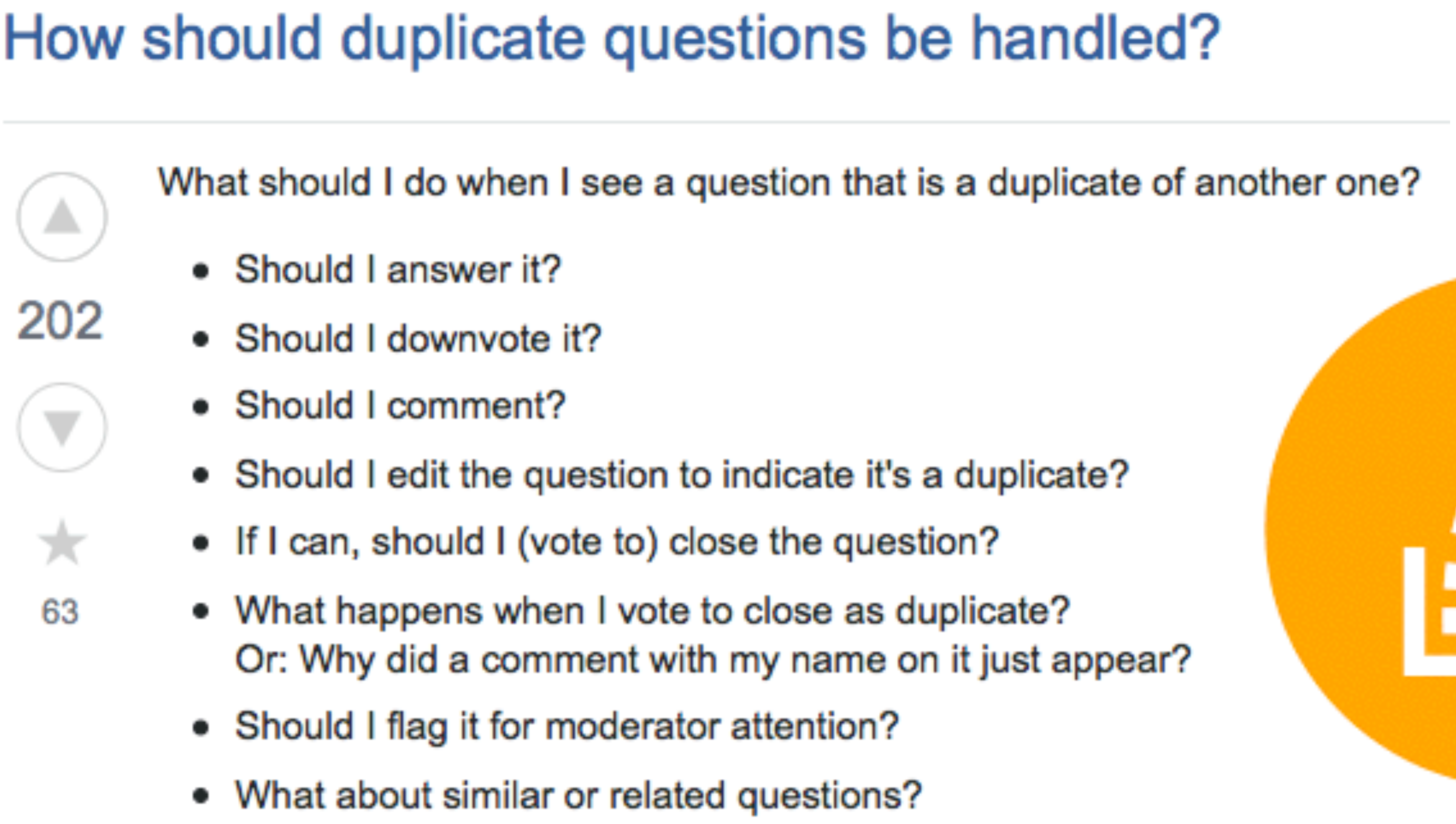
12 pull requests
9 merged
3 pending



SergeStinckwich commented on Aug 24

Thank you for increasing our test coverage ;-)

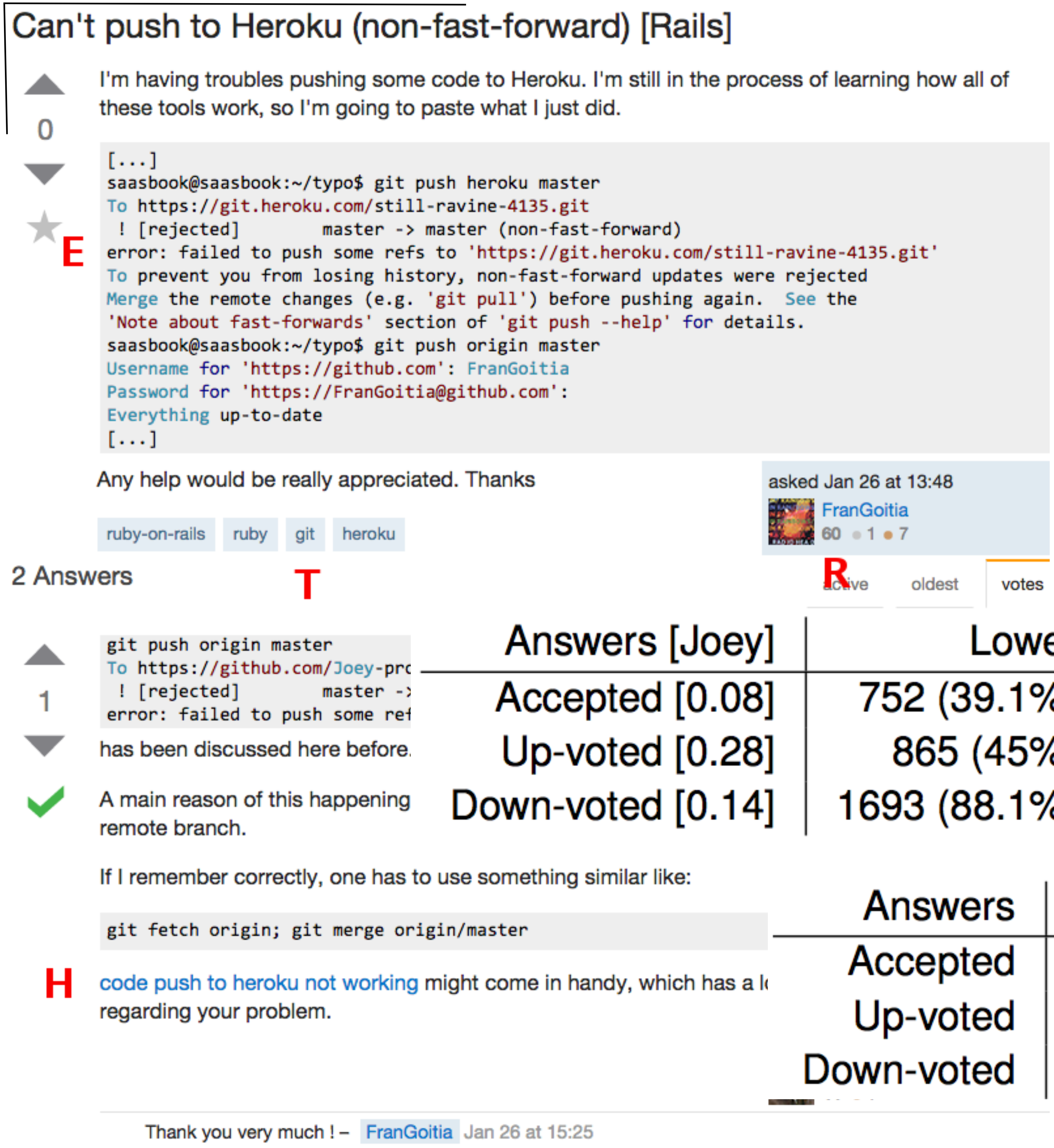
Q&A support



How should duplicate questions be handled?

What should I do when I see a question that is a duplicate of another one?

- Should I answer it?
- Should I downvote it?
- Should I comment?
- Should I edit the question to indicate it's a duplicate?
- If I can, should I (vote to) close the question?
- What happens when I vote to close as duplicate?
  Or: Why did a comment with my name on it just appear?
- Should I flag it for moderator attention?
- What about similar or related questions?



Can't push to Heroku (non-fast-forward) [Rails]

0

I'm having troubles pushing some code to Heroku. I'm still in the process of learning how all of these tools work, so I'm going to paste what I just did.

```
[...]
saasbook@saasbook:~/typo$ git push heroku master
To https://git.heroku.com/still-ravine-4135.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://git.heroku.com/still-ravine-4135.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
saasbook@saasbook:~/typo$ git push origin master
Username for 'https://github.com': FranGoitia
Password for 'https://FranGoitia@github.com':
Everything up-to-date
[...]
```

Any help would be really appreciated. Thanks

ruby-on-rails ruby git heroku

asked Jan 26 at 13:48
FranGoitia
60 ● 1 ● 7

2 Answers

active oldest votes

1

```
git push origin master
To https://github.com/Joey-pro
 ! [rejected]        master -›
error: failed to push some ref
```

has been discussed here before.

A main reason of this happening
remote branch.

If I remember correctly, one has to use something similar like:

```
git fetch origin; git merge origin/master
```

code push to heroku not working might come in handy, which has a l
regarding your problem.

Thank you very much ! – FranGoitia Jan 26 at 15:25

Answers [Joey]	Lower	Equal	Higher
Accepted [0.08]	752 (39.1%)	3 (0.2%)	1167 (60.7%)
Up-voted [0.28]	865 (45%)	3 (0.2%)	1054 (54.8%)
Down-voted [0.14]	1693 (88.1%)	0 (0%)	229 (11.9%)

Answers	Joey	Answer_Bot
Accepted	4/50 (0.08)	1/13 (0.08)
Up-voted	14/50 (0.28)	0
Down-voted	7/50 (0.14)	3/13 (0.23)

12.Conclusion

Summary (i)

- You should know the answers to these questions
  - + Name 3 items from the code of ethics and provide a one-line explanation.
  - + If you are an independent consultant, how can you ensure that you will not have to act against the code of ethics?
  - + What would be a possible metric for measuring the amount of innovation of a manufacturing company?
  - + Explain the 2 main steps of test amplification: input amplification and assertion amplification

Summary (i) - Continued

“No Silver Bullet”

- What’s the distinction between essential and accidental complexity?
- Name 3 reasons why the building of software is essentially a hard task? Provide a one-line explanation.
- Why is “object-oriented programming” no silver bullet?
- Why is “program verification” no silver bullet?
- Why are “components” a potential silver bullet?

“Killer Robot”

- Which regression tests would you have written to prevent the “killer robot”?
- Was code reviewing applied as part of the QA process? Why (not)?
- Why was the waterfall process disastrous in this particular case?
- Why was the user-interface design flawed?

Summary (ii)

- Can you answer the following questions?
  - + You are an experienced designer and you heard that the sales people earn more money than you do. You want to ask your boss for a salary-increase; how would you argue your case?
  - + Software products are usually released with a disclaimer like “Company X is not responsible for errors resulting from the use of this program”. Does this mean that you shouldn’t test your software? Motivate your answer.
  - + Your are a QA manager and are requested to produce a monthly report about the quality of the test process. How would you do that?
  - + Why is “explainable Artificial Intelligence” so important when creating bots for software engineering tasks?
- When you chose the “No Silver Bullet” paper
  - + Explain why incremental development is a promising attack on conceptual essence. Give examples from the different topics addressed in the course.
  - + “Software components” are said to be a promising attack on conceptual essence. Which techniques in the course are applicable? Which techniques aren’t?
- When you chose the “Killer Robot” paper
  - + Recount the story of the Killer Robot case. List the three most important causes for the failure and argue why you think these are the most important.

Summary (i)

- You should know the answers to these questions
  + Name 3 items from the code of ethics and provide a one-line explanation.
  + If you are an independent consultant, how can you ensure that you will not have to act against the code of ethics?
  + What would be a possible metric for measuring the amount of innovation of a manufacturing company?
  + Explain the 2 main steps of test amplification: input amplification and assertion amplification

When you chose the “No Silver Bullet” paper

- What’s the distinction between essential and accidental complexity?
- Name 3 reasons why the building of software is essentially a hard task? Provide a one-line explanation.
- Why is “object-oriented programming” no silver bullet?
- Why is “program verification” no silver bullet?
- Why are “components” a potential silver bullet?

When you chose the “Killer Robot” paper

- Which regression tests would you have written to prevent the “killer robot”?
- Was code reviewing applied as part of the QA process? Why (not)?
- Why was the waterfall process disastrous in this particular case?
- Why was the user-interface design flawed?

Summary (ii)

- Can you answer the following questions?
  - + You are an experienced designer and you heard that the sales people earn more money than you do. You want to ask your boss for a salary-increase; how would you argue your case?
  - + Software products are usually released with a disclaimer like “Company X is not responsible for errors resulting from the use of this program”. Does this mean that you shouldn’t test your software? Motivate your answer.
  - + Your are a QA manager and are requested to produce a monthly report about the quality of the test process. How would you do that?
  - + Why is “explainable Artificial Intelligence” so important when creating bots for software engineering tasks?

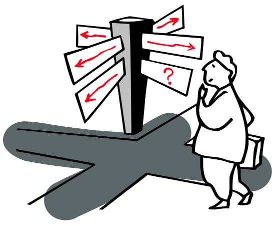
When you chose the “No Silver Bullet” paper

- + Explain why incremental development is a promising attack on conceptual essence. Give examples from the different topics addressed in the course.
- + “Software components” are said to be a promising attack on conceptual essence. Which techniques in the course are applicable? Which techniques aren’t?

When you chose the “Killer Robot” paper

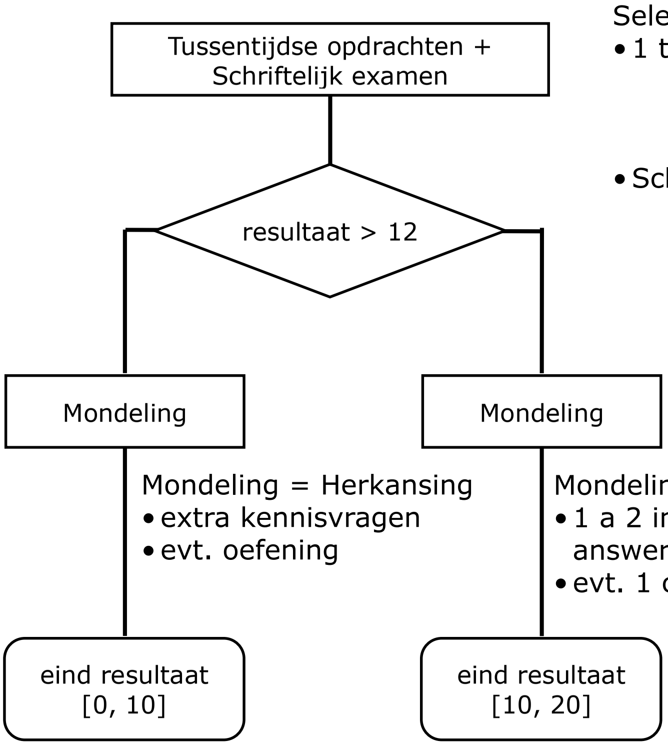
- + Recount the story of the Killer Robot case. List the three most important causes for the failure and argue why you think these are the most important.

13. Appendix. Questions



- + 01. Introduction
- + 02. Requirements
- + 03. Software Architecture
- + 04. Project Management
- + 05. Design By Contract
- + 06. Testing
- + 07. Formal Specifications
- + 08. Domain Models
- + 09. Software Quality
- + 10. Software Metrics
- + 11. Refactoring
- + 12. Conclusion

Examen



Selectie (minimumnorm)

- 1 tussentijdse opdracht per hoofdstuk
  - Tijdens het jaar op te leveren
  - Presentatie + Kwalitatieve feedback
  - Resultaat > 12
- Schriftelijk examen
  - 1 kennis vraag per hoofdstuk
    - (cfr. "You should know the answer to these questions")
  - beperkte oefeningen
  - Resultaat > 12

Mondeling = Herkansing

- extra kennisvragen
- evt. oefening

Mondeling = Diversificatie

- 1 a 2 inzichtsvragen (cfr. "Can you answer the following questions")
- evt. 1 creatieve vraag

1. Introduction (1/2)

- You should know the answers to these questions:
  - + How does Software Engineering differ from programming?
  - + Why is programming only a small part of the cost of a “real” software project ?
  - + Give a definition for “traceability”.
  - + What is the difference between analysis and design?
  - + Explain verification and validation in simple terms.
  - + Why is the “waterfall” model unrealistic? Why is it still used?
  - + Can you explain the difference between iterative development and incremental development?
  - + How do you decide to stop in the spiral model?
  - + How do you identify risk? How do you asses a risk? Which risks require action?
  - + What is Failure Mode and Effects Analysis (FMEA)?
  - + List the 6 principles of extreme programming.
  - + What is a “sprint” in the SCRUM process?
  - + Give the three principal roles in a scrum team. Explain their main responsibilities.
  - + Draw a UML class diagram modelling marriages in cultures with monogamy (1 wife marries 1 husband), polygamy (persons can be married with more than one other person), polyandry (1 woman can be married to more than one man) and polygyny (1 man can be married to more than one woman).
  - + Draw a UML diagram that represents an object “o” which creates an account (balance initially zero), deposits 100$ and then checks whether the balance is correct.

Introduction (2/2)

- Can you answer the following questions?
  - + What is your preferred definition of Software Engineering? Why?
  - + Why do we choose “Correctness” & “Traceability” as evaluation criteria? Can you imagine some others?
  - + Why is “Maintenance” a strange word for what is done during the activity?
  - + Why is risk analysis necessary during incremental development?
  - + How can you validate that an analysis model captures users’ real needs?
  - + When does analysis stop and design start?
  - + When can implementation start?
  - + Can you compare the Unified Process and the Spiral Model?
  - + Can you explain the values behind the Agile Manifesto?
  - + Can you identify some synergies between the techniques used during extreme programming?
  - + Can you explain how the different steps in the scrum process create a positive feedback loop?
  - + How does scrum reduce risk?
  - + Is it possible to apply Agile Principles with the Unified Process?
  - + Did the UML succeed in becoming *the* Universal Modeling Language? Motivate your answer.

2. Requirements (1/2)

- You should know the answers to these questions
  - + Why should the requirements specification be understandable, precise and open?
  - + What’s the relationship between a use case and a scenario?
  - + Can you give 3 criteria to evaluate a system scope description? Why do you select these 3?
  - + Why should there be at least one actor who benefits from a use case?
  - + Can you supply 3 questions that may help you identifying actors? And use cases?
  - + What’s the difference between a primary scenario and a secondary scenario?
  - + What’s the direction of the <<extends>> and <<includes>> dependencies?
  - + What is the purpose of technical stories in scrum?
  - + List and explain briefly the INVEST criteria for user stories.
  - + Explain briefly the three levels of detail for Product Backlog Items (Epic, Features, Stories).
  - + *What is a minimum viable product?*
  - + Define a misuse case.
  - + Define a safety story.

- You should be able to complete the following tasks
  - + Write a requirements specification for your bachelor capstone project.

CAPSTONE PROJECT

Requirements (2/2)

- Can you answer the following questions?
  - + Why do use cases fit well in an iterative/incremental development process?
  - + Why do we distinguish between primary and secondary scenarios?
  - + What would you think would be the main advantages and disadvantages of use cases?
  - + How would you combine use-cases to calculate the risky path in a project plan?
  - + Do use-cases work well with agile methods? Explain why or why not.
  - + Can you explain the use of a product roadmap in scrum?
  - + Choose the three most important items in your “Definition of Ready” checklist. Why are these most important to you?
  - + Can you relate scrum user stories to some of the principles in the Agile Manifesto?
  - + How would you turn an FMEA analysis into a misuse case diagram?
  - + Elaborate on the relationship between an FMEA analysis and the variants of safety stories.

3. Architecture (1/2)

You should know the answers to these questions

- What’s the role of a software architecture?
- What is a component? And what’s a connector?
- What is coupling? What is cohesion? What should a good design do with them?
- What is a pattern? Why is it useful for describing architecture?
- Can you name the components in a 3-tiered architecture? And what about the connectors?
- Why is a repository better suited for a compiler than pipes and filters?
- What’s the motivation to introduce an abstract factory?
- Can you give two reasons not to introduce an Adapter (Wrapper)?
- What problem does an abstract factory solve?
- List three tradeoffs for the Adapter pattern.
- How do you decide on two architectural alternatives in scrum?
- What’s the distinction between a package diagram and a deployment diagram?
- Define a sensitivity point and a tradeoff point from the ATAM terminology.

You should be able to complete the following tasks

- Take each of the patterns and identify the components and connectors. Then assess the pattern in terms of coupling and cohesion. Compare this assessment with the tradeoffs.

Architecture (2/2)

Can you answer the following questions?

- What do architects mean when they say “architecture maps function onto form”? And what would the inverse “map form into function” mean?
- How does building architecture relate to software architecture? What’s the impact on the corresponding production processes?
- Why are pipes and filters often applied in CGI-scripts?
- Why do views and controllers always act in pairs?
- Explain the sentence “Restricts communication between subject and observer” in the Observer pattern
- Can you explain the difference between an architecture and a pattern?
- Explain the key steps of the ATAM method?
- *How can you balance emergent design with intentional architecture?*
- *What happens when your team goes outside the boundaries of the guardrail?*
- How would you organize an architecture assessment in your team?

4.Project Management (1/2)

- You should know the answers to these questions
  - + Name the five activities covered by project management.
  - + What is a milestone? What can you use them for?
  - + What is a critical path? Why is it important to know the critical path?
  - + What can you do to recover from delays on the critical path?
  - + How can you use Gantt-charts to optimize the allocation of resources to a project?
  - + What is a "Known kown", and "Unknown known" and an "Unknown Unknown"?
  - + How do you use PERT to calculate the risk of delays to a project?
  - + Why does replacing a person imply a negative productivity?
  - + What's the difference between the 0/100; the 50/50 and the milestone technique for calculating the earned value?
  - + Why shouldn't managers take on tasks in the critical path?
  - + What is the "definition of done" in a Scrum project?
  - + Give a definition for a Squad, Tribe, Chapter and Guild in the spotify scrum model.

- You should be able to complete the following tasks
  - + draw a PERT Chart, incl. calculating the critical path and the risk of delays
  - + draw a Gant chart, incl. allocating and optimizing of resources
  - + draw a slip line and a timeline

Project Management (2/2)

- Can you answer the following questions?
  - + Name the various activities covered by project management. Which ones do you consider most important? Why?
  - + How can you ensure traceability between the plan and the requirements/system?
  - + Compare PERT-charts with Gantt charts for project planning and monitoring.
  - + How can you deal with "Unknown Unknowns" during project planning?
  - + Choose between managing a project that is expected to deliver soon but with a large risk for delays, or managing a project with the same result delivered late but with almost no risk for delays. Can you argue your choice?
  - + Describe how earned-value analysis can help you for project monitoring.
  - + Would you consider bending slip lines as a good sign or a bad sign? Why?
  - + You're a project leader and one of your best team members announces that she is pregnant. You're going to your boss, asking for a replacement and for an extension of the project deadline. How would you argue the latter request?
  - + You have to manage a project team of 5 persons for building a C++ compiler. Which team structure and member roles would you choose? Why?
  - + Can you give some advantages and disadvantages of scrum component teams and scrum feature teams.

5. Design by Contract (1/2)

- You should know the answers to these questions
  - + What is the distinction between Testing and Design by Contract? Why are they complementary techniques?
  - + What’s the weakest possible condition in logic terms? And the strongest?
  - + If you have to implement an operation on a class, would you prefer weak or strong conditions for pre- and postcondition? And what about the class invariant?
  - + If a subclass overrides an operation, what is it allowed to do with the pre- and postcondition? And what about the class invariant?
  - + Compare Testing and Design by contract using the criteria “Correctness” and “Traceability”.
  - + What’s the Liskov substitution principle? Why is it important in OO development?
  - + What is behavioral subtyping?
  - + When is a pre-condition reasonable?

- You should be able to complete the following tasks
  - + What would be the pre- and post-conditions for the methods top and isEmpty in the Stack specification? How would I extend the contract if I added a method size to the Stack interface?
  - + Apply design by contract on a class Rectangle, with operations move() and resize().
  - + Write consumer-driven contracts for a given REST-API .

Design by Contract (2/2)

- Can you answer the following questions?
  - + Why are redundant checks not a good way to support Design by Contract?
  - + You’re a project manager for a weather forecasting system, where performance is a real issue. Set-up some guidelines concerning assertion monitoring and argue your choice.
  - + If you have to buy a class from an outsourcer in India, would you prefer a strong precondition over a weak one? And what about the postcondition?
  - + Do you feel that design by contract yields software systems that are defect free? If you do, argue why. If you don’t, argue why it is still useful.
  - + How can you ensure the quality of the pre- and postconditions?
  - + Why is (consumer-driven) contract testing so relevant in the context of micro-services?
  - + Assume you have an existing software system and you are a software quality engineer assigned to apply design by contract. How would you start? What would you do?

6.Testing (1/2)

You should know the answers to these questions

- What is (a) Testing, (b) a Testing Technique, (c) a Testing Strategy
- What is the difference between an error, a failure and a defect?
- What is a test case? A test stub? A test driver? A test fixture?
- What are the differences and similarities between basis path testing, condition testing and loop testing?
- How many tests should you write to achieve MC/DC coverage? And multiple condition coverage?
- Where do you situate alpha/beta testing in the four quadrants model?
- What are the differences and similarities between unit testing and regression testing?
- How do you know when you tested enough?
- What is Alpha-testing and Beta-Testing? When is it used?
- What is the difference between stress-testing and performance testing?

You should be able to complete the following tasks

- Complete test cases for the Loop Testing example (Loop Testing on page 19).
- Rewrite the binary search so that basis path testing and loop testing becomes easier.
- Write a piece of code implementing a quicksort. Apply all testing techniques (basis path testing, conditional testing [3 variants], loop testing, equivalence partitioning) to derive appropriate test cases.

- Write FIT test cases for the user stories in you Bachelor Capstone Project
- *Apply fuzz testing to the REST-API of your project*

CAPSTONE PROJECT

Testing (2/2)

Can you answer the following questions?

- You’re responsible for setting up a test program. To whom will you assign the responsibility to write tests? Why?
- Why do we distinguish between several levels of testing in the V-model?
- Explain why basis path testing, condition testing and loop testing complement each other.
- Why is mutation coverage a better criterion for assessing the strength of a test suite?
- *Explain fuzzing (fuzz testing) in your own words.*
- Explain what FIT tables are.
- When would you combine top-down testing with bottom-up testing? Why?
- When would you combine black-box testing with white-box testing? Why?
- Is it worthwhile to apply white-box testing in an OO context?
- What makes regression testing important?
- Is it acceptable to deliver a system that is not 100% reliable? Why (not)?
- Explain the subtle difference between code coverage and test coverage.

7.Formal Specifications (1/2)

You should know the answers to these questions

- Why is an UML class diagram a semi-formal specification?
- What is an automated theorem prover?
- What is the distinction between “partially correct” and “totally correct”?
- Give the mathematical definition for the *weakest precondition* of Hoare triple {P} S {Q}
- Why is it necessary to complement sequence diagrams with statecharts?
- What is the notation for the start and termination state on a state-chart? What is the notation for a guard expression on an event?
- What does it mean for a statechart to be
  (a) consistent, (b) complete, and (c) unambiguous?
- How does a formal specification contribute to the correctness of a given system?

You should be able to complete the following tasks

- Use a theorem prover (Daphny) to prove that a given piece of code is correct.
- Create a statechart specification for a given problem.
- Given a statechart specification, derive a test model using path testing.

Formal Specifications (2/2)

Can you answer the following questions?

- (Based on the article “A Formal Approach to Constructing Secure Air Vehicle Software”.)
  + What is according to you the most effective means to achieve “provably secure against cyberattacks”?
- Why is it likely that you will encounter formal specifications?
- Explain why we need both the loop variant and the loop invariant for proving total correctness of a loop?
- What do you think happened with the bug report on the broken Java.utils.Collection.sort ()? *Why* do you think this happened?
- Explain the relationship between “Design By Contract” on the one hand “State based specifications” on the other hand.
- Explain the relationship between “Testing” on the one hand and “State based specifications” on the other hand.
- You are part of a team build a fleet management system for drones transporting medical goods between hospitals. You must secure the system against cyber-attacks. Your boss asks you to look into formal specs; which ones would you advise and why?

8. Domain Modelling (1/2)

- You should know the answers to these questions
  - + Why is it necessary to validate and analyze the requirements?
  - + What’s the decomposition principle for functional and object-oriented decomposition?
  - + Can you give the advantages and disadvantages for functional decomposition? What about object-oriented decomposition?
  - + How can you recognize “god classes”?
  - + What is a responsibility? What is a collaboration?
  - + Name 3 techniques to identify responsibilities.

  - + What do feature models define?
  - + Give two advantages and disadvantages of a “clone and own” approach
  - + Explain the main difference between a social fork and a variant fork

  - + How does domain modeling help to achieve correctness? Traceability?
- You should be able to complete the following tasks
  - + Apply noun identification & verb identification to (a part of) a requirements specification.
  - + Create a feature model for a series of mobile phones.

Domain Modelling (2/2)

- Can you answer the following questions?
  - + How does domain modeling help to validate and analyze the requirements?
  - + What’s the problem with “god classes”?
  - + Why are many responsibilities, many collaborators and deep inheritance hierarchies suspicious?
  - + Can you explain how role-playing works? Do you think it helps in creative thinking?
  - + Can you compare Use Cases and CRC Cards in terms of the requirements specification process?
  - + Do CRC cards yield the best possible class design? Why not?
  - + Why are CRC cards maintained with paper and pencil instead of electronically?
  - + What would be the main benefits for thinking in terms of “system families” instead of “one-of-a-kind development? What would be the main disadvantages?
  - + Can you apply scrum to develop a product line? Argue your case.

9. Software Quality (1/2)

You should know the answers to these questions

- Why is software quality more important than it was a decade ago?
- Can a correctly functioning piece of software still have poor quality? Why?
- If quality control can’t guarantee results, why do we bother?
- What’s the difference between an external and an internal quality attribute? And between a product and a process attribute?
- What’s the distinction between correctness, reliability and robustness?
- How can you express the “user friendliness” of a system?
- Can you name three distinct refinements of “maintainability”? What do each of these names mean?
- What is meant with “short time to market”? Can you name 3 related quality attributes and provide definitions for each of them?
- Name four things which should be recorded in the review minutes.
- Explain briefly the three items that should be included in a quality plan.
- What’s the relationship between ISO9001, CMMI standards and an organization’s quality system? How do you get certified?
- Can you name and define the 5 levels of CMMI?
- Where would “use-cases” as defined in chapter 3 fit in the table of core process areas (p. 32)? Motivate your answer shortly.

Software Quality (2/2)

You should be able to complete the following tasks

- Given a piece of code and a coding standard, review the code to verify whether the standard has been adhered to.

Can you answer the following questions?

- Given the Quality Attributes Overview table, argue why the crosses and blanks occur at the given positions.
- Why do quality standards focus on process and internal attributes instead of the desired external product attributes?
- Why do you need a quality plan? Which topics should be covered in such a plan?
- How should you organize and run a review meeting?
- Why are coding standards important?
- What would you include in a documentation review checklist?
- How often should reviews by scheduled?
- Could you create a review check-list for ATAM?
- Would you trust software from an ISO 9000 certified company? And if it were CMMI?
- You are supposed to develop a quality system for your organization. What would you include?
- Where would “testing” fit in the table of core process areas (p. 32). Does it cover a single row or not? Argue why (not)?

10.Software Metrics (1/2)

You should know the answers to these questions

- Can you give three possible problems of metrics usage in software engineering? How does the measurement theory address them?
- What’s the distinction between a measure and a metric?
- Can you give an example of a direct and an indirect measure?
- What kind of measurement scale would you need to say “A specification error is worse than a design error”? And what if we want to say “A specification error is twice as bad as a design error?”
- Explain the need for a calibration factor in Putnam’s model.
- Fill in the blanks in the following sentence. Explain briefly, based on the Putnam’s model.
  + If you want to finish earlier (= decrease scheduled time), you should ... the effort ... .
- Give three metrics for measuring size of a software product.
- Discuss the main advantages and disadvantages of Function Points.
- What does it mean for a coupling metric not to satisfy the representation condition?
- Can you give 3 examples of impreciseness in Lines of Code measurements?

You should be able to complete the following tasks

- Given a set of use cases (i.e. your project) calculate the use case points.
- Given a set of user stories, perform a poker planning session.

Software Metrics (2/2)

Can you answer the following questions?

- During which phases in a software project would you use metrics?
- Why is it so important to have “good” product size metrics?
- Can you explain the two levels of calibration in COCOMO (i.e. C & S vs. M)? How can you derive actual values for these parameters?
- Can you motivate why in software engineering, productivity depends on the scheduled time? Do you have an explanation for it?
- Can you explain the cone of uncertainty? And why is it so relevant to cost estimation in software projects?
- How can you decrease the uncertainty of a project bid using Putnam’s model?
- Why do we prefer measuring Internal Product Attributes instead of External Product Attributes during Quality Control? What is the main disadvantage of doing that?
- You are a project manager and you want to convince your project team to apply algorithmic cost modeling. How would you explain the technique?
- Where would you fit coupling/cohesion metrics in a hierarchical quality model like ISO 9126?
- Why are coupling/cohesion metrics important? Why then are they so rarely used?
- Do you believe that “defect density” says something about the correctness of a program? Motivate your answer?

11.Refactoring (1/2)

You should know the answers to these questions:

- Can you explain how refactoring differs from plain coding?
- Can you tell the difference between Corrective, Adaptive and Perfective maintenance? And how about preventive maintenance?
- Can you name the three phases of the iterative development life-cycle? Which of the three does refactoring support the best? Why do you say so?
- Can you give 4 symptoms for code that can be "cured" via refactoring?
- Can you explain why add class/add method/add attribute are behaviour preserving?
- Can you give the pre-conditions for a "rename method" refactoring?
- Which 4 activities should be supported by tools when refactoring?
- Why can't we apply a "push up" to a method "x()" which accesses an attribute in the class the method is defined upon (see Refactoring Sequence on page 27–31)?

You should be able to complete the following tasks

- Two classes A & B have a common parent class X. Class A defines a method a() and class B a method b() and there is a large portion of duplicated code between the two methods. Give a sequence of refactorings that moves the duplicated code in a separate method x() defined on the common superclass X.
- What would you do in the above situation if the duplicated code in the methods a() and b() are the same except for the name and type of a third object which they delegate responsibilities too?

- Monitor the technical debt of you bachelor capstone project.

CAPSTONE PROJECT

Refactoring (2/2)

Can you answer the following questions?

- Why would you use refactoring in combination with Design by Contract and Regression Testing?
- Can you give an example of a sequence of refactorings that would improve a piece of code with deeply nested conditionals?
- How would you refactor a large method? And a large class?
- Consider an inheritance relationship between a superclass "Square" and a subclass "Rectangle". How would you refactor these classes to end up with a true "is-a" relationship? Can you generalise this procedure to any abusive inheritance relationship?

12.Conclusion (1/2)

- You should know the answers to these questions
  - + Name 3 items from the code of ethics and provide a one-line explanation.
  - + If you are an independent consultant, how can you ensure that you will not have to act against the code of ethics?
  - + What would be a possible metric for measuring the amount of innovation of a manufacturing company?
  - + Explain the 2 main steps of test amplification: input amplification and assertion amplification

When you chose the “No Silver Bullet” paper

- What’s the distinction between essential and accidental complexity?
- Name 3 reasons why the building of software is essentially a hard task? Provide a one-line explanation.
- Why is “object-oriented programming” no silver bullet?
- Why is “program verification” no silver bullet?
- Why are “components” a potential silver bullet?

When you chose the “Killer Robot” paper

- Which regression tests would you have written to prevent the “killer robot”?
- Was code reviewing applied as part of the QA process? Why (not)?
- Why was the waterfall process disastrous in this particular case?
- Why was the user-interface design flawed?

Conclusion (2/2)

- Can you answer the following questions?
  - + You are an experienced designer and you heard that the sales people earn more money than you do. You want to ask your boss for a salary-increase; how would you argue your case?
  - + Software products are usually released with a disclaimer like “Company X is not responsible for errors resulting from the use of this program”. Does this mean that you shouldn’t test your software? Motivate your answer.
  - + Your are a QA manager and are requested to produce a monthly report about the quality of the test process. How would you do that?
  - + Why is “explainable Artificial Intelligence” so important when creating bots for software engineering tasks?

When you chose the “No Silver Bullet” paper

- + Explain why incremental development is a promising attack on conceptual essence. Give examples from the different topics addressed in the course.
- + “Software components” are said to be a promising attack on conceptual essence. Which techniques in the course are applicable? Which techniques aren’t?

When you chose the “Killer Robot” paper

- + Recount the story of the Killer Robot case. List the three most important causes for the failure and argue why you think these are the most important.