consorzio
interuniversitario
nazionale
per l'informatica

CINECA SCAI
SuperComputing Applications and Innovation

Software engineering for scientists

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

25th Summer School
on Parallel Computing

Bologna May 15 2017

Agenda

- Engineering software for science
- The software development lifecycle
- Sw development best practices in HPC

Making software for science

- The software is not the end goal in itself
- Performance is a key quality (HPC)
- Time limitations
- Target audience are expert
- Coders often from research domains
- Individual “ownership” of software products common

Why a lecture on Software Engineering?

Although productivity and quality of software crafting for HPC is on demand, not much interest from the “scientific” user community in SwEng. Why?

- The goal is to publish results. Not productivity or quality. (computational scientists vs. software engineers).
- In HPC increasing performance is a goal, other qualities are less important
- Productivity is not really measured
- Quality of research products is not measured at all

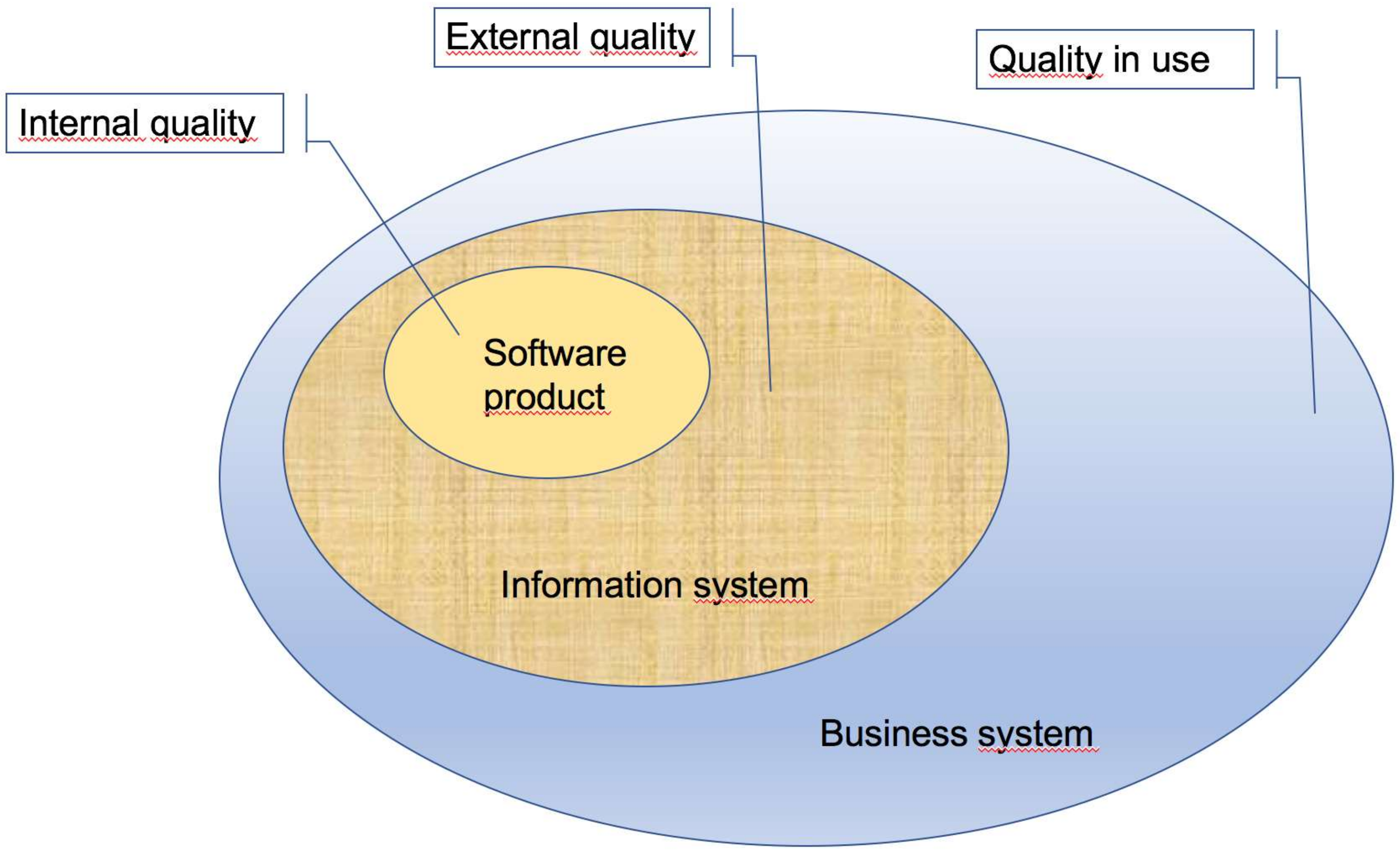
Software Engineering for HPC

- SwEng deals with designing, implementing, and modifying software so that it is faster to build, of higher quality, more usable and maintainable
- In HPC we have all the problems of software development, plus the specific problem that developers have little knowledge of software engineering best practices
- In this lecture we will deal with some of these problems and suggest some solutions

Targets of sw engineering

- The quality of the software products
- The process by which quality products are built
- Tools and practices to obtain quality products

Software qualities



What is software quality?

- Software *functional quality* (internal) reflects how well it complies with or conforms to a given specification, based on some functional requirements
- Software *structural quality* (external) refers to non-functional qualities, like performance or cost
- Software *quality in use* refers to how well it satisfies actual users' needs

IMPORTANT: Software qualities can be measured!

Do you agree?

- Scientists want to repeatedly tweak queries and analyses on their data sets and get immediate feedback. As long as you can clearly explain what you did to get the results published in a scientific paper, you don't need to pursue the code any further
- As science becomes increasingly computational in nature, it will become more important that scientific code does not end its **development cycle** on publication of the paper.
- Data exploration drives primary scientific discovery, but in order for future scientists to fully leverage the work of their predecessors, **robust**, **reproducible**, and **sustainable** software is needed to automate the parts we already know how to do

Software qualities: examples

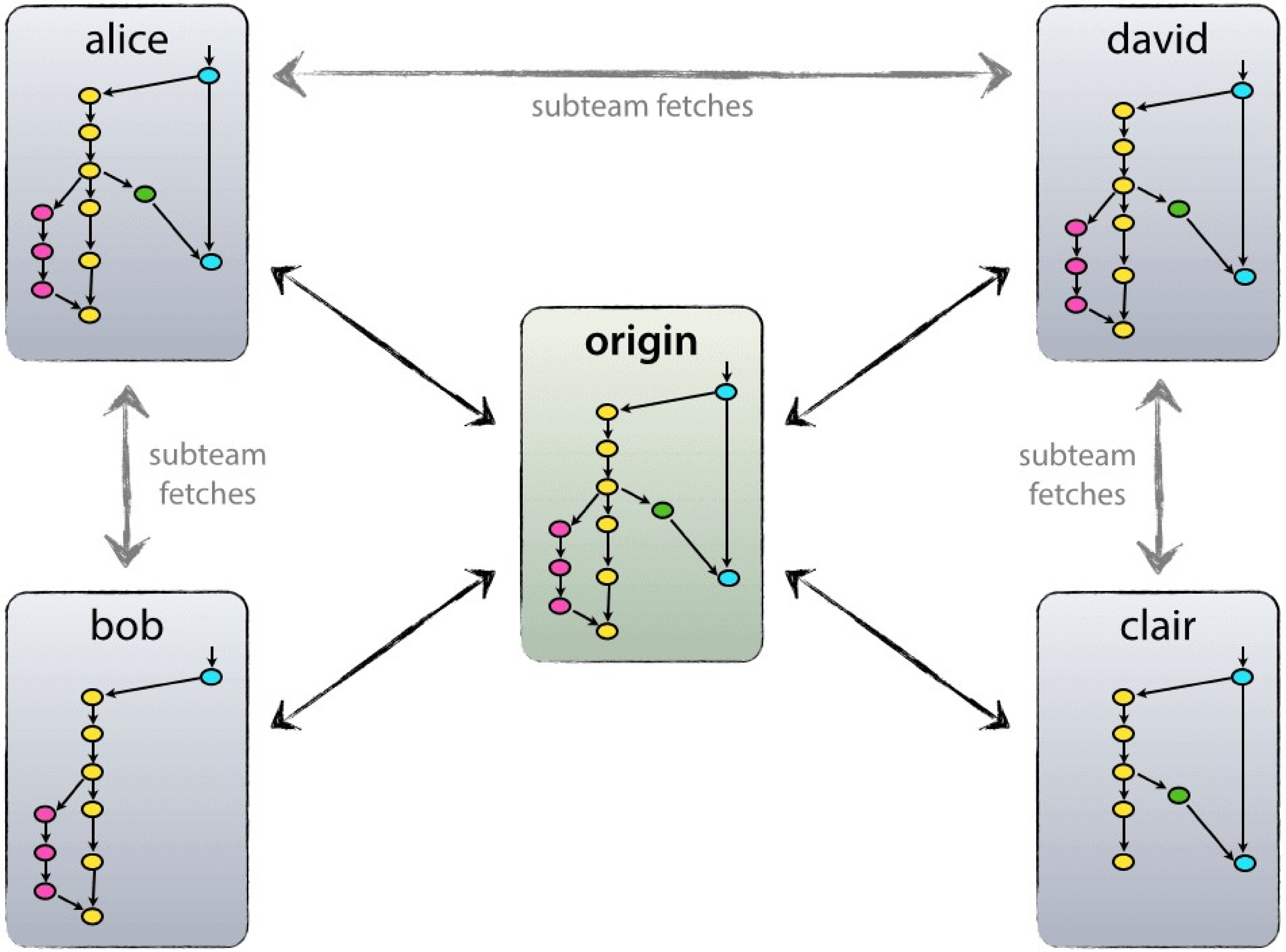
- **Robust** software: able to cope with errors during execution
- **Reproducible** software: version control
- **Sustainable** software: long lasting software able to cope with changes

Reproducibility

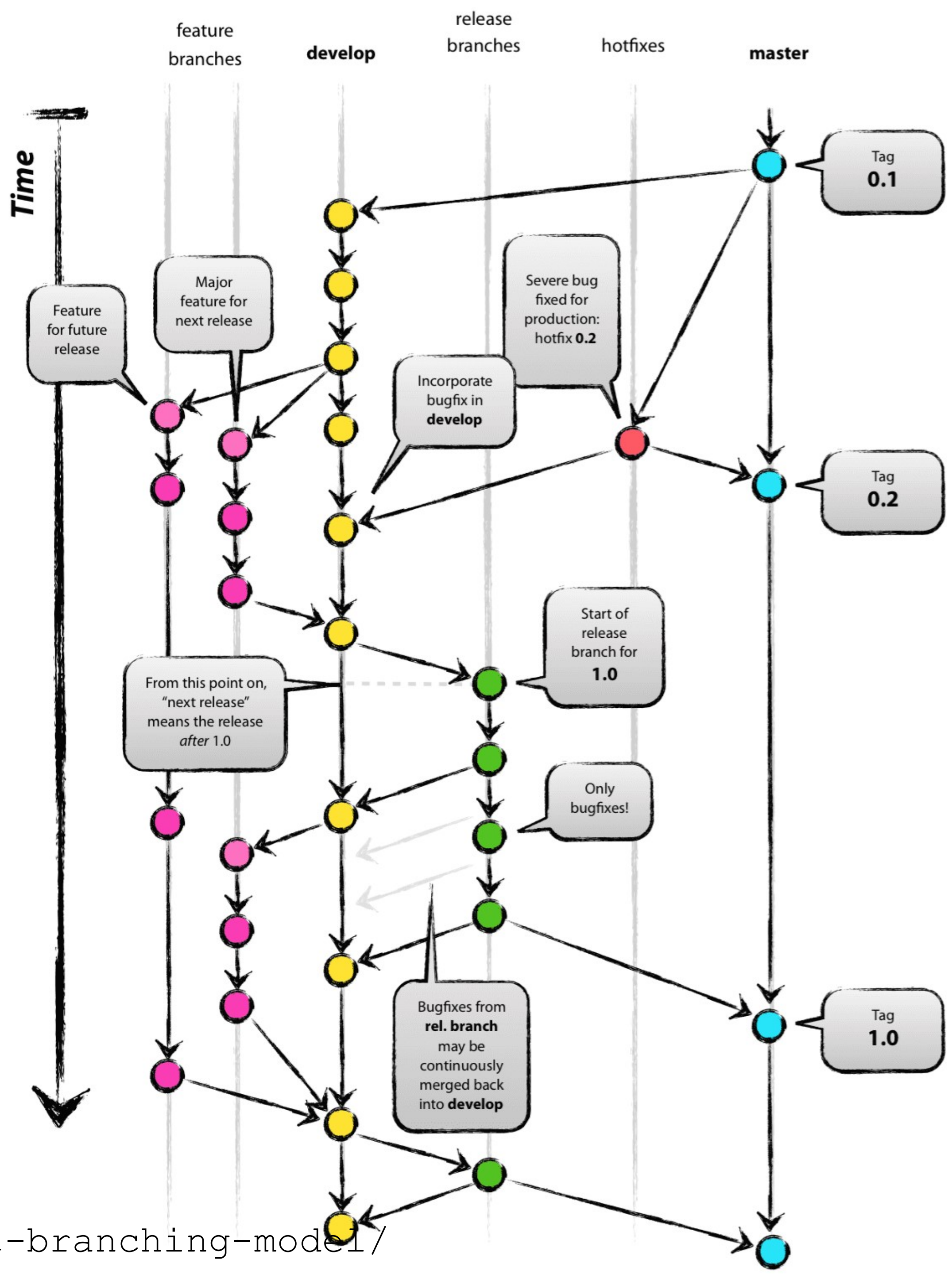
- Tracking and recreating every step of your work – including testing
- Git: an enabling tool – use version control for everything
  - Paper writing
  - Grant writing
  - Everyday research
- Advantages:
  - A time machine view tracking every result
  - Distributed backup
  - Collaborate with collegues

Git

- Efficient distributed version control system
  - Advanced branching mechanisms
- Many hosting services available online
- GitHub (github.com)
  - Hosting service
  - Developers community
  - Web-based interface
  - Access control
  - Collaboration features (including wikis, etc.)


alice
david
subteam fetches
origin
subteam
fetches
subteam
fetches
bob
clair

Git flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix **0.2**
Major feature for next release
Feature for future release
Incorporate bugfix in **develop**
Tag 0.2
Start of release branch for **1.0**
From this point on, "next release" means the release *after* 1.0
Only bugfixes!
Bugfixes from **rel. branch** may be continuously merged back into **develop**
Tag 1.0

hnvie.com/posts/a-successful-git-branching-model/

Reproducibility of experiments based on sw

- Software publications coming of age
- Example: SoftwareX (Elsevier)
- Your software should be be available, credited, versioned, licensed, citable

www.elsevier.com/authors/author-services/research-elements/software-articles/original-software-publications

Example: a study about how software is mentioned in scientific papers

- Paper: Howison & Bullard, Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature, JAIST 67:9, 2016
- Random sample of 90 biology articles
- Software is formally cited only 40% of times
- Software is frequently inaccessible (15%–29% of packages in any form; between 90% and 98% of specific versions; only between 24-40% provide source code).
- Cites to publications are poor at providing version information, whereas informal mentions are poor at providing crediting information.

Software development: typical problems

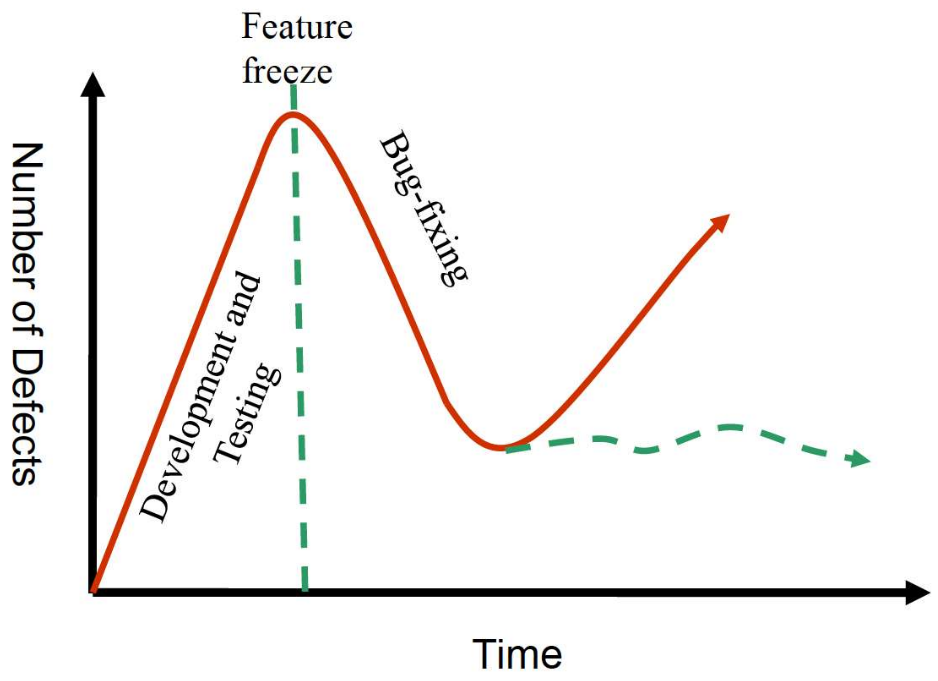
- Unacceptable software performance
- Software hard to maintain or extend
- Inaccurate understanding of user needs
- Inability to deal with changing needs (requirements)
- Late discovery of serious flaws

→ **poor software quality**

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



Poor practices for sw development

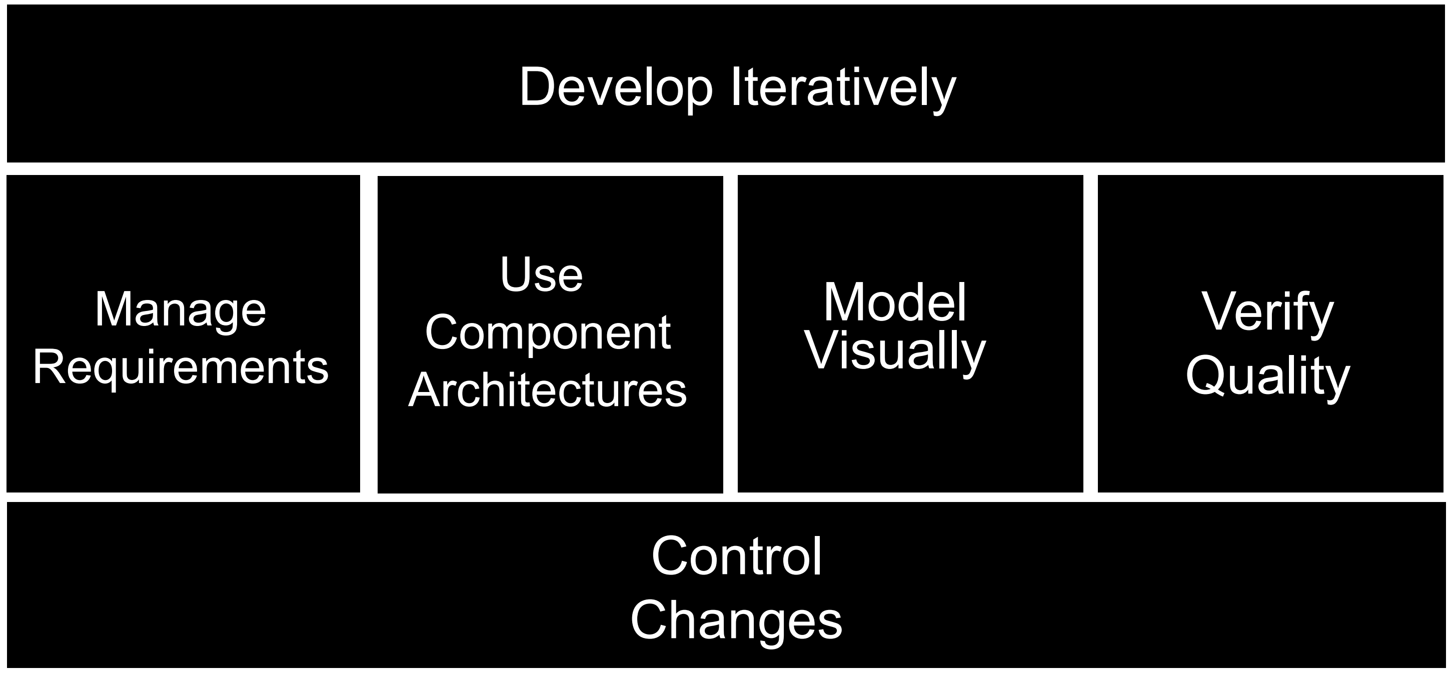
- Under-evaluation of development risks
- Overwhelming complexity
- Ambiguous communication
- Insufficient testing
- Insufficient requirements management
- Inconsistencies among requirements, designs, implementations, and tests
- Fragile software architecture

Best practices of software development

- Develop iteratively
- Control changes

- Manage requirements
- Verify quality
- Use components
- Model software architecture visually

Best practices of software development



Know these!

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
  - Changes to the software 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

Challenges

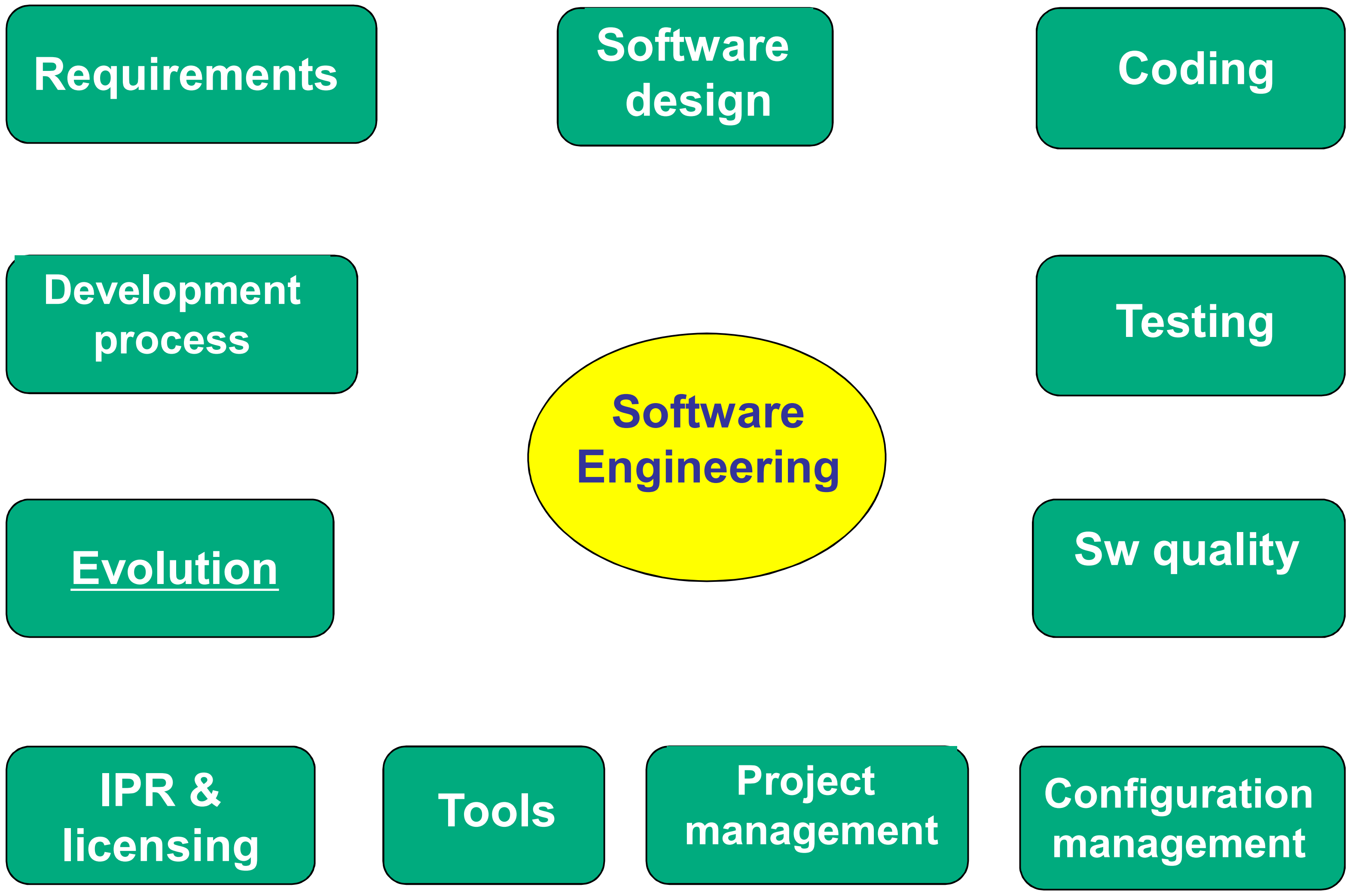
- Technical
  - All parts of the cycle can be under research
  - Needs change throughout the lifecycle as knowledge grows
  - Verification complicated by floating point representation
- Sociological
  - Real world is messy, so is the software
  - Need for interdisciplinary interactions
  - Competing priorities and incentives
  - Limited resources
  - Perception of overhead without benefit

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
IPR & licensing
Tools
Project management
Configuration management

Topics of the discipline

- Product and process standards for software
- Project management for software systems
- Software development models: planned vs agile
- Requirement analysis
- Software design by visual modeling
- Measuring, verifying, and ensuring software quality
- Software evolution and maintenance
- Typical tools used by software engineers:
    - Version control, configuration management

Some international standards on sw

- ISO/IEC 6592:2000, Guidelines for the documentation of computer-based application systems
- ISO/IEC 9126:1991, Product quality characteristics
- ISO 9127:1988, User documentation and cover information for consumer software packages
- ISO/IEC TR 9294:1990, Management of software documentation
- ISO/IEC 12119:1994, Software packages: Quality requirements and testing
- ISO/IEC TR 12182:1998, Categorization of software
- ISO/IEC 12207:1995, Software life cycle processes
- ISO/IEC 14143-1:1998, Functional size measurement

Roadmap

- Engineering software for science
- **The software development lifecycle**
- Sw development best practices in HPC

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

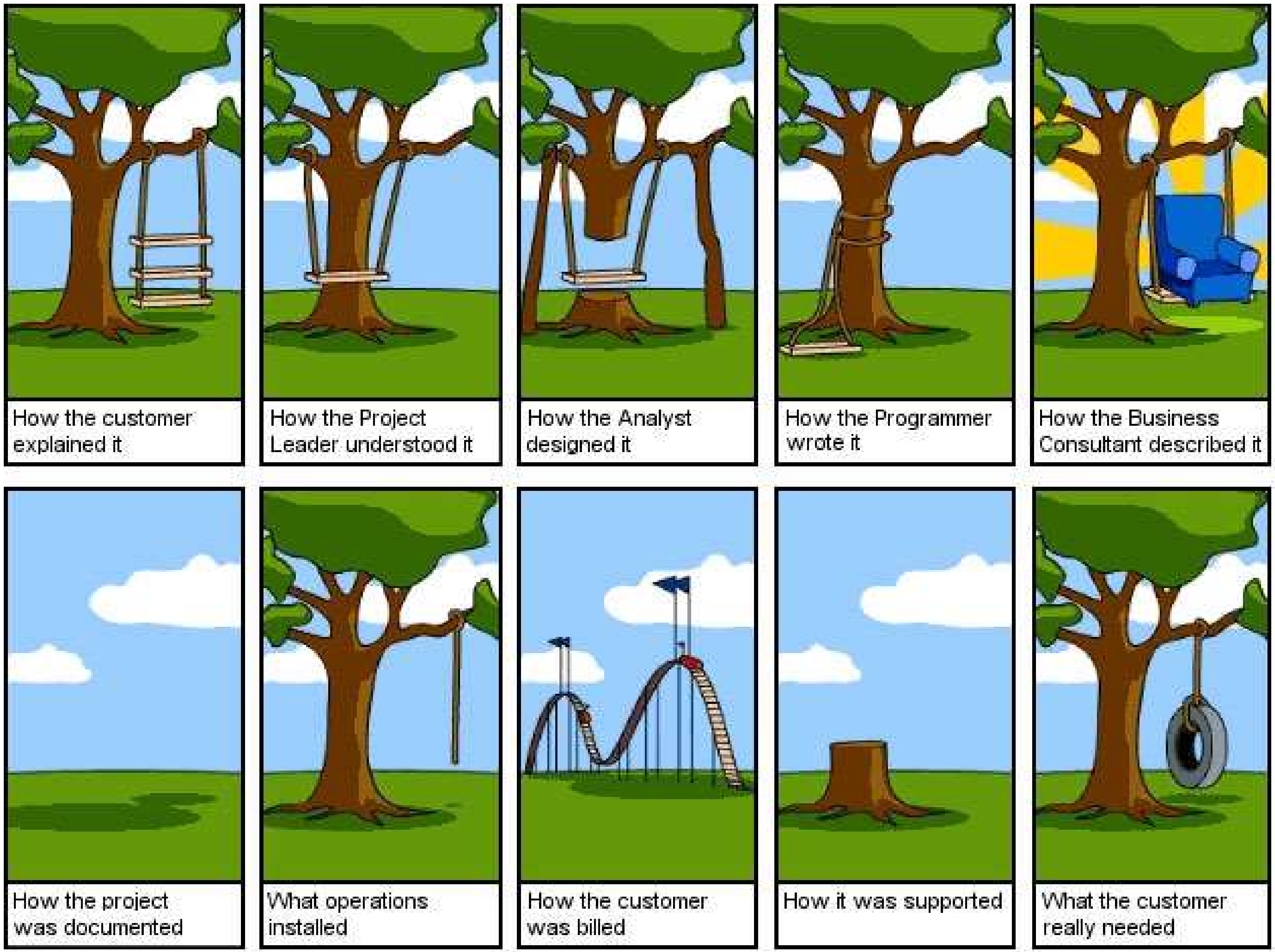
Stakeholders

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

The software development process

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

Activities

- Each organization differs in the 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!

Traditional approach

Sw development is a sequence including the following phases:

- Requirements Analysis
- Design
- Coding
- Testing: first check the units, then the system

The entire development process goes through these phases **linearly**: first all the requirements are defined, then the design is completed, and finally the code is written and tested.

The key assumptions are that when design begins, requirements no longer change. When coding starts, the design ceases to change. etc.

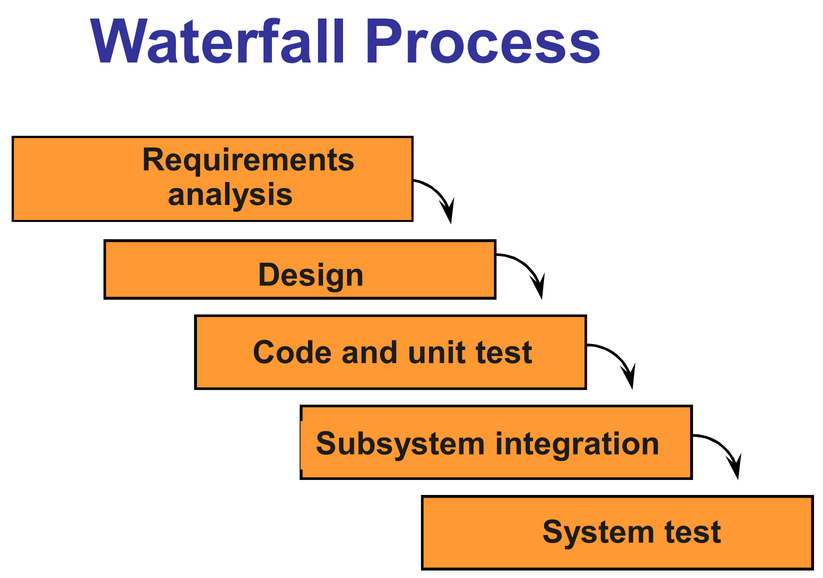
NB: This “traditional” approach is sometimes called "waterfall development”

Models for the software process

A model for the sofware development process is a method to describe the roles, the tasks, and the documents to be developed

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

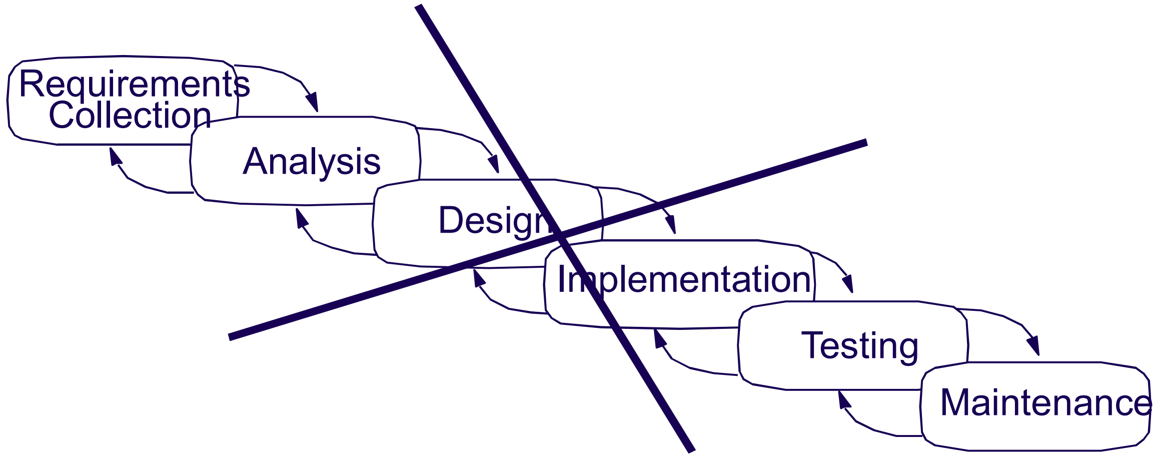
Waterfall characteristics



- 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"

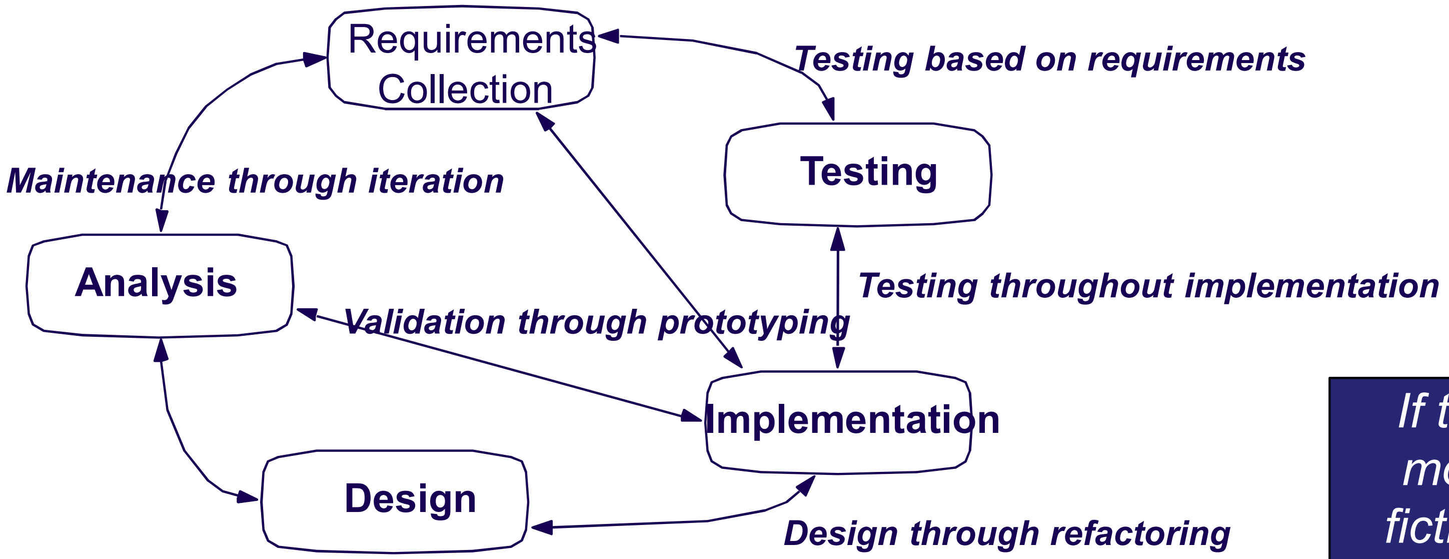
2. "It is often *difficult* for the customer *to state all requirements* explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects."

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


Requirements Collection
Testing based on requirements
Testing
Maintenance through iteration
Analysis
Testing throughout implementation
Validation through prototyping
Implementation
Design
Design through refactoring

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

Iterative development

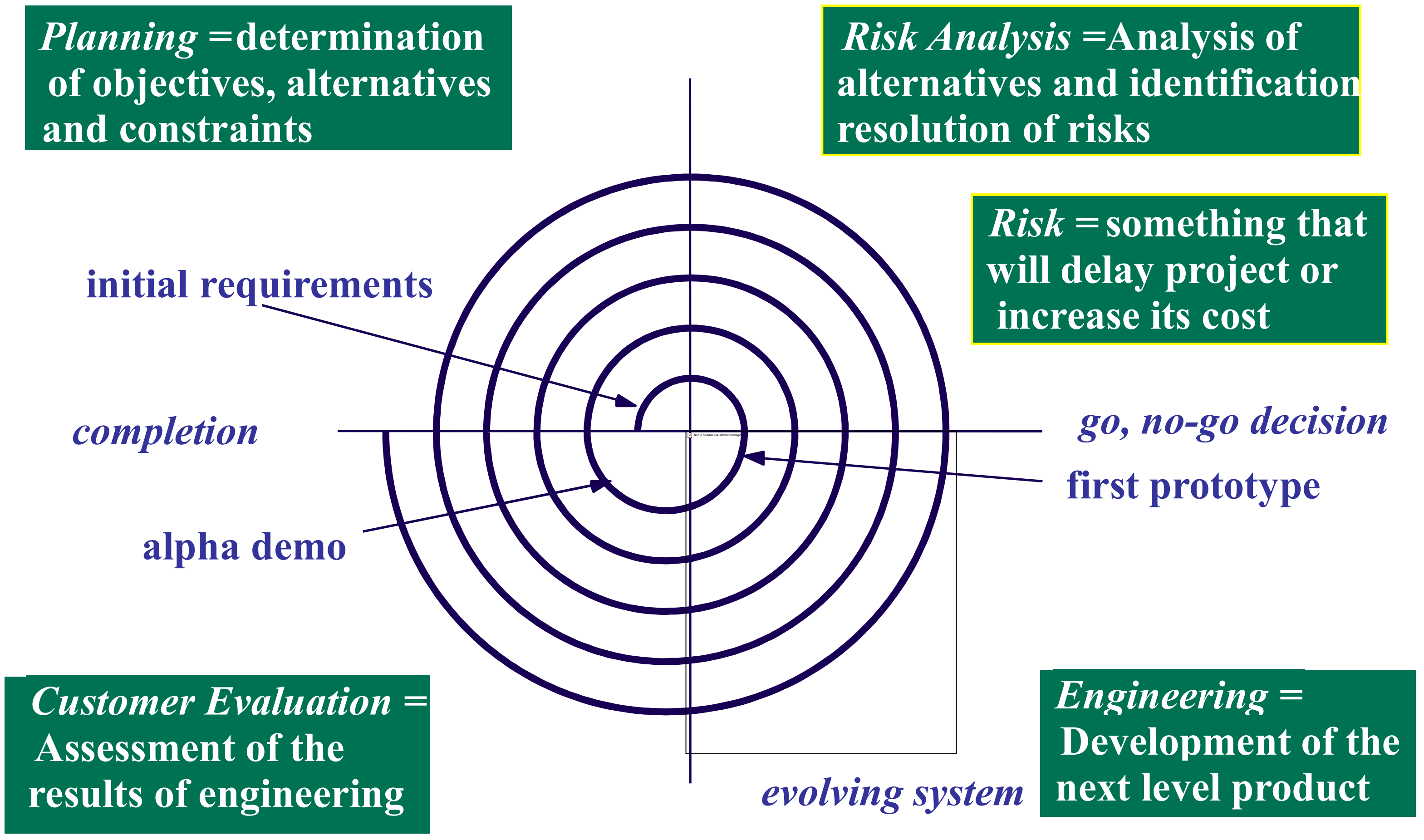
- Plan to iterate your analysis, design and implementation
  - You will not get it right the first time, so integrate, validate and test as frequently as possible
  - During software development, more than one iteration of the software development cycle may be in progress at the same time
  - This process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

Incremental development

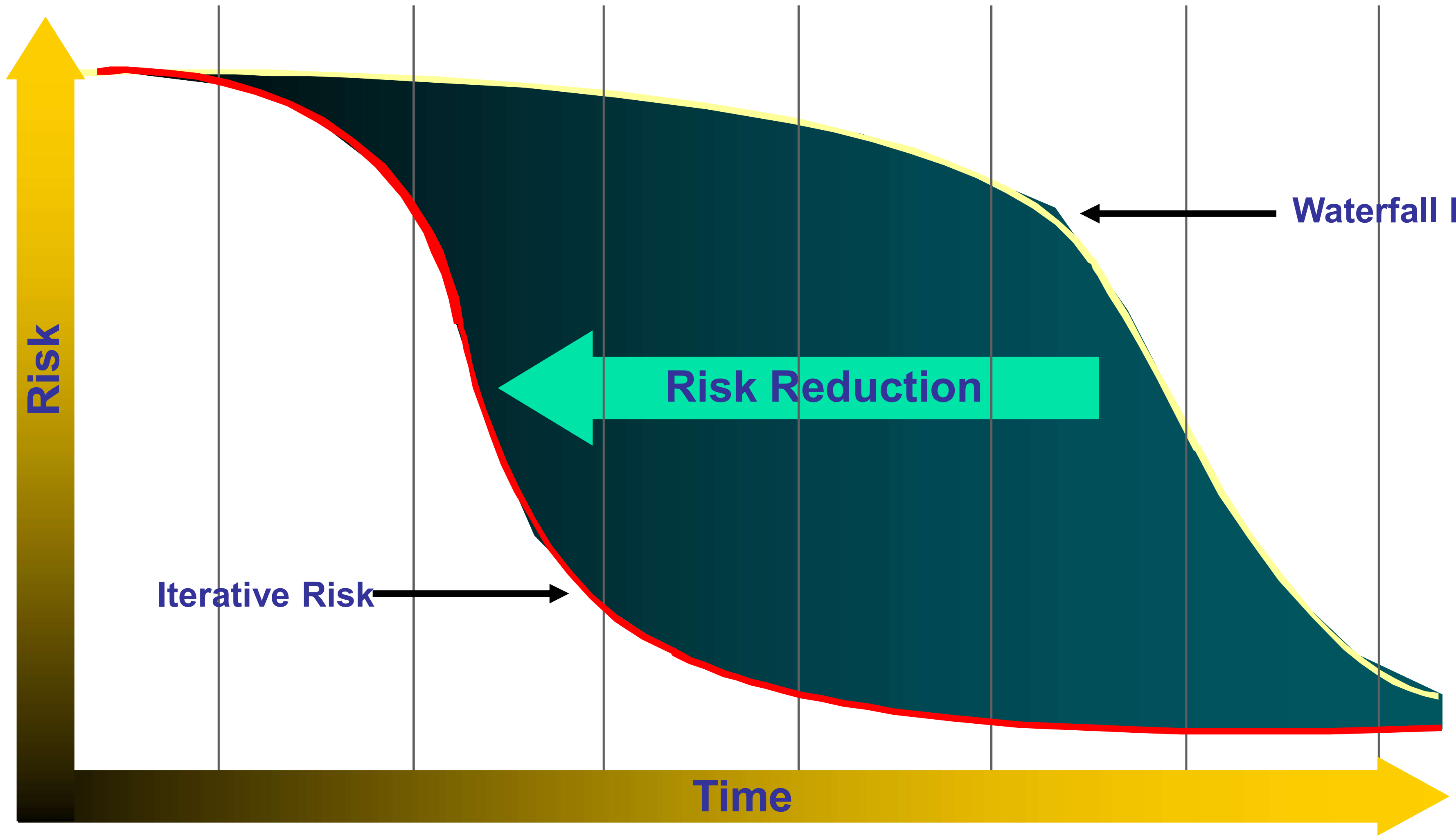
Plan to *incrementally* develop (i.e., prototype) the system

- 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


Planning =determination of objectives, alternatives and constraints
Risk Analysis =Analysis of alternatives and identification resolution of risks
Risk = something that will delay project or increase its cost
initial requirements
completion
go, no-go decision
first prototype
alpha demo
Customer Evaluation = Assessment of the results of engineering
evolving system
Engineering = Development of the next level product

Risk: waterfall vs iterative



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

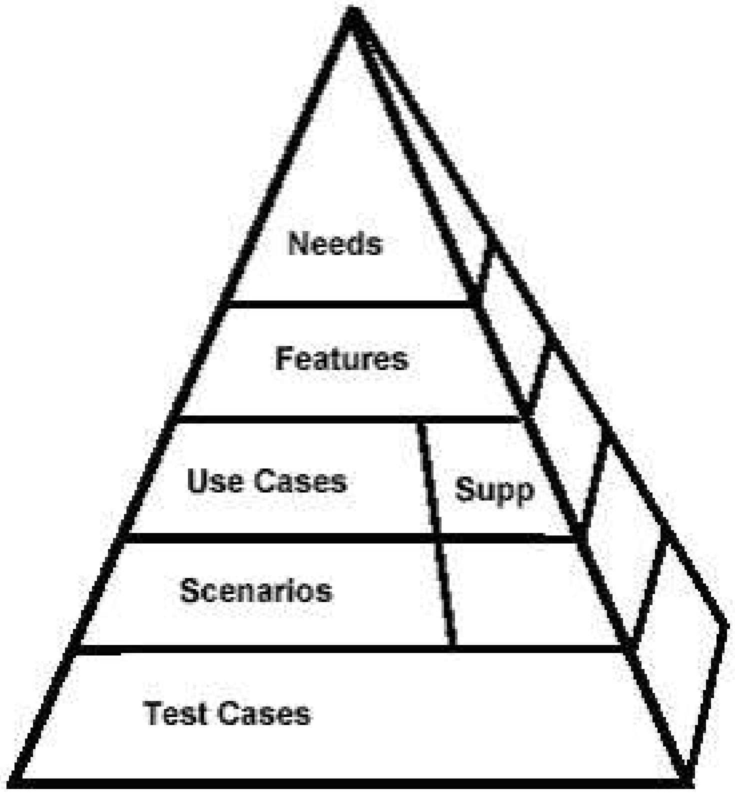
The requirements pyramid

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

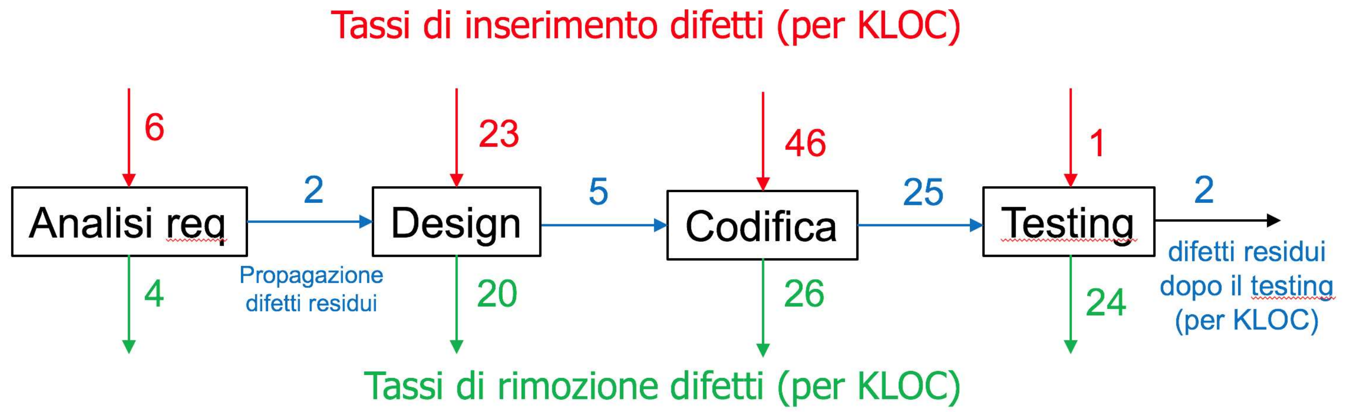
Features and requirements can be aggregated in “scenarios”: after the code is built, testing it in the scenario will prove that its features satisfy the user’s needs



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

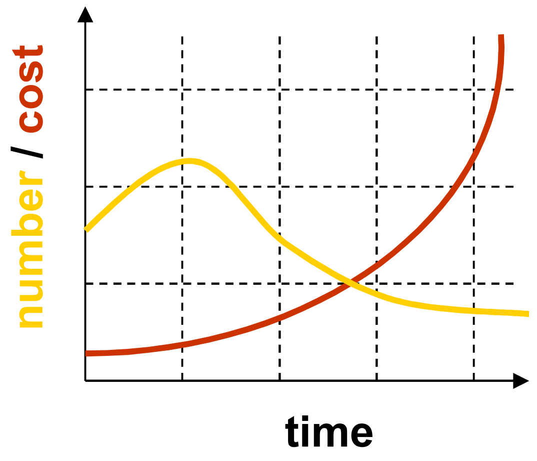
Beware your requirements

- Most errors are introduced during requirements analysis and design



Tassi di inserimento difetti (per KLOC)
6
23
46
1
Analisi req
2
Design
5
Codifica
25
Testing
2
Propagazione difetti residui
difetti residui dopo il testing (per KLOC)
4
20
26
24
Tassi di rimozione difetti (per KLOC)

- The later an error is detected, the more expensive it is to address
  - 1 hour to fix in the design
  - 10 hours to fix in the code
  - 100 hours to fix after it's gone live…



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

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!

Implementation and testing

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

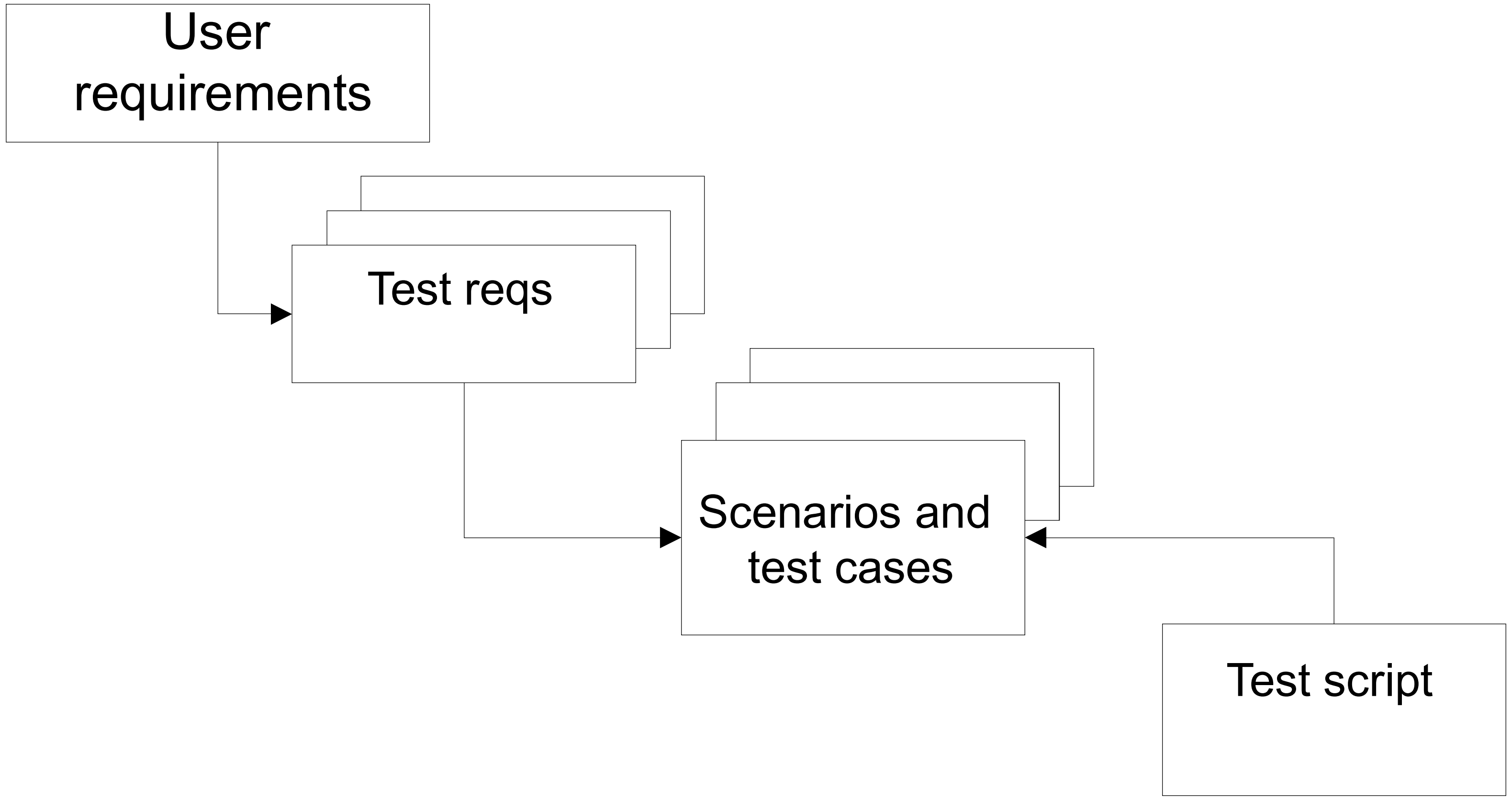
Testing is the process of *verifying* that the solution meets the requirements.

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

Invite others to read your code

- Rigorous inspections can remove 60-90% of errors before first test is run
  - Fagan (1975) “Design and Code Inspections to Reduce Errors in Program Development"
- The first review and hour matter most
  - Cohen (2006) “Best Kept Secrets of Peer Code Review”
- Develop code in small, readable, reviewable chunks

Requirements and tests

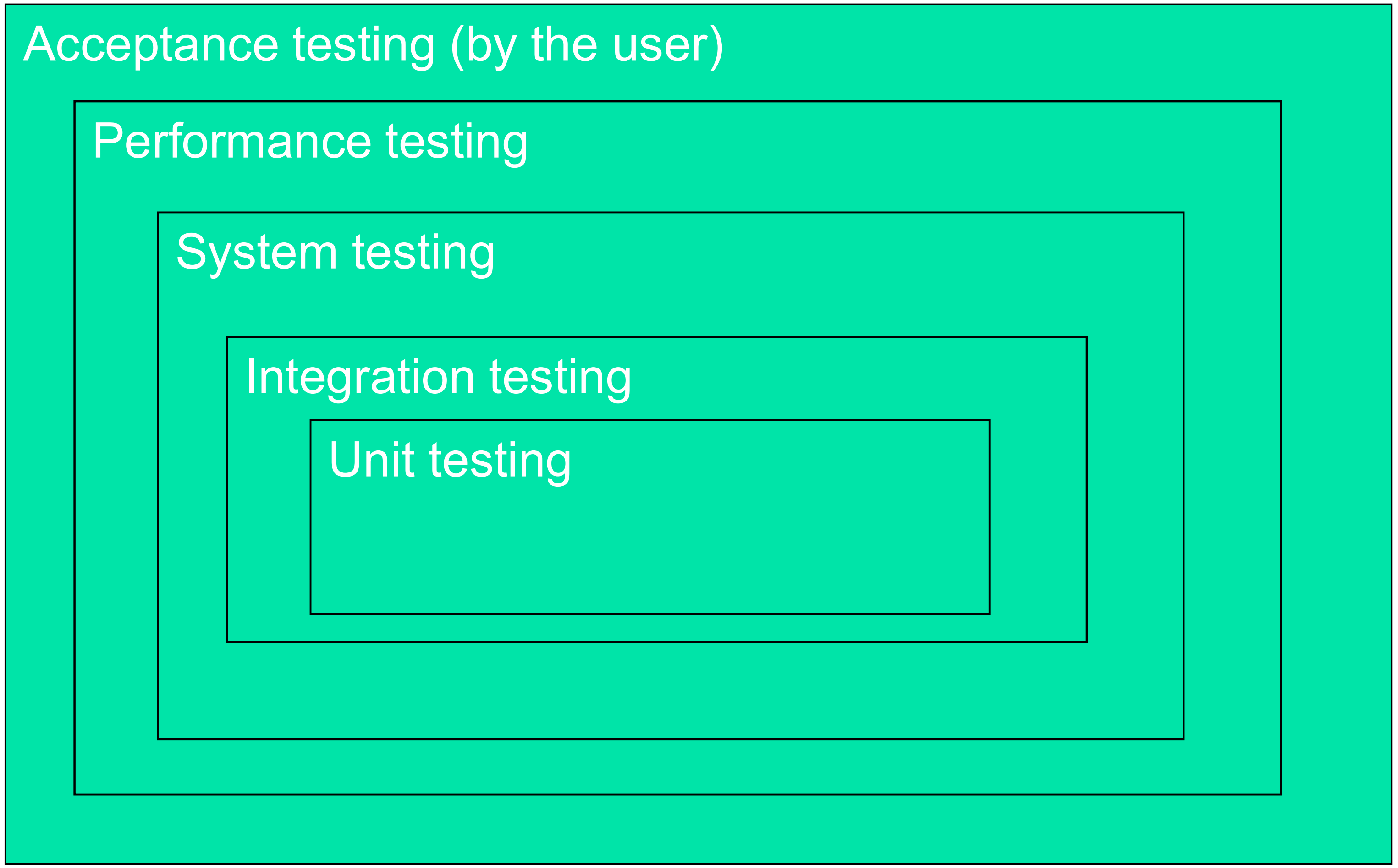


Errors tend to socialize

- About 80% of the defects come from 20% of the modules, while about half the modules are error free
  - Boehm and Basili (2001)

- When you identify more errors than expected in some program module, keep looking!

Types of testing



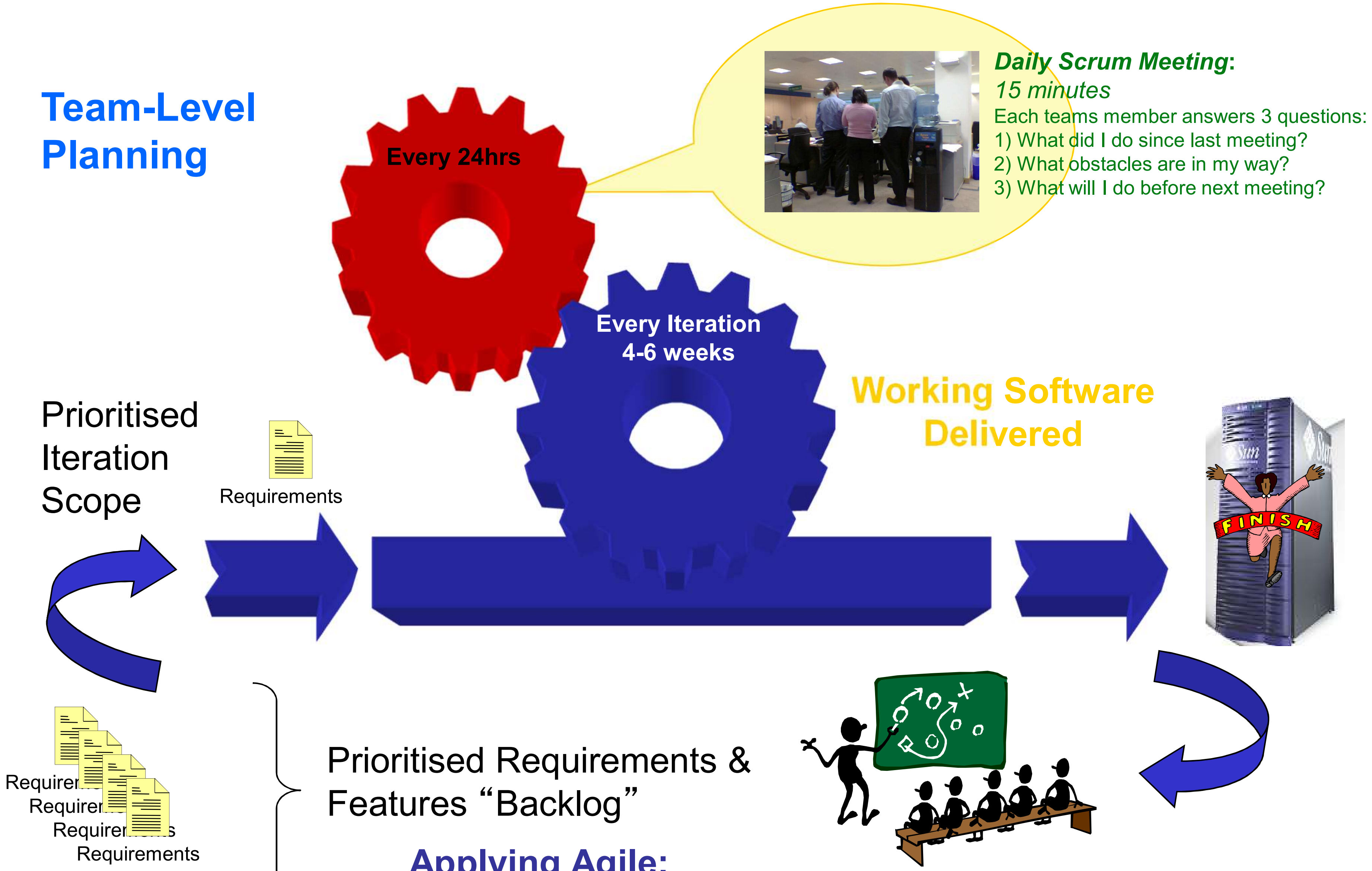
Testing before designing

- 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

SCRUM□



Daily Scrum Meeting:
15 minutes
Each teams member answers 3 questions:
1) What did I do since last meeting?
2) What obstacles are in my way?
3) What will I do before next meeting?

Applying Agile:
Continuous integration; continuously monitored progress

From Version Control to Continuous Integration

- When a new version is ready, a number of quality control activities can be executed
  - Testing
  - Analysis
  - …
- Why not executing them regularly?
  - Or after every commit?

Legacy code

- Code without tests is bad code.
- It does not matter how well written it is; it does not matter how pretty or structured or well-encapsulated it is.
- With tests, we can change the behavior of our code quickly and verifiably.
- Without them, we really don't know if our code is getting better or worse.

Build and test

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

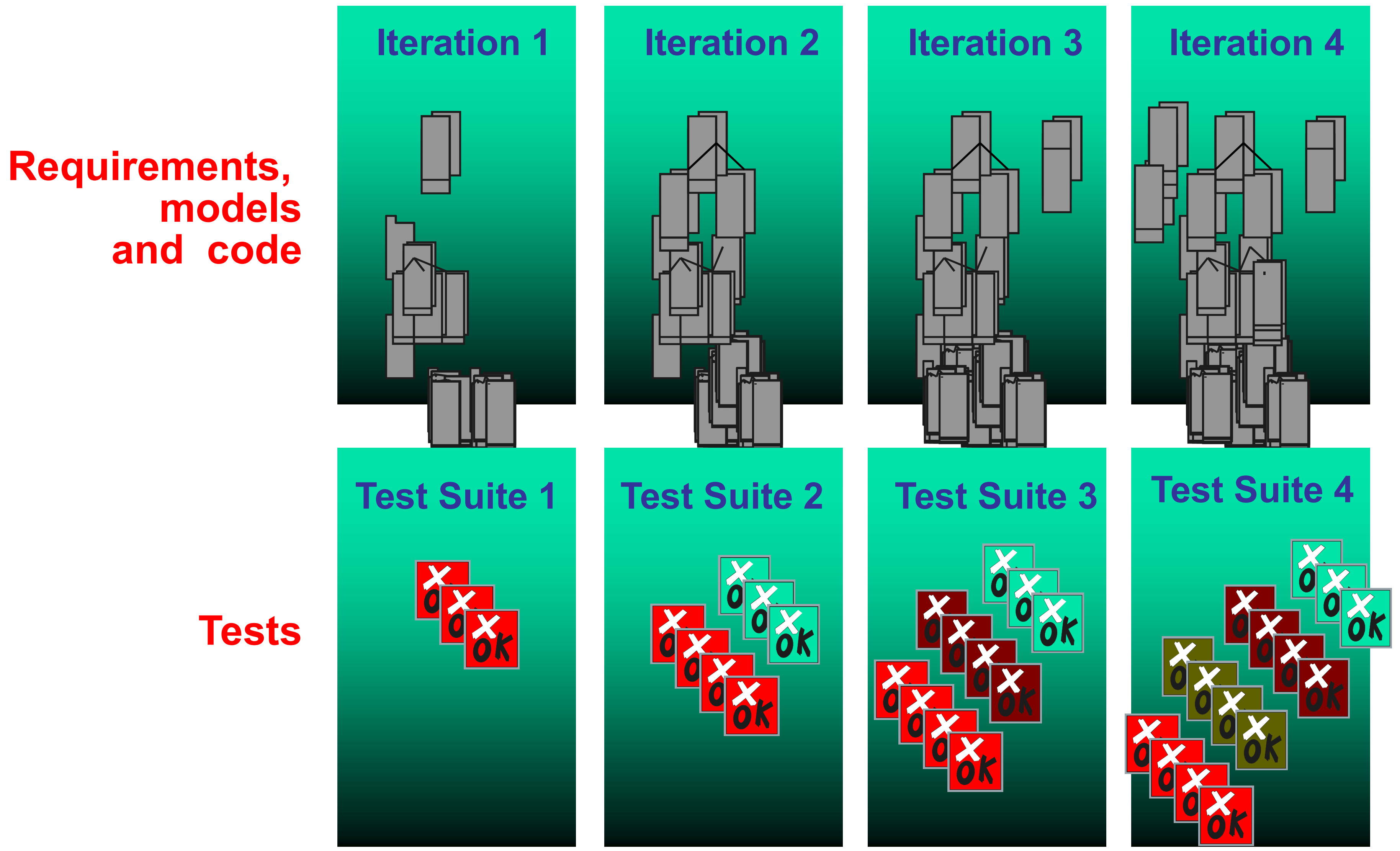
- Infrastructure support
  - Nightly builds – run build & test overnight, send reports
  - Continuous integration - run build & test when codebase changes

***anytime releasable* code!**

Example: a Continuous Integration Policy

- A <u>time</u> (say 5pm) for <u>delivery</u> of system components is agreed
- A new version of a system is <u>built</u> from these components by compiling and linking them
- The new version is <u>tested</u> using pre-defined tests
  - See the second part of the lecture for information about testing and analysis
- <u>Faults</u> that are discovered during testing time are documented and <u>returned to</u> the system <u>developers</u>

Test each iteration



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:

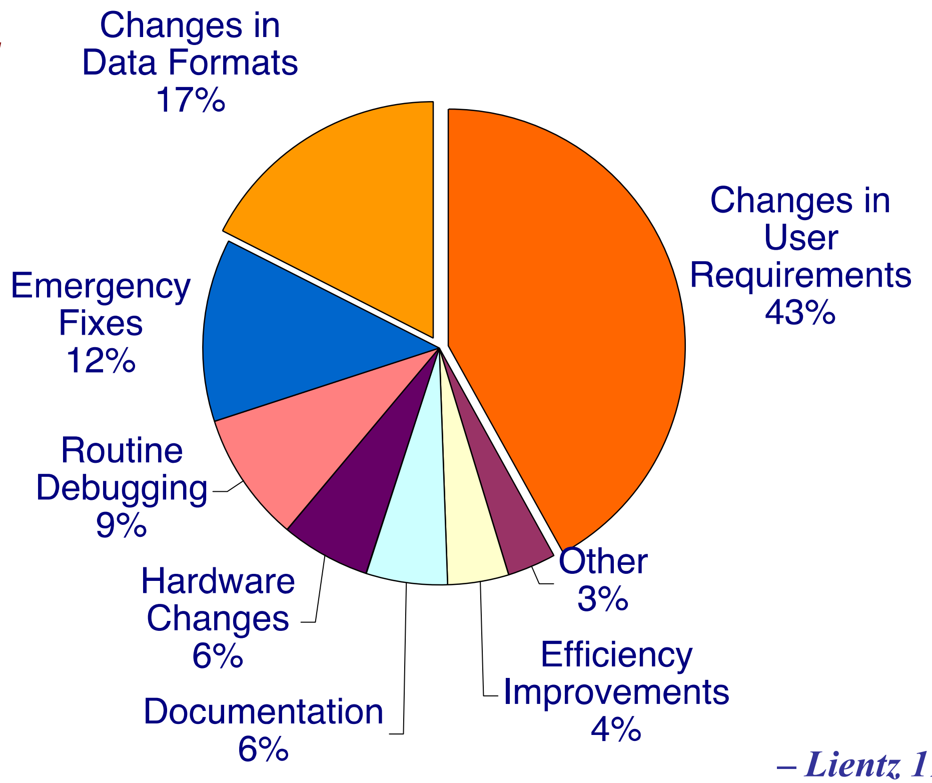
- configuration and version management
- reengineering (redesigning and refactoring)
- updating all analysis, design and user documentation

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

Roadmap

- Engineering software for science
- The software development lifecycle
- **Sw development best practices in HPC**

Eroic programming

- Spending huge amounts of (coding) effort by talented people to overcome shortcomings in software process, management, architecture configuration or any other shortfalls in the execution of a development project in order to complete it
- Heroic Programming is often the only course of action left when poor planning, insufficient funds, and impractical schedules leave a project stranded and unlikely to complete successfully

Best practices for scientific computing

- Write programs for people, not computers
- Use a tool to automate workflows
- Make incremental changes, use a version control system
- Try to reuse code instead of rewriting it
- Plan for finding mistakes
- Optimize software only after it works correctly
- Document design and purpose (not mechanics)
- Collaborate (eg. by pair programming or by using an issue tracking tool)

Wilson et al., Best Practices for Scientific Computing, PLOS Biology 2014

Sw development in HPC

- Waterfall is not used in HPC
  - Near the application level (tools - libraries)
  - Standard is test-driven development
  - Tight development loop: requirements, development and documentation, evaluation, test, deploy.
- Orientation towards minimized maintenance
  - Thorough testing: unit, functional, system, integration, at scale (time dedication)
- Instrument codes at the application level are tested too for development and acceptance.
  - End-to-end testing using codes.

Risks when developing HPC sw

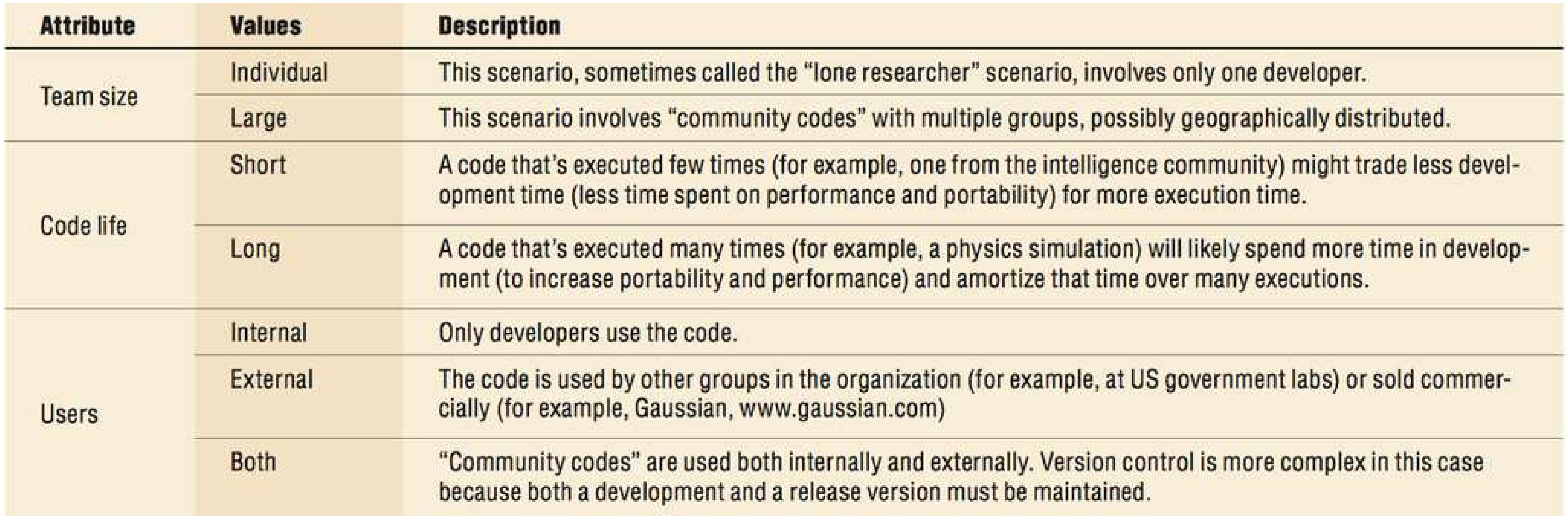
- Risks in sw development cycle
- Risks in the development environment
- Risks in the programming environment

See: Kendall, A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications, SEI 2007

Examples

- Typical risk in the development cycle: misunderstanding requirements
- Typical risk in the development environment: too many manual activities
- Typical risk in the programming environment: underestimating dependencies

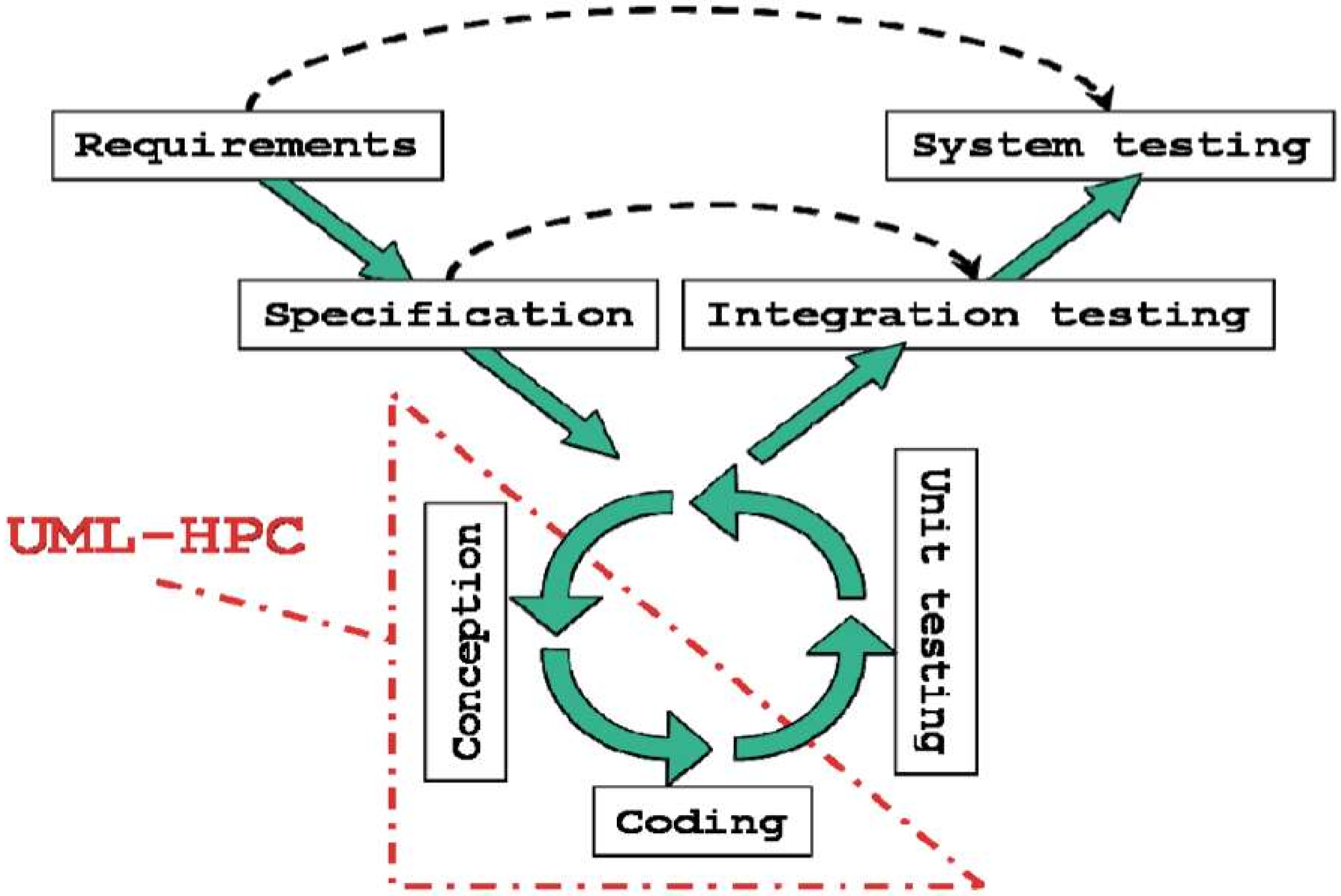
HPC stakeholders attributes



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

A process for HPC [Lugato 2010]



Community codes

- Popularizing the code alone does not build a community: different users want different capabilities
- Enabling contributions from users and providing support for them
- Including policy provisions for balancing the IP protection with open source needs
- Relaxed distribution policies – giving collective ownership to groups of users so they can modify the code and share among themselves as long as they have the license
- More inclusivity => greater success in community building
- An investment in robust and extensible infrastructure, and a strong culture of user support is a pre-requisite

Community codes

- Open source with a governance structure in place
- Trust building among teams
- Commitment to transparent communications
- Strong commitment to user support
- Either an interdisciplinary team, or a group of people comfortable with science and code development
- Attention to software engineering and documentation
- Understanding the benefit of sharing as opposed to being secretive about the code

flash.uchicago.edu/cc2012/

Measuring software

- How can we measure the impact of a scientific software package over time?
- When a system has no price, no purchase contracts and no buyers or sellers it can be difficult to judge its impact on the world
- We should try to measure its usage, quality, reusability, and sustainability in order to understand how much it should be rewarded

www.software.ac.uk/blog/2017-05-09-software-metrics-why-and-how

Measuring the use

- Depsy `depsy.org`
- www.nature.com/news/the-unsung-heroes-of-scientific-software-1.19100

- Depsy counts the uses of your software stored under github

Deployment

- Virtual Machines
  - Easy: Software pre-installed, ready to run
  - 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

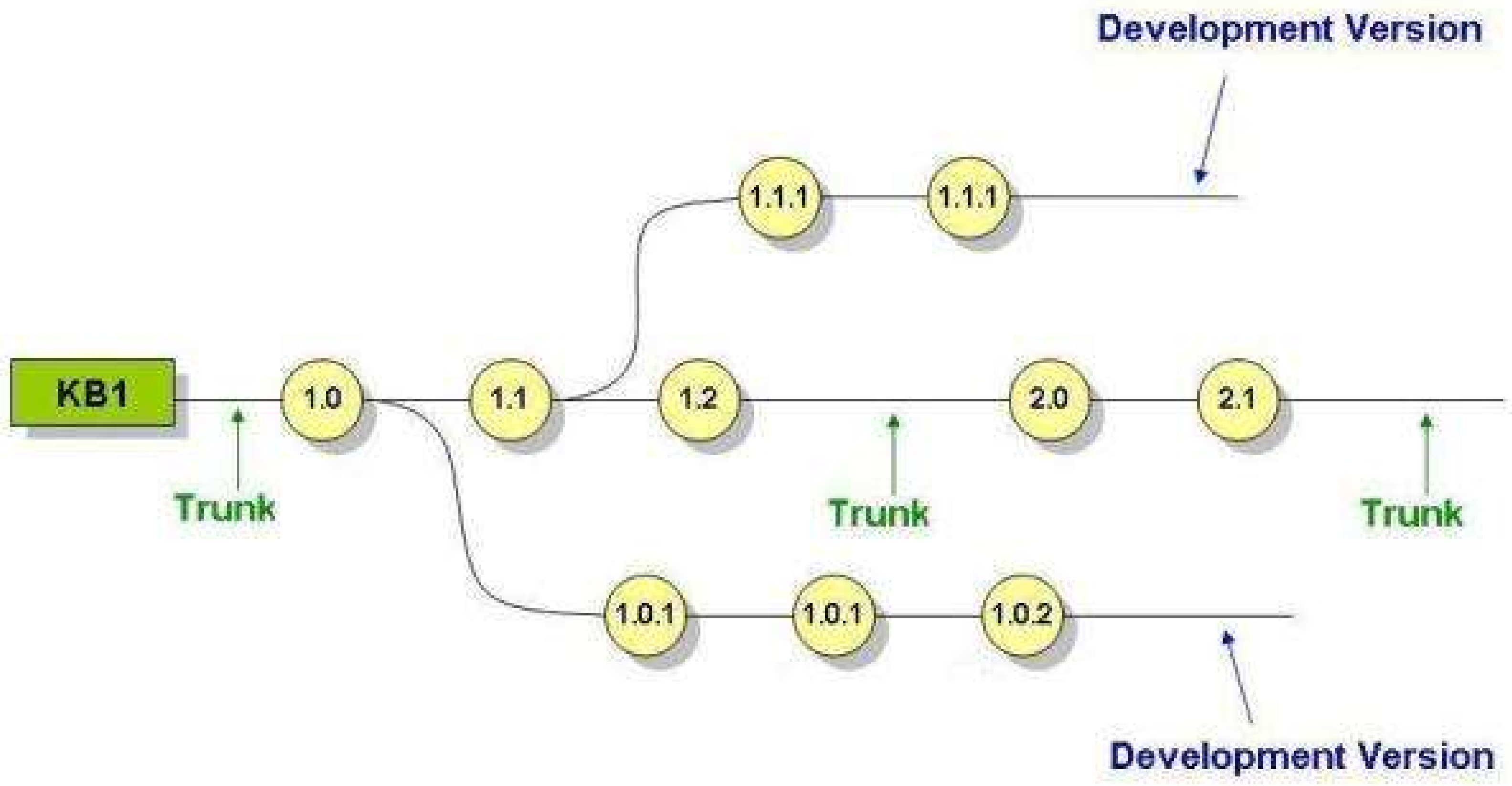
Configuration management

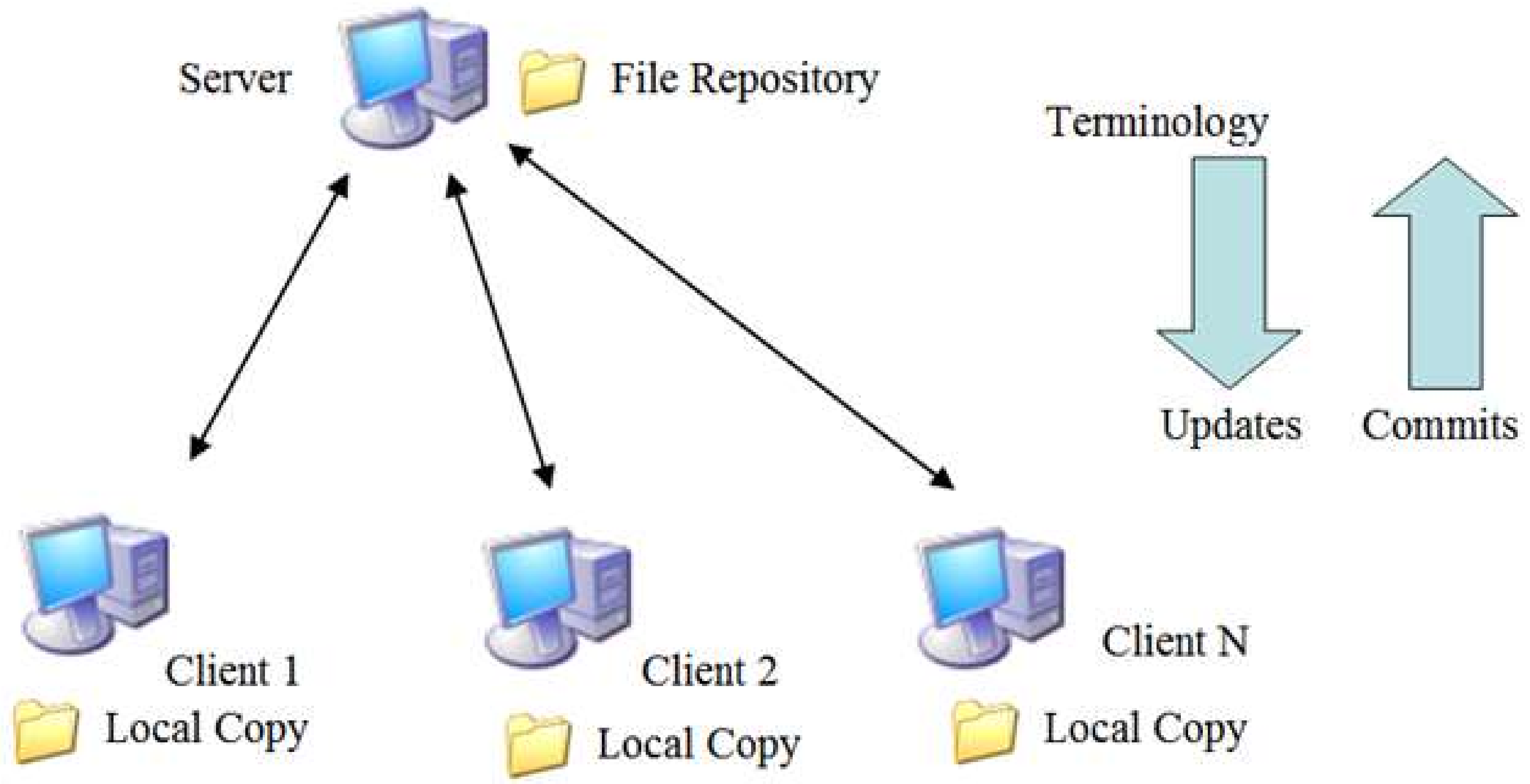
- Run your own CM system, if you have the resources
  - Generally easy to set up
  - Full control, but be sure to back it up!
- Some public solutions can offer most of these for free
  - SourceForge, GoogleCode, GitHub, Codeplex, Launchpad, Assembla, Savannah, …
  - BitBucket for private code base (under 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

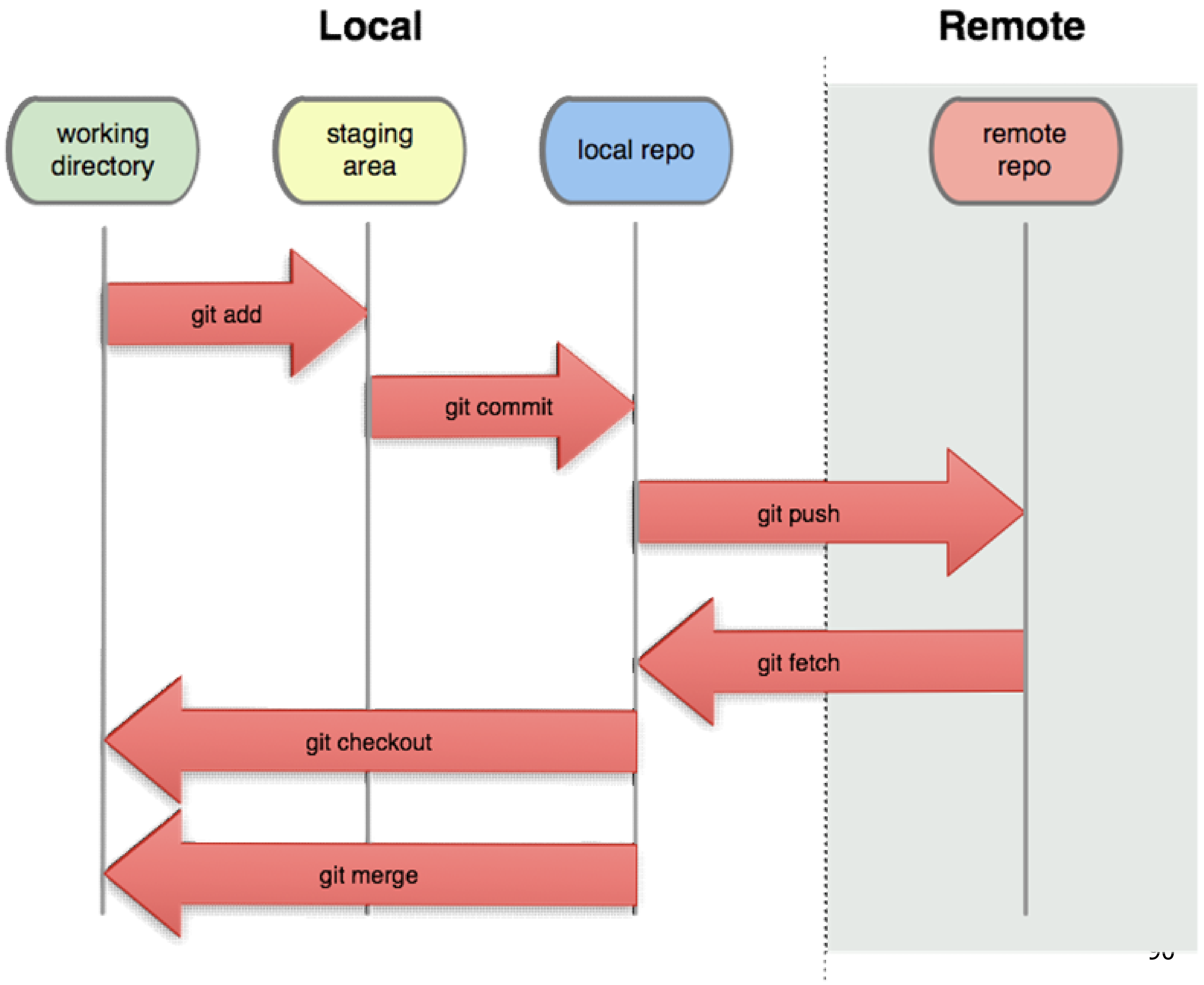
"If you're not using version control, whatever else you might be doing with a computer, it's not science" – Greg Wilson, Software Carpentry

Version control

- Version management allows you to control and monitor changes to files
  - What changes were made?
  - Revert to previous versions
  - When were changes made ?
  - What code was present in release 2.7?
- Earliest tools were around 1972 (SCCS)
- Older tools – RCS, CVS, Microsoft Source Safe, PVCS Version Manager, etc…
- Current tools – Subversion, Mercurial, Git, Bazaar


Development Version
1.1.1
1.1.1
KB1
1.0
1.1
1.2
2.0
2.1
Trunk
Trunk
Trunk
1.0.1
1.0.1
1.0.2
Development Version


Server
File Repository
Terminology
Updates
Commits
Client 1
Local Copy
Client 2
Local Copy
Client N
Local Copy


Local
Remote
working directory
staging area
local repo
remote repo
git add
git commit
git push
git fetch
git checkout
git merge

Version control concepts

- checkout – get a local copy of the files
  - I have no files yet, how do I get them?
- add – add a new file into the repository
  - I created a new file and want to check it in
- commit – send locally modified files to the repository
  - I made changes, how do I send them to the group?
- update – update all files with latest changes
  - Other people made changes, how do I get them?
- tag / branch – label a “release”
  - I want to “turn in” a set of files

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 and measures)

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?

Reference: papers

- V.Basili et al., Understanding the High-Performance- Computing Community: A Software Engineer's Perspective, *IEEE Software*, 2008
- G. Wilson et al., Best Practices for Scientific Computing. *PLoS Biol* 12(1), 2014
- Kendall et al., A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications, TN-039 CMU, 2007
- D.Lugato et al., Model-driven engineering for HPC applications, *Proc. Modeling Simulation and Optimization Focus on Applications, Acta Press* (2010): 303-308.

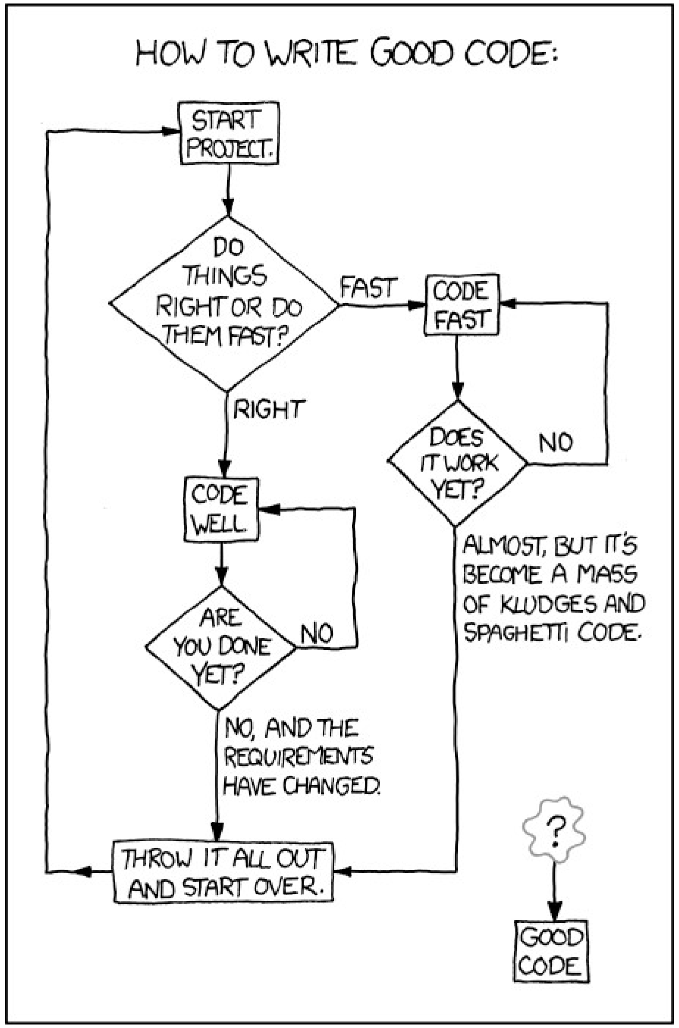
References: books

- Pressman, *Software engineering a practictioner approach*, 8th ed., McGrawHill, 2014
- The Computer Society, *Guide to the Software Engineering Body of Knowledge*, 2013
  www.computer.org/portal/web/swebok

Useful references

- `software.ac.uk` Software Sustainability Institute
- `software.ac.uk/resources/case-studies`
- `software-carpentry.org` Software carpentry
- www.software.ac.uk/blog/2016-09-26-scientific-coding-and-software-engineering-whats-difference
- Proc. 2016 Int. Workshop on Sw Engineering for HPC in Science
- `www.journals.elsevier.com/softwarex/`
- `ccpforge.cse.rl.ac.uk/gf/`

Questions?


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

http://xkcd.com/844/