Software Development Methodologies

Elena Punskaya, op205@cam.ac.uk

Software Process

- "**Analysis** is discovering and describing those aspects of a software development about which there is no choice, that is, to which the project is already committed"
- **Design** is creating a definition of how the project goals are going to be achieved
- **Implementation** is the process of writing code, typically would be partitioned in many subprojects
- **Building** is creating a "complete" version of the software, i.e. putting all chunks of code together, including any custom configurations for target deployment
- **Testing** is about making sure that small independent parts of code work correctly (unit tests), that all code parts work together (integration tests), that the functionality meets requirements (acceptance), that any new code doesn't break the old (regression)

Software Process

- **Deployment** – actual release of the software into the end user environment, e.g. publishing the mobile app in the App Store or launching the service on bank's servers

- **Maintenance** – supporting the software during its lifetime, e.g. releasing compatibility updates when a new version of mobile OS is released or improving performance as user base grows
- Traditionally, **80-90%** of software system **Total Cost of Ownership** is attributed to **maintenance**. Once the system is operational, the cost of change is high (each new release requires a new full cycle: analyse, design, implement, build, test, deploy). Also, to achieve a better **Return on Investment**, the preference is naturally to extend the existing system than develop a new one.
- Nowadays, in some software systems (web apps), the line between maintenance and continuous development is less clear, consider Google and Facebook – is scaling up to *hundreds of millions of users* and *PetaBytes of data* maintenance or new development?

1970: The Waterfall Model

- **Sequential design process, progress is flowing steadily downwards through the phases (Royce actually suggested a number of improvements, including iterations and prototyping**

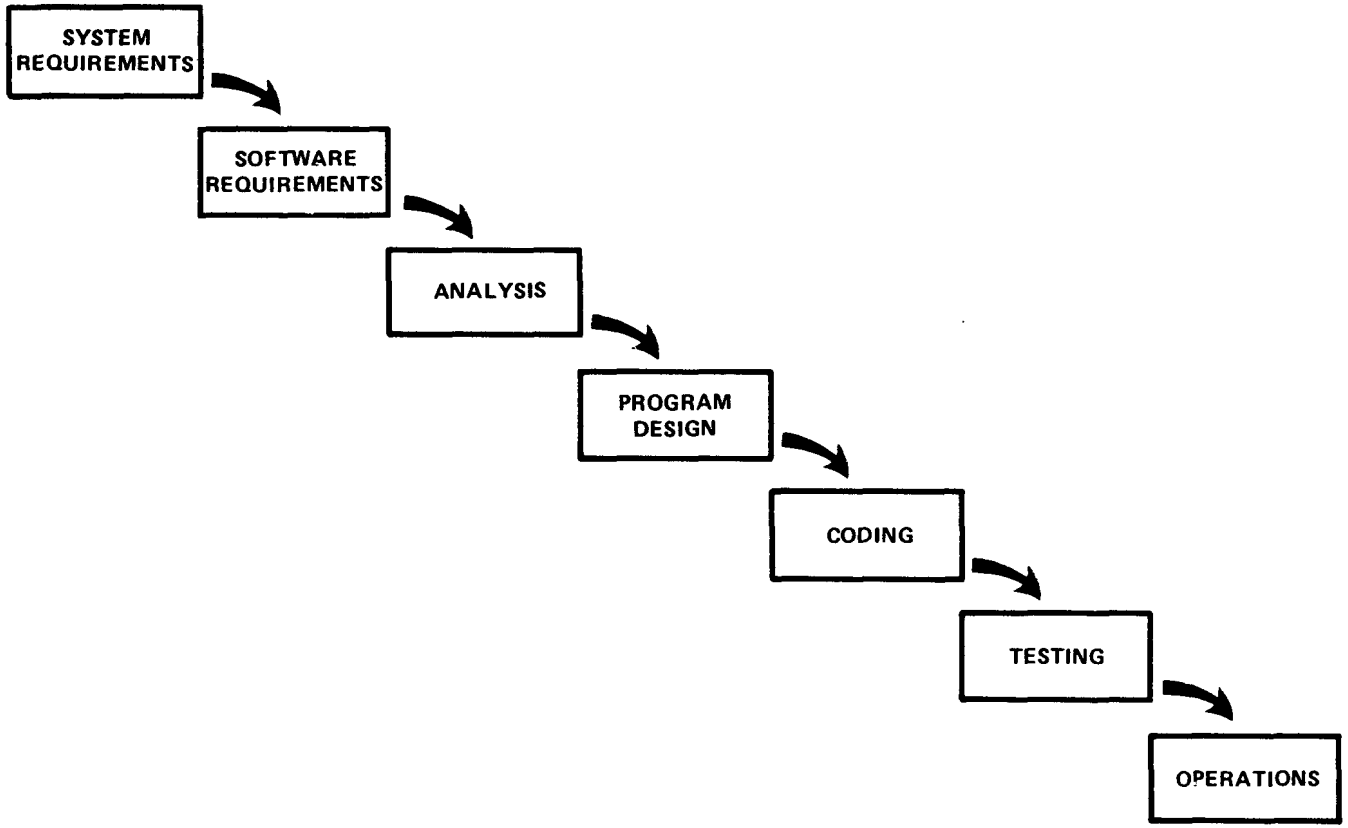
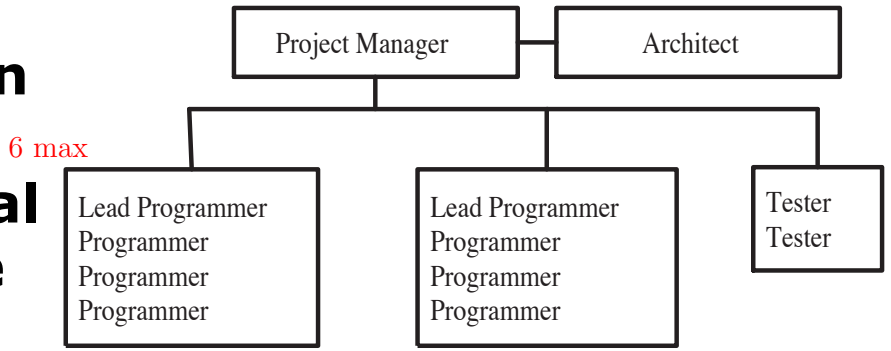


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

"Managing the development of large computer systems", Winston W. Royce, 1970

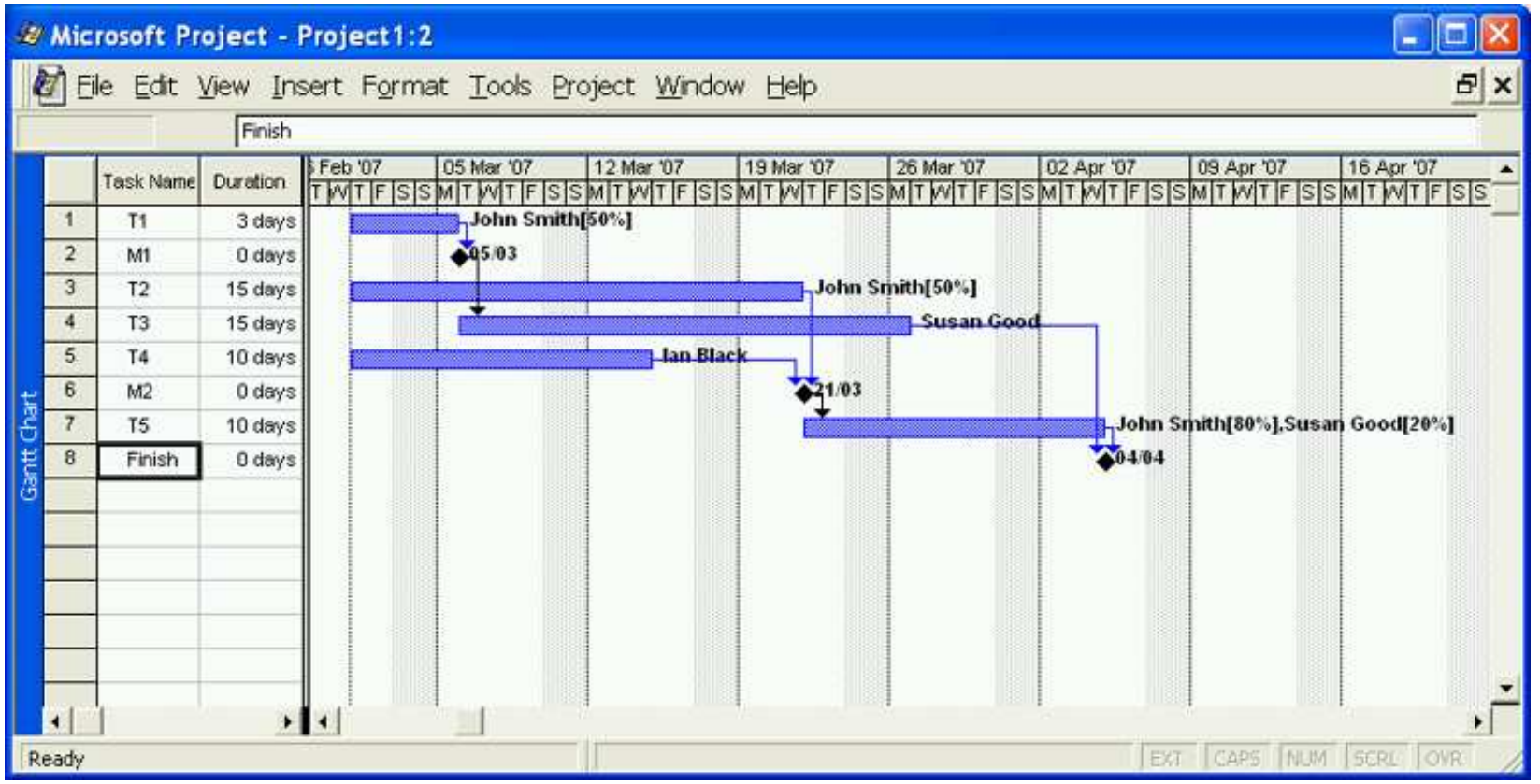
Traditional Team

- **Architect: The principal designer, defines the overall architecture, module structure and all major interfaces, usually also an expert in the associated technology. Responsible for Specification and High Level Design.**
- **Project Manager: Responsible for scheduling/rescheduling the work, tracking progress and ensuring that all of the process steps are properly completed (on time, on budget).**
- **Lead Programmer: Leader of a programming team. Will typically spend 30% of his/her time managing the rest of the team.**
- **Programmer: Implements specific modules and often implements module test procedures.**
- **Tester: Designs test and validation procedures for the completed software. Tests are based on initial specification and will focus on the overall product, rather than the individual modules.**



Traditional Tools

- **Famous Gantt chats in Microsoft Project shows tasks on a calendar**



The Waterfall Model

Advantages

- **Early clarification of system goals**
- **Can charge for changes to the requirements**
- **Works well with management tools**
- **If it is really possible to define all the requirements in advance a viable option**

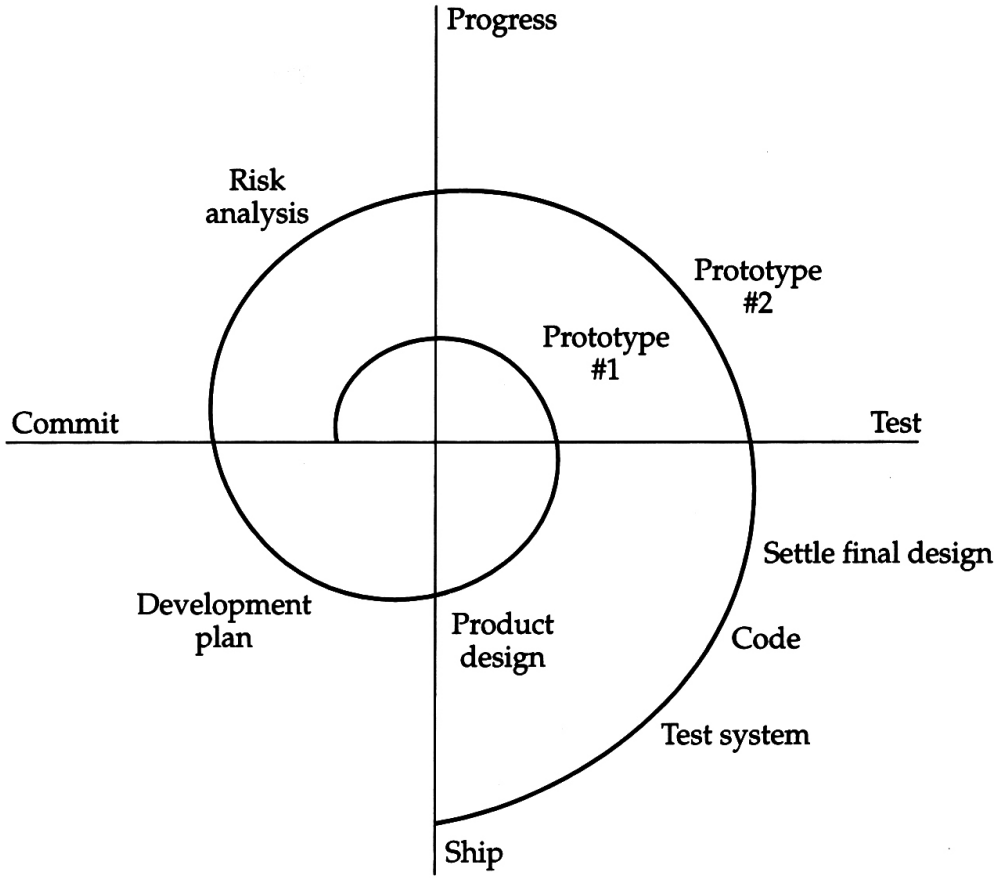
Disadvantages

- **Iterations are critical to software development process (requirements are not always understood by the customer and developer, technology and environment are changing, etc.)**
- **A lot of time is spent on system spec, then functional spec, then programming spec, etc. getting the system absolutely right from the beginning maybe impossible or unimportant over the system lifecycle ("right" today is not necessarily "right" tomorrow, doing everything "right" takes a long time)**

The Spiral Model

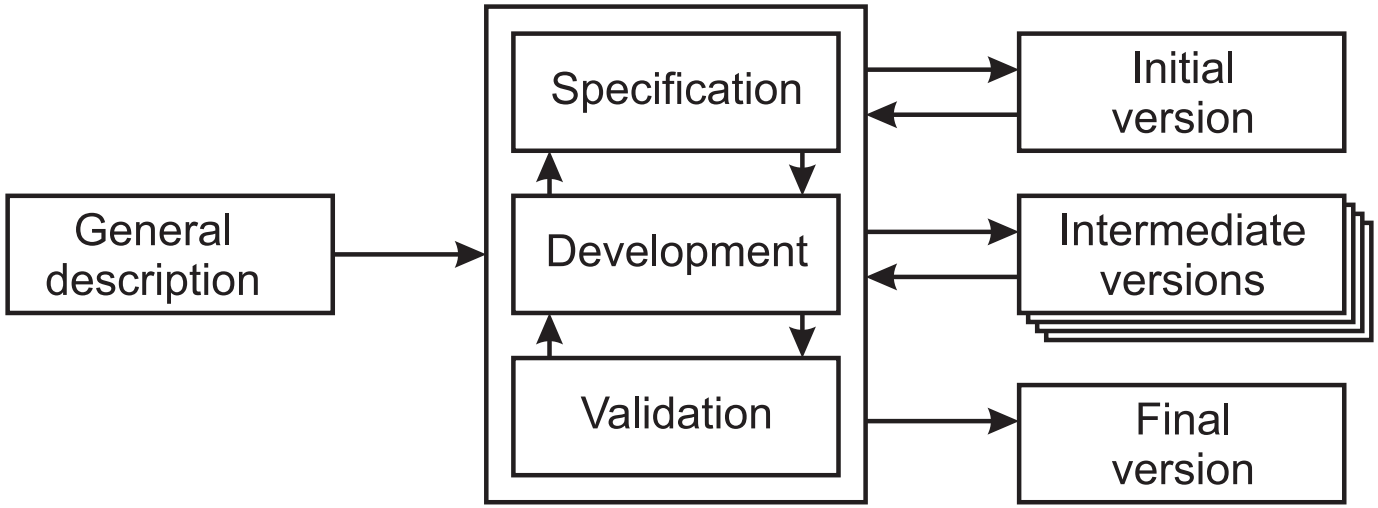
- Decide in advance on fixed number of iterations (engineering prototype, pre-production prototype, production prototype, etc.)
- Allows to manage risk – prototype bits considered "risky" by you or customer first

- Iterative development with systematic aspects of the waterfall mode
- Iterative incremental refinement though each time around spiral



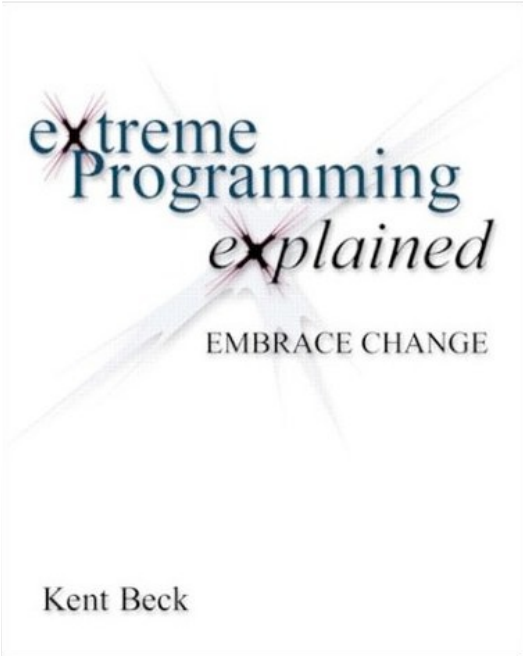
Evolutionary Model

- **In reality many modern systems "evolve" daily and the arrival of *automatic testing* not only made evolutionary models possible but also enabled a development of a new *lightweight* software development methodology**



Lightweight methods emerging

- In mid-90th a number of *lightweight* (and later known as *agile*) software development methods emerged as a reaction against *one cycle, big bang regulated* methods
- Examples include Scrum, Crystal Clear, Adaptive Software Development, Feature Driven Development, etc.
- One of such earlier methods is Extreme Programming
- In 1996, Chrysler brought Kent Beck, a SmallTalk practitioner, as a project leader in "C3" (Chrysler Comprehensive Compensation System) - a project to replace several payroll applications
- Beck noted several problems with the development process and took the opportunity to implement some changes (working with other collaborators)
- In 1999, the book Extreme Programming Explained was published to spread the ideas (2004 - 2nd Edition with Cynthia Andres)



Extreme Programming (XP)

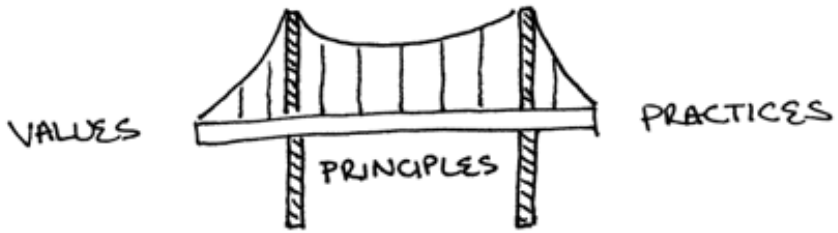
- **Everything changes (requirements, business, technology, team). The problem isn't change but our inability to cope with it.**
- **The driving factor: improve software quality and responsiveness to changing requirements, reduce cost of change.**
- **Introduces not just agile development framework but also philosophy, values, practices and principles.**
- **Although XP is new many of the practices have been around for some time, takes "best practices" to extreme.**
- **The paradigm of XP: "Stay aware. Adapt. Change"**
- **Driving metaphor (customers steer the content, the team steers the project)**

"Driving is not about getting the car going in the right direction. Driving is about constantly paying attention, making a little correction this way, a little correction that way."

Kent Beck and Cynthia Andres, Extreme Programming Explained: Embrace Change

XP not just a Software Development Framework

- **Practices – things you do day to day, clear and objective, good starting point but not enough (I can "correctly prune a branch" but will not be able to "see that the whole tree should come out")**
- **Values – the roots of the things we like and don't like about the situation (when the developer does not want to estimate his task it's in fear of being accountable not due to lack of technique; he values protection more than communication); implicit values bring purpose to practice**
- **Values and practices are ocean apart – values are expressed on too high level, anything in the name of it (1000 page document as I value communication?); they need to be brought together (find defect, value: learning (don't brush it off, practice: root-cause analysis)**
- **Principle bridge the gap - domain-specific guidelines**



Kent Beck and Cynthia Andres, Extreme Programming Explained: Embrace Change

XP Values

- **Communication – most valuable, when problems arise someone knows the solution but that knowledge doesn't get through to someone with the power to make changes**
- **Simplicity – most challenging, making system simple enough to gracefully solve only today's problem (What is the simplest thing that can possibly work?) - hard work!**
- **Feedback – being satisfied with improvement rather than expecting instant perfection, one uses feedback to get closer and closer to the goal, need to differentiate important feedback from any other feeedback**
- **Courage – facing fear, courage to accept responsibility, to speak truth, to take action (rather than spend time defending yourself)**
- **Respect – if the team members don't care about their work, each other or the project nothing will help; the contributions of each person to the team need to be respected**

XP Principles

- **Humanity – people develop software and their needs have to be met**
- **Economics – somebody has to pay for all this, make sure there is business value**
- **Mutual Benefit – every activity should benefit everyone concerned (automated testing, refactoring and clean code - some times there is no need for extensive documentation to benefit some future unknown person)**
- **Self-similarity – try copying the structure of new solution into a different context, even at different scales**
- **Improvement – "Best is the enemy of good enough"**
- **Diversity – teams need to bring together a variety of skills, opinions, perspectives - a team of everyone alike is not effective but expose them and learn from them"**

XP Principles

- **Reflection – "good teams don't just do their work, they think about how they are working and why they are working; they analyse why they succeeded or failed; they don't hide mistakes"**
- **Flow – deliver "a steady flow of valuable software by engaging in all the activities of development simultaneously"; no big chunks or discrete phases**
- **Opportunity – problems are opportunities for change**
- **Redundancy - resolve difficult problems in several different ways**
- **Failure – failure is not a waste if it imparts knowledge**
- **Quality – people need to do work they are proud of**
- **Baby Steps – less overhead than aborting big changes**
- **Accepted responsibility – responsibility cannot be assigned, only accepted**

Primary XP Practices

- Sit Together to “communicate with all our senses” whether it’s a chair next to someone, conference room for problem solving or open office - physical proximity enhances communication
- Whole Team - include people with ALL the skills and perspectives necessary for the project to succeed, create a sense of a “team’
- Informative Workspace - an observer should be able to work into the team space and get an idea of how the project is going in 15 seconds; create “Stories” on the wall with areas for Done, This week, This Release, Future
- Energised Work - “software development is a game of insight, and insight comes to the prepared, rested, relaxed mind,” burning yourself out unproductively today, or coming in sick is not good for the team in long run
- Pair Programming (Buddy Programming) - write all production code with two people seating at one machine to keep on track, brainstorm for refinements, clarify the ideas (more intense and satisfying but also more tiring)

Note how the first Primary Practices are about PEOPLE and Organising Teams, Peopleware is much more important than Software

Primary XP Practices

- **Stories - move from “requirements” (mandatory) to stories with description of one functionality per story (“add email button”) plus estimated required effort - helps to make informed decisions (Do I want a Castle for 100 mln or a flat for 100K)**
- **Weekly cycle and Quarterly Planning - a meeting per week to review last week, pick stories for the next week, subdivide into tasks for team members to take responsibility and estimate; plan quarterly to reflect and look at bigger picture**
- **Slack - in any plan include any minor tasks that can be dropped**

Note these are about Project management

Primary XP Practices

- **Ten Minute Build – *automatically* build the *whole* system and run *all* of the tests in 10 minutes; automated build is a "stress-reliever at crunch time"**
- **Continuous Integration – integrate and test changes after no more than a couple of hours**
- **Test-First Programming – write a failing automated test before changing any code (defines scope, identifies other problems, builds trust)**
- **Incremental Design – constantly bring the design back into alignment with your ever increasing understanding**

XP Practices Corollary Practices

- **Real Customer Involvement**
- **Incremental Deployment – gradually, no big bang**
- **Team Continuity – keep effective teams together**
- **Root-Cause Analysis – each time eliminate not only a defect but also its cause**
- **Shared Code – anyone on the team can improve any part of the code any time – collective ownership**
- **Code and Tests – maintain Code and Tests, generate other documents from Code and Tests**
- **Single Code Base – there is only one code stream, temporary branches may live no longer than a few hours**
- **Daily Deployment – put new software into production every night, do not let the programmer to be out of sync**

Agile Manifesto

- By 2001 Extreme Programming was one of several early implementations of what will soon be knowns as *agile software development* methods
- These methodologies had a lot in common, shared a lot of the same values, principles and practices (and many of these values, principles and practices were not new, and were discussed earlier)
- On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people who thought very much alike met to talk, ski, relax, and try to find common ground and of course, to eat. In particular, they were discussing the *lightweight* methods. This is how the Manifesto for Agile Software Development emerged.
- They also defined 12 Principles

Agile Manifesto

http://agilemanifesto.org/

Agile Methodologies

Wikipedia-known agile software development methods include:

- **Agile Modeling**
- **Agile Unified Process (AUP)**
- **Dynamic Systems Development Method (DSDM)**
- **Essential Unified Process (EssUP)**
- **Exia Process (ExP)**
- **Extreme Programming (XP)**
- **Test Driven Development (TDD)**
- **Feature Driven Development (FDD)**
- **Open Unified Process (OpenUP)**
- **Scrum**
- **Crystal Clear**
- **Velocity tracking**
- **Kanban (development)**
- **GSD**

Continuous Integration (CI)

- **Was introduced as part of Extreme Programming by Kent Beck**
- **Martin Fowler is also a big advocate of CI**
- http://martinfowler.com/articles/continuousIntegration.html
- **The idea: integration is unpredictable so has to become a part of each "baby step" - integrate and test changes after no more than a couple of hours**
- **Check in changes, complete the build, run entire test suite and only then proceed**
- **Getting this all work smoothly is obviously a lot more than that**
- **Key ingredients:**
  - **Source repository**
  - **An Automated Build**
  - **Agreement of the Team to adopt this**

Source Repository

- **Keeping track of so many files is a major effort**
- **The tools for the job: Source Code Management tools called under configuration management, version control systems, repositories**
- **Source Repository is a MUST**
- **The current tools of choice are still: Subversion (open source, used to be CVS and is still used, there are newer versions such as distributed revision control system Mercurial, Git, Bazaar) and Perforce (need to pay for)**
- **A source repository provides**
  - a central place to store all source code
  - a historical record of what has been done over time
  - a facility to record a set of sources as a “release”
  - an ability to reconstruct a project as it was at any time in the past
  - a facility to create separate code branches and merge them later
- **Check in Everything (!!!) you need to create, install, run and test: code, tests, database scripts, build and deployments scripts, etc.**

Automated Build

- Getting code turned into a running system can be a complicated task involving compilation, moving files around, clicking through dialog boxes, typing strange commands - waste of time and a source for mistakes - like everything can be and must be automated
- Automated environments are a common feature of a system: Ant for Java, Nant or MSBuild for .NET etc.
- Everything must be included (everyone should be able to bring a new machine, check everything out, and have a running system on their machine)
- Depending on your needs you need to build different targets: with or without test code, some components might be built as stand-alone, etc.
- Many IDEs (integrated development environment such as Eclipse, Microsoft Visual Studio, Xcode) have *build management process* - convenient for developers but might be fragile
- Everything should be launched using a single command line even if it is just to say to order your IDE to do it

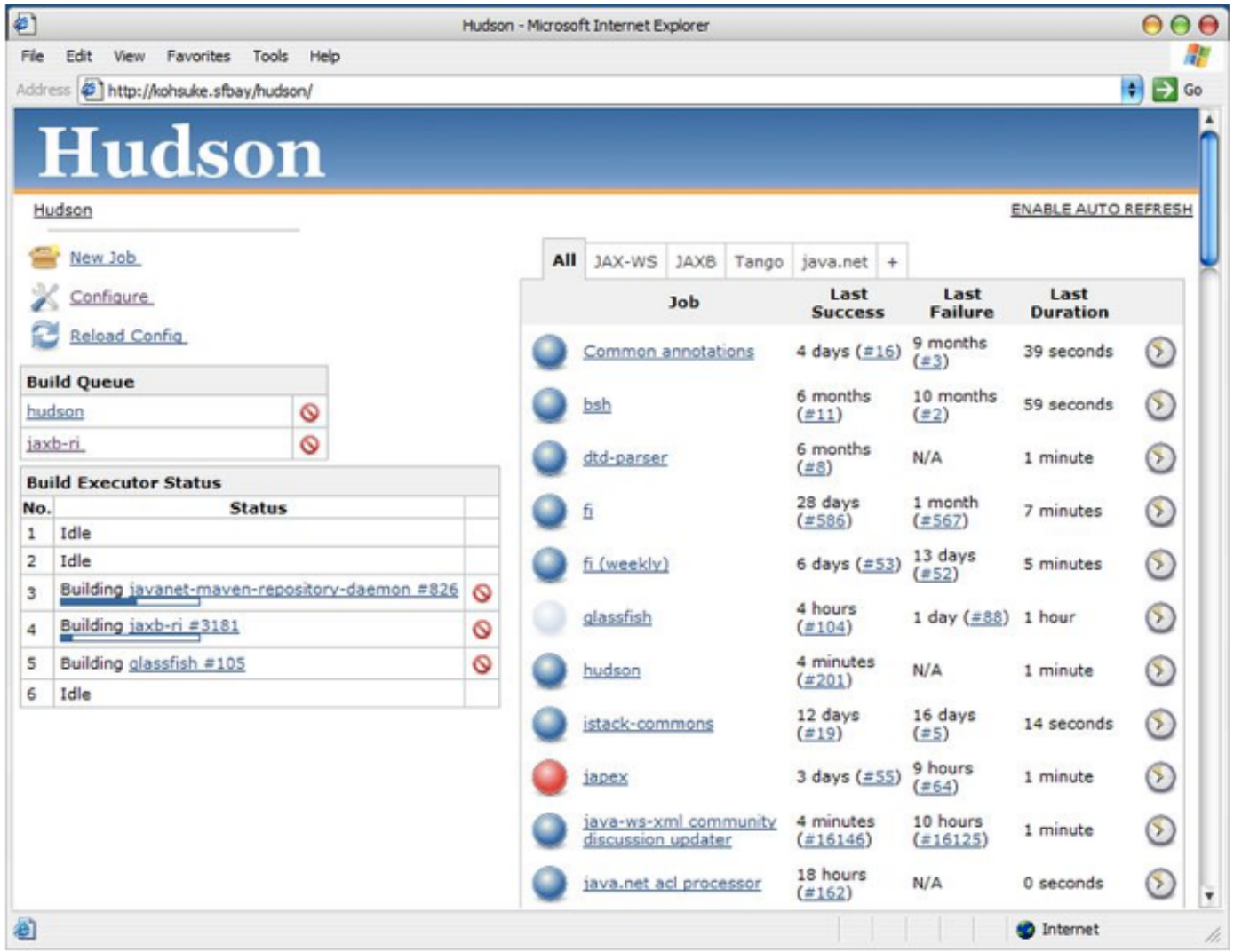
CI Key practices

- Maintain a Single Source Repository - merging anything is difficult, keep the number of branches to a minimum (temporary experiment or pre-production prototypes bug fixes are needed of course)
- Automate the Build - run via the command line without your IDE to automate, treat build scripts as codebase (test and refactor), be independent of IDE configurations, etc.
- Make Your Build Self-Testing - a program runs but does it do the same thing?
- Everyone Commits To the Mainline Every Day
- Every Commit Should Build the Mainline on an Integration Machine
- Keep the Build Fast
- Test in a Clone of the Production Environment
- Make it Easy for Anyone to Get the Latest Executable
- Everyone can see what's happening
- Automate Deployment

CI Software

- Basic functionality: check with your repository if any commits have occured, if so, check out the latest version of the software, run your build script to compile the software, run the tests to notify about the results
- Components:
  - long-running process that executes a workflow at regular intervals
  - a view of the results, notification of success of failure and access to the reports
  - a web server is often included to show a list of builds that run and to show the report
- One of the most common tools: open source Jenkins released under hte MIT licence (previously known as Hudson)

Jenkins (previously known as Hudson)



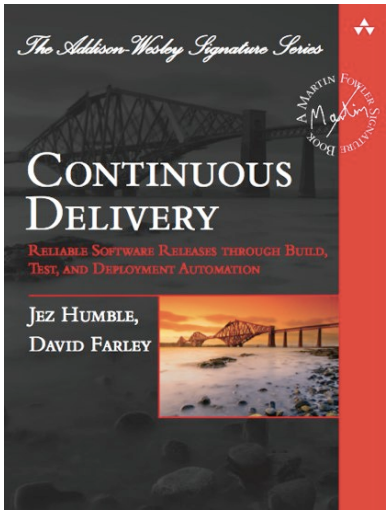
Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*

Continuous Delivery

- **Continuous Integration was the foundation for Continuous Delivery**
- **Traditional Release Candidate - a change to the code may or may not be releasable (or sufficient quality and functionally complete), release candidate is identified at the end of the process, means testing is significantly delayed, expensive and stressful**

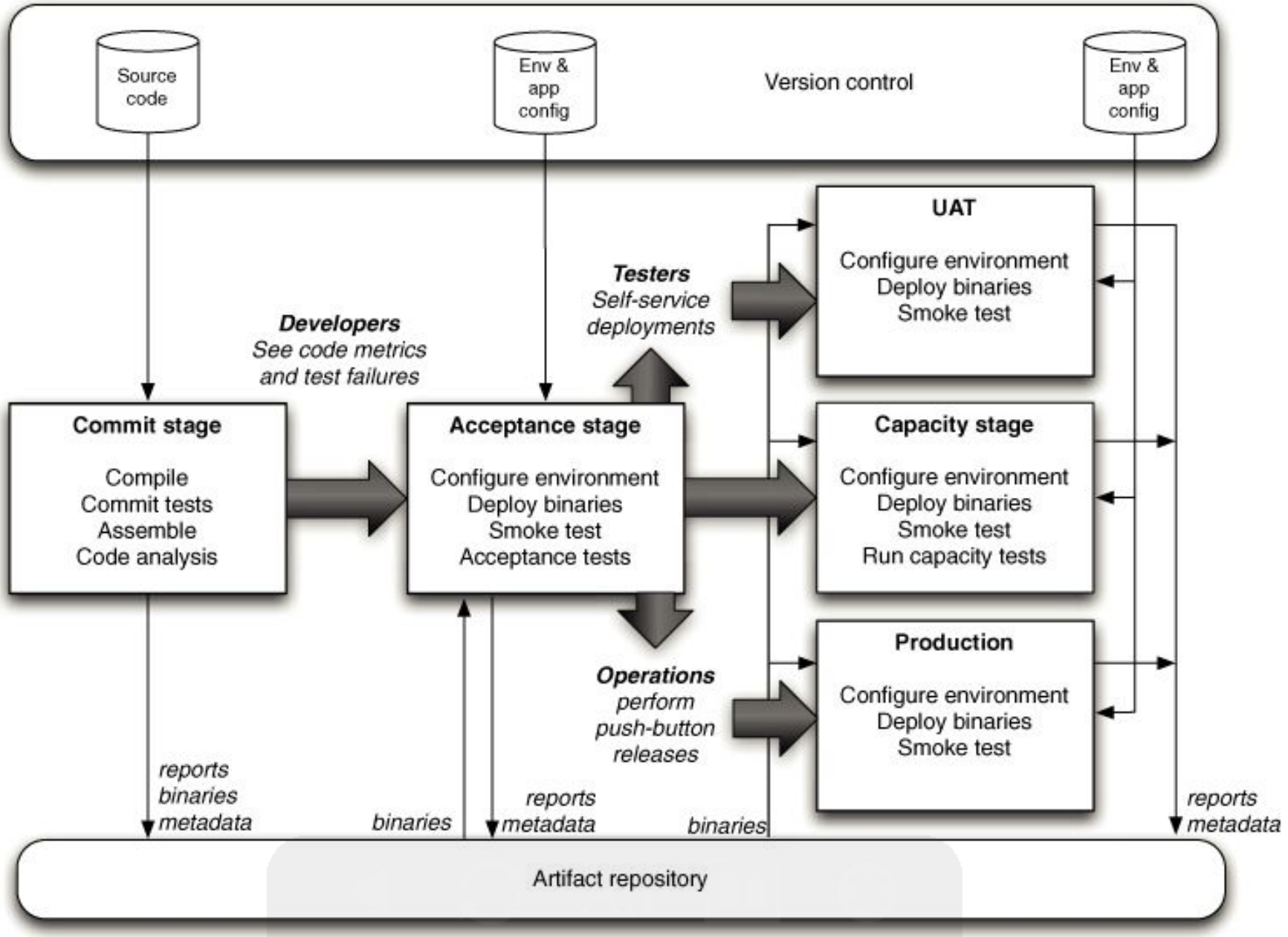
Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*

- **A different approach: build the software that is always in a production ready state**
- **CI? yes, but not enough**
- **This is achieved by constantly running a deployment pipeline that tests whether the deployment is in a state to be delivered**

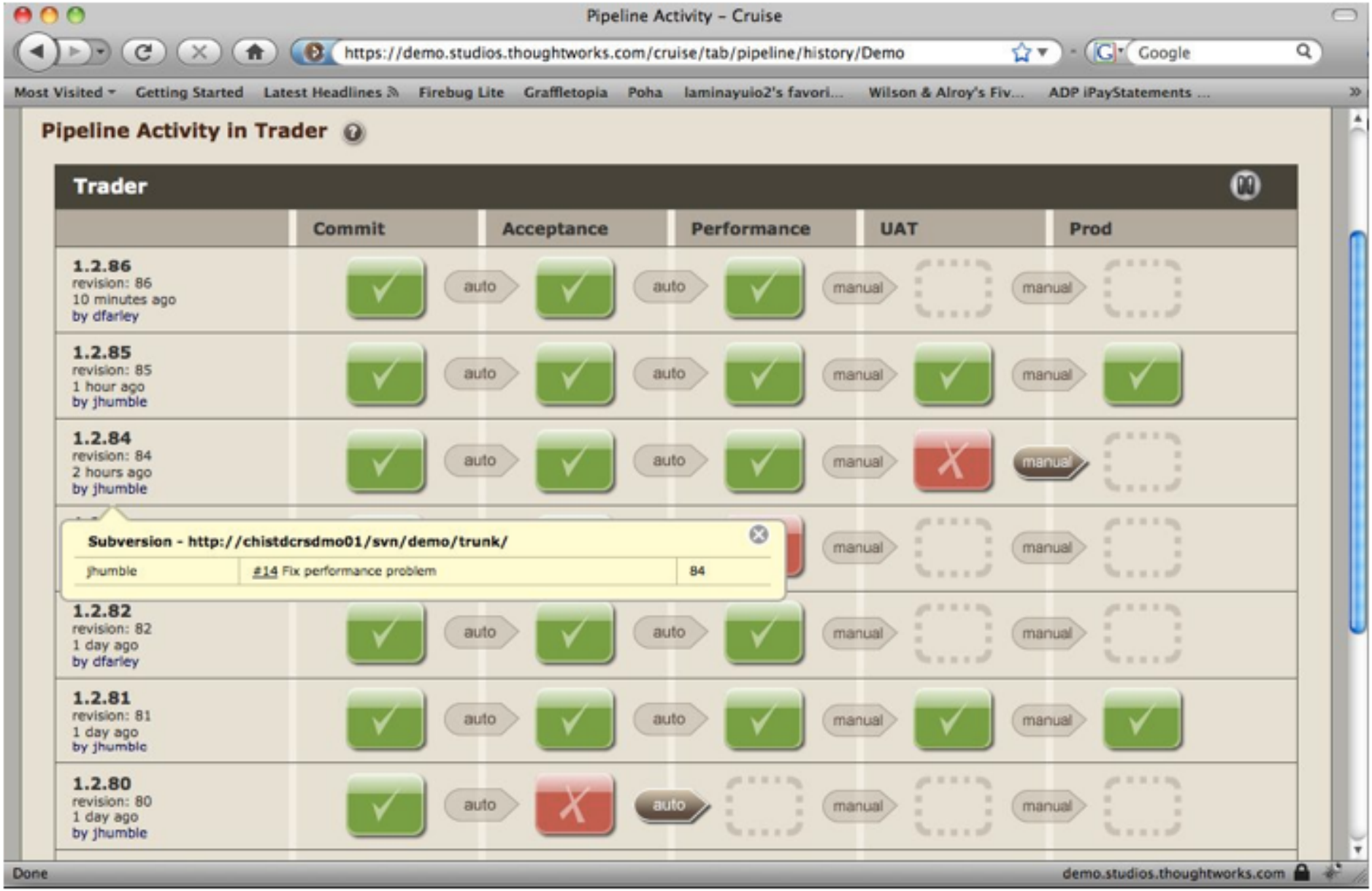


Deployment Pipeline

- CI mainly focuses on development teams - not enough
- Deployment pipeline reflects the process of getting the software from source control into the hands of the users - automated software delivery system
- First Stage: developers commit changes into their source repository; the first (commit) stage compiles the code, runs the tests, creates installers; if all ok assemble the executable code into binaries and store them in an artifact repository
- Second Stage: automatically triggered by successful first stage - longer running automated acceptance tests
- Third stage: pipeline branches to enable independent deployment of your builds in various environments (user acceptance tests, capacity tests and production). The testers should be able to see the release candidates available to them and their status plus any comments and at a press of a button deploy in a relevant environment


Source code
Env & app config
Version control
Env & app config
UAT
Configure environment
Deploy binaries
Smoke test
Testers
Self-service deployments
Developers
See code metrics and test failures
Commit stage
Compile
Commit tests
Assemble
Code analysis
Acceptance stage
Configure environment
Deploy binaries
Smoke test
Acceptance tests
Capacity stage
Configure environment
Deploy binaries
Smoke test
Run capacity tests
Operations
perform push-button releases
Production
Configure environment
Deploy binaries
Smoke test
reports
binaries
metadata
binaries
reports
metadata
binaries
reports
metadata
Artifact repository

Go showing which changes passed which stages



Is it strategically important?

- Keeping the system constantly production has huge benefits
  - allows more frequent feedback from business colleagues, user, etc.
  - can release important features earlier (edge to competitors)
  - but being closer and understanding users better developers may create something new
  - delivering better reliability and stability
  - automating repeated tasks saves time
- BUT also comes at a cost
  - more intense collaboration between departments
  - more investment in automation
  - more effort required to deploy regularly
- One needs to assess whether this is strategically important to your particular business case

Test Driven Development (TDD)

- Having a comprehensive test suite is essential for Continuous Deployment
- "Test - Code - Refactor" rhymes rediscovered by Kent Beck and is related to the Test-First Programming practice from *Extreme Programming*
- Repetition of a very short development cycle:
  - write an automated test that defines new improvement/functionality and make it fail
  - write code to introduce the improvement/functionality that passes the test by any means necessary
  - refactor the code to acceptable standard
- More recently has created more general interest in its own right
- The technique is heavily emphasized by those using agile software development methodologies
- There might be slight variations among TDD practitioners
- Has benefits and shortcomings so choose your strategy wisely

TDD Benefits

- The suit of unit tests provides constant feedback that everything is still working
- The unit tests also act as documentation that never goes out of date, everything else is generated from this documentation when and only when needed - no time or effort wasted
- Test passes and refactoring done means “it’s done, move on”
- The developer has much better understanding what the result should be before he/she even starts - better design
- The developer has confidence and freedom: the code always works as the tests are always running, the code always can be refactored so no need to get it right immediately - better design
- Significantly reduced debugging time

What is a Good Unit Test?

- Runs fast (otherwise is not used often enough)
- Is very limited in scope, tests one and one thing only (if fails, obvious where to look for a problem)
- Run and pass in isolation no matter where it is run (don't depend on the environmental set up)
- Simulates any dependencies (no calls out to a database or filesystem)
- Clearly reveals its purpose (anyone can look and understand what the production code should do)

TDD Shortcomings

- Difficult to use when full functional tests are required to determine success, for example, UI, software that works with databases or needs special network setup, etc
- Both tests and code is written by the same person - blind spots might exist
- Might create a false sense of security and lead to fewer integration and other tests
- Tests become a maintenance overhead (they may have bugs, may need refactoring, etc.)
- High dependence on earlier written tests that become of huge significance

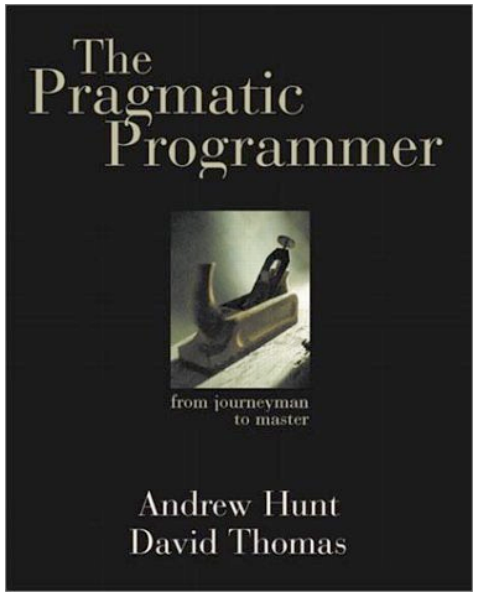
Refactoring

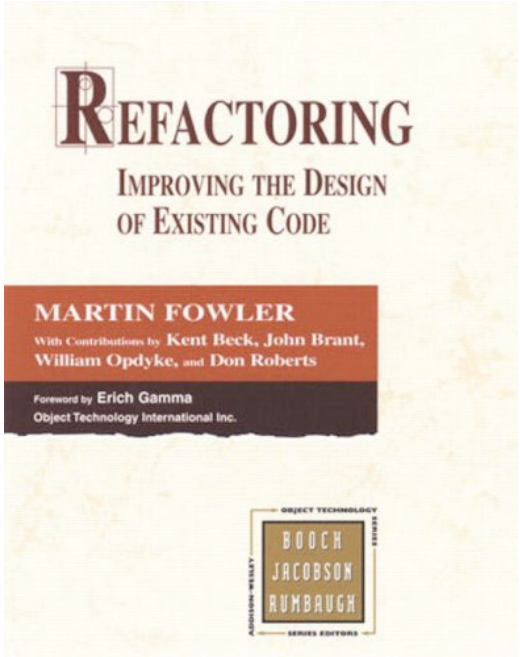
- Building software is not building constructions (an architect draws up blueprints; contractors dig the foundation and build the structure; people move in and live happily ever after) - software evolves
- As it evolves it becomes necessary to rethink earlier decisions and to rework portions of the code
- Refactoring - rewriting, reworking and re-architecting the code
- The questions are:
- When to refactor?
  - if something looks wrong - change it if possible - there is no time like the present; if not possible “leave the programmer’s debt”
  - the sooner the cheaper (hurts more later)
- How to refactor? Depends on your methodology ...
  - subdivide into baby steps
  - make sure solid tests are in place
  - avoid adding new functionality while refactoring unless necessary

Code Smells

- Code Smell - is a hint that something might be wrong
- "Bad Smells in Code" an essay by KentBeck and MartinFowler, Chapter 3 of *Refactoring* by Martin Fowler

- Among Other "Smells"
  - -Duplicated code
  - -Long Method
  - -Large Class
  - -Long Parameter List
  - -Parallel Inheritance Hierarchies
  - -etc.
- Pragmatic Approach: consider on a case by case basis; leave 'programmer debt"





Behaviour Driven Development (BDD)

- **BDD is a technique developed by Dan North as a response to the problems he experience teaching TDD (where to start, what to do, what to call, etc.)**
- http://dannorth.net/introducing-bdd/
- **"Behaviour" is a better word than "test" - different focus**
  - Unit tests are behaviours that are described by sentences starting with "Should .."
  - Acceptance tests are User Stories "As a [role] I want [feature] so that [benefit]"
  - Acceptance criteria in terms of scenarios: "Given [initial context], when [event occurs], then [ensure some outcomes]"
- **All must be in order of importance or business value**
- **BDD provides a "ubiquitous language" for analysis process**
- **Andrew Glover (the founder of the easyb BDD framework and the co-author of "Continuous Integration", "Groovy in Action" and "Java Testing Patterns"): "BDD is TDD done right"**

Root-Cause Analyses - Five Whys

- When using an adaptive or agile methodology there is a need in a natural feedback loop.
- Kent Beck recommends the use of root-cause analyses technique Five Whys - a bit like a child who keeps asking “Why?”
- The idea: each time eliminate not only a defect but also its cause
- “Five Whys” was originally introduced by Taiichi Ohno, the Father of the Toyota Production System and the Lean Manufacturing: preserve value with less work by eliminating waste
- The Lean Startup 2012 takes it and develops it further by introducing incremental proportional investment at each stage
- http://ecorner.stanford.edu/authorMaterialInfo.html?mid=2296
- The idea is: repeating “Why?” five times helps uncover the root of the problem and correct it
- Don’t let Five Whys become Five Blames!

Five Whys in Action

- **Eric Ries, 'The Lean Startup," the IGN (American global entertainment website that focuses on video games, films, music and other media) story**
- **http://www.ign.com/blogs/ign-tech/2011/02/17/blogs-outage-and-five-whys**

Why couldn't you add or edit posts on the blogs?

Answer: The article content api posts were returning 500 errors *Proportional Investment:* Allow users to edit drafts without errors, better user experience.

Why was the content api returning 500 errors?

Answer: The bson_ext gem *[a packaged Ruby application or library]* was incompatible with other gems it depends upon. *Proportional Investment:* Remove the gem.

Why was the gem incompatible?

Answer: We added a new version of the gem in addition to the existing version and the app started using it unexpectedly. *Proportional Investment*: convert our app to use Bundler for gem management.

IGN Example

Why did we add a new version of a gem in production without testing?

Answer: We didn't think we needed a test in these cases. *Proportional Investment:* Write a test for the api that failed in this case

Why do we add additional gems that we don't intend to use right way?

Answer: In preparation for a code push we wanted to get all new gems ready in the production environment. *Proportional Investment:* Automate gem management and installation into Continuous Integration and Continuous Delivery process.

Bonus Why - Why are we doing things in production on Friday night?

Answer: Noone says we can’t and it was convenient for the developer. *Proportional Investment:*

Until we are fully CD, no production changes on the apis on Friday, Saturday, and Sunday unless an exception has been made and approved.

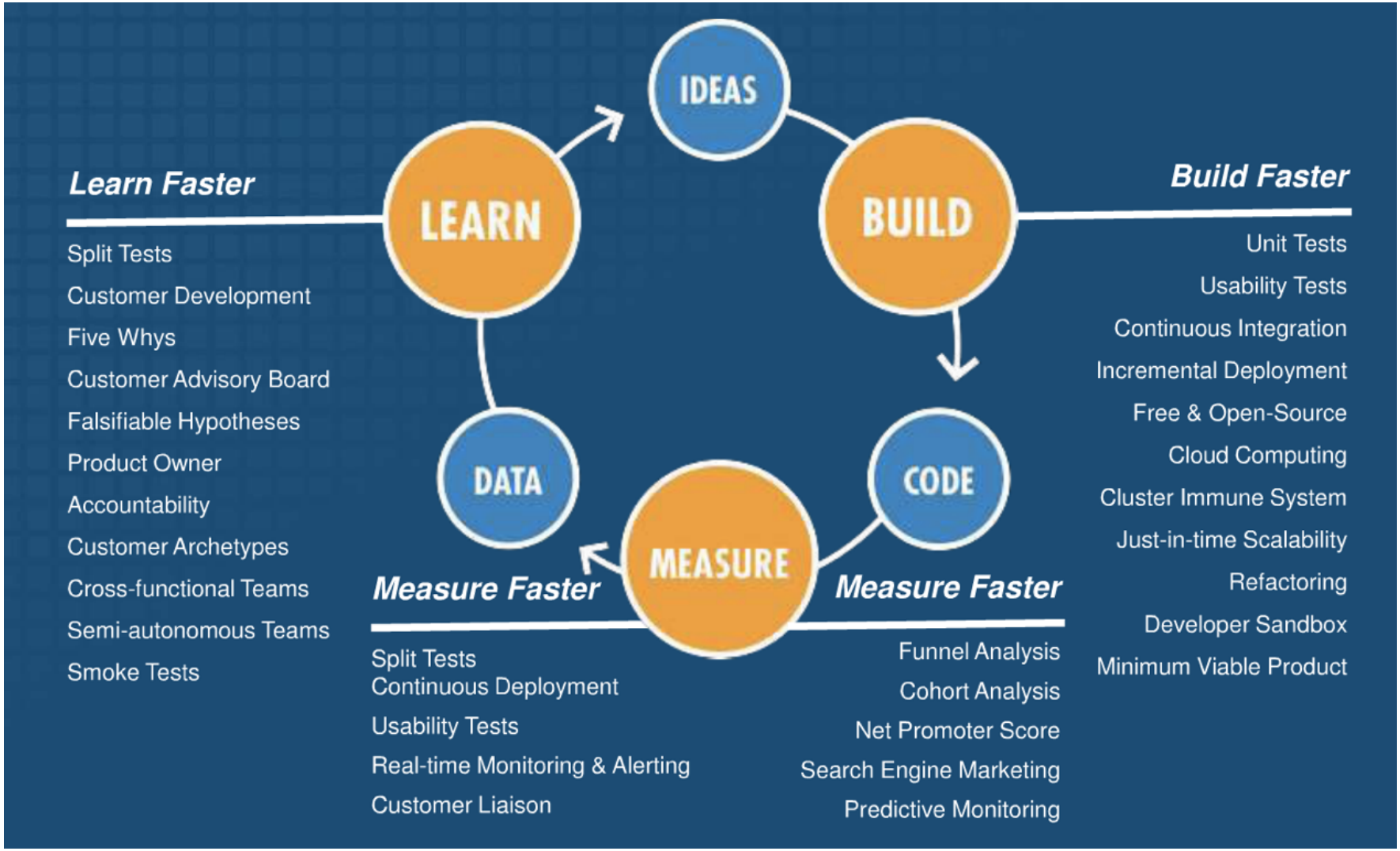
Benefits

- **Without Five Whys one developer would be told not to do silly things and move on**

- **With Five Whys the whole deployment was made easier, quicker and never ever the process will allow a developer to place gems into production system with unintended consequences**

Software Industry Evolving: 2012 – Lean Startup

These methods are constantly evolving and other approaches keep emerging such as very recent Lean Startup



Eric Ries - The Lean Startup - Google Tech Talk, http://www.slideshare.net/startuplessonslearned/eric-ries-the-lean-startup-google-tech-talk

Where Next?

"… I can only show you the door. You're the one that has to walk through it …"

…anything is possible. Where we go from there is a choice I leave to you."