INF2001/ICT2201

Introduction to Software Engineering

Dr. Alex Q. Chen

Alex.Q.Chen@SingaporeTech.edu.sg

@AlexQChen

TODAY

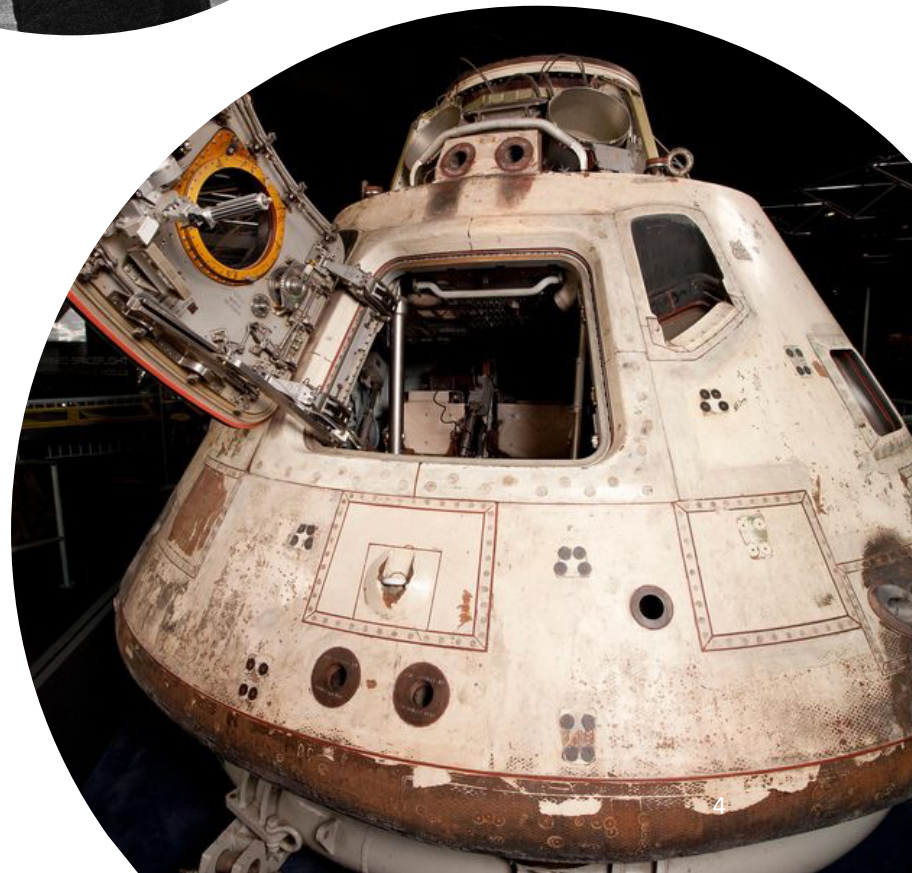
- Motivation for Software Engineering?
- What is Software Engineering?

Motivation for Software Engineering

In 1968…

- Reveal of Boeing 747

- Apollo 8 - 1st manned spacecraft (soon followed by Apollo 11)



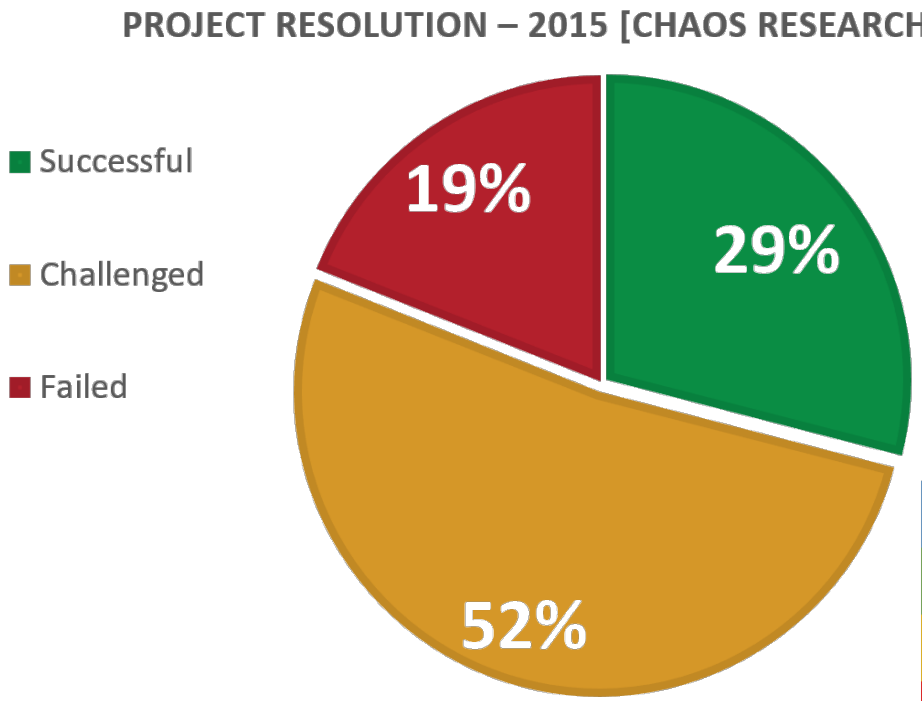
In 1968: Raising Demand for Software

- ↑ demand for software
- ↑ complexity for software
- ↓ productive for programmers

- Search for “Million Line of code” to see reported size of applications.

IN 1968 THE MAJOR COMPLAINTS WERE:

Software Engineering: The Reality In Numbers



	2004	2006	2008	2010
Successful	29%	35%	32%	37%
Challenged	53%	46%	44%	42%
Failed	18%	19%	24%	21%

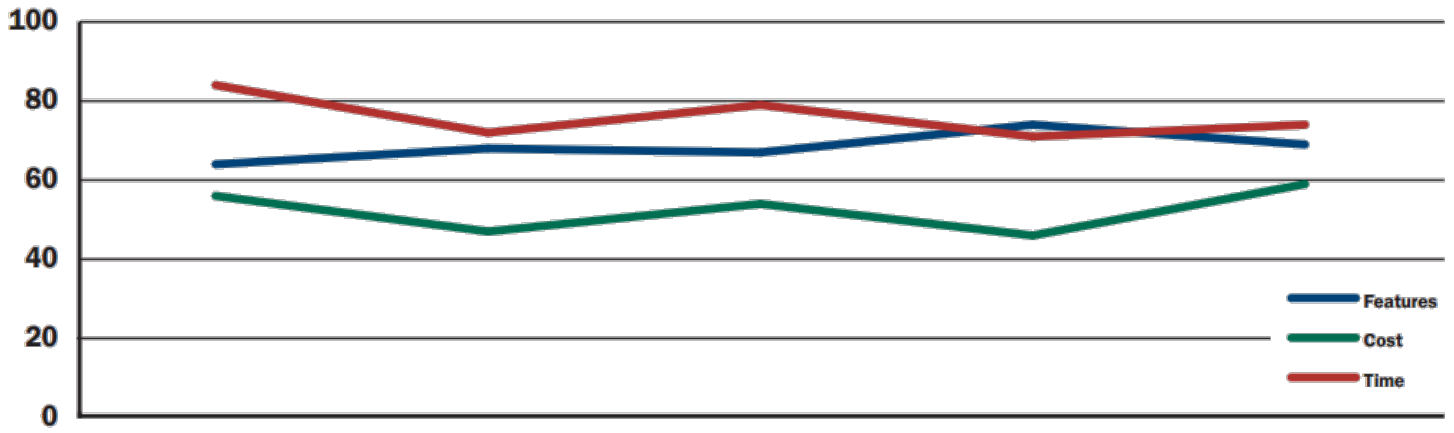
	2011	2012	2013	2014	2015
Successful	29%	27%	31%	28%	29%
Challenged	49%	56%	50%	55%	52%
Failed	22%	17%	19%	17%	19%

Chart Area

Software Engineering: The Reality In Numbers

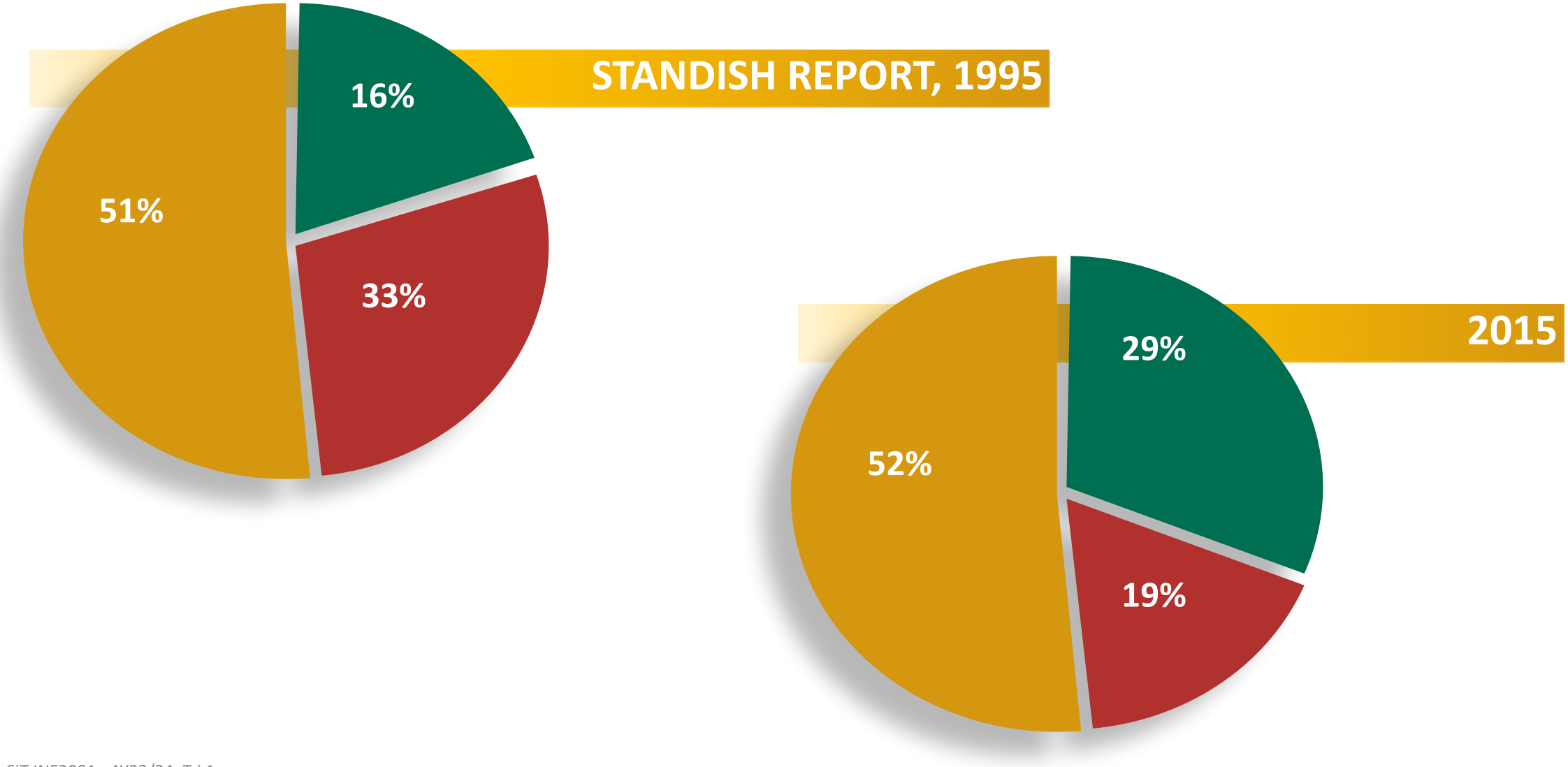
OVERRUNS AND FEATURES

Time and cost overruns, plus percentage of features delivered from CHAOS research for the years 2004 to 2012.



	2004	2006	2008	2010	2012
TIME	84%	72%	79%	71%	74%
COST	56%	47%	54%	46%	59%
FEATURES	64%	68%	67%	74%	69%

We Are Getting Better…



CHAOS Resolution by Project Size

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
Total	**100%**	**100%**	**100%**

The resolution of all software project by size from FY2011-FY2015 with in the new CHAOS database

Cutter Consortium Data

- 2002 survey of information technology organisations
  - **78%** have been involved in disputes **ending in litigation**

- For the organisations that entered into litigation:
  - In 67% of the disputes, the **functionality** of the information system as delivered **did not meet up** to the claims of the developers.
  - In 56% of the disputes, the promised **delivery date** slipped several times.
  - In 45% of the disputes, the **defects** were so **severe** that the information system was **unusable.**

The software crisis from 1968 is still relevant and has not been solved yet…

What is Software Engineering?

Defining “SOFTWARE ENGINEERING”

Create account Log in

Article Talk Read Edit View history Search

Software

Computer software, or simply **software** is any set of machine-readable instructions that directs a computer's processor to perform specific operations. Computer software contrasts with computer hardware, which is the physical component of computers. Computer hardware and software require each other and neither can be realistically used without the other. Using a musical analogy, hardware is like a musical instrument and software is like the notes played on that instrument.

Computer software includes computer programs, libraries and their associated documentation. The word software is also sometimes used in a more narrow sense, meaning application software only. Software is stored in computer memory and is intangible, i.e. it cannot be touched.[1]

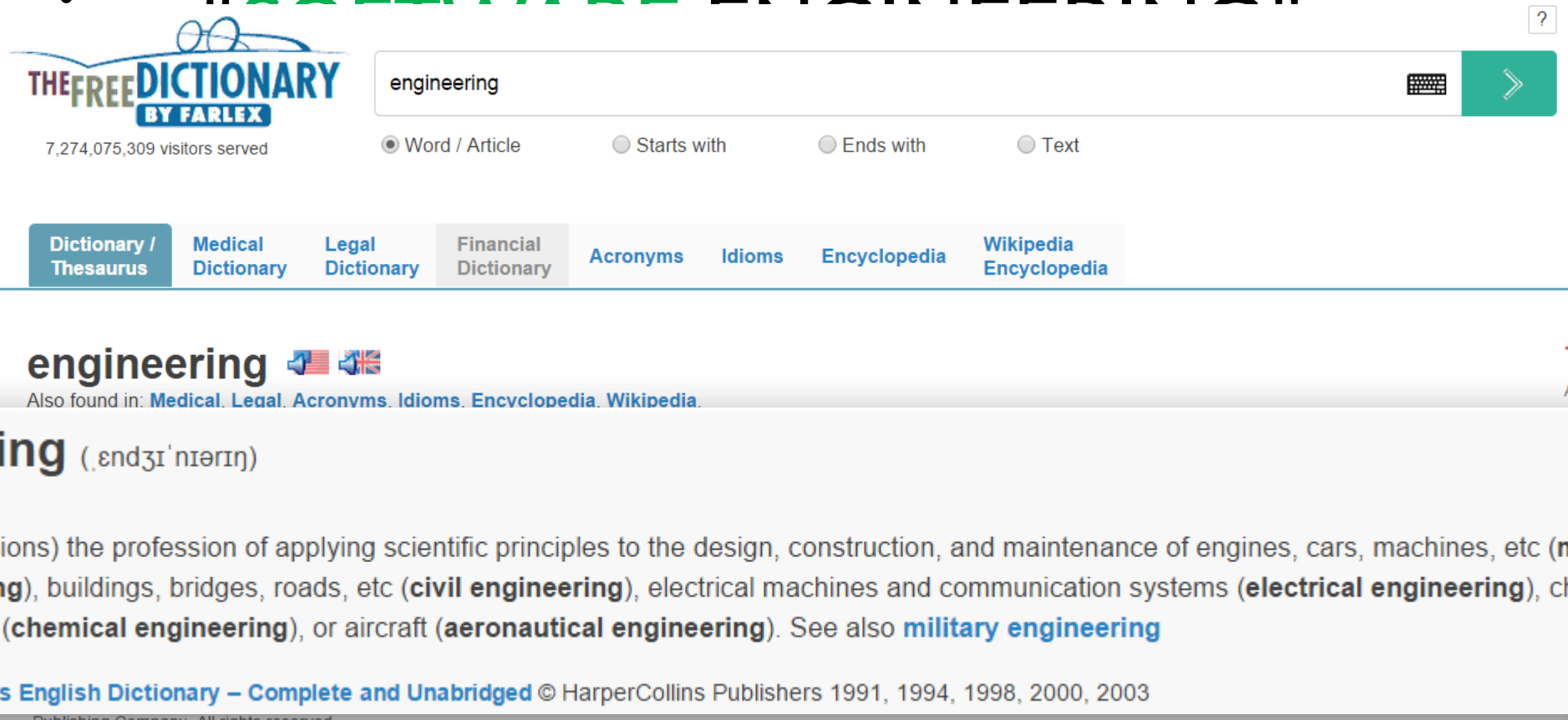
software require each other and neither can be realistically used without the other. Using a musical analogy, hardware is like a musical instrument and software is like the notes played on that instrument.

Computer software includes computer programs, libraries and their associated documentation. The word software is also sometimes used in a more narrow sense, meaning application software only. Software is stored in computer memory and is intangible, i.e. it cannot be touched.[1]

About Wikipedia
Community portal
Recent changes
Contact page

Tools

Defi … "SOFTWARE ENGINEERING"



Defining “SOFTWARE ENGINEERING”

“The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software.”

IEEE
Advancing Technology for Humanity

Defining “SOFTWARE ENGINEERING”

“The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software.”

systematic

Also found in: **Medical**, **Legal**, **Financial**, **Encyclopedia**, **Wikipedia**.

sys·tem·at·ic (sĭs'tə-măt'ĭk) also **sys·tem·at·i·cal** (-ĭ-kəl)

adj.

1. Characterized by, based on, or constituting a system: *systematic thought.*
2. Working or done in a step-by-step manner; methodical: *a systematic worker; a systematic approach.*

IEEE
Advancing Technology for Humanity

Defining “SOFTWARE ENGINEERING”

“The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software.”

discipline

Also found in: Medical, Legal, Acronyms, Idioms, Encyclopedia, Wikipedia.

dis·ci·pline (dĭs'ə-plĭn)

n.

1. Training expected to produce a specific character or pattern of behavior, especially training that produces moral or mental improvement: *was raised in the strictest discipline.*
2.
   a. Control obtained by enforcing compliance or order: *military discipline.*
   b. Controlled behavior resulting from disciplinary training; self-control: *Dieting takes a lot of discipline.*
   c. A state of order based on submission to rules and authority: *a teacher who demanded discipline in the classroom.*
3. Punishment intended to correct or train: *subjected to harsh discipline.*
4. A set of rules or methods, as those regulating the practice of a church or monastic order.
5. A branch of knowledge or teaching: *the discipline of mathematics.*

IEEE
Advancing Technology for Humanity

Defining “SOFTWARE ENGINEERING”

“The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software.”

quan·ti·fy (kwŏn'tə-fī')

tr.v. **quan·ti·fied**, **quan·ti·fy·ing**, **quan·ti·fies**

1. To determine or express the quantity of.
2. *Logic* To limit the variables of (a proposition) by prefixing an operator such as *all* or *some*.

[Medieval Latin `quantificāre` : Latin `quantus`, *how great*; see **quantity** + Latin `-ficāre`, *-fy*.]

quan'ti·fi'a·ble *adj.*

quan'ti·fi·ca'tion (-fī-kā'shən) *n.*

IEEE
Advancing Technology for Humanity

Defining “SOFTWARE ENGINEERING”

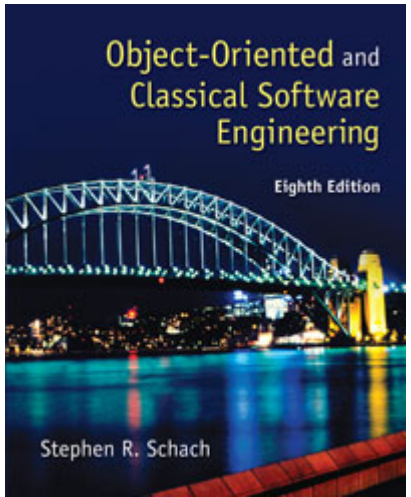
“The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software.”

It’s not just coding!

IEEE
Advancing Technology for Humanity

Defining “SOFTWARE ENGINEERING”

“Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user’s needs***”



Defining “SOFTWARE ENGINEERING”

- Programming in the large
  - many people, more complex systems
- Disciplined development
- Repeatable practices
- Full life-cycle of activities

“Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user’s needs***”

- A dual emphasis:
  - Product: what is produced
  - Process: how the product is produced

Fault-free Software

- What are Faults?
  - software behaviour unaccounted for in its design
- Examples
  - Patriot missile timing error
  - Therac-25 medical linear accelerator incident
  - The pervasive Y2K bug
    - ❖ Global economic impact
- How can we lessen our chances of faults?
  - use a well-defined process that includes rigorous design, testing, and programming techniques
  - ICT2106, ICT3101…

image from: http://www.financialtraining.ca/wp-content/uploads/2012/01/Blame.jpg January 2015

Why Is It So Hard To Develop Software That Is…

- On time,
- Within budget
- Satisfies the user’s needs (functionalities and usability)

Time And Budget Case Studies

California DMV software (1987-1993)

- attempt to merge driver & vehicle registration systems
- Plan: $28 million over 5 years.
- spent 7 years and $50 million …

before **pulling the plug**

❖Estimated cost of over $200 million and at least 5 years delay

"Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user's needs***"

Round 2:

California DMV software (2006-2013)

- ➢ Project stopped after spending $134 million due to "significant concerns with the lack of progress"
- ➢ 3 months before the original expected delivery date
- ➢ Some of the functions were delivered

Data from: IEEE SPECTRUM, 2013 Los Angeles Times, 2013

Time And Budget Case Studies

Digital Media Initiative (DMI) by BBC, UK

- To modernise the Corporation's production and archiving methods by using connected digital production and media asset management systems.
- Plan: ~£81.7 million
- Actual cost: >£98 million (>SGD200 million)
- 2008 - 2013
- Project scrapped due to bad management and outpaced by changing technology.
- By 2013, much cheaper Commercial Off The Shelf (COTS) alternatives by then existed.

"Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user's needs***"

Data from: BBC abandons £100m digital project

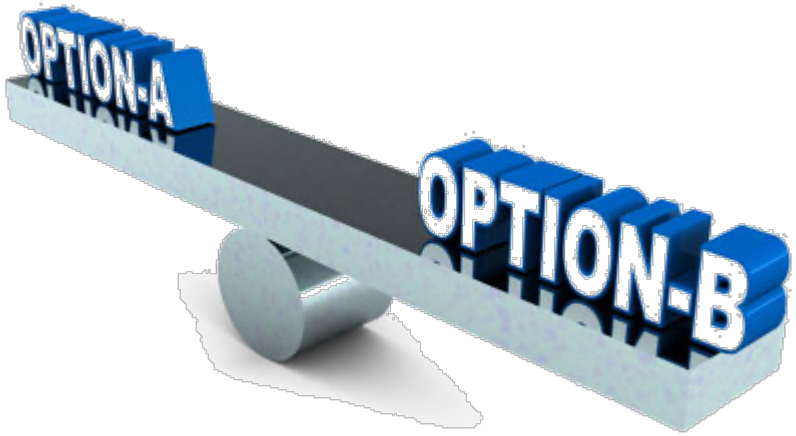
Economic Aspects: Example

- Coding Method CM_{new} is 10% faster than currently used Coding Method CM_{old}.

  Should it be used?

- Common sense answer:

  Of course!



- Software Engineering answer:
  - Consider the cost of training
  - Consider the impact of introducing a new technology
  - Consider the effect of CM_{new} on maintenance

Satisfies The User's Needs

- Functionality and Usability
  - does what the user's tasks require (ICT2108, ICT3101)
  - efficient to use & error rates kept to a minimum (ICT2102, ICT3102)
  - easy to learn
  - leads to high user satisfaction
- Bad (even if correct) user interfaces cost
  - money (5%↑ satisfaction ⇒ up to 85%↑ profits)
  - lives (Therac-25)
- User interfaces hard to get right
  - people are unpredictable
  - ICT2102 is all about how to get UIs right

"Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user's needs***"

Software Life Cycle

- A life cycle: the actual steps performed when building a product

- A life cycle **model**:
  - The steps to follow when building software
  - A theoretical description of what should be done

- Having a defined process (model) is essential
- Maturity of the process is some gauge of success of organization

What are the Activities Involved in Building Software?

Software Development Activities

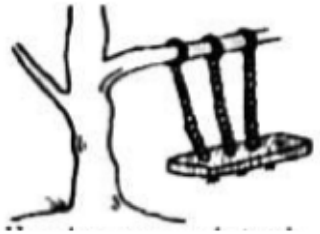
- Understand User requirements(what the client needs)
- Analysis and design software architecture(high level and detailed)
- Implement and document the design
- Testing(verify and validate)
- Deploy the product
- Fix bugs after deployment
- Add features after deployment
- Planning and Team Management

Among the Activities…

Which Activity is Known to be the Most Problematic?

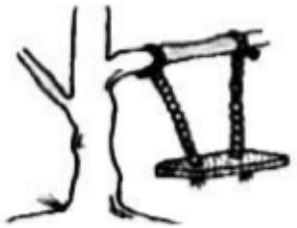
What the user asked for during meetings

Analysis



Design

Implementation



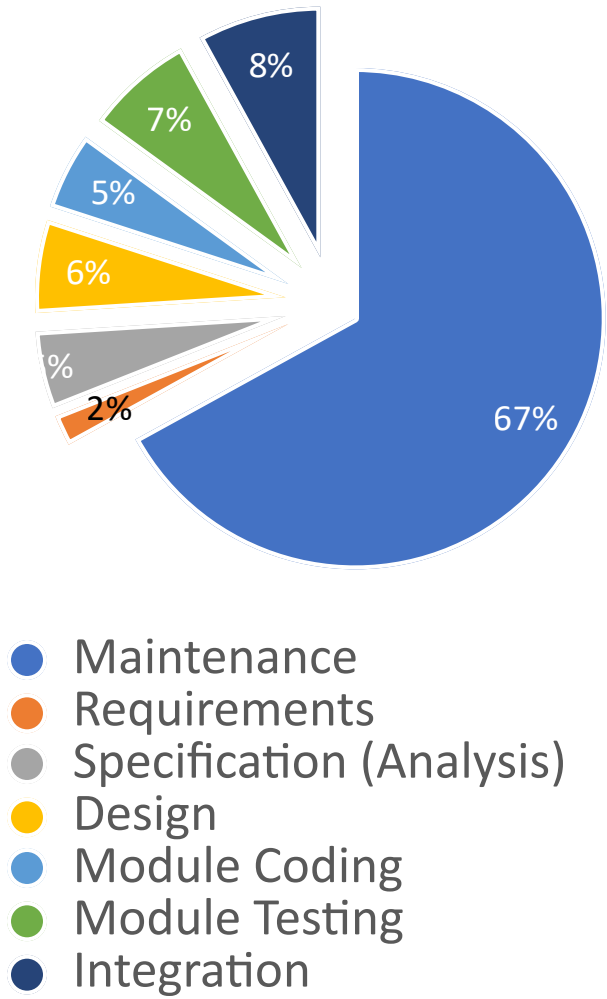
What the user really wanted

Which Phase Costs the Most?

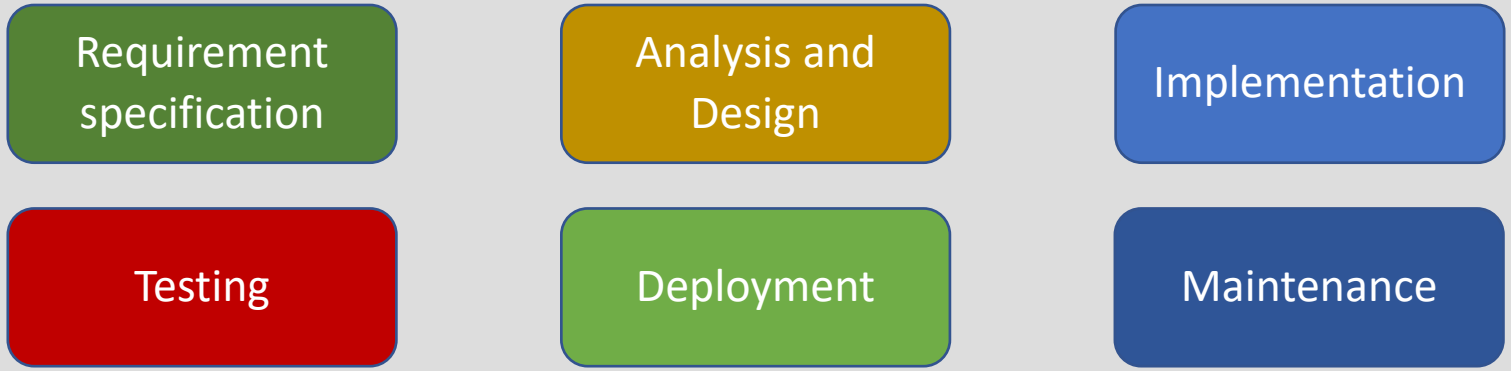
Phase Cost Approximation

- 1976–1981 data
  Maintenance constitutes 67% of total cost
- Good software is maintained
  (for 10, 20 years or more)
- Bad software is discarded
- We should design our software with
  maintenance in mind

We need techniques, tools and practices to reduce maintenance costs!



What is the best way to arrange the phases in a workflow?



- Which phase should come first?
- Should we work in parallel or sequential?
- Can we go through a phase more than once?
- Would your choice be affected by the type, size, and context of the project and hosting organization?

Typical Classical Phases

- Requirements phase (ICT2108)
  - Explore the concept
  - Elicit the client's requirements

- Analysis (specification) phase (ICT2108)
  - Analyze the client's requirements
  - Draw up the **specification document**
  - Draw up the software project **management plan**
  - "What the product is supposed to do"

Typical Classical Phases

- Design phase (ICT2106 and others)
  - Architectural design, followed by
  - Detailed design
  - “How the product does it”

- Implementation phase (ICT1002, ICT1009, ICT2105 and many others)
  - Coding
  - Unit testing
  - Integration
  - Acceptance testing

Typical Classical Phases

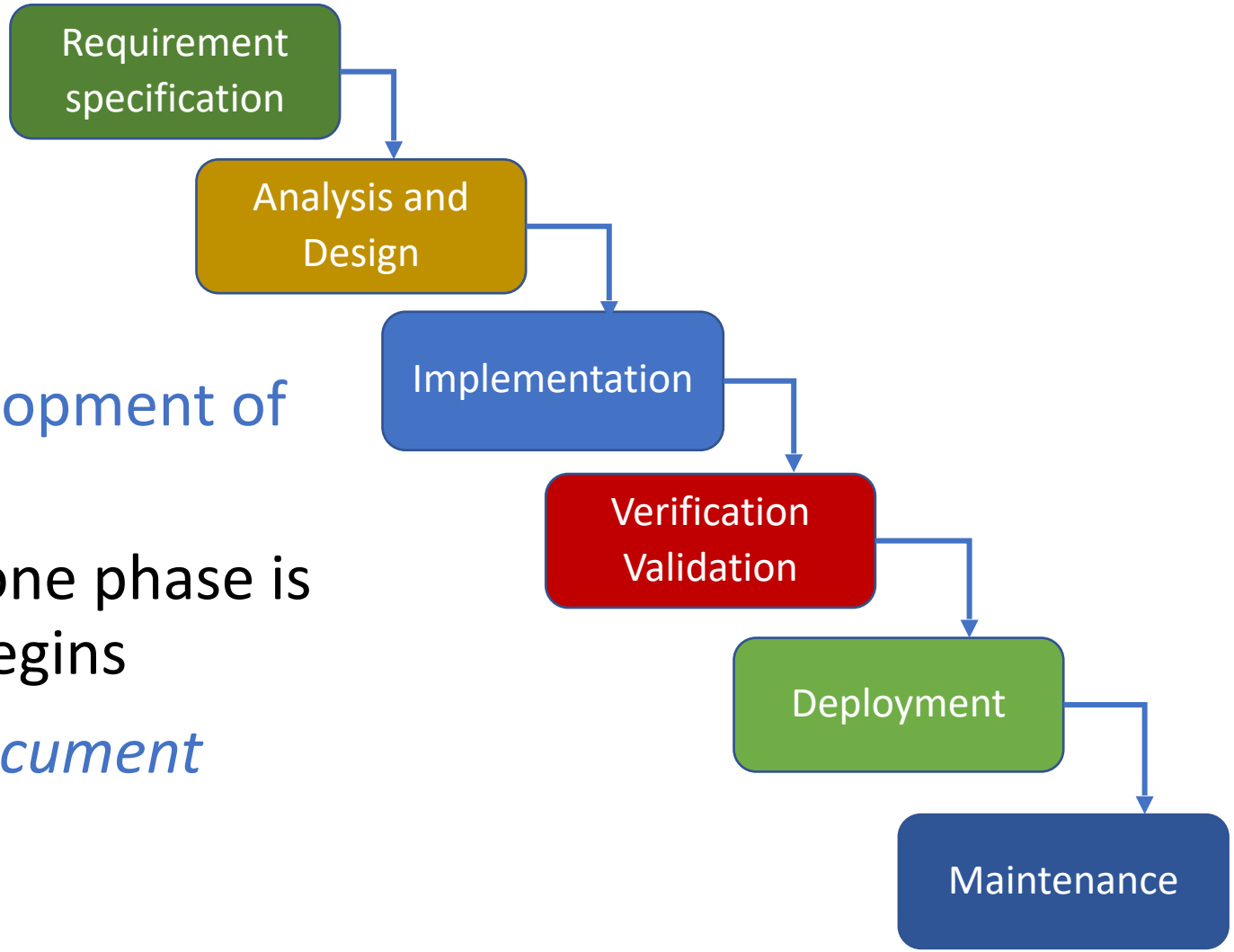
- Postdelivery maintenance (ICT3104 and others)
  - Corrective maintenance
  - Perfective maintenance
  - Adaptive maintenance
- Retirement

Waterfall Life-cycle Model

Classical model (1970)

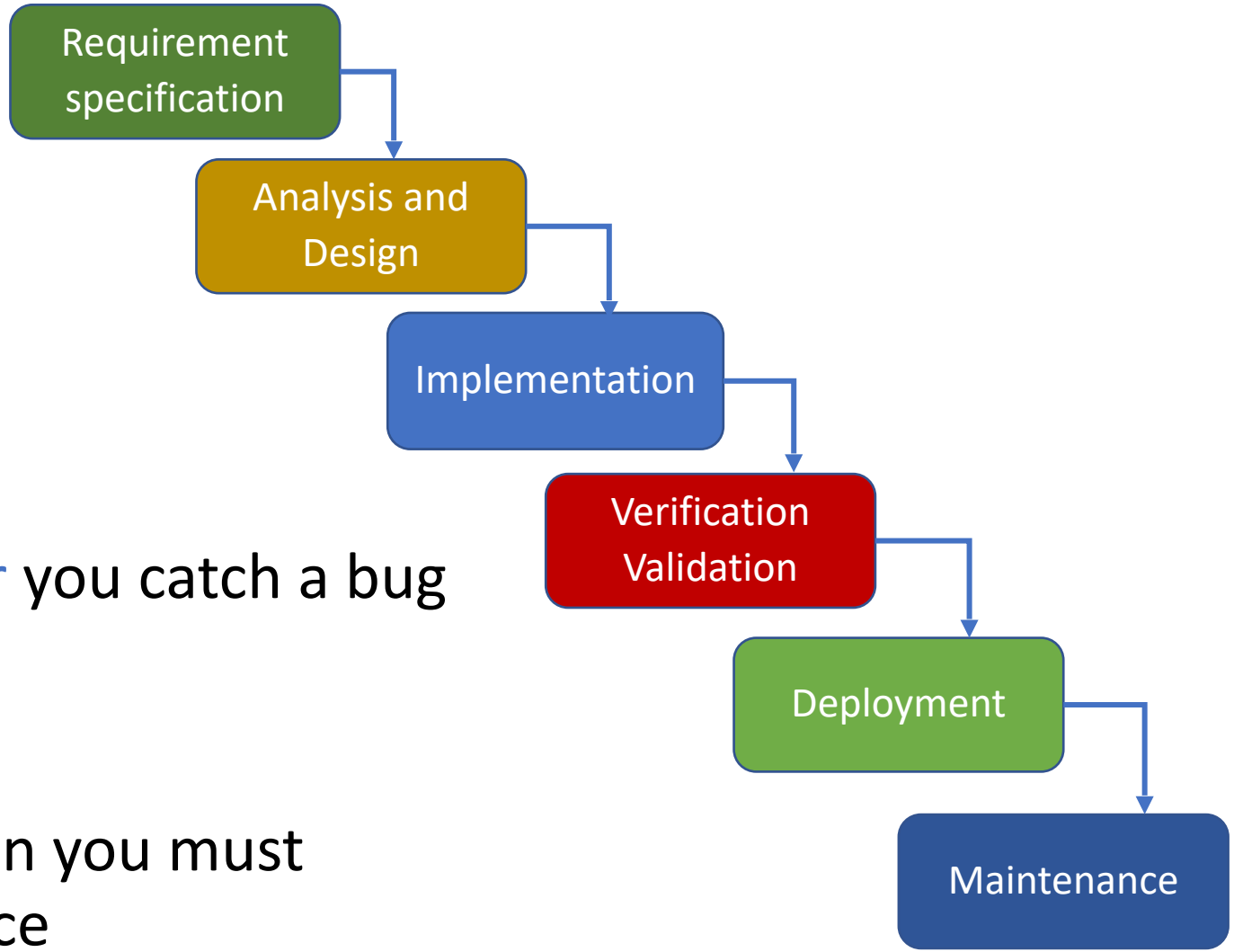
1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance
6. Retirement

The Waterfall Process - 1970



- W. Royce "Managing the Development of Large Software Systems"
- Sequential & Distinct phases: one phase is completed and the next one begins
- Document-driven *(Plan and document approach)*
- Big Design Up Front (BDUF)

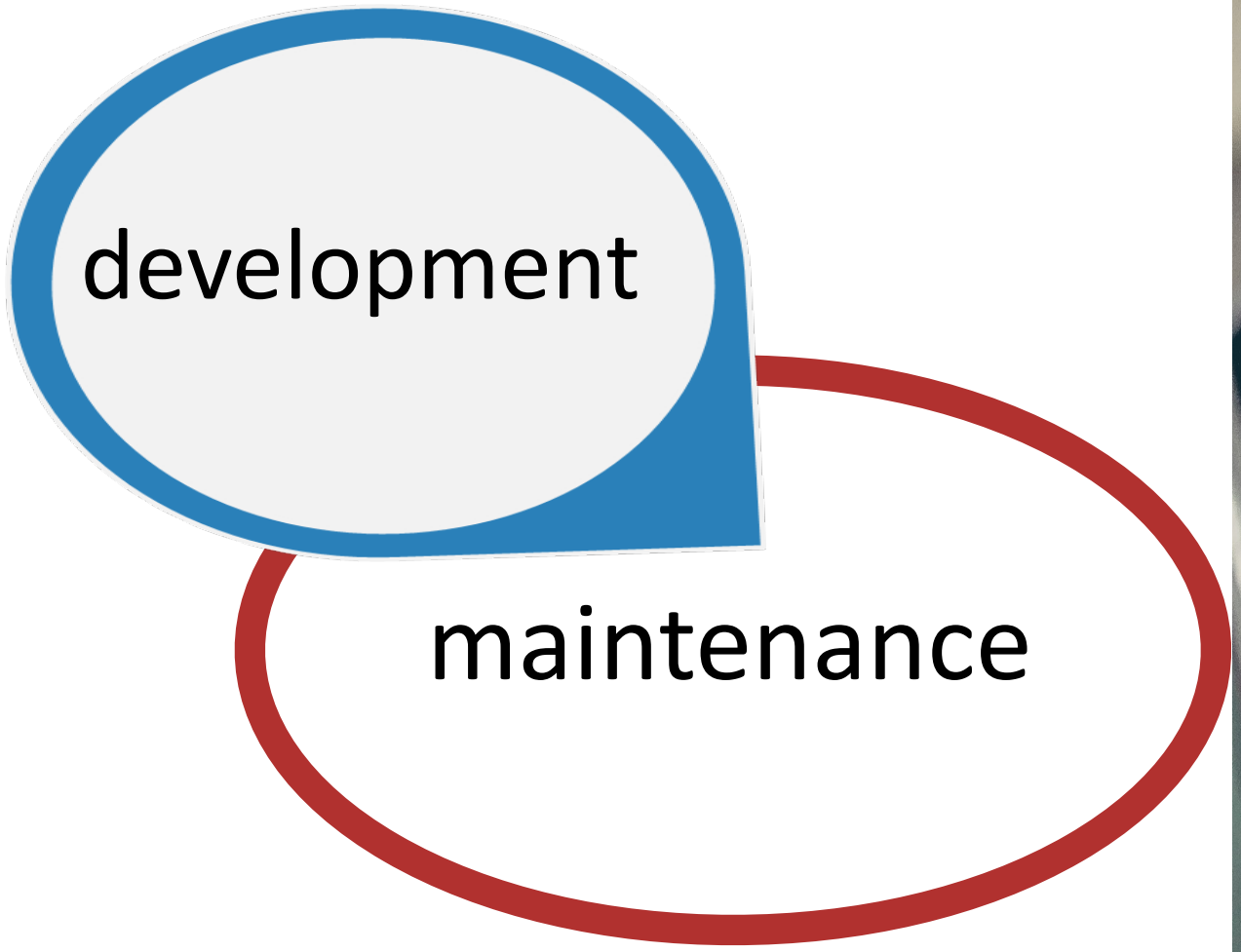
The Waterfall Process – Feedback loops



- The rationale is that the earlier you catch a bug the cheaper to fix it
- However, it is not realistic
- Thus, you keep going back, then you must move forward again in sequence

Maintenance

Classical perceptions:



MAINTENANCE

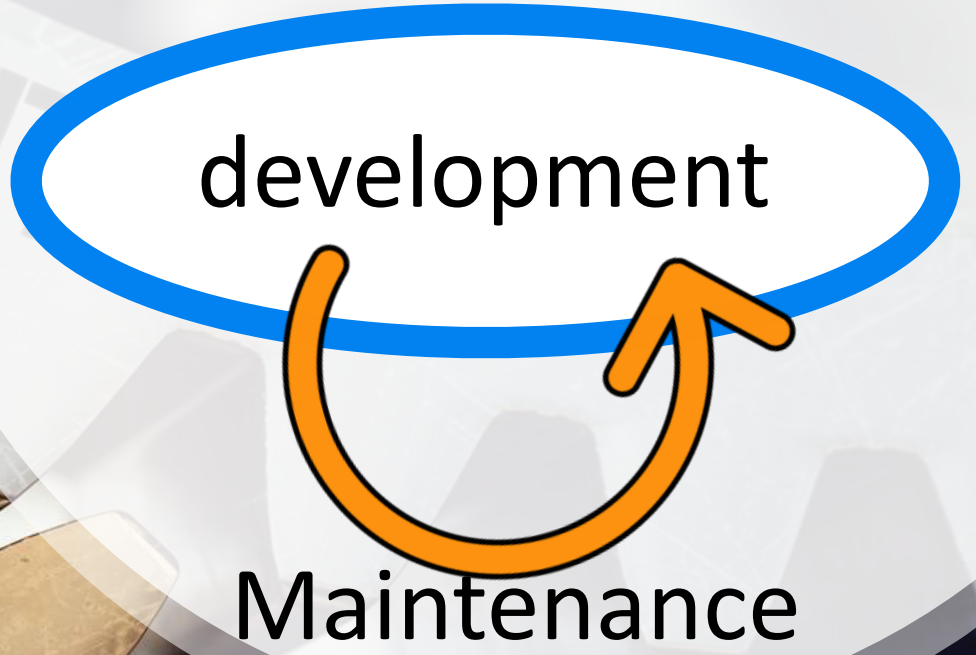
Modern perception:

The process that occurs when a software artefact is **modified** because of a problem or because of a need for improvement or adaptation

(adopted by IEEE)

MAINTENANCE

Modern perception:

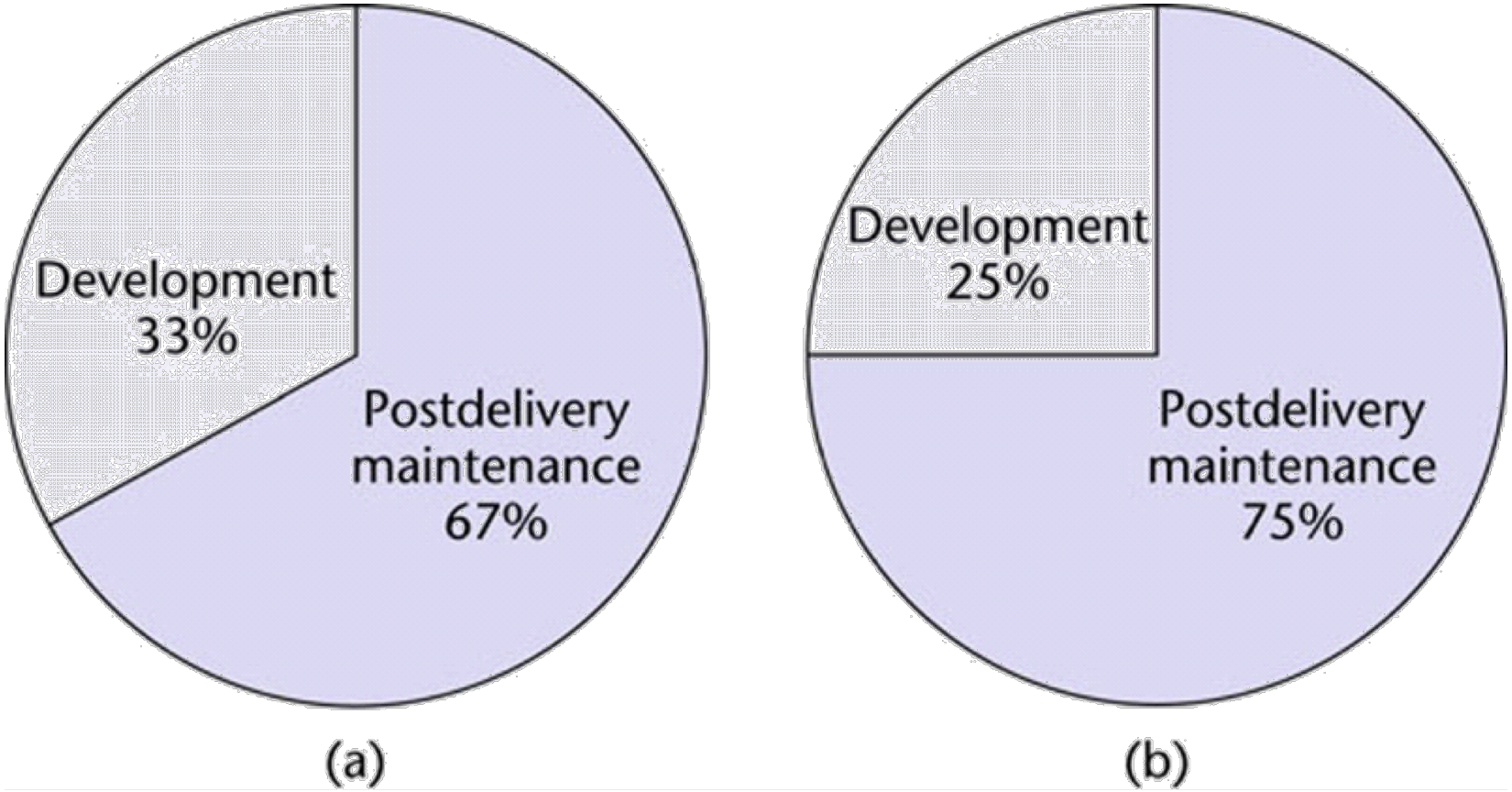


Importance Of Post-delivery Maintenance

- **Good** software is **maintained** (for 10, 20 years and more),
- **Bad** software is **discarded**

- Software is a model of reality, which is constantly changing
- Different types of maintenance
  - Corrective maintenance [about 20%]
  - Enhancement
    - ❖ Perfective maintenance [about 60%]
    - ❖ Adaptive maintenance [about 20%]

TIME (= COST) Of Post-delivery Maintenance

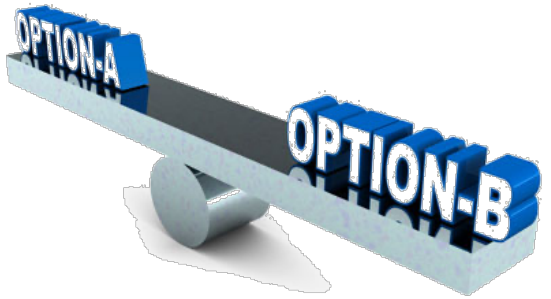


Between 1976 and 1981

Between 1992 and 1998

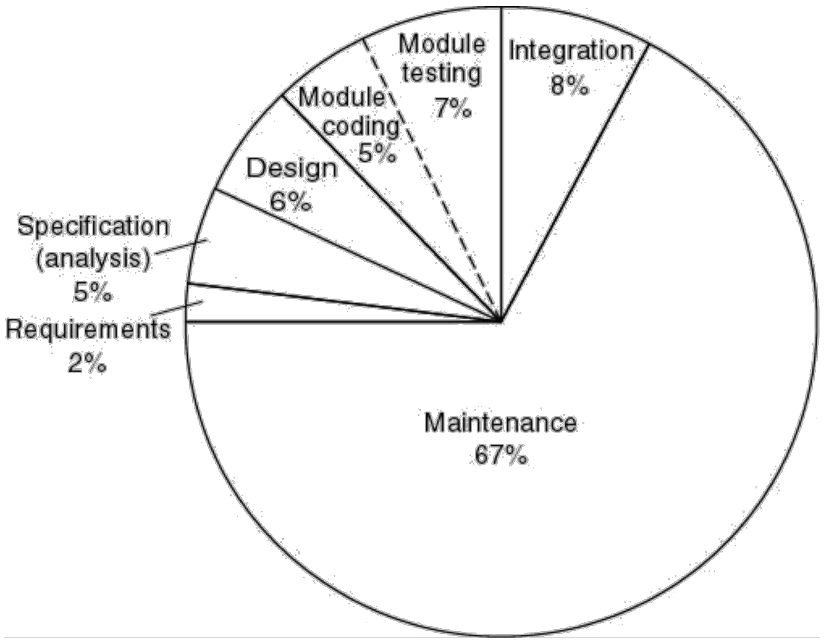
Consequence Of Relative Costs Of Phases

- Return to CM_{old} and CM_{new}

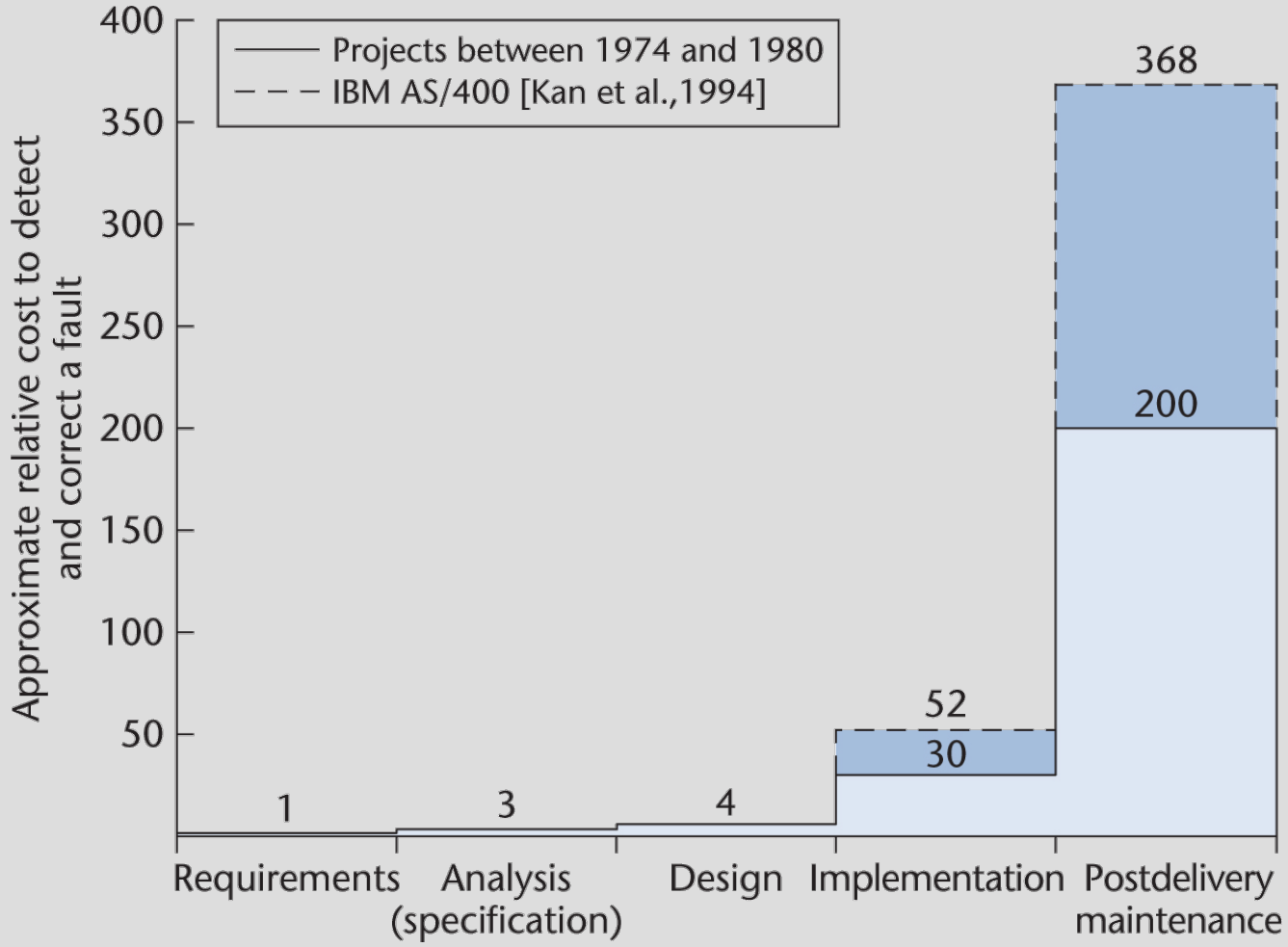


- Reducing the coding cost by 10% yields at most a 2% reduction in total costs
  - Consider the expenses and disruption incurred

- Reducing postdelivery maintenance cost by 10% yields a 6-7% reduction in overall costs



Requirements, Analysis, And Design Aspects



The cost of detecting and correcting a fault at each phase

Requirements, Analysis, And Design Aspects

- To correct a fault early in the life cycle
  - Usually just a document needs to be changed

- To correct a fault late in the life cycle
  - Change the code and the documentation
  - Test the change itself
  - Perform regression testing
  - Reinstall the product on the client’s computer(s)

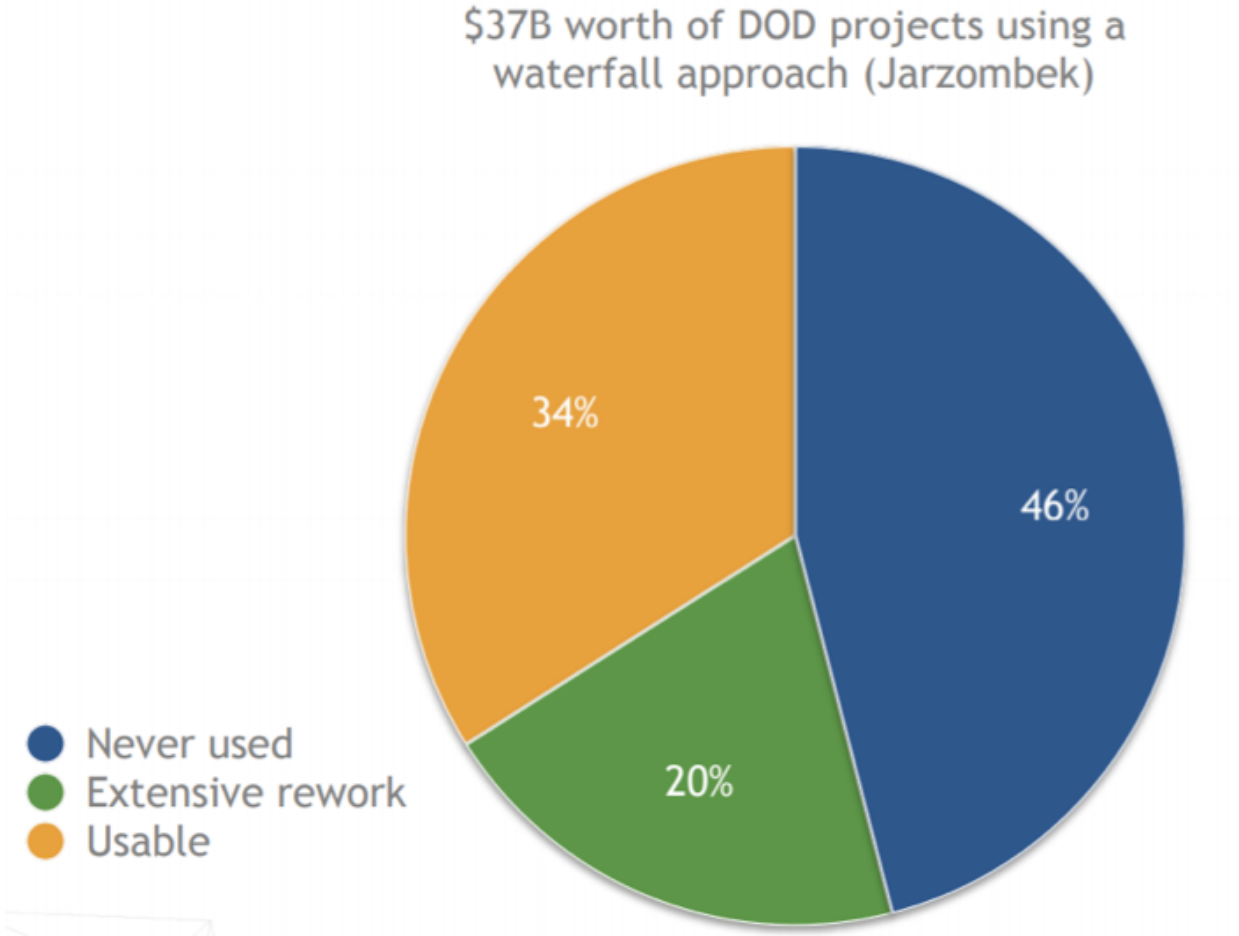
It is vital to improve our requirements, analysis, and design techniques

- To find faults as early as possible
- To reduce the overall number of faults (and, hence, the overall cost)
- To reduce the cost of maintenance

Did the Waterfall Model Work?

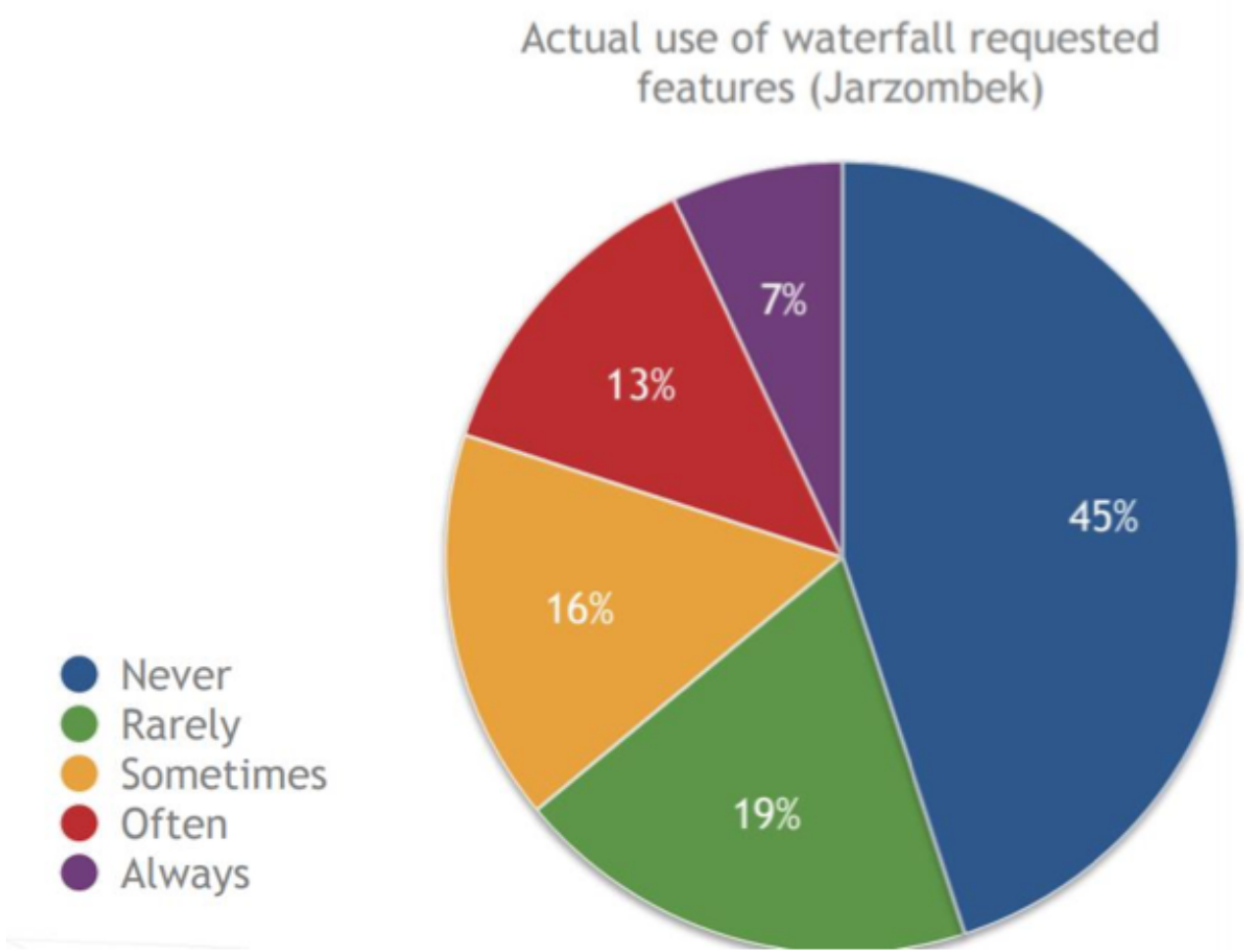
US Department of Defense (DOD) experienced an alarming amount of software failure using Waterfall

They hired an expert, Jarzombek (1999), to investigate the cause of failure



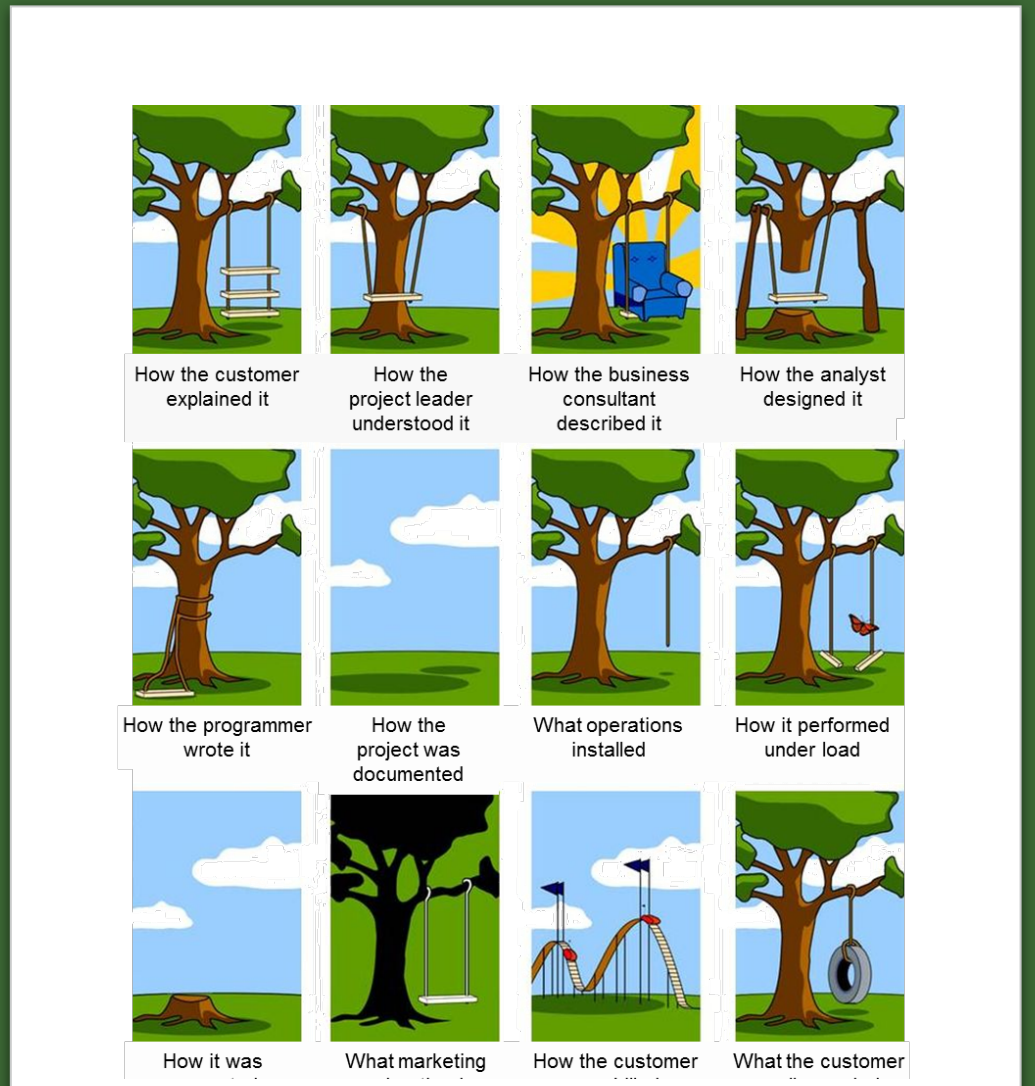
Id. at 87 (citing Lt Col (ret.) Joe Jarzombek, The 5th Annual JAWS S3 Proceedings (1999))

Among the systems that worked, not all features were used



Id. at 87 (citing Lt Col (ret.) Joe Jarzombek, The 5th Annual JAWS S3 Proceedings (1999))

Which phase caused significant failures when using the Waterfall model?



Misinterpretation of requirements from users

Attributes of the Waterfall Process

The Good

- Waterfall model is easy to understand due to its simple linear structure and clearly defined stages
- Provides structure to less experienced staff
- Facilitates project management
- It helps to define the goals and deliverables at the early stage of the project

The Bad

- Success depends on precise requirements
- No working model of the software until the end of the life cycle
- High amounts of risk / uncertainty.
- When in a later stage, it is very difficult to make any changes to the product
- Not realistic: Does not reflect real processes
- Expensive

When to Use Waterfall Model?

When to Use

- Requirements are very well known and understood
- Technology is understood
- If you are building a new version of an existing product (with some exceptions)
- Low-risk in-house projects
- Small and simple Projects
- Government projects that heavily regulated.

When Not to Use

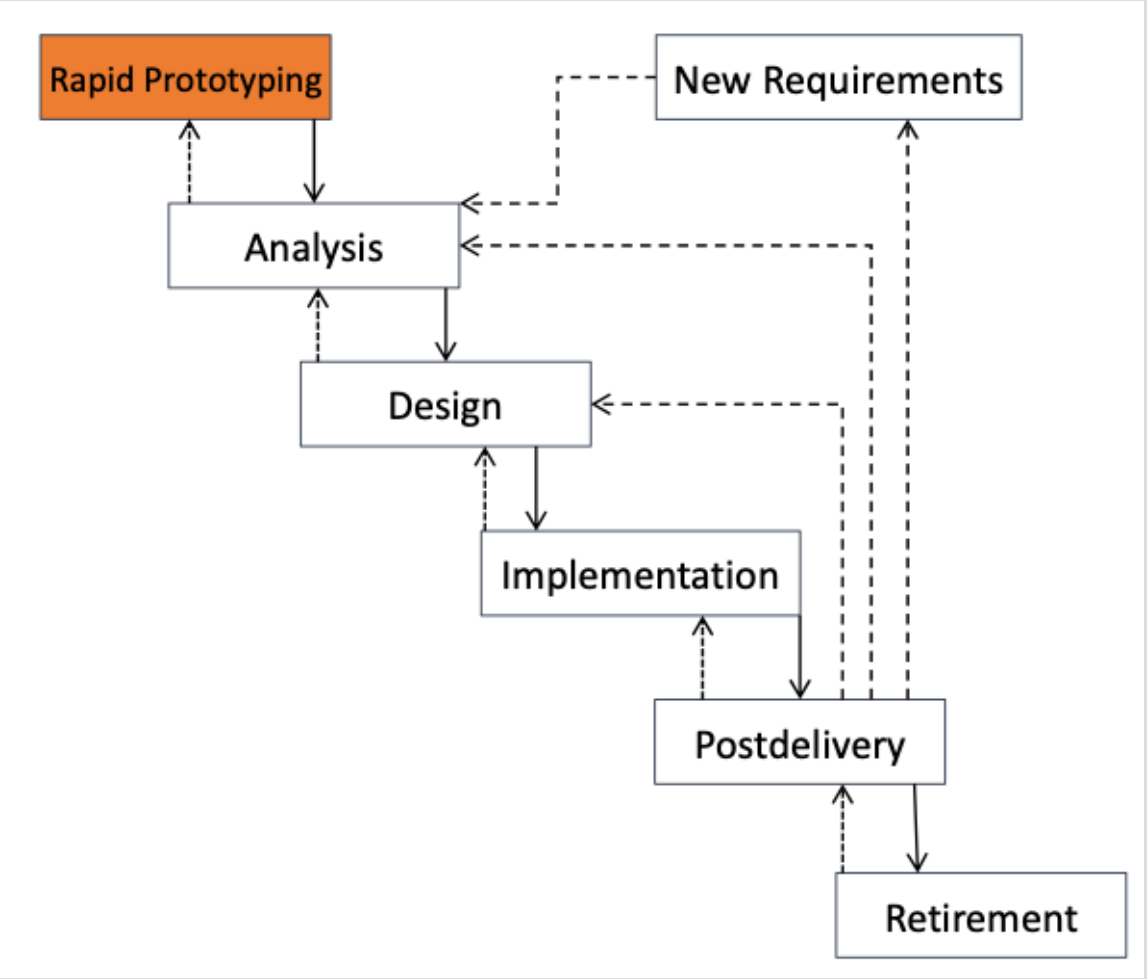
- Not a great choice for complex and long-term project
- Doesn't work for maintenance type project
- When client is strict with timeline and budget
- New idea that have not done before
- Technology is new or team doesn't know it.

Rapid Prototyping Model

A variation of the Waterfall model

Rapid Prototyping Model

- Linear model
- Instead of 'Requirements':
  - Listen to customer
  - Build prototype
  - Customer "test drives" prototype
  - Should be quick (can't be long)
- Challenges:
  - "Throw-away" phenomenon
  - Demos can set unrealistic expectations
  - Compromises that solidify



KEY POINTS: Rapid Prototyping

- Do not turn the prototype into product
- Rapid prototyping may replace specification phase — never the design phase
- Comparison:
  - Waterfall model — try to get it right first time
  - Rapid prototyping — frequent changes, then discard
  - But both method only have one shot for delivery at the end

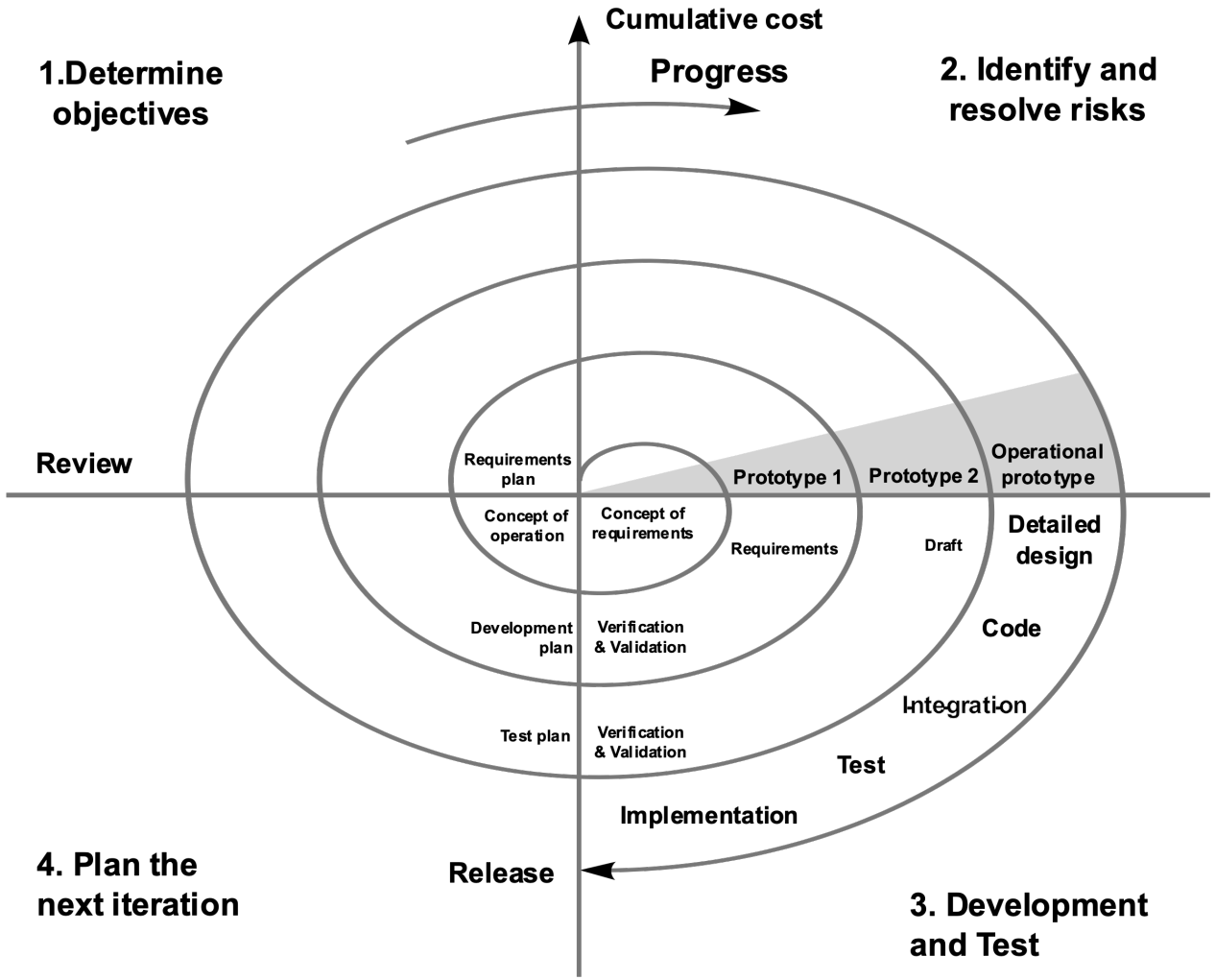
Did the Rapid Prototyping Model Work?

- When do customers see anything working?
  (at the end)
- The customers don't know what they want till they see something working.
- Same issues as Waterfall
  (one shot delivery and expensive)

The Spiral Model

The Spiral Model

- 15 years later…
- 4 Specific phases
- Uses in iterations
- Combines planning and documentation with prototyping in iterations
- There is an emphasis on risk analysis
- The radius of the iteration reflects the accumulated cost involved
- Customer is involved throughout



Attributes of the Spiral Model

The Good

- Customers see the product as it evolves.
- Risk management is part of the life-cycle (in every iteration)
- Project monitoring and scheduling are easy because of the clear phases
- Features can be added

The Bad

- Iterations are very long - they could be 0.5-2 years
- A lot of documentation for every iteration
- You can't start a phase till the other ends
- Need staff who are experts in risk
- identification and resolution
- Cost of the process is high (eg. time in prototyping)
- Requires Stakeholder engagement

When to Use the Spiral Model?

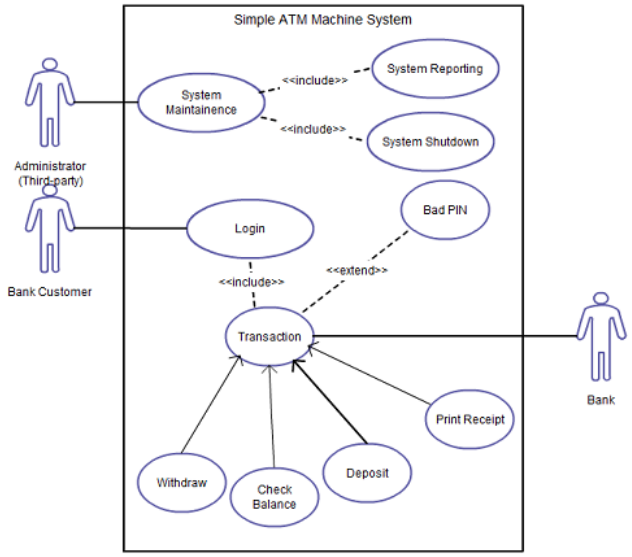
When to Use

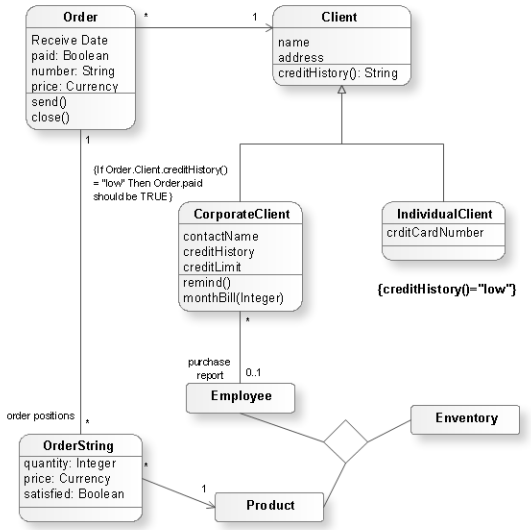
- High Risk and large systems
- Can be used for totally new ideas

When Not to Use

- Client is not available
- Progress is urgent
- When client is strict with timeline and budget
- Low risk and low budget projects (unnecessary expenses)

The Rational Unified Process

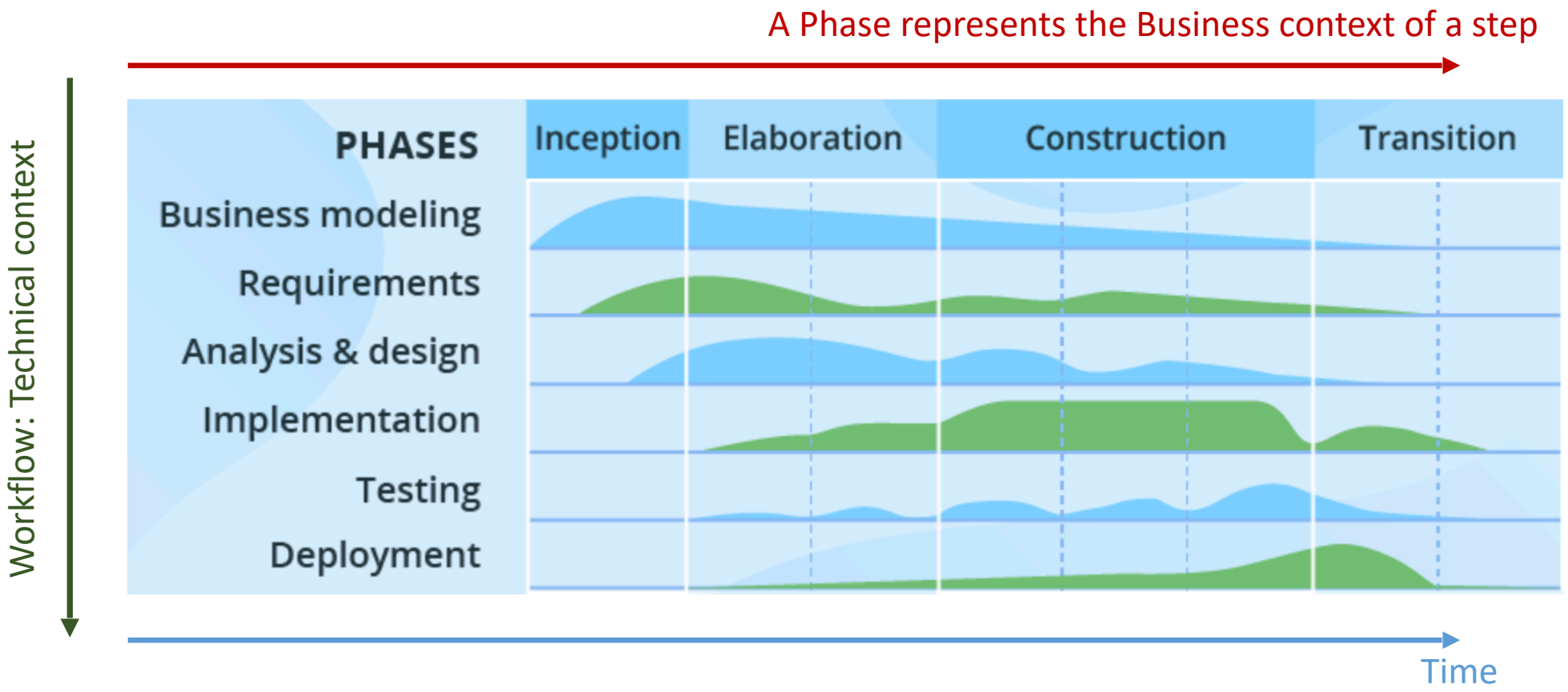




The Rational Unified Process - 2003

- Another 15 years later ...
- Closely tied to UML (Unified model language) and component-based modelling
- The Unified Process is *not* a series of steps for constructing a software product
  - No such single “one size fits all” methodology could exist
  - There is a wide variety of different types of software
- The Unified Process is an adaptable methodology
  - It must be modified for the specific software product to be developed

The Rational Unified Process (Framework)



https://www.scnsoft.com/blog/software-development-models

Phases of Business Context

Inception

- Begin to make the initial business case
- Set tentative schedule and budget
- Risk assessment
- Understand the domain
- Usually short

Elaboration

- To refine the initial requirements and define priorities of use cases
- Refine the software architecture
- Refine the business case
- Refine the project management plan

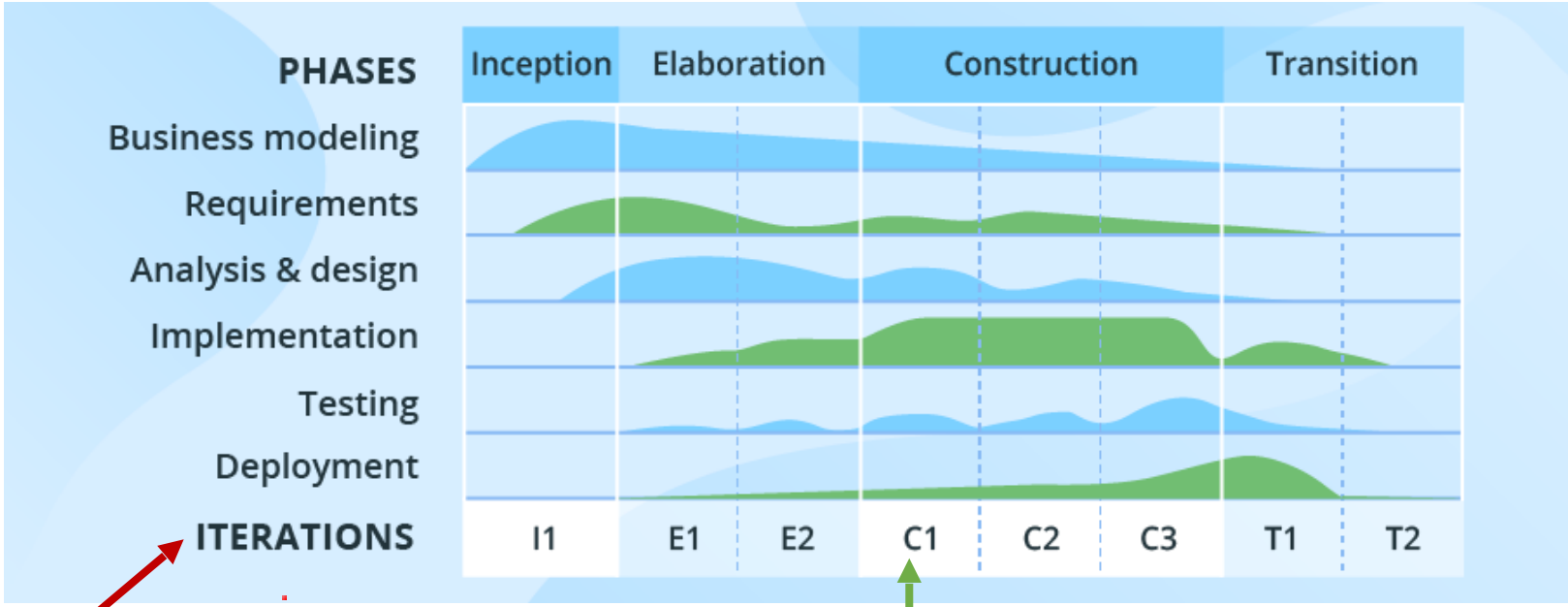
Construction

- Emphasis is on Implementation and Testing
  - Integration testing of subsystems
  - Product testing of the overall system
- Operational releases
- Usually longer than the rest

Transition

- Move to customers' real environment (deployment)
- Ensure that the requirements are met
- Correct Faults
- Complete manuals
- Driven by feedback from client

Example of the Rational United Process - 2003



Technically you can have as many iterations as you want.
The above case is typically used.

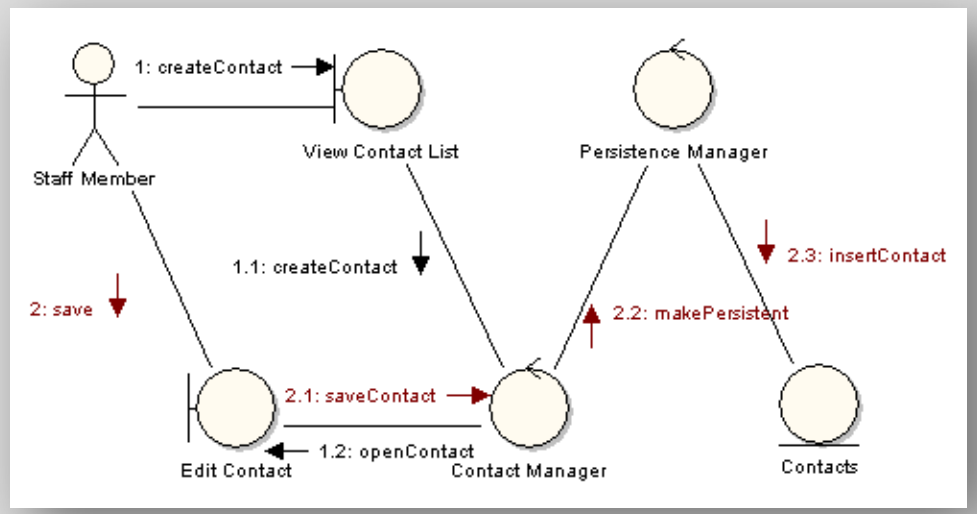
You can deliver a component here or deliver all of them at the end in parallel

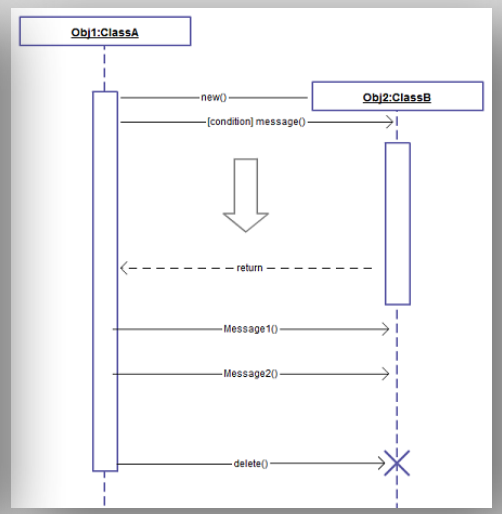
https://www.scnsoft.com/blog/software-development-models

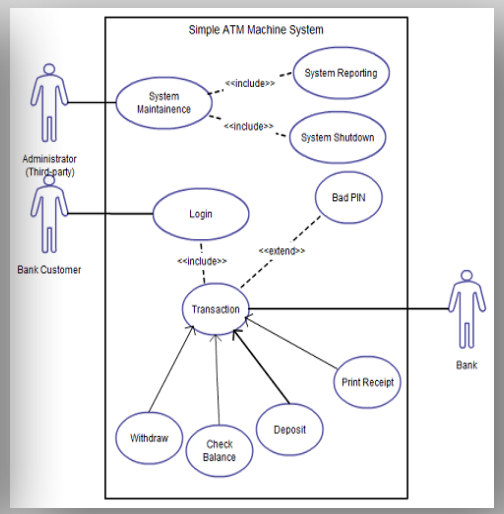
Key Points for the Rational Unified Process

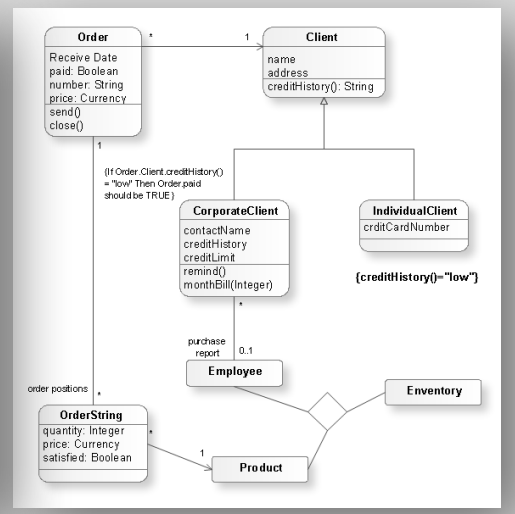
- Use case and architecture centric
- Deals with software in components with defined interfaces
- The Unified Process framework is an adaptable methodology

  It has to be modified for the specific software product to be developed









Attributes of the Rational Unified Process

The Good

- Business Process tied to the development process
- Tool support for gradual improvement of a project
- Risk Mitigation
- Focus on quality of design
- A framework that allows the use of other models
- Doesn’t need all requirements to be known at the beginning
- Deliver value early (if needed)

The Bad

- Complicated
- Expensive tools are needed
- Only good for medium and large- scale projects
- Extensive Documentation and planning (a lot of overhead)

When to Use the Rational Unified Process?

When to Use

- Medium to large projects
- Budget and schedule can be strict
- You need to show value early (you can show something working in the first iteration of construction phase)
- Engineers experienced with object-oriented design

When Not to Use

- Small simple projects
- Limited budget projects

Document & Plan Driven Approaches

- **Plan-and-document** methods were created to make software as dependable as civil engineering
  - Extensive documentation and planning
  - Experienced project managers

    Write Contracts
    Hire talents
    Evaluate development Team
- Were we able to deliver **high quality** products **on time** and **within budget**?

In 1987, Fred Brooks, a very well-known expert on how to produce software systems led a task force to find out what is going wrong with all these software failures from traditional techniques.

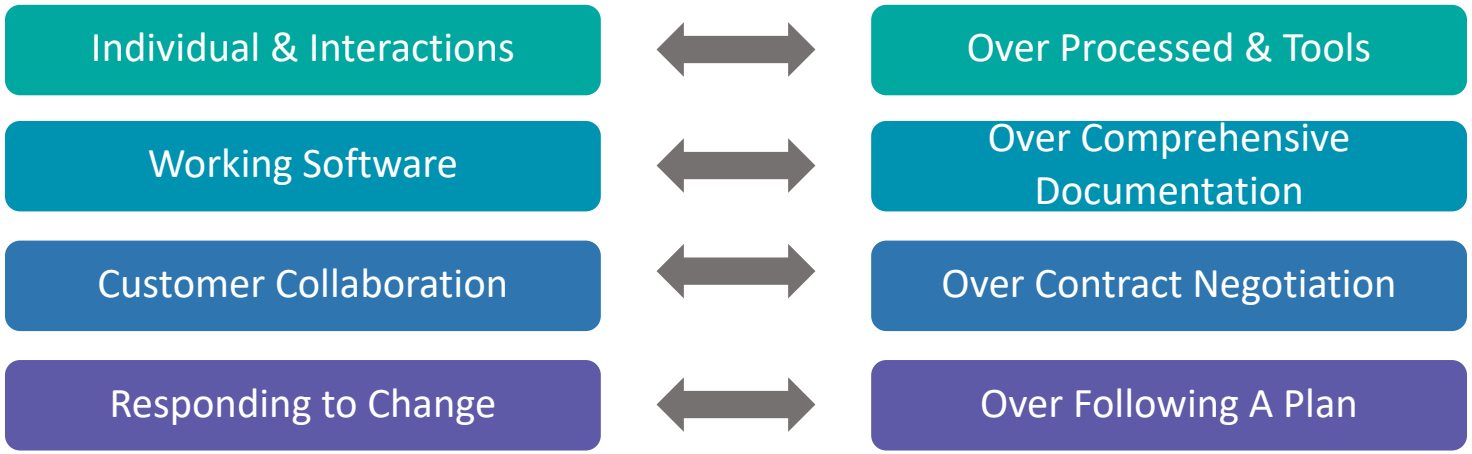
"The document-driven, specify-then-build approach lies at the heart of so many software problems."

Birth of Agile Approaches

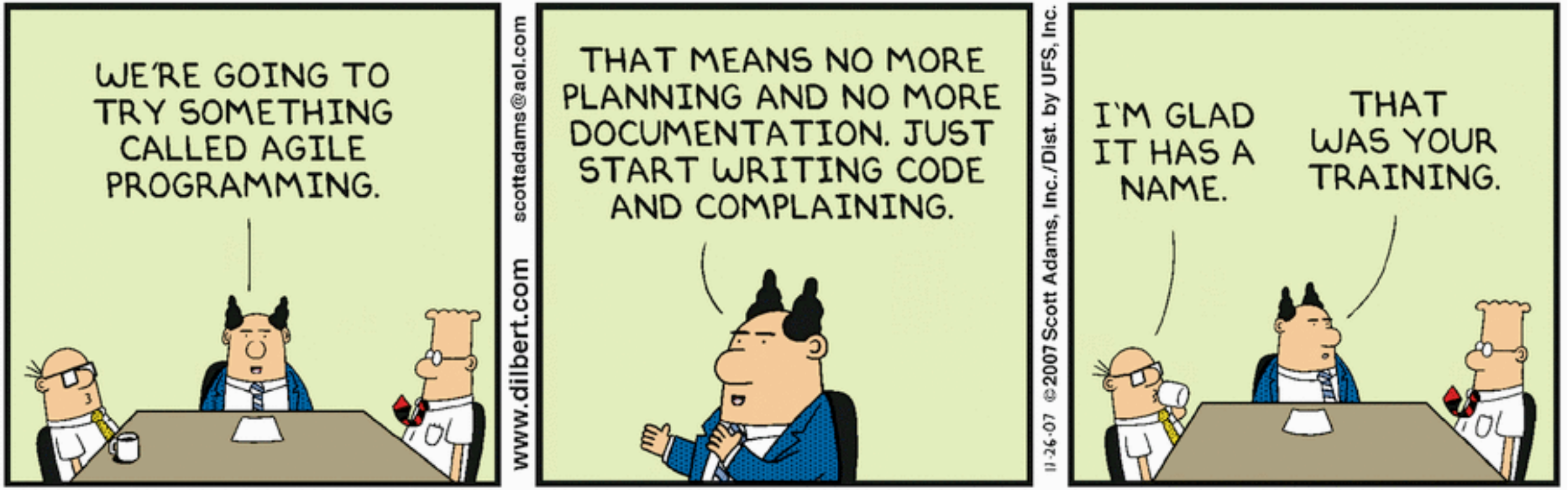
Agile Manifesto

- Signed in 2001 by “independent-minded practitioners”

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



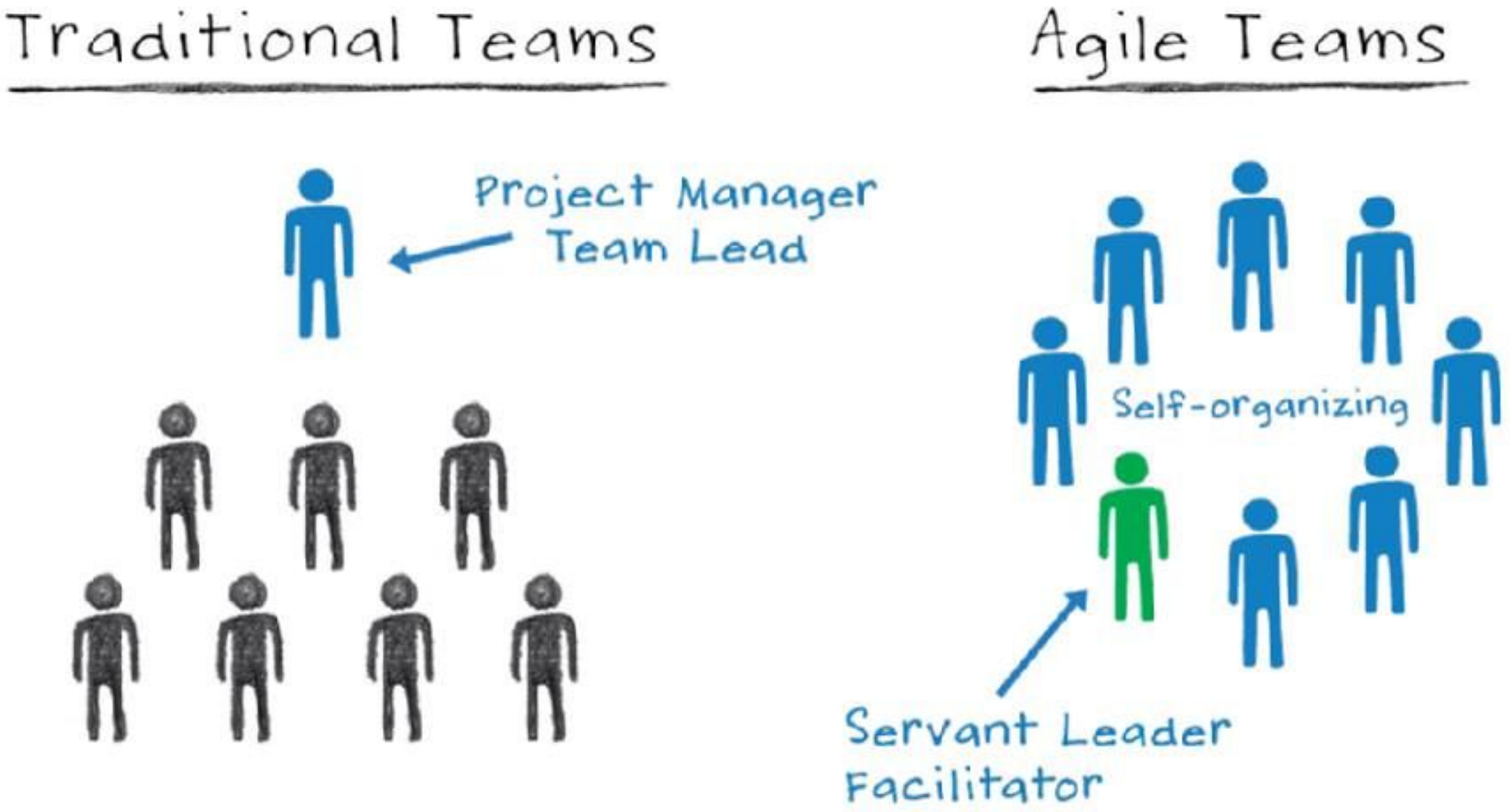
- That is *, while there is **value in the items on the right**, we **value the items on the left more**.”*



Agile is a Mindset

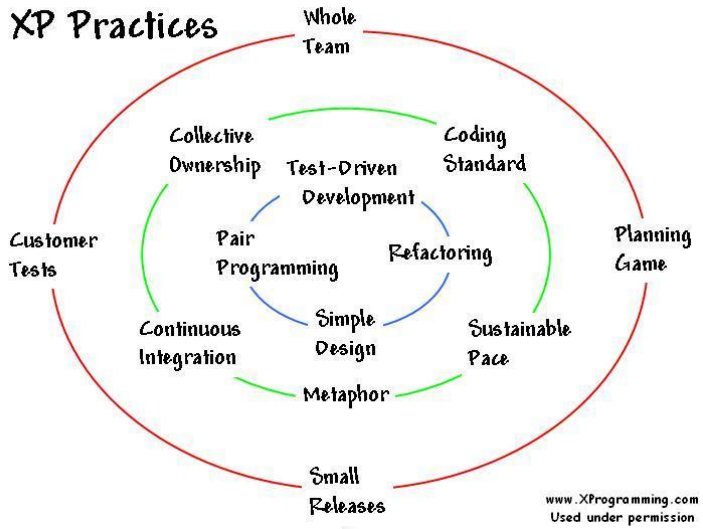
Principles of Agile Method

- Embraces change as a fact of life
  - Continuous improvement over fixed phase
- Incremental delivery (1-4 weeks)
- Increments have value
- Emphasize Test driven development
- Small teams
- Customer involvement (not during iterations)
- The automation of tasks where possible
- Light-weight documentation
- Velocity is the way to measure progress (how to predict progress with past progress)
- Self management



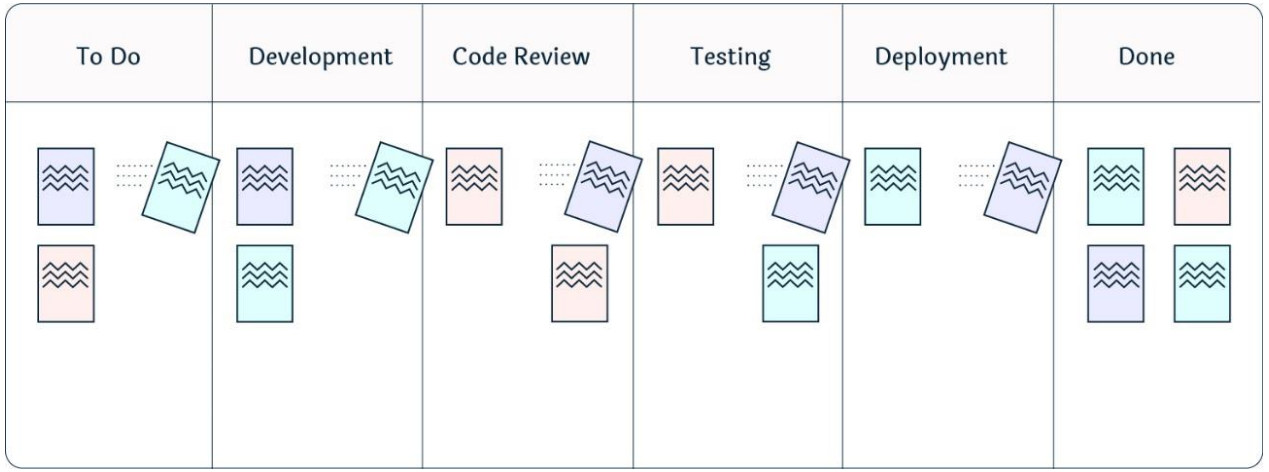
Source: [Emmanuel Peña Alvarez]

Types of Agile Methods



Scrum

Kanban



https://getnave.com/blog/kanban-workflow/

Attributes of the Agile Method

The Good

- Flexible to change and continuous feedback which increases the change of building the right product
- Customer Satisfaction
- Early value delivery and early to market
- Team Ownership (self organizing team)

The Bad

- May require some rework (since we didn't know EVERYTHING upfront)
- Requires close collaboration with the client
- Good tools for automation are a must have (poor ones might delay you)
- Not every individual/team can adopt Agile values, setup needs trust and communication

Practising Agile

- Gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks

- The developers know that they will have 3 weeks (but no more) to deliver a new iteration
  - Without client interference of any kind

- If it is impossible to complete the entire task in the timebox, the work may be reduced ("descoped")
  - Agile processes demand fixed time, not fixed features

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

CHAOS Resolution by Agile Versus Waterfall

The resolution by Agile versus Waterfall from FY2011-FY2015 with in the new CHAOS database

When to Use the Agile Method?

When to Use

- Lightweight methods suit small- to medium-size projects (or large projects divided into components)
- Used for time-critical applications and prototypes
- Requirements are sure to change, new or uncertain
- Technology is new

When Not to Use

- Do not have a good team (attitude and skills)
- For large projects where customer needs specific documentation and formal communication
- Large Systems that can't be broken into modules for smaller teams

Introduction to Scrum

- https://vimeo.com/334800839/3a7b2fa063
- Watch the video (4 mins) by Prof. Tan Chek Tien

What We Have Learned Today

- Motivation
  - Developing software is not simple, and often not done well.
  - **Developing software is not the same as coding**. There is much more to it.
- We need to have good **systems and processes** in place.
- We want to keep **maintenance** in mind when developing.
- We want to **detect faults as early as possible**.
- Software Development Life Cycle (SDLC)
- Software models – their Pros, Cons, and when to use them