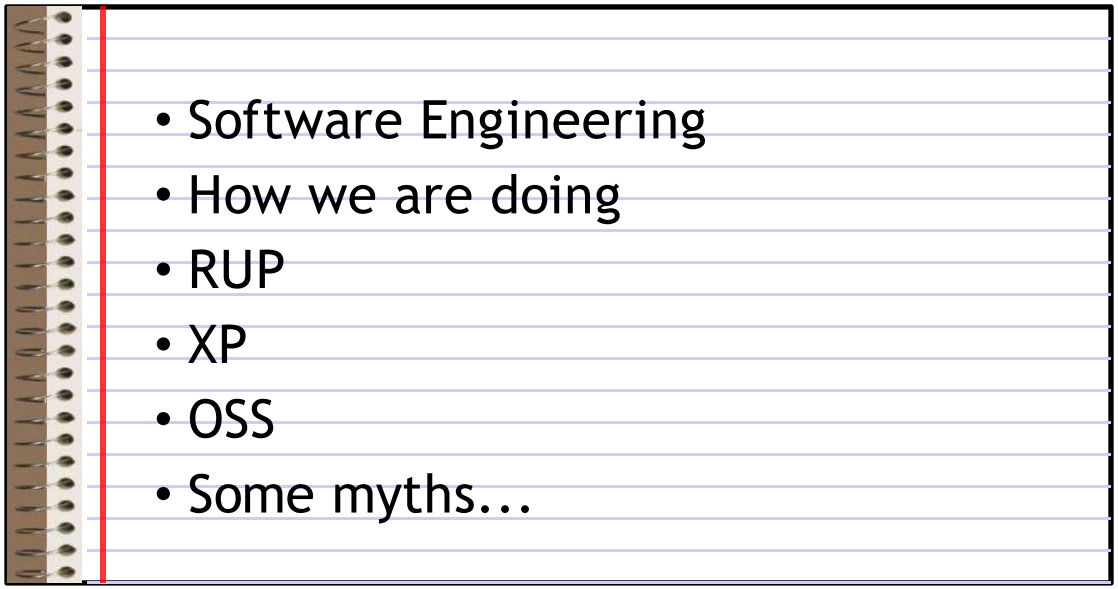
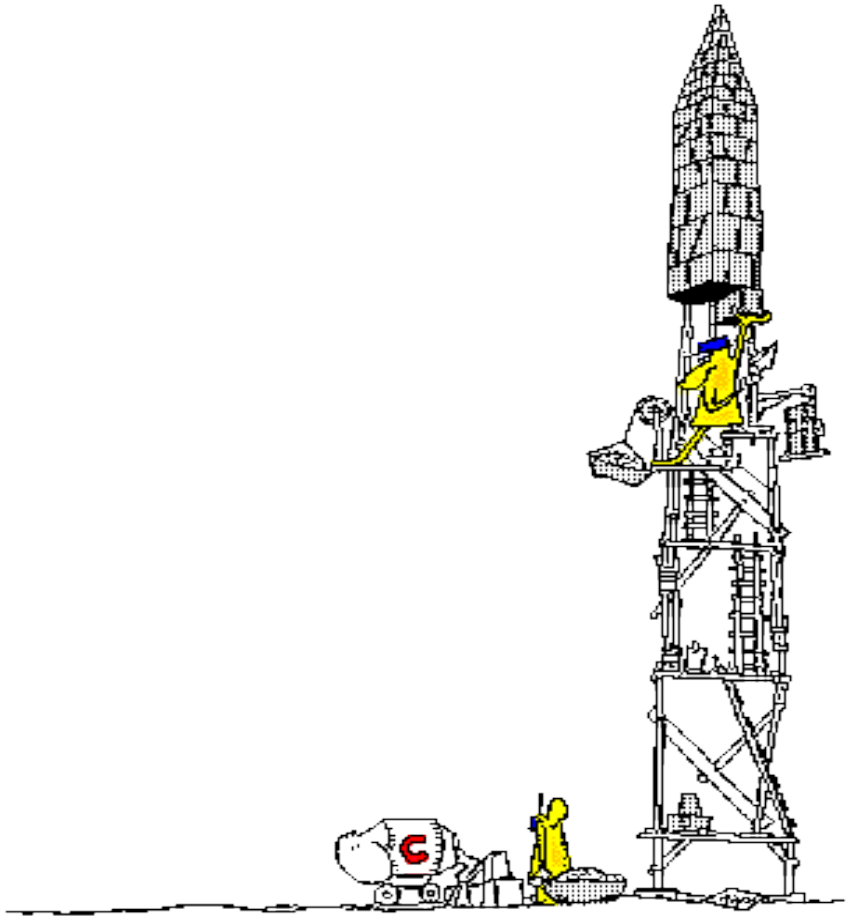
Andrew Boake,
University of Pretoria

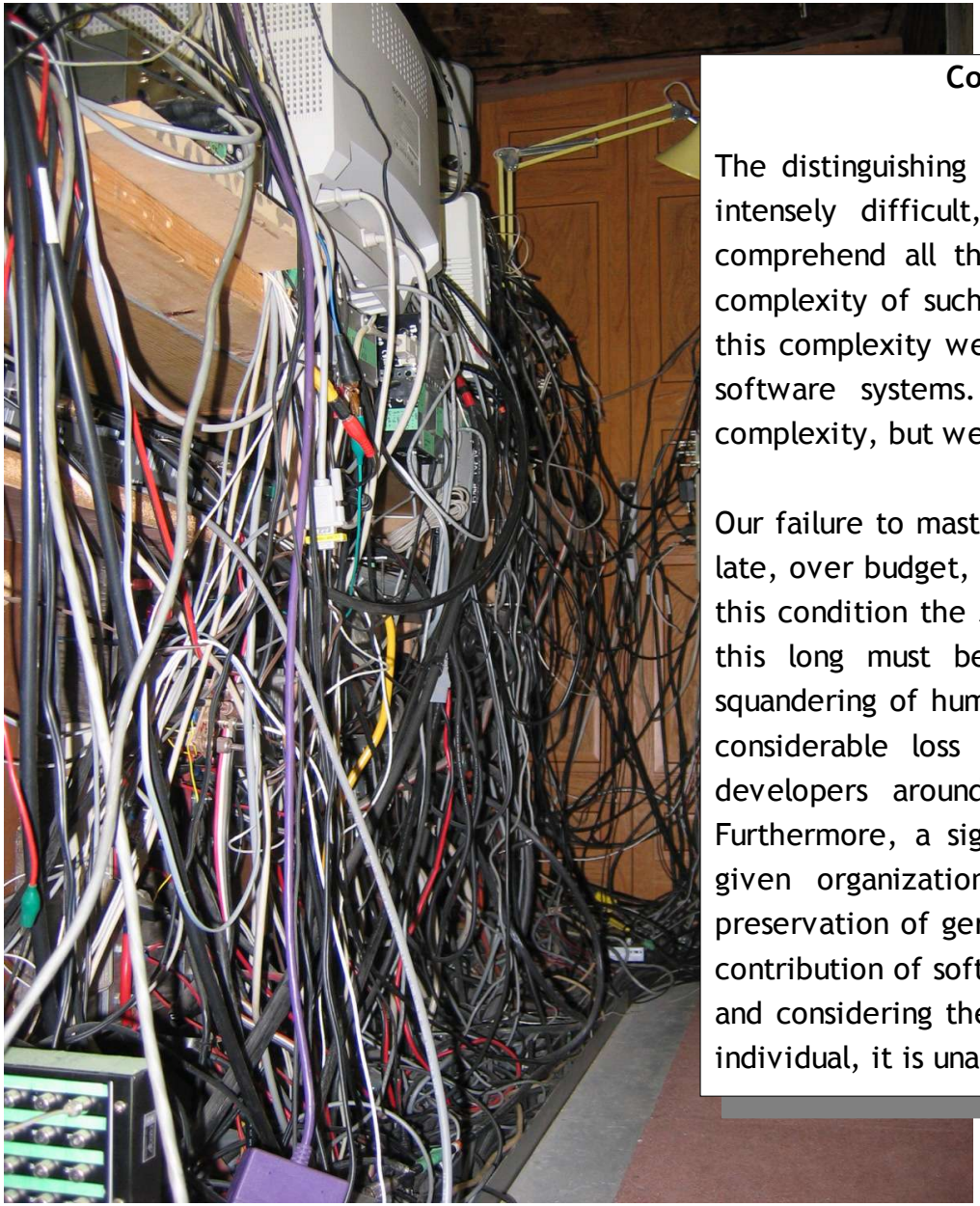
Why should we as software engineers concern ourselves about Open Source Software?





The term "software engineering" came to prominence when it was used as the name of a NATO workshop in 1968. It was used then to draw attention to software development problems. It was then, as to a large extent it remains now, a phrase of aspiration, not of description.

Mary Shaw, *Three patterns that help explain the development of software engineering*, March 1997.



Consequences of Unrestrained Complexity

The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design. Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By *essential* we mean that we may master this complexity, but we can never make it go away.

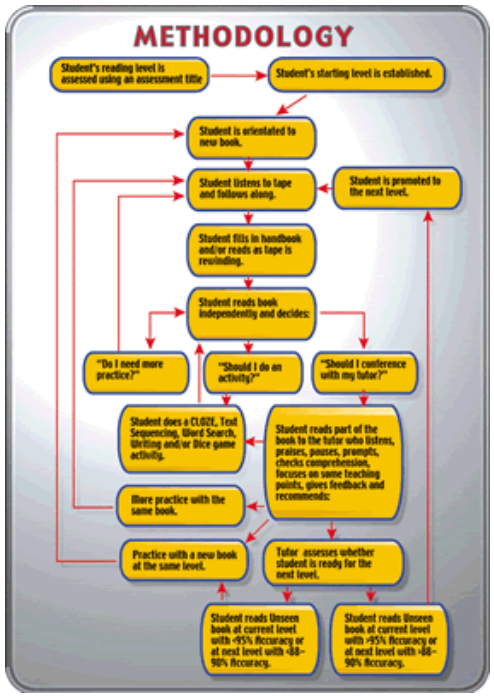
Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition the *software crisis*, but frankly, a malady that has carried on this long must be called normal. Sadly, this crisis translates into the squandering of human resources – a most precious commodity – as well as a considerable loss of opportunities. There are simply not enough good developers around to create all the new software that users need. Furthermore, a significant number of the developmental personnel in any given organization must often be dedicated to the maintenance or preservation of geriatric software. Given the indirect as well as the direct contribution of software to the economic base of most developed countries, and considering the ways in which software can amplify the powers of the individual, it is unacceptable to allow this situation to continue.

Grady Booch,
***Object Oriented Design with Applications*, 2003.**

Software Engineering: How we are doing

There are some software developments where predictability is possible. Organizations such as NASA's space shuttle software group are a prime example of where software development can be predictable. It requires a lot of ceremony, plenty of time, a large team, and stable requirements. There are projects out there that are space shuttles. However I don't think much business software fits into that category. For this you need a different kind of process.

Martin Fowler, *The New Methodology*, April 2003.



Most software development is a chaotic activity, often characterized by the phrase "code and fix". The software is written without much of an underlying plan, and the design of the system is cobbled together from many short term decisions. This actually works pretty well as the system is small, but as the system grows it becomes increasingly difficult to add new features to the system. Furthermore bugs become increasingly prevalent and increasingly difficult to fix. A typical sign of such a system is a long test phase after the system is "feature complete". Such a long test phase plays havoc with schedules as testing and debugging is impossible to schedule.

We've lived with this style of development for a long time, but we've also had an alternative for a long time: Methodology. Methodologies impose a disciplined process upon software development with the aim of making software development more predictable and more efficient. They do this by developing a detailed process with a strong emphasis on planning inspired by other engineering disciplines - which is why I tend to refer to them as **engineering methodologies**. Engineering methodologies have been around for a long time. They've not been noticeable for being terribly successful. They are even less noted for being popular. The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down.

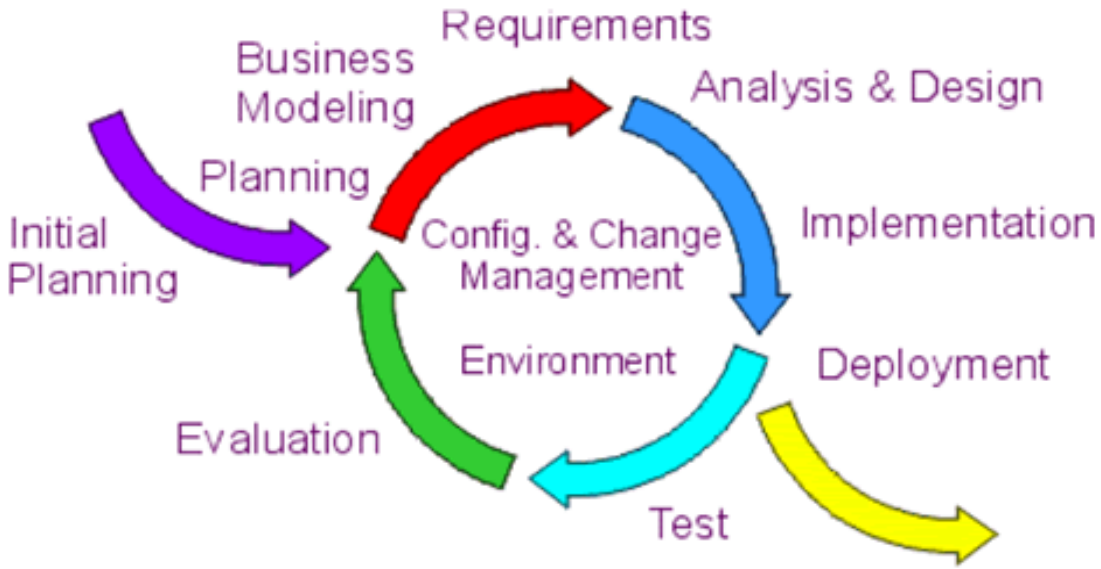
As a reaction to these methodologies, a new group of methodologies have appeared in the last few years. For a while these were known as lightweight methodologies, but now the accepted term is **agile methodologies**. For many people the appeal of these agile methodologies is their reaction to the bureaucracy of the monumental methodologies. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff.

Martin Fowler, *The New Methodology*, April 2003.

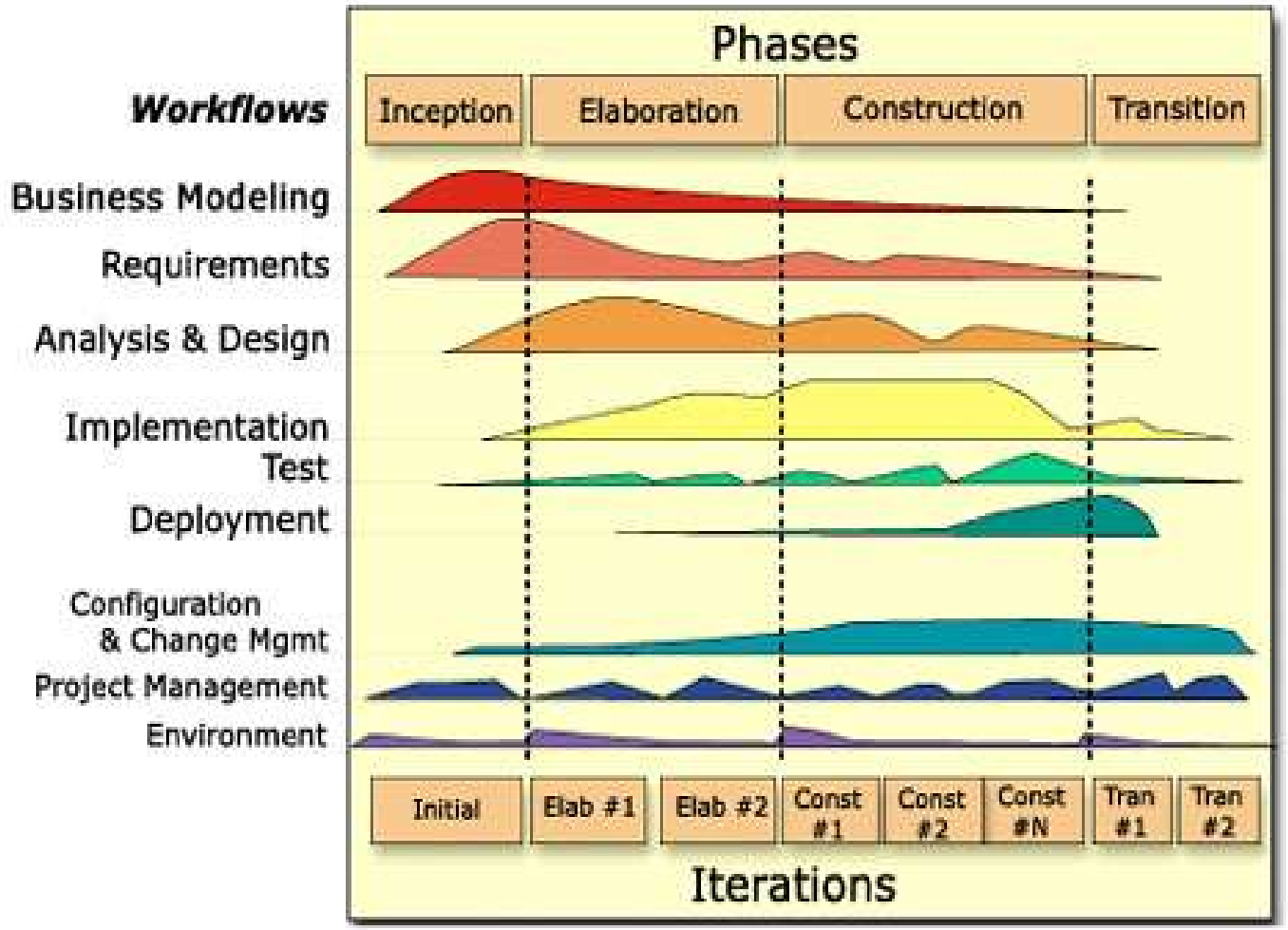

RATIONAL UNIFIED PROCESS

- **Develop Software Iteratively**
  - Increasing understanding of problem through multiple refinements
  - Incremental growth of an effective solution over multiple iterations
- **Manage Requirements**
  - Elicit, organise, document required functionality + constraints
  - Track trade-offs and decisions
  - Drive design, implementation and testing
- **Use Component-Based Architecture**
  - Early development of a robust executable architecture prior to full-scale development
  - Strive for a resilient, flexible, intuitive design
  - Compose new and existing components
- **Visually Model Software**
  - Visually model to capture various aspects of structure and behaviour
  - Hide the details using graphical building blocks
  - Clear communication
- **Verify Software Quality**
  - Continuously review functionality and qualities of system (reliability, performance)
  - As part of process, by all participants
- **Control Changes to Software**
  - Control, track and monitor changes to all artefacts
  - Use configuration control and build management to isolate participants from change

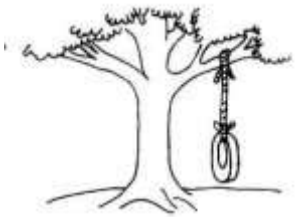
Rational Unified Process - Best Practices for Software Development Teams



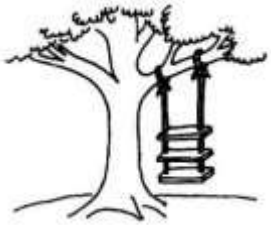
Rational Unified Process: Process



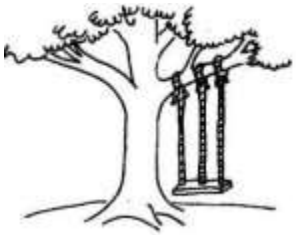
Rational Unified Process: Workflows



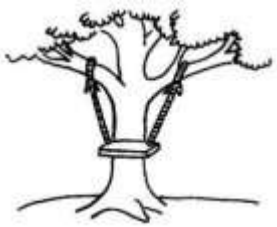
What the customer wanted



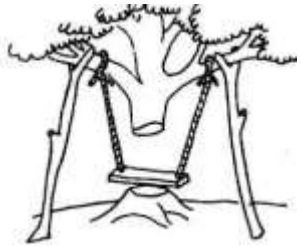
What the analysts specified



What the architects designed



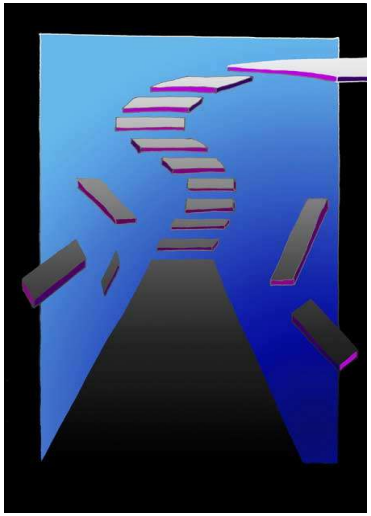
What the developers built



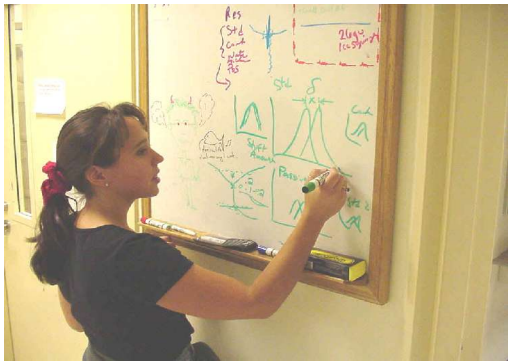
What was installed

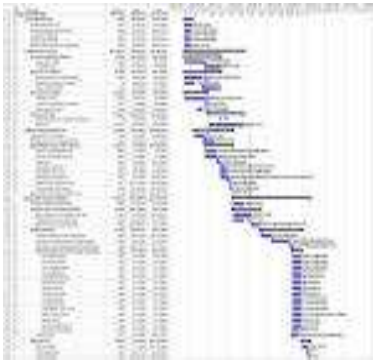
- **Broken telephone**
  - Many stakeholders, long communication paths from actual users to actual programmers
- **Lack of agility**
  - More documentation and ceremony means project inertia
- **Over-design**
  - Up-front design and anticipating change often leads to a complex architectural framework

Rational Unified Process: Weaknesses



- **Speed**
  - Market windows dictate products and services, and their software support
- **Uncertainty**
  - Requirements are vague – users also need to iterate
- **Change**
  - Requirements change as more clarity is reached or the business direction changes
  - Change is unavoidable, and time-scales unrelenting – mergers, legislation
- **Business bottom line prevails**
  - Return on investment, good enough software,
  - No IT-only projects, decentralization for control
- **Systems are transient**
  - Systems have short useful lifespans





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

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

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

Agile Manifesto (Feb 2001, Utah, USA): http://agilemanifesto.org

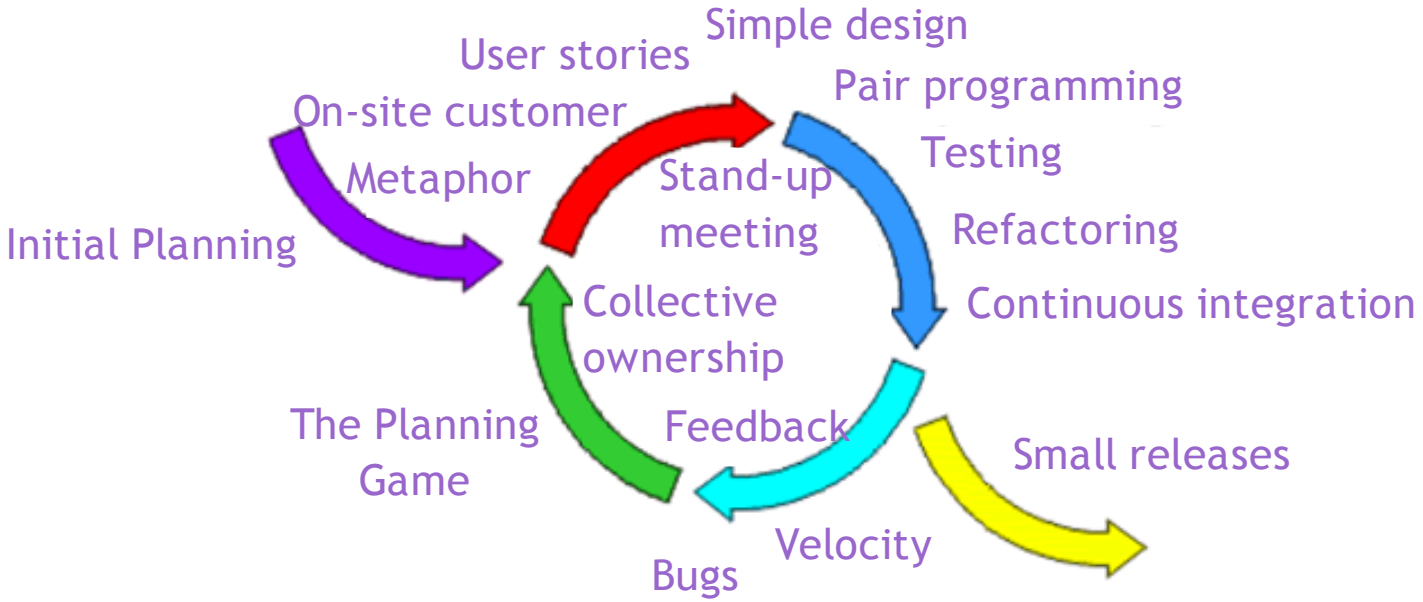
The Agile Manifesto


Hey You!
Why aren't you
CODING?

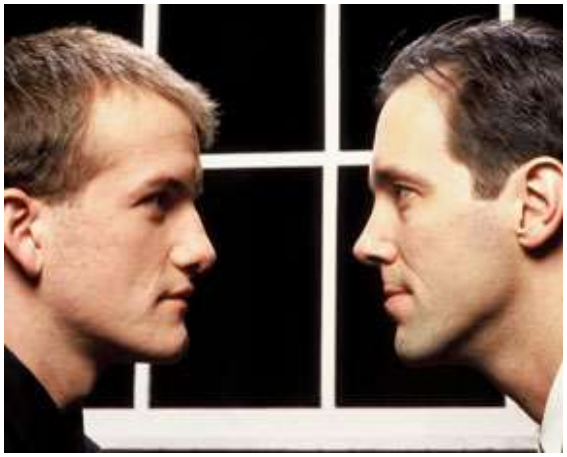
- **Planning Game**
  - Short cycles, frequent updates, stories
- **Small Releases**
  - As small as possible, containing most valuable business requirements; sense of accomplishment, frequent feedback
- **Metaphor**
  - Overall coherent theme to which both developers and business clients can relate
- **Simple Design**
  - Design for defined functionality, not potential future (cost of redoing when change); create the simplest design for now
- **Refactoring**
  - Ongoing redesign to improve system – quality and functionality
  - Redesign in such a way that the external behaviour of the code does not change, yet improves its internal structure.

Testing

  - Test cases before coding points out lacking functionality; automatic ongoing checking of existing functionality
- **Pair Programming**
  - Continuous code inspections, knowledge transfer, collaboration
- **Collective Ownership**
  - Everyone may work on any part of the code. Encourage moving around.
- **Continuous Integration**
  - Frequent builds (every few hours); Perils of integration avoided
- **40 hour week**
  - Don't burn out the troops. Enthusiasm is more productive.
- **On-site Customer**
  - Continuous detailed user involvement
- **Coding Standards**
  - So that everyone can read and write the communal code

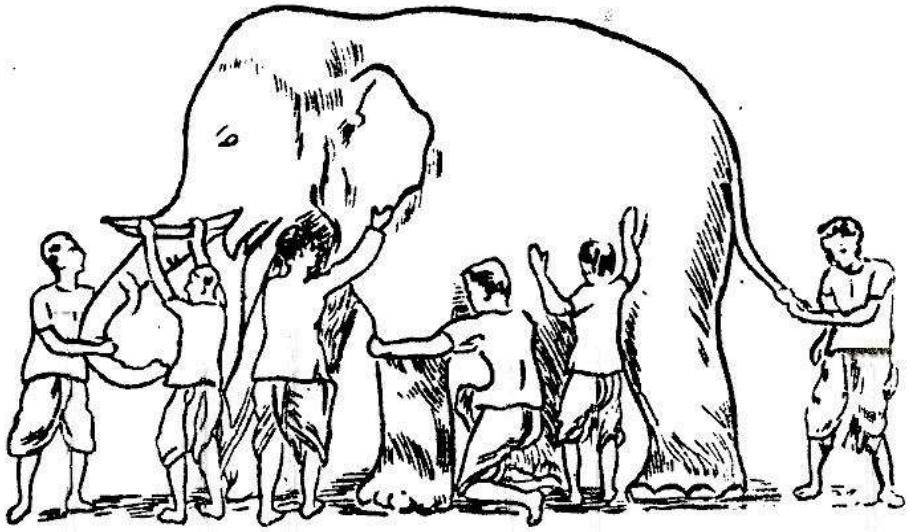


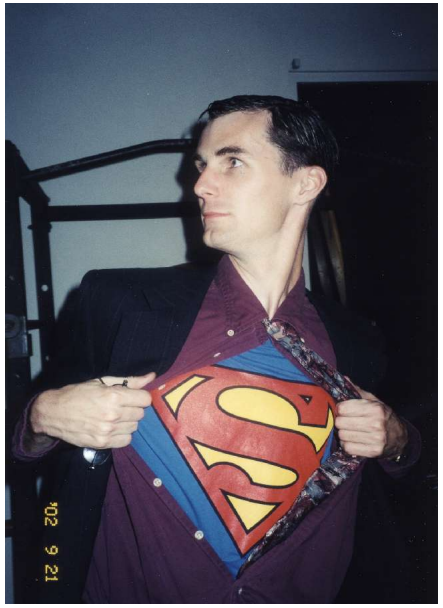
eXtreme Programming: Process



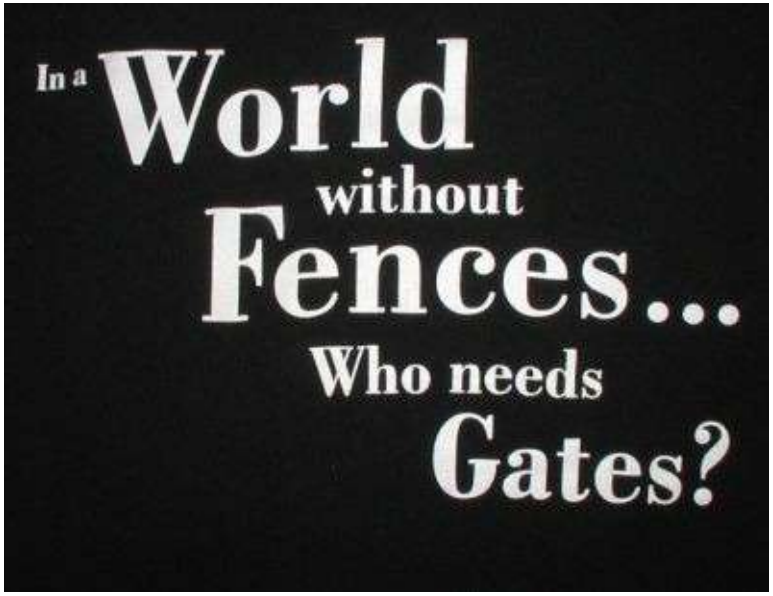
- **Communication**
  - Face to face, mutual understanding
- **Simplicity**
  - Make it simple today, so that cost of change is low tomorrow
- **Feedback**
  - developers are eternal optimists – feedback is the treatment
  - In designing, coding, building, testing (tests, builds, stories)
- **Courage**
  - do what is right even when pressured to do otherwise
  - discipline requires courage
  - courage requires conviction and confidence

eXtreme Programming: Values





- **Self-management**
  - XP teams prefer to manage themselves, but this disregards the need for a strong management role in larger projects with more stakeholders.
- **Design**
  - Can good design emerge from naive simplicity?
- **Large projects**
  - There are some aspects in the nature of XP projects which seem to inhibit scaling to larger projects.
  - eg face-to-face communication, little documentation, little management, 'global consciousness'
- **Team responsibility**
  - It is difficult to assess any one team member's performance
- **Availability of the super-user**
  - The super-user is always the most valuable business resource, and is therefore scarce


In a World without Fences... Who needs Gates?

He had worked on a cathedral once - Exeter. At first he had treated it like any other job. He had been angry and resentful when the master builder had warned him that his work was not quite up to standard; he knew himself to be rather more careful than the average mason. But then he had realized that the walls of a cathedral had to be not just good, but *perfect*. This was because the cathedral was for God, and also because the building was so *big* that the slightest lean in the walls, the merest variation from the absolutely true and level, could weaken the structure fatally. Tom's resentment turned to fascination. The combination of a hugely ambitious building with merciless attention to the smallest detail opened Tom's eyes to the wonder of his craft. He learned from the Exeter master about the importance of proportion, the symbolism of various numbers, and the almost magical formulas for working out the correct width of a wall or the angle of a step in a spiral staircase. Such things captivated him. He was surprised to learn that many masons found them incomprehensible.

"The Pillars of the Earth", Ken Follet

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development—release early and often, delegate everything you can, be open to the point of promiscuity—came as a surprise. No quiet, reverent cathedral-building here—rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

"The Cathedral and the Bazaar", Eric Raymond

The image I have is of hackers encamped just outside a stone gate, carefully but joyfully building, well, cathedrals, just like those within the town. From a long distance, they appear huddled by the gate as if frightened by what may lie beyond in the unknown. It is a slightly bitter image.

Their philosophy is to build up a single reliable layer. Although the classic open-source licences permit forking it rarely happens because a fork is a failure – there is a right place to go and a right thing to build. The design centre is whatever the hacker community likes, because the motivation is to create a world of open source that companies can never take away from them. It's a matter of liberation, and that's why it can be hard for companies to launch successful open-source projects.

"Richard Gabriel & Ron Goldman, "Mob Software: The Erotic Life of Code"

Let me say it plainly: We know how to produce small portions of software using small development teams – up to 10 or so – but we don't know how to make software any larger except by accident or by rough trial and error. Because the software we're trying to build is too massive – it is simply too difficult to plan it all out, and we have no idea how to coordinate the number of people it takes. Every piece of software built requires tremendous attention to detail and endless fiddling to get it right.

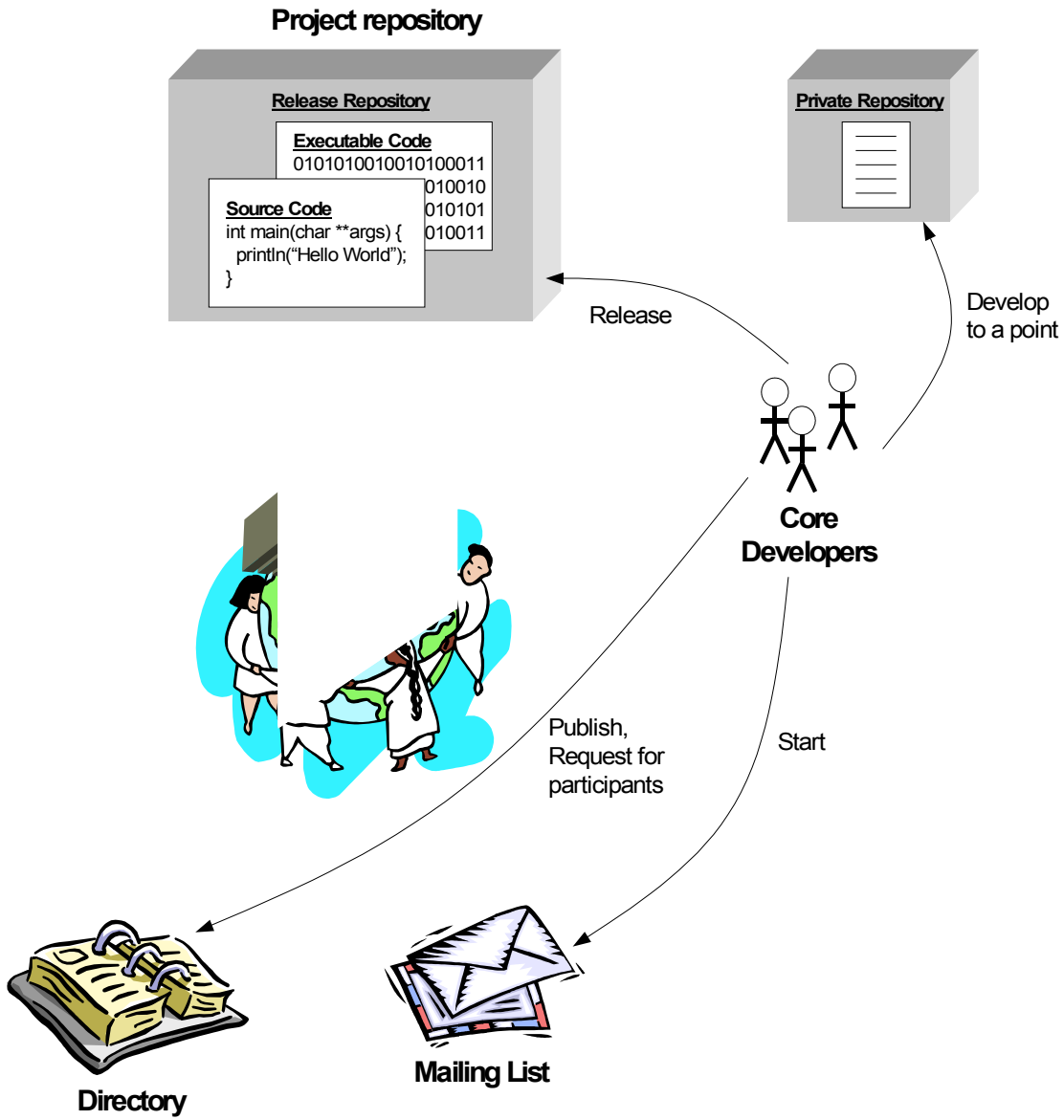
In response to this problem we have clung to fads: structured and object-oriented programming, UML, software patterns, and extreme programming. We grasp for mathematics or engineering to come to our rescue…

The way out of this predicament is simple: Set up a fairly clear architectural direction, produce a decent first cut at some of the functionality, let loose the source code, and then turn it over to a mob.

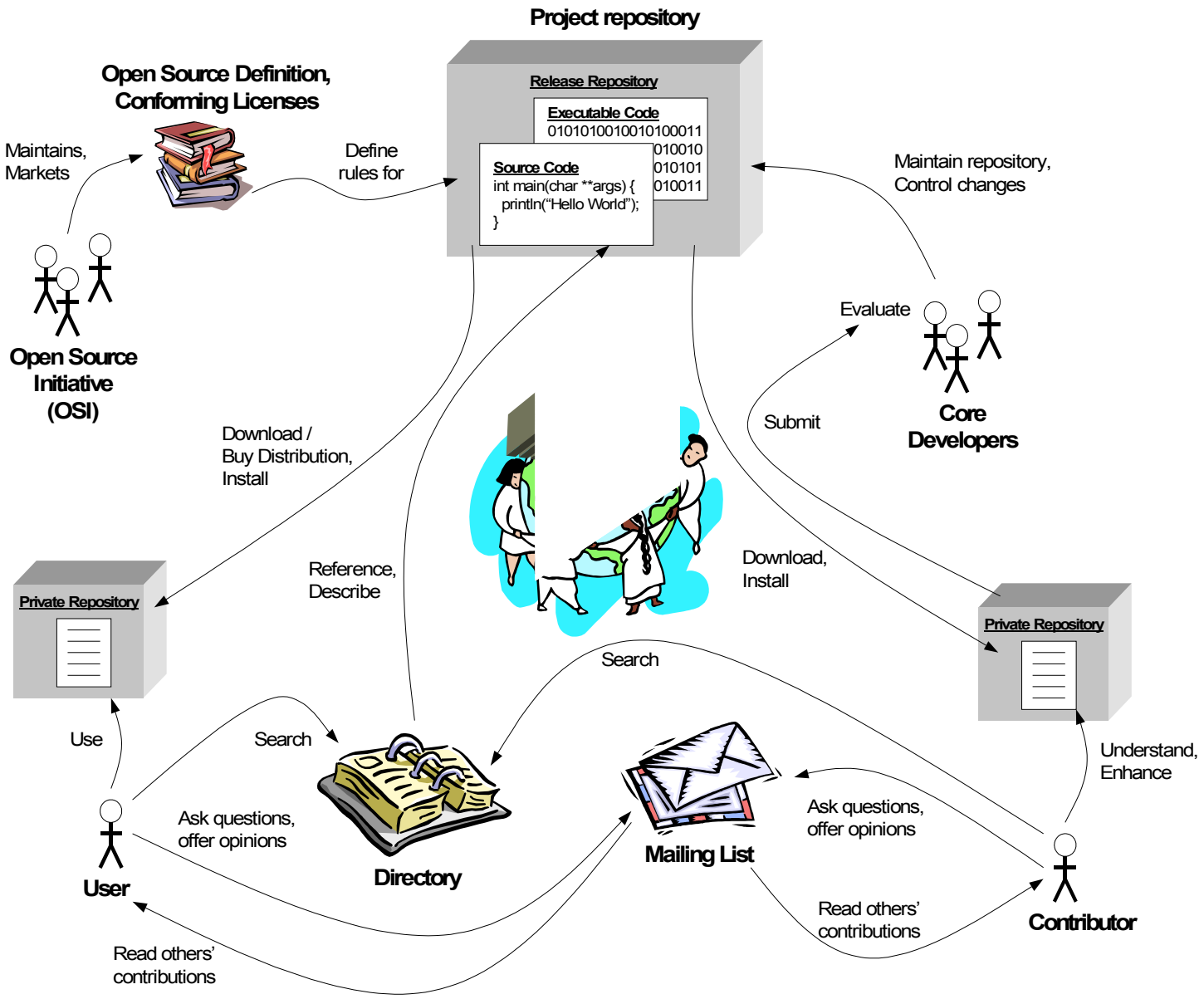
"Richard Gabriel & Ron Goldman, "Mob Software: The Erotic Life of Code"

In a development which is changing the traditional software development paradigm, voluntary teams of expert developers are collaborating on the development and maintenance of common “pools” of software. These experts are typically from different companies, and are collaborating driven by needs such as developing, fixing or enhancing a piece of software that is particularly relevant to their area of interest, and sharing in a community that is intimately familiar with, and has a passion for, their area of expertise, and hence appreciates the skill in their work.

Members of these teams work individually or in small groups, cooperating with the bigger group via the Internet, on a single body of software. They make this freely available, on agreement that further enhancements will be fed back to the community.

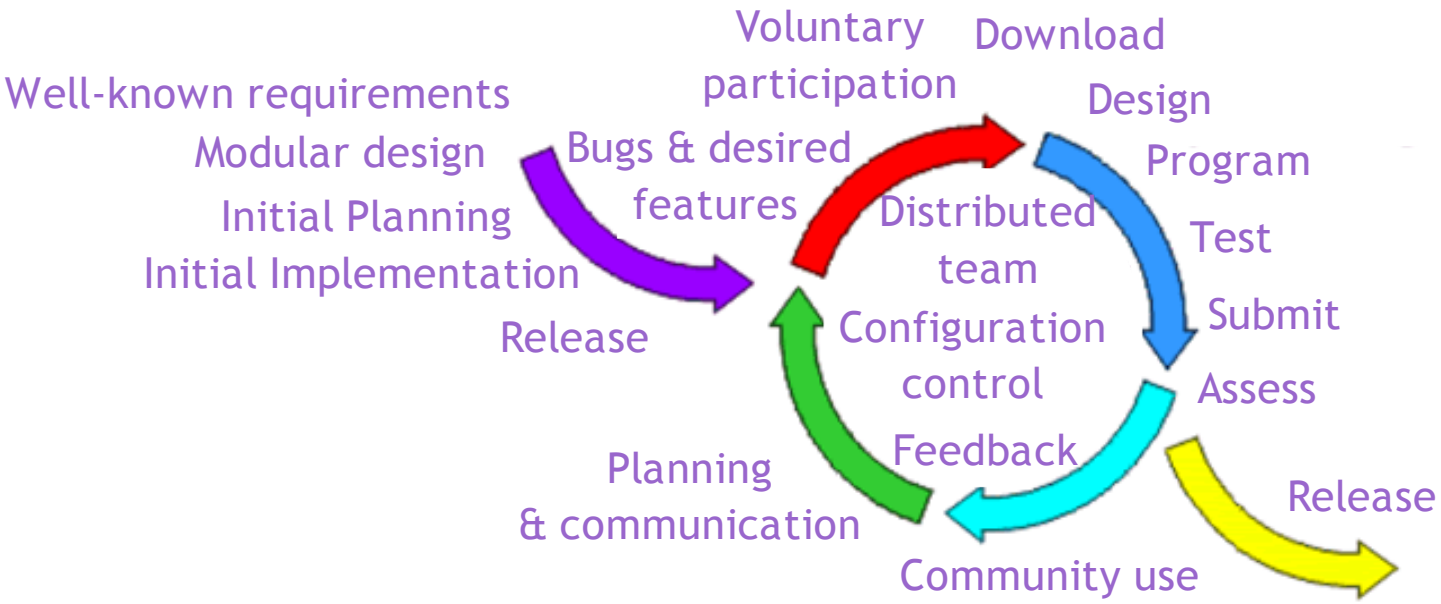


Open Source Software: Starting an Open Source project



Open Source Software: Using and contributing to Open Source

University of Pretoria



Open Source Software: Process

Myth #1

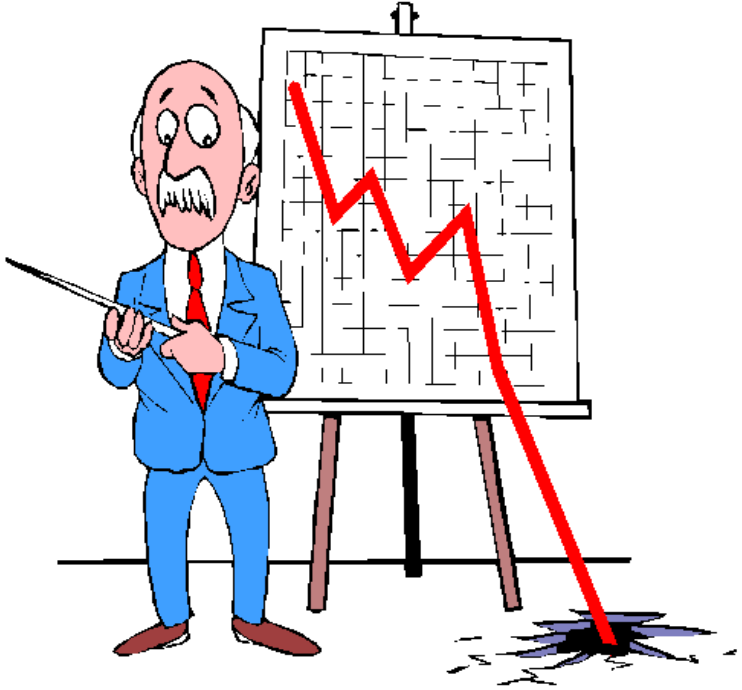
Open Source Software is a completely new way of doing things.

Before the current era of protecting your software's precious 'intellectual capital', source code supporting research was freely exchanged in the academic world, where peer review, having others use what you have created, and perhaps even take it further than you did, was valued highly. In the early Unix days, the source code for the X window system was distributed so that people could understand *exactly* why graphical user interfaces behaved as they did.

Myth #2

All talented developers are in Redmond / Santa Clara / Redwood Shores.

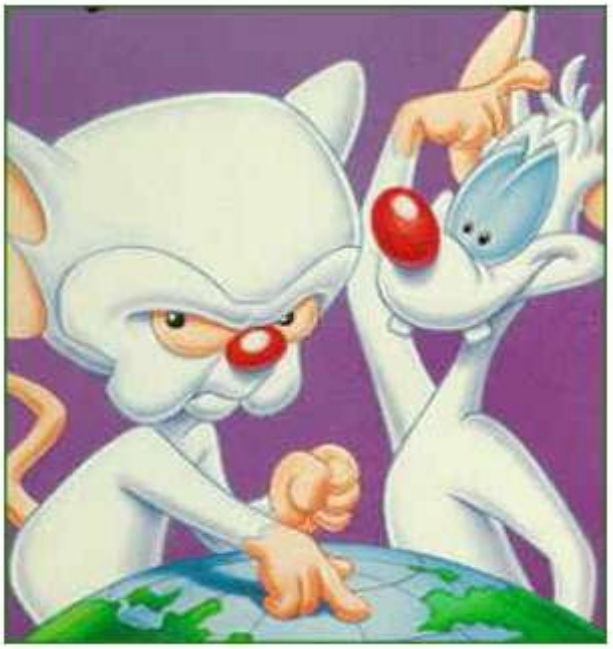
Given sufficient opportunity, there is no reason why experts in various software areas should not be distributed around the world. Chances are your in-house talent pool will not be able to compete with the available wealth of talent around the world. But, you will have to convince them to participate in your project.



Myth #3

Open Source cannot be an economically sustainable effort.

Software developers need money to survive. If they give away what they work on, where does the money come from to pay them? Many companies are “open sourcing” parts of their software, and providing proprietary paid-for add-ons. Others are providing services for open source software, like maintenance, installation, customization, and training. In both cases, the open source software provides an enabling layer that supports many other activities. This layer must be open and standard to allow many other participants in the game. Share where you should, and compete elsewhere.



Myth #4

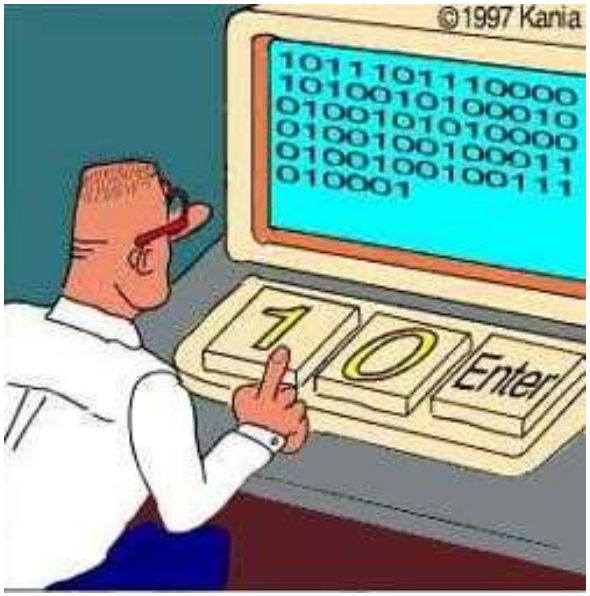
All software will soon be Open Source.

The successful Open Source projects have a few characteristics in common: clear requirements and design consensus in a commoditised (usually infrastructural) domain. It makes sense for people to share their knowledge and tools in these domains, to provide a base for the common good. In areas where software holds the promise of financial gain, or where design expertise is specialized to a particular industry or company, that software has less success in open source form. In addition, software to be used by non-technical users needs considerably more support than the Open Source community currently gives.

Myth #5

Reduce your costs by "open sourcing" your project's code, because outsiders will be writing some of the code, and code will be developed faster because of the increased workforce.

A typical OSS project attracts relatively few outside developers. Only a few really successful projects have an active global community. But, if you attract a community to your interesting and useful software, you will gain insightful, experienced testers: with many eyes, many bugs *are* shallow. Peer review is healthy. Share this and you may benefit elsewhere.



Myth #6

Open Source Software is a radical departure from established Software Engineering principles.

"On closer inspection, the bazaar model of OSS does not seem to depart as wildly from many of the sensible and proven fundamental software engineering principles as was first assumed. The argument then that OSS begins as a bazaar with a chaotic development process and evolves mysteriously into a coordinated process with an exceptionally high quality end product is perhaps too simplistic a characterization of what is actually taking place in practice."

Understanding Open Source Software Development Joseph Feller & Brian Fitzgerald, 2002

Myth #7

Access to source code provides the keys to all of the secrets of the software.

Access to source code is cited by many to allow governments to inspect the security characteristics of the code, students to learn from real-world software, and ICT industries to use existing bodies of code to enable a thriving software industry. It takes considerable effort and expertise to understand millions of lines of code, weaving together the results of many different design decisions. Very few people have the skill, tools (or inclination) to be able to unravel the rationale behind the software only by looking at the source code. Much more useful is the ability to be able to use or write modular plug-in components that extend a particular platform. This leads to a community (like Eclipse), where these components are shared, and the whole provides an extremely useful tool set.

- *The Ten Essentials of RUP - The Essence of an Effective Development Process* - Leslee Probasco
- *Using the Rational Unified Process for Small Projects* - Gary Pollice
- *Extreme Programming* - Jim Highsmith
- *CMM vs Agile Software Development* - Ken Orr
- *The New Methodology* - Martin Fowler
- *Is Design Dead?* - Martin Fowler
- *The Cathedral and the Bazaar*, Eric Raymond, 2000
- *Simple Open Source Economics*, Lerner & Tirole, June 2002
- *Open Source: Beyond the Fairytales*, Gabriel & Goldman, Sept 2002
- *Mob Software: The Erotic Life of Code*, Gabriel & Goldman, Oct 2000
- *Understanding Open Source Software Development*, Feller & Fitzgerald, Addison-Wesley, 2002
- *An Empirical Study of Open Source and Closed Source Software Products*, Paulson et al, IEEE Transactions on Software Engineering, Vol 30, No 4, April 2004

References