24th Summer School on PARALLEL COMPUTING

Software engineering for HPC

Paolo Ciancarini - paolo.ciancarini@unibo.it
Department of Informatics - University of Bologna

SCAI SuperComputing Applications and Innovation

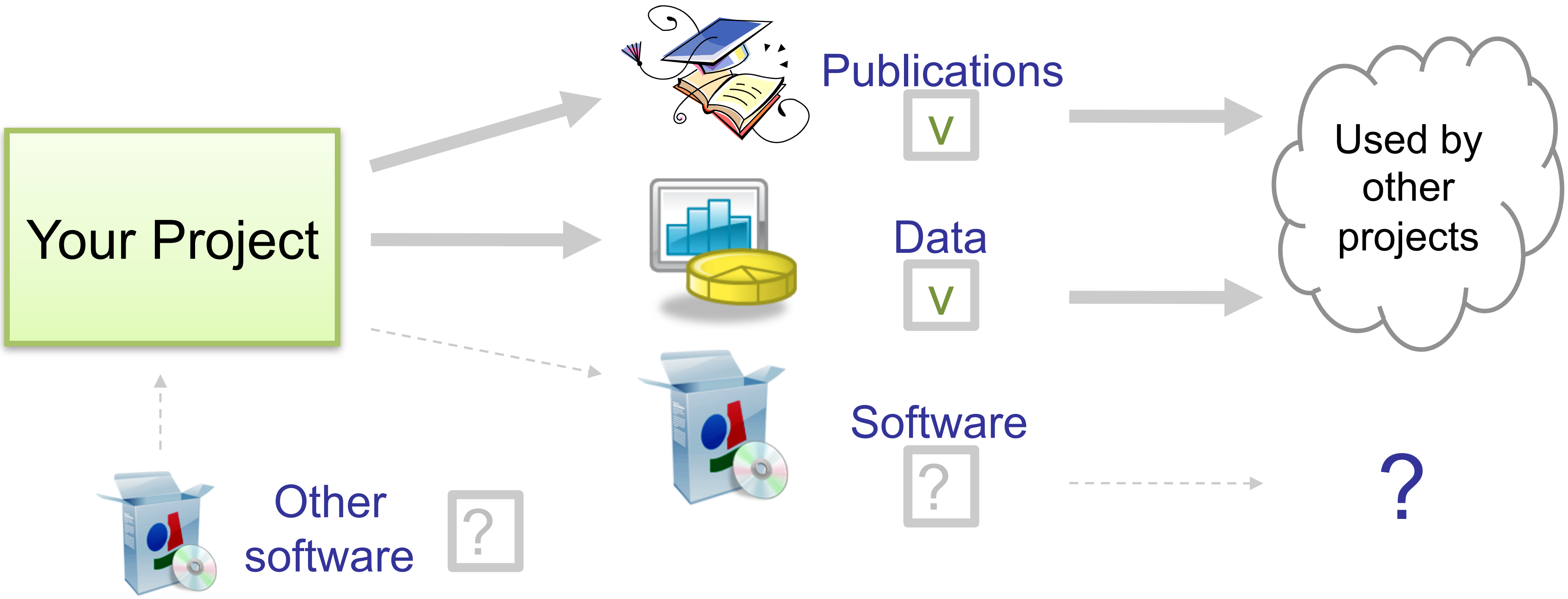
consorzio interuniversitario nazionale per l'informatica

Agenda

- What is Software Engineering?
- The Software Development Lifecycle
- Software Development Activities
- Methods and tools

you build software for your research

Using software in a research project



What is the future of your software?

Typical development of software for science

Research Project → Research Questions → Develop Software → Run Software → Analyse Data → Publish Paper → Project Ends → ?

What happens to the software?

- Thrown away
- Kept on some systems, possibly in different versions
- Dumped on a code repository

What happens when…

- You have a follow-on project?
- Someone wants to (re)use the code?
- Someone wants to reproduce your results?
- Maintenance or future reuse should be considered?

Beware of software aging!

Software can *age*

- Ill-conceived design or modifications
- Functional operation degrades over time
- It becomes unsustainable, unusable
- Lack of proper maintenance
- Infrastructure (os, libraries, language platform) evolves
- Some software types more susceptible

Enters Software Engineering

"Software engineering is the discipline concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use"

[Sommerville 2007]

Software Engineering

"The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines." [Naur & Randell, 1968]

Software Engineering

- A definition and some issues
  - “developing quality software on time and within budget”
- Trade-off between a system perfectly engineered and the available resources
  - SwEng has to deal with real-world issues
- State of the art
  - Community decides on “best practices” + life-long education

What is Software Engineering?

A naive view:

Problem Specification $\xrightarrow{\textit{coding}}$ Final Program

But ...

- Where did the *problem specification* come from?
- How do you know the problem specification corresponds to and satisfies 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 up* if you have more than a one person in the developing team?
- How do you reuse exisiting software for solving similar problems?

What is Software Engineering?

“multi-person construction of multi-version software”

— Parnas

- Software is complex and difficult to build
- Team-work
  - Scale issue (“program well” is not enough) + communication issues: Conway’s law
- Successful software systems must evolve or perish
  - Change is the norm, not the exception

Conway's Law

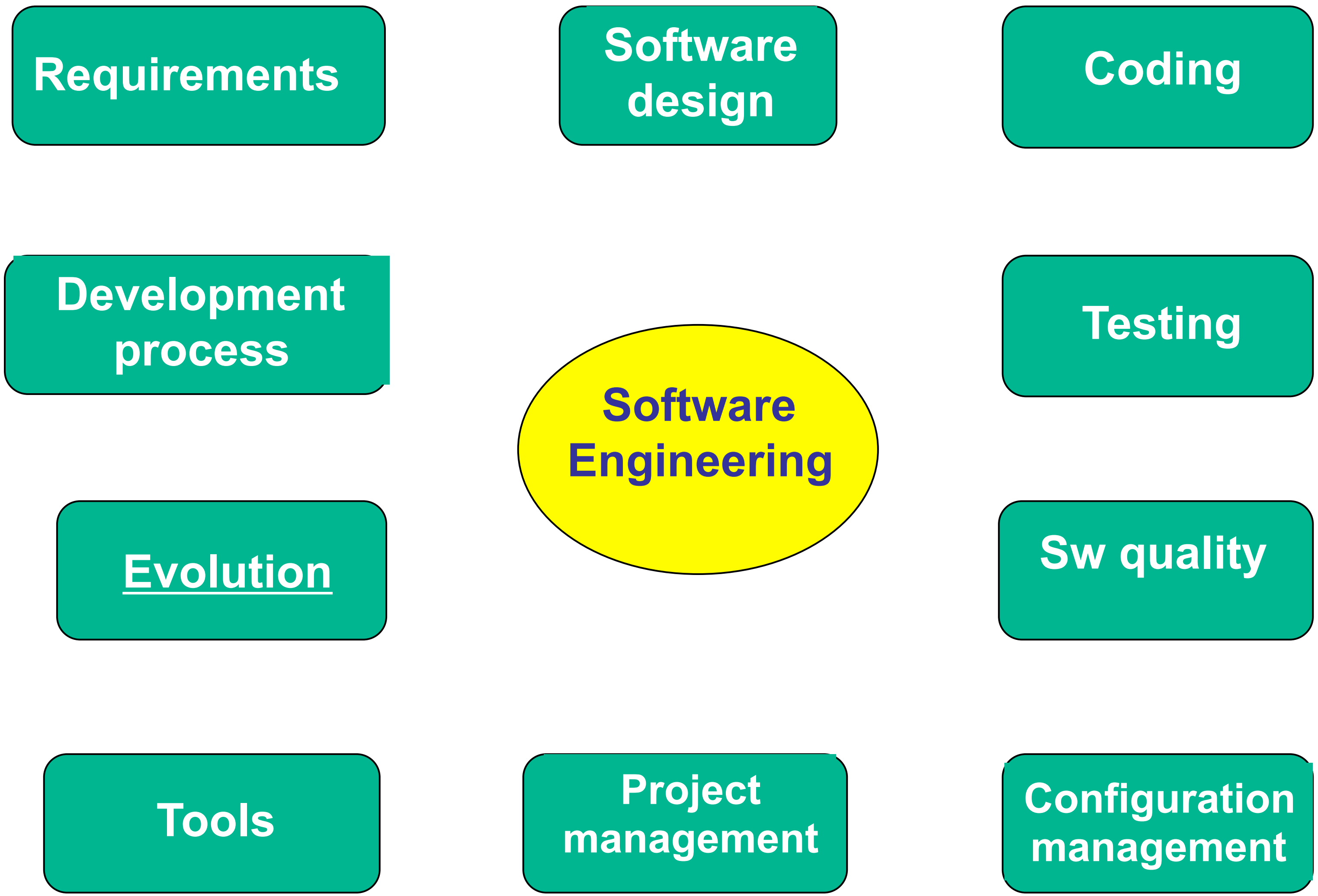
- The law: *Organizations that design software systems are constrained to produce designs that are copies of the communication structures of these organizations*
- Example: "If you have four groups working on a compiler, you'll get a 4-pass compiler"
- Several studies found significant differences in modularity when software is outsourced, consistent with a view that distributed teams tend to develop more modular products

What is Software Engineering?

"software engineering is different from other engineering disciplines"

— Sommerville

- It is not constrained by physical laws
  - limit = human knowledge
- It is constrained by social forces
  - Balancing stakeholders needs
  - Consensus on functional and especially non-functional requirements


Requirements
Software design
Coding
Development process
Software Engineering
Testing
Evolution
Sw quality
Tools
Project management
Configuration management

Topics of the discipline

- Standard methods and techniques for software
- Software product lifecycles
- Requirement analysis
- Software modeling and design
- Project Management for software systems
- Measuring and ensuring software quality
- Software evolution and maintenance
- Typical tools used by software engineers

Software Engineering for HPC

- Software engineering aims to designing, implementing, and modifying software so that it is faster to build, of higher quality, more maintainable
- In HPC there are all the general problems of software development, and the specific problem that software developers have scarce knowledge of software engineering best practices
- In the following slides we will deal with some of these problems and suggest some solutions

Roadmap

- What is Software Engineering?
- **The Software Development Lifecycle**
- Software development activities
- Methods and tools

Software: the product of a process

- Many kinds of software products
  → many kinds of development processes
- "Study the process to improve the product"

- A software development process can be described according to some specific "model"
- Examples of process models: waterfall, iterative, agile, explorative,…
- These models differ mainly in the roles and activities that the stakeholders cover

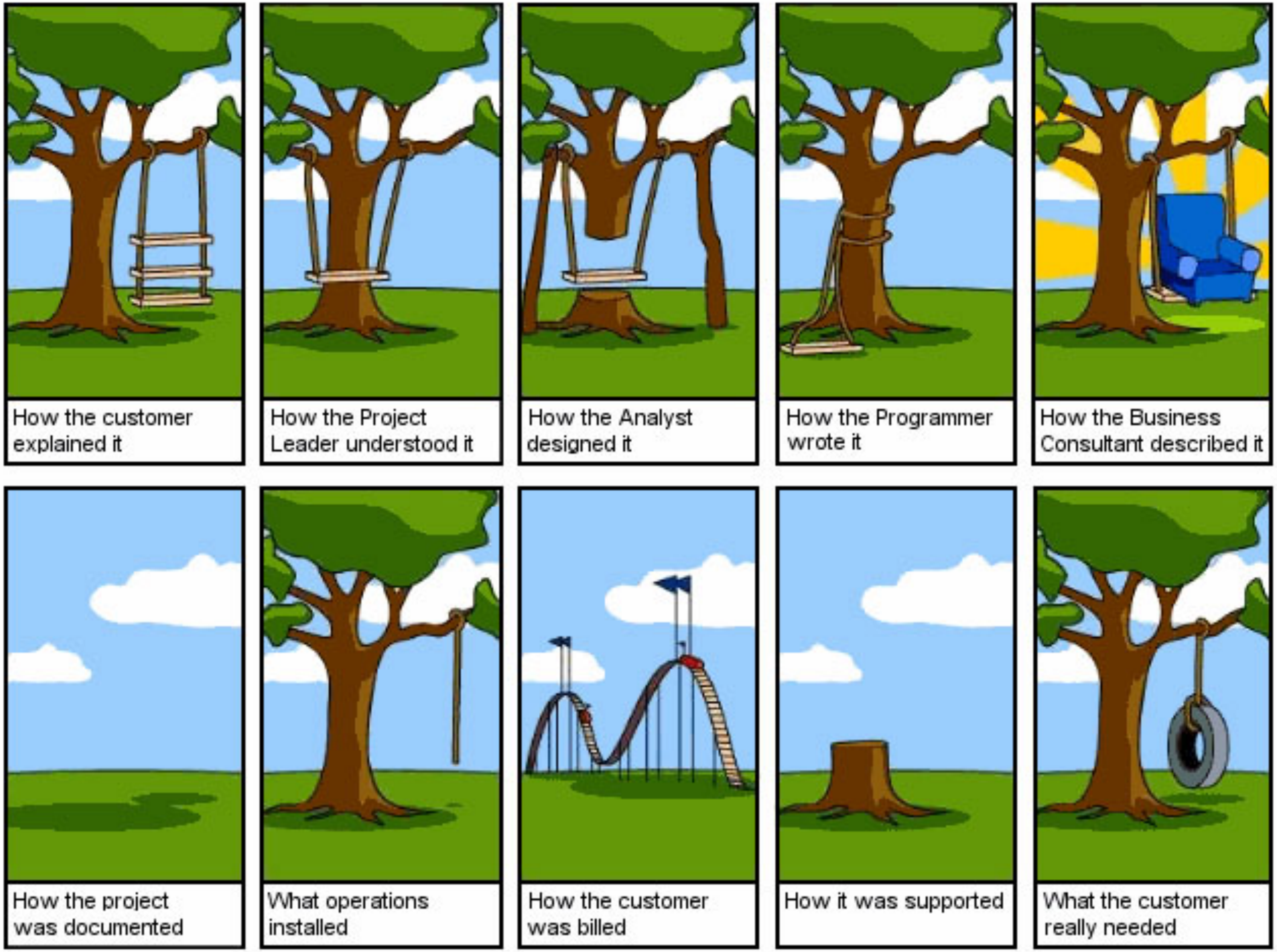
Stakeholders

Typical stakeholders in a sw process

- Users
- Decisors
- Designers
- Management
- Technicians
- Funding people
- …

Each stakeholder has a specific viewpoint on the product and its development process

Just a joke?


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

HPC stakeholders attributes

Attribute	Values	Description
Team size	Individual	This scenario, sometimes called the "lone researcher" scenario, involves only one developer.
	Large	This scenario involves "community codes" with multiple groups, possibly geographically distributed.
Code life	Short	A code that's executed few times (for example, one from the intelligence community) might trade less development time (less time spent on performance and portability) for more execution time.
	Long	A code that's executed many times (for example, a physics simulation) will likely spend more time in development (to increase portability and performance) and amortize that time over many executions.
Users	Internal	Only developers use the code.
	External	The code is used by other groups in the organization (for example, at US government labs) or sold commercially (for example, Gaussian, www.gaussian.com)
	Both	"Community codes" are used both internally and externally. Version control is more complex in this case because both a development and a release version must be maintained.

V.Basili et al., Understanding the High-Performance-Computing Community: A Software Engineer's Perspective, IEEE Software, 2008

The software development process

- **Software process**: set of roles, activities, and artifacts necessary to create a software product
- Possible roles: stakeholder, designer, developer, tester, maintainer, ecc.
- Possible artifacts: source code, executables, specifications, comments, test suite, etc.

Activities

- Each organization differs in products it builds and the way it develops them; however, most development processes include:
  - Specification
  - Design
  - Verification and validation
  - Evolution
- The development activities must be modeled to be managed and supported by automatic tools

Software development activities

Requirements Collection	Establish customer’s needs
Analysis	Model and specify the requirements (“what”)
Design	Model and specify a solution (“how”)
Implementation	Construct a solution in software
Testing	Validate the software against its requirements
Deployment	Making a software available for use
Maintenance	Repair defects and adapt the sw to new requirements

NB: these are ongoing activities, not sequential phases!

First development step: requirements

- The first step in any development process consists in understanding the needs of someone asking for a software
- The needs should be stated explicitly in “requirements”, which are statements requiring some function or property to the final software system

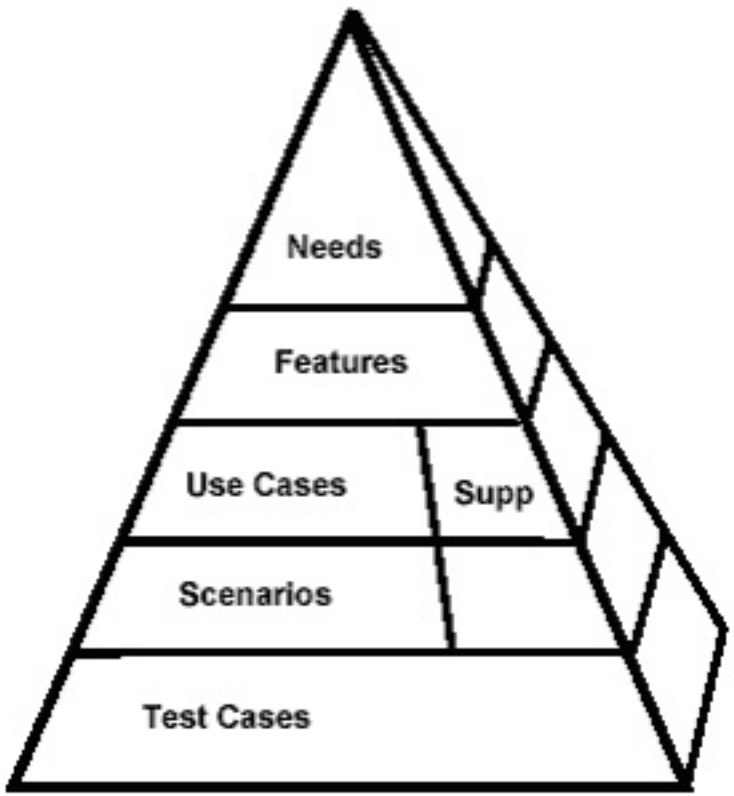
The requirements pyramid

Some user has some need

Needs are answered by “features” that some system must have

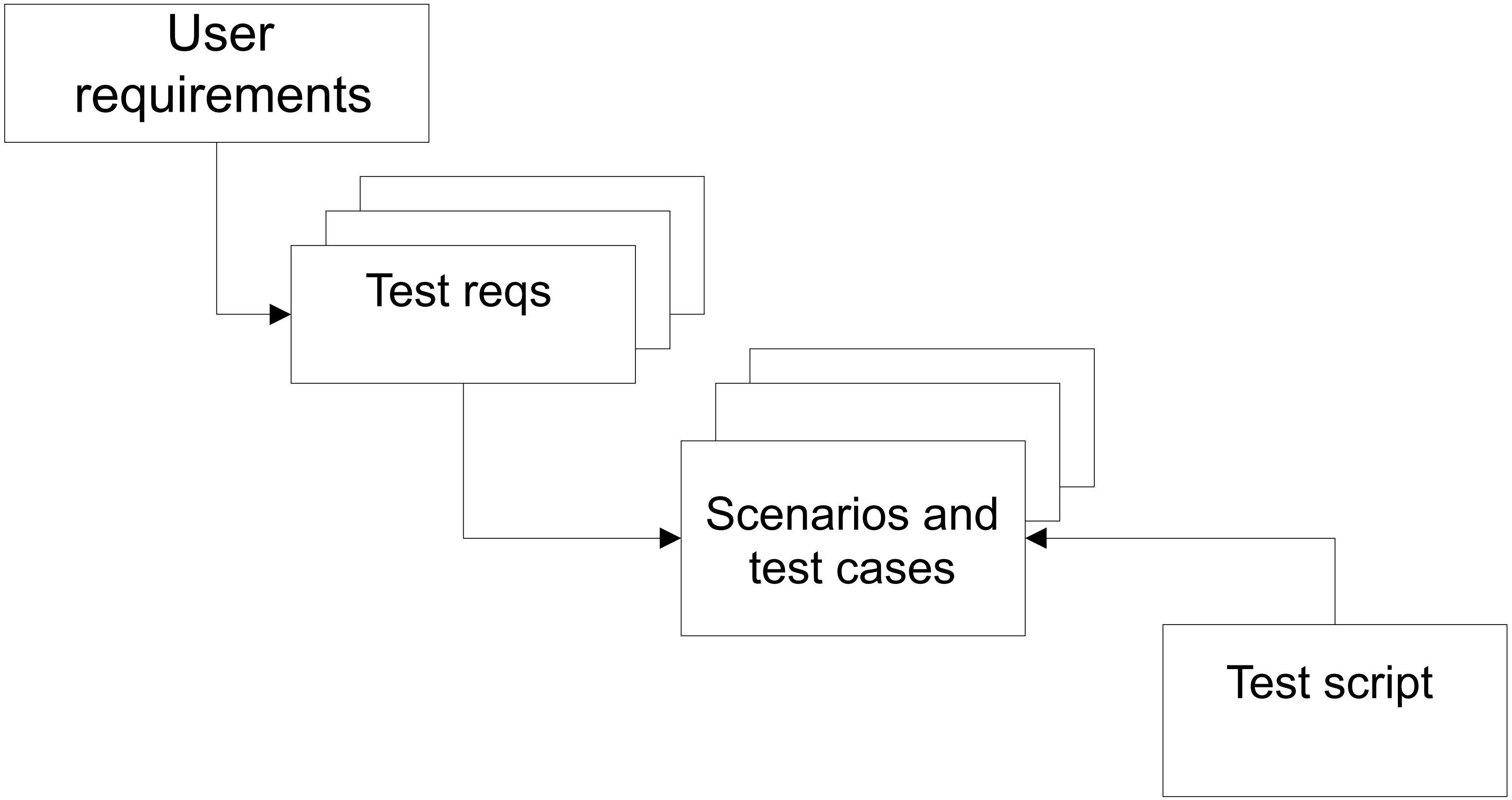
Each feature corresponds to a need and is a collection of requirements

Features and requirements can be aggregated in “scenarios” where testing can prove that the features will satisfy the needs



www.ibm.com/developerworks/rational/library/04/r-3217

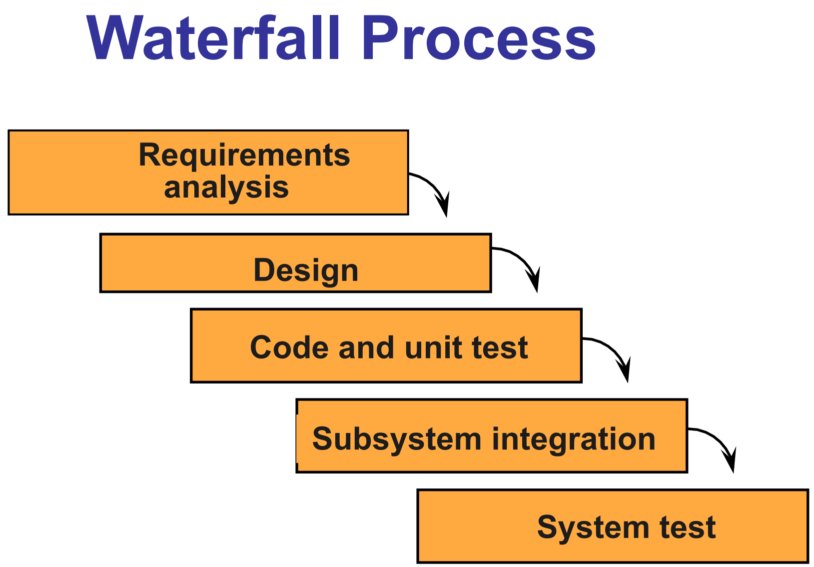
Requirements and tests



Models for the software process

- Waterfall (planned, linear)
- Spiral (planned, iterative)
- Agile (unplanned, test driven)

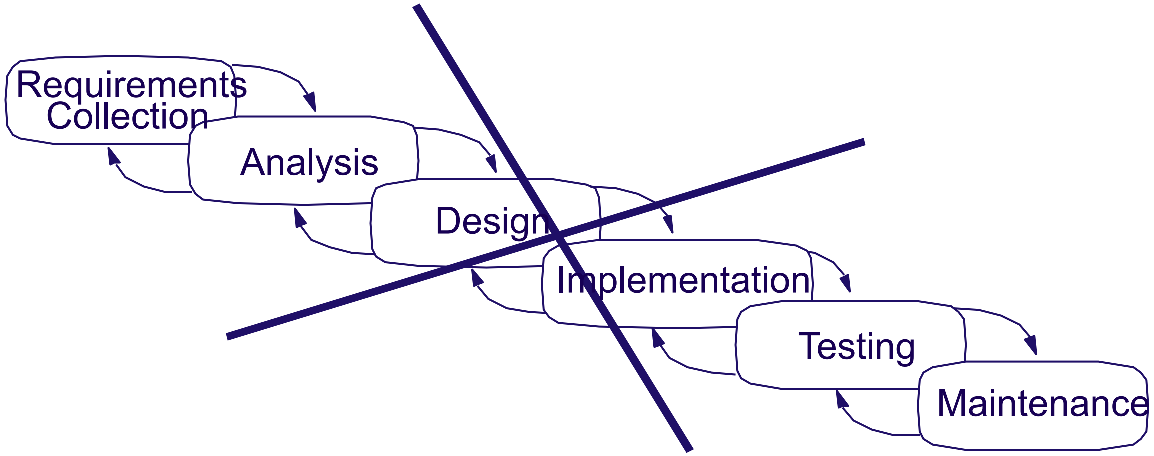
Waterfall characteristics



- One way communicatons
- Delays confirmation of critical risk resolution
- Measures progress by assessing work-products that are poor predictors of time-to-completion
- Delays and aggregates integration and testing
- Precludes early deployment
- Frequently results in major unplanned iterations

The classical software lifecycle

The classical software lifecycle models the software development as a step-by-step "waterfall" between the various development phases



The waterfall model is flawed for many reasons:

- **Requirements must be *frozen too early* in the life-cycle**
- **User requirements are *validated too late***
- **Risks in costructing wrongly the software are high**

Problems with the waterfall lifecycle

1. “Real projects rarely follow the sequential flow that the waterfall model proposes. *Iteration* always occurs and creates problems in the application of the paradigm”

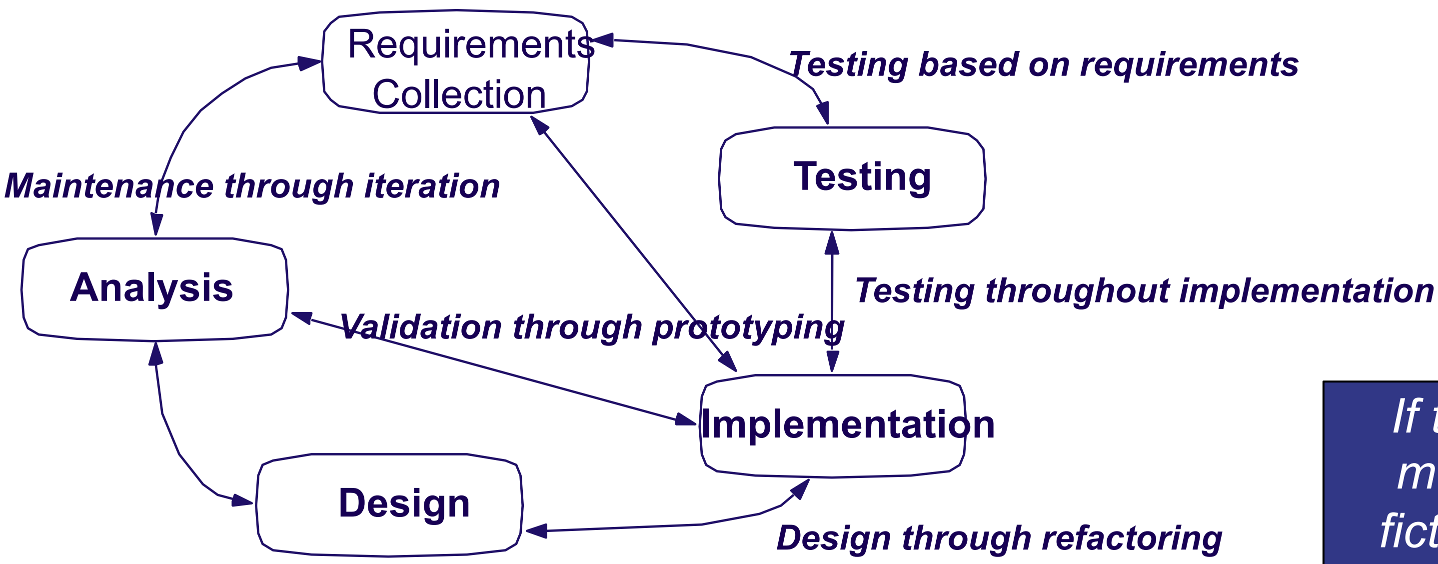
2. “It is often *difficult* for the customer *to state all requirements* explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.”

3. “The customer must have patience. A *working version* of the program(s) will not be available until *late in the project* timespan. A major blunder, if undetected until the working program is reviewed, can be disastrous.”

— *Pressman, SE, p. 26*

Iterative development

In practice, development is always iterative,
and *most* activities can progress in parallel



If the waterfall model is pure fiction, why is it still the dominant software process?

Iterative development

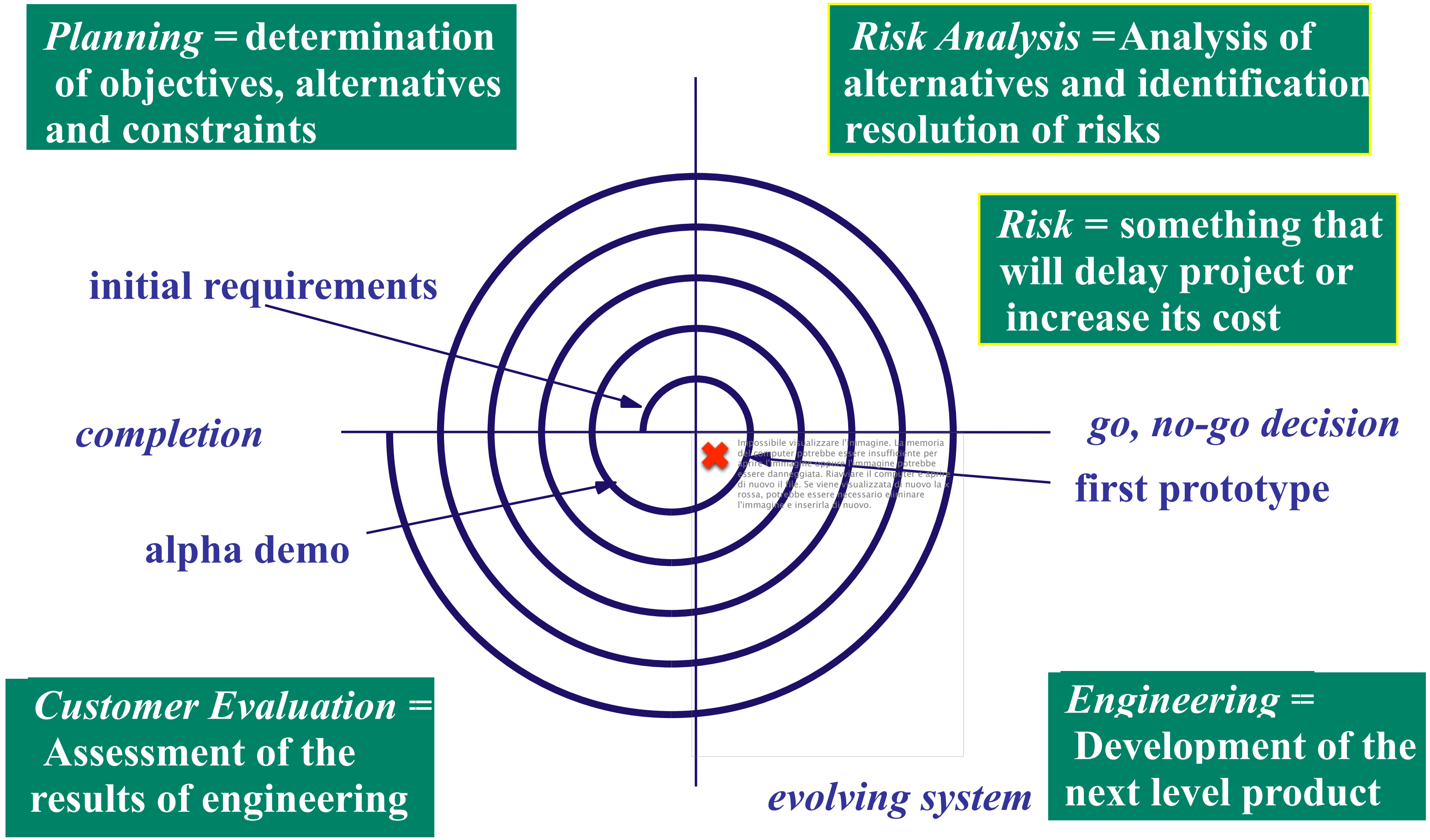
- Plan to iterate your analysis, design and implementation
  - You will not get it right the first time, so integrate, validate and test as frequently as possible
  - During software development, more than one iteration of the software development cycle may be in progress at the same time
  - This process may be described as an 'evolutionary acquisition' or 'incremental build' approach

Iterative development

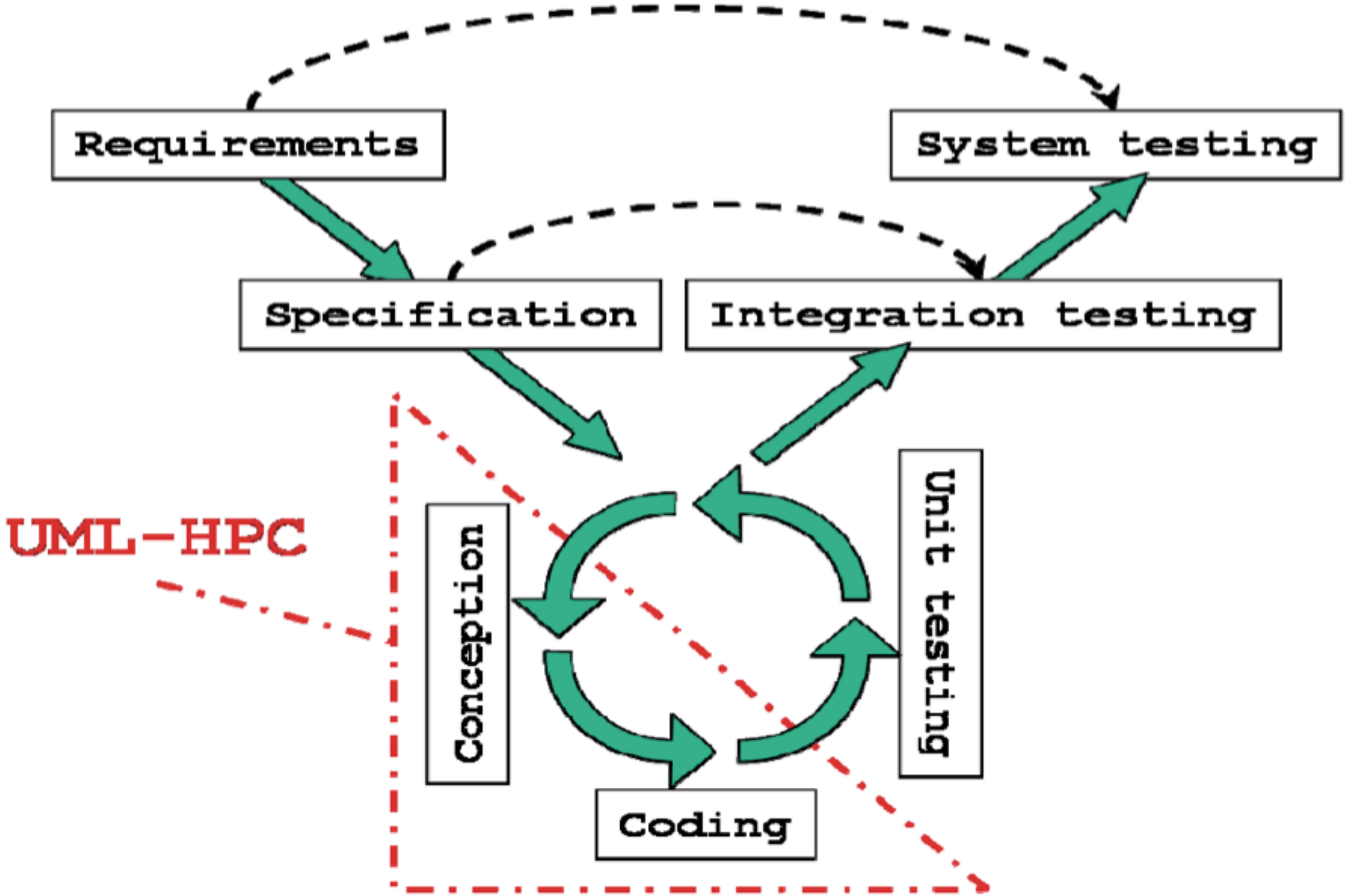
Plan to *incrementally* develop (i.e., prototype) the system

- If possible, *always have a running version* of the system, even if most functionality is yet to be implemented
- *Integrate* new functionality as soon as possible
- *Validate* incremental versions against user requirements.

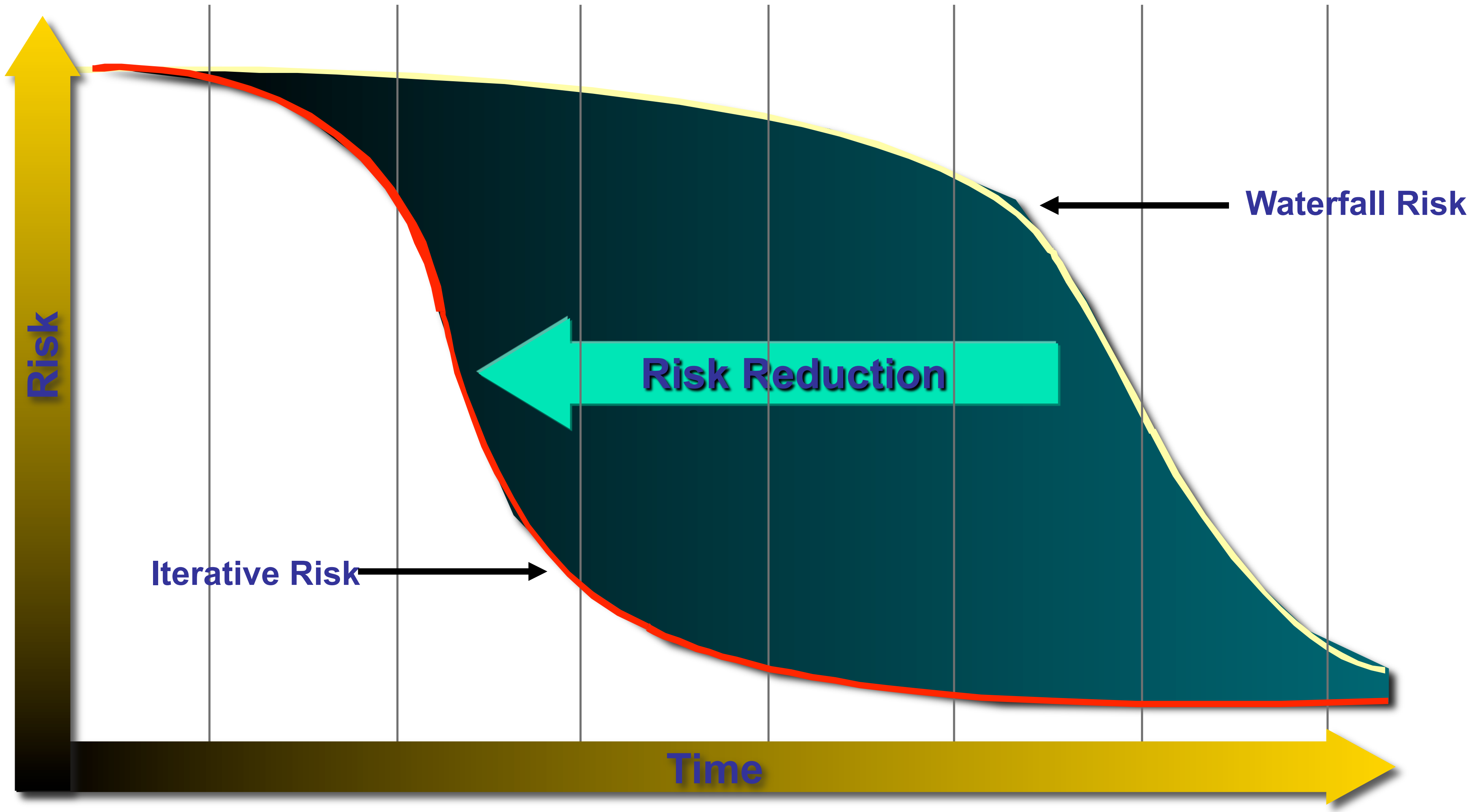
The spiral lifecycle



A process for HPC [Lugato 2010]



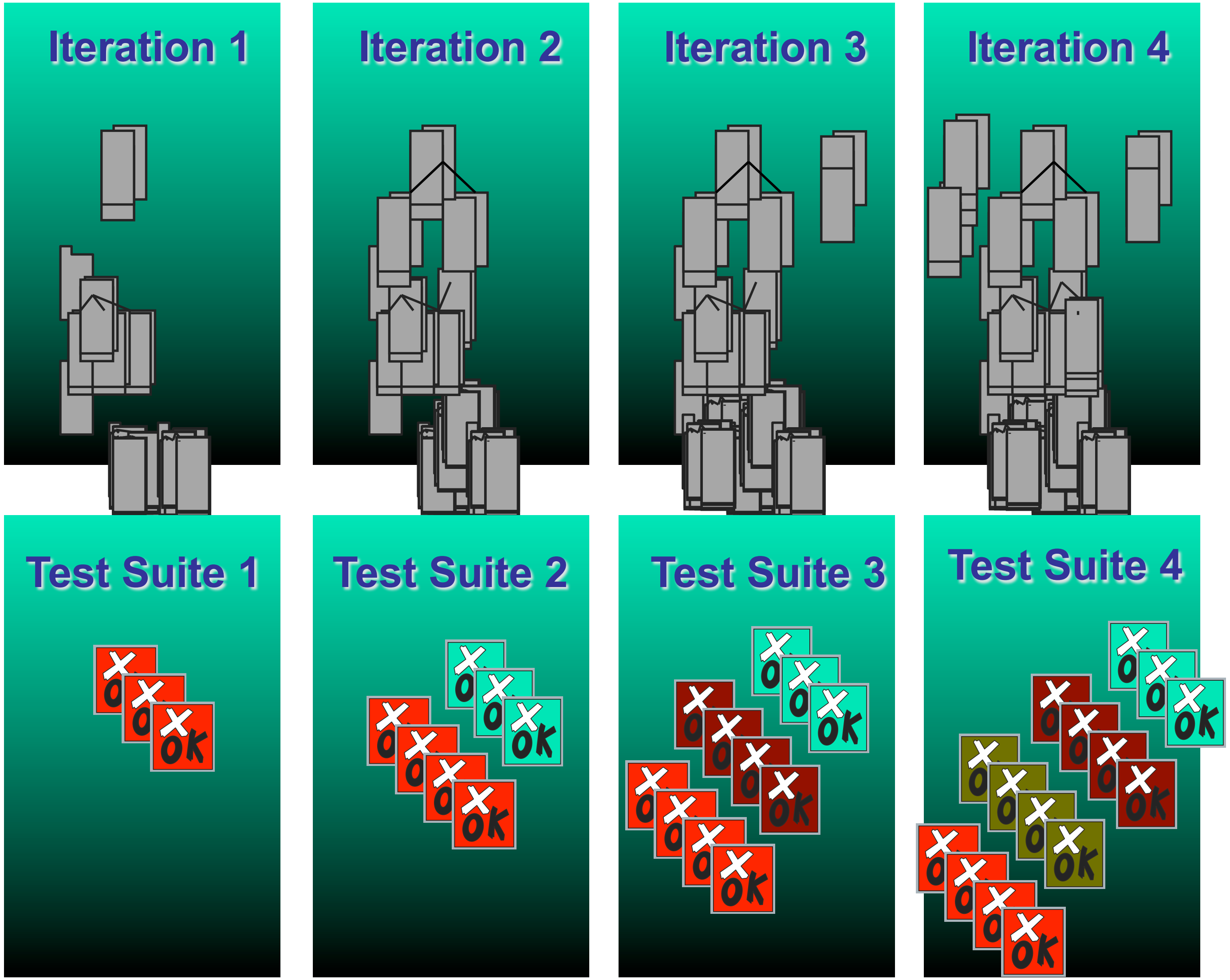
Risk: waterfall vs iterative



Test each iteration

Requirements, models and code

Tests



Testing before designing

- What is software testing? an investigation conducted to provide information about the quality of some software product
- In planned process models testing happens after the coding, and checks if the code satisfies the requirements
- What happens if we define the tests before the code they have to investigate?

Agile development processes

- There are many agile development methods; most minimize risk by developing software in short amounts of time
- The requirements are initially grouped in stories and scenarios
- Then the tests for each scenario are agreed with the user, before any code is written
- Each code is tested against its scenario tests, and integrated after it passes its unit tests

Agile ethics

- www.agilemanifesto.org

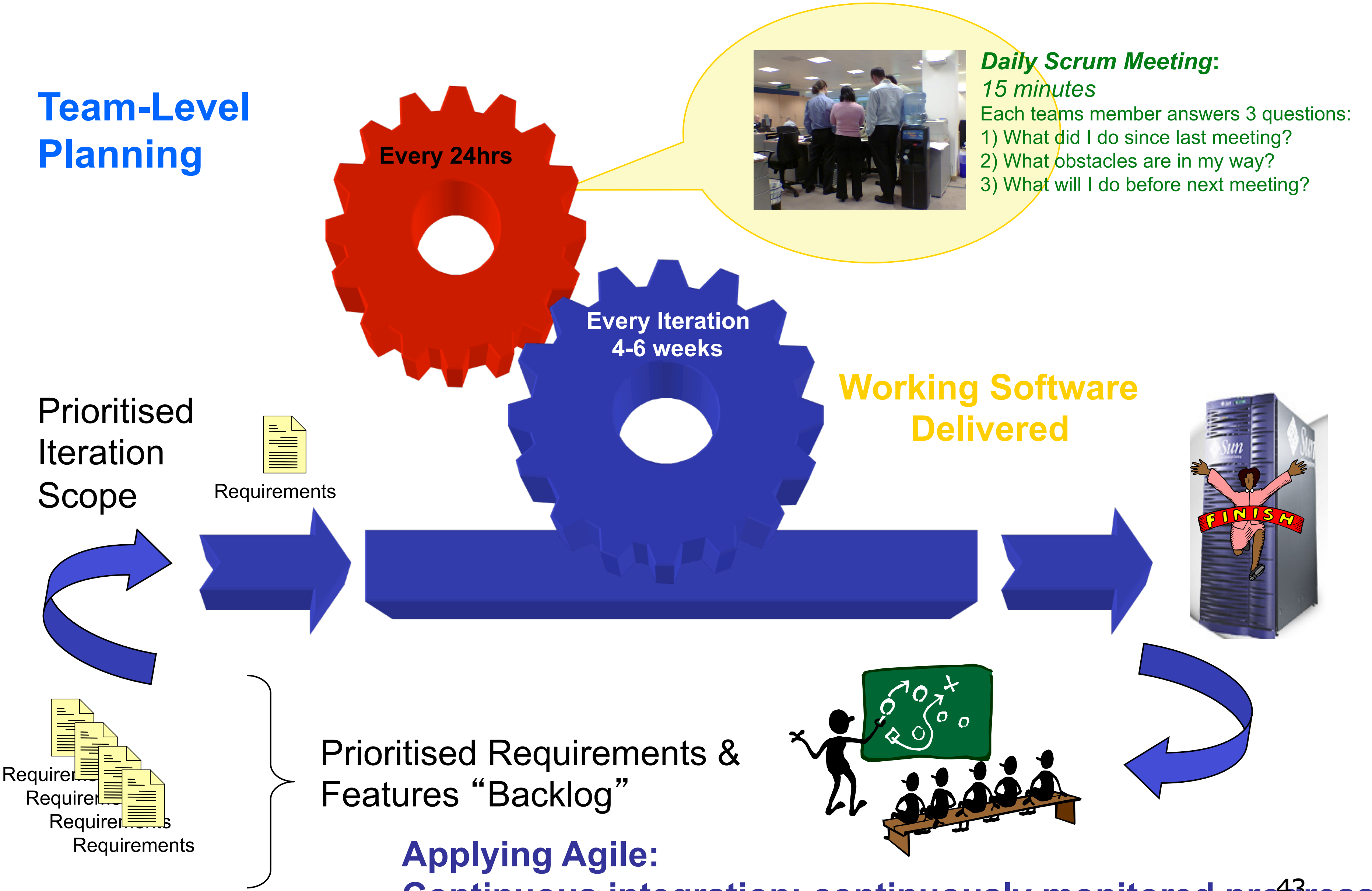
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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 prefer the items on the left.

- Management can tend to prefer the things on the right over the things on the left

SCRUM



Applying Agile:
Continuous integration; continuously monitored progress

Roadmap

- What is Software Engineering?
- The Software Development Lifecycle
- **Software Development Activities**
- Methods and tools

Requirements collection

User requirements are often expressed *informally*:

- They are grouped in *features*
- They are put in context in usage scenarios

Even if requirements are documented in written form, they may be *incomplete*, *ambiguous*, or *incorrect*

Changing requirements

Requirements *will* change!

- *inadequately captured* or expressed in the first place
- user and business *needs may change* during the project

Validation is needed *throughout* the software lifecycle, not only when the “final system” is delivered!

- build constant *feedback* into your project plan
- plan for *change*
- early *prototyping* [e.g., UI] can help clarify requirements

Requirements analysis

Analysis is the process of specifying *what* a system will do

- The goal is to provide an understanding of what the system is about and what its underlying concepts are

The result of analysis is a *specification document*

Does the requirements specification correspond to the users' actual needs?

Object-oriented analysis

An *object-oriented analysis* results in a **model** of the system which describes:

- *classes* of objects that exist in the system
  - *responsibilities* of those classes
- *relationships* between those classes
- *use cases* and *scenarios* describing
  - *operations* that can be performed on the system
  - allowable *sequences* of those operations

Design

Design is the process of specifying *how* the specified system behaviour will be realized from software components. The results are *architecture* and *detailed design documents*.

Object-oriented design delivers models that describe:

- how system operations are implemented by *interacting objects*
- how classes refer to one another and how they are related by *inheritance*
- *attributes* and *operations* associated to classes

Design is an iterative process, proceeding in parallel with implementation!

Prototyping

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

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

Implementation and testing

Implementation is the activity of *constructing* a software solution to the customer's requirements.

Testing is the process of *verifying* that the solution meets the requirements.

- The result of implementation and testing is a *fully documented* and *verified* solution.

Testing!

1

- Provide automated build process
  - Far easier & quicker to validate changes
  - e.g. Make, Ant, Maven

2

- Provide automated regression test suite - TDD
  - Do changes break anything?
  - JUnit, CPPUnit, xUnit, fUnit, …

3

- Join together: automated build & test
  - A 'fail-fast' environment

4

- Infrastructure support
  - Nightly builds – run build & test overnight, send reports
  - Continuous integration - run build & test when codebase changes

Towards *anytime releasable* code!

Iterativity of design, Implementation and testing

Design, implementation and testing are iterative activities

- The implementation does not “implement the design”, but rather the design document *documents the implementation*!

- System tests reflect the requirements specification
- Testing and implementation go hand-in-hand
  - Ideally, test case specification *precedes* design and implementation

Maintenance

Maintenance is the process of changing a system after it has been deployed.

- *Corrective maintenance*: identifying and repairing *defects*
- *Adaptive maintenance*: *adapting* the existing solution to new platforms
- *Perfective maintenance*: implementing *new requirements*
- *Preventive maintenance*: repairing a software product before it breaks

In a spiral lifecycle, everything after the delivery and deployment of the first prototype can be considered "maintenance"!

Maintenance activities

"Maintenance" entails:

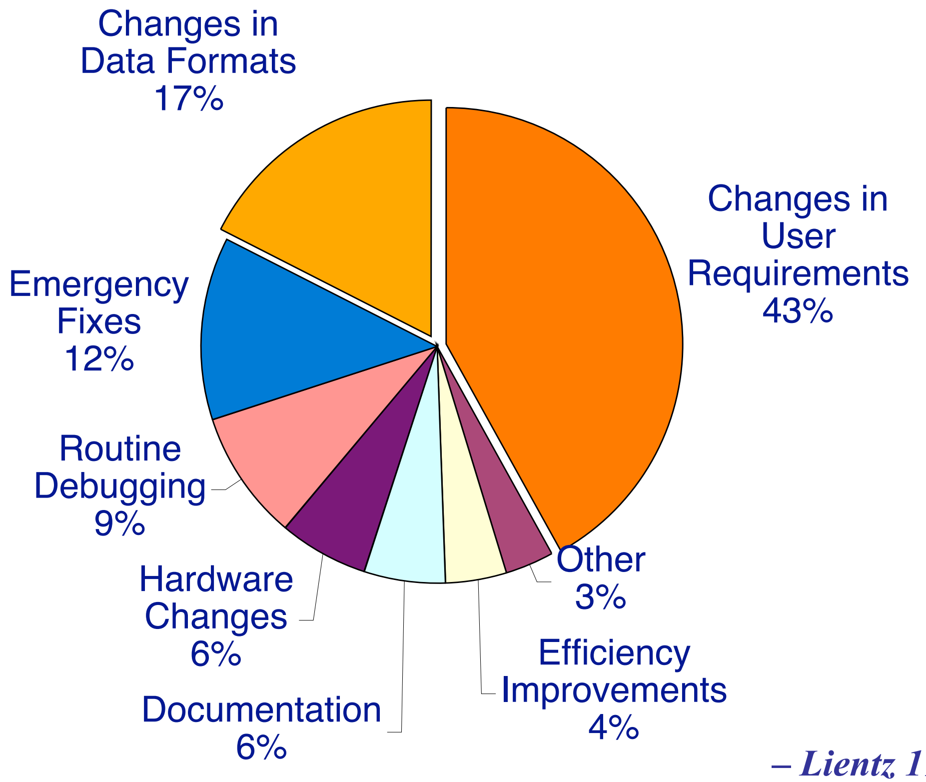
- configuration and version management
- reengineering (redesigning and refactoring)
- updating all analysis, design and user documentation

Repeatable, automated tests enable evolution and refactoring

Maintenance costs

"Maintenance"
typically accounts for
70% of software costs!

Means: **most project costs concern continued development *after* deployment**



– Lientz 1979

Deployment

- Virtual Machines
  - Software pre-installed, ready to run
  - Often easiest
  - Not enough in itself – documentation!
- Release software
  - Prioritise & select requirements -> Develop -> Test -> Commit changes to repository -> Test -> Release
  - Documentation (minimum: quick start guide)
- Licencing
  - Specify rights for using, modifying and redistributing

Configuration management

- Run your own CM system, if you have the resources
  - Generally easy to set up
  - Full control, but be sure to back it up!
- Some public solutions can offer most of these for free
  - SourceForge, GoogleCode, GitHub, Codeplex, Launchpad, Assembla, Savannah, …
  - BitBucket for private code base (under 5 users)
  - See (for hosting code and related tools) http://software.ac.uk/resources/guides/choosing-repository-your-software-project
  - See (for hosted continuous integration) http://www.software.ac.uk/blog/2012-08-09-hosted-continuous-integration-delivering-infrastructure

"If you're not using version control, whatever else you might be doing with a computer, it's not science" – Greg Wilson, Software Carpentry

Conclusions

Software engineering deals with

- the way in which software is made (process),
- the languages to model and implement software,
- the tools that are used, and
- the quality of the result (testing)

Self test questions

- How does Software Engineering differ from programming?
- Why is the “waterfall” model unrealistic?
- What is the difference between analysis and design?
- Why plan to iterate? Why develop incrementally?
- Why is programming only a small part of the cost of a “real” software project?

References: books

- Pressman, *Software engineering a practictioner approach*, 7th ed., McGrawHill, 2009
- Larman, *Agile and Iterative Development: a managers' guide*, Addison Wesley, 2003
- The Computer Society, *Guide to the Software Engineering Body of Knowledge*, 2013 www.computer.org/portal/web/swebok

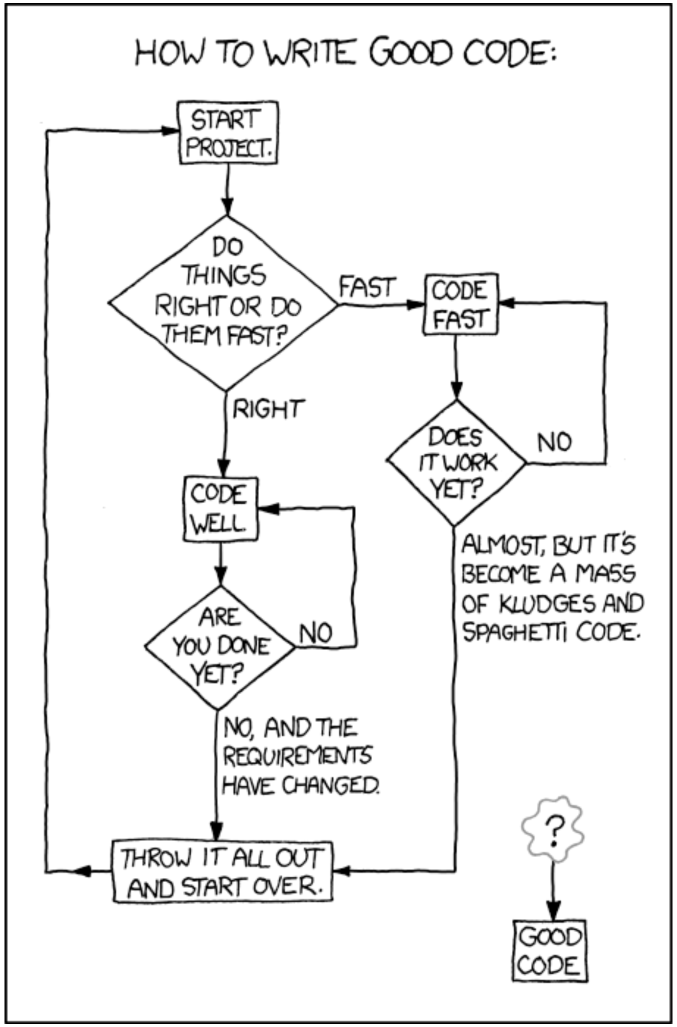
Reference: papers

- V.Basili et al., Understanding the High-Performance- Computing Community: A Software Engineer’s Perspective, IEEE Software, 2008
- D.Lugato et al., Model-driven engineering for HPC applications, *Proc. Modeling Simulation and Optimization Focus on Applications, Acta Press* (2010): 303-308.
- M.Palyart et al, MDE4HPC: An Approach for Using Model-Driven Engineering in High-Performance Computing, Proc. SDL, LNCS 7083, 2011.

Useful sites

- `software-carpentry.org` Software carpentry
- `software.ac.uk/resources/case-studies`
- Int. Workshop on Sw eng for HPC, 2013 etc.
  `sehpccse13.cs.ua.edu`

Questions?



http://xkcd.com/844/