Develpment Practices

ENGF0002: Design and Professional Skills

Mark Handley, University College London
based on slides from Prof. George Danezis

Term 1, 2018

Outline

This topic is all about **scaling up** the development process, while maintaining **feasibility and quality**.

- Traditional software **development methods & disasters**.
- The **Agile** software development framework.
- **Tools** to support development teams.

Traditional software development.

Software development methods & project management.

Different than traditional engineering management:

- Recent discipline, immature processes and tools (**incidental complexity**).
- **Intrinsic complexity** increased through **no physical constraints**.
- Tight integration between requirements, specification, implementation, operation – no **manufacturing** stage.
- Complexity increased through **flexibility** – ever changing & evolving requirements.
- Difficult to **estimate** time to completion & defect rate.
- Orders of magnitude differences in **programmer productivity**.

Anti-pattern: The Waterfall model.

Adapted from other engineering (construction). The **Waterfall model** structures a project as a sequence of:

- **Requirements gathering**
- **System Design & Specification**.
  Also known as ‘Big Design Up Front’.
- **Implementation**
- **Testing & Validation**
- **Deployment & Maintenance**

Winston Royce (Director at Lockheed Software Technology Center) in 1970 presents the model as a ‘bad idea’. It is later (1985) formalized – in terms of auditing steps – as a methodology in US Dept. of Defense ‘Defense Systems Software Development’ (DOD-STD-2167A). Subsequent MIL-STD-498 in 1994 states a preference for ‘Iterative and incremental development’.

Problems with the Waterfall model.

All activities included in the model need to take place at some point. However, a linear end sequential application is not appropriate, because:

- **Users do not know** or cannot formulate requirements, until they see working software (prototypes, early versions, . . .)
- **Designers may not foresee** all issues related to the implementation, or be able to chose between design options, without prototyping and testing.
- When **different people** are employed in different phases, **communication breakdown** may occur; and only a **fraction of the team is engaged** at any point.

Poorly adapted to **‘brown-field’ development**, where existing software needs to be extended, rather than ‘green-field’ new projects.

The tar pit (Brooks 1975)

The Mythical Man-Month. Essays on Software Engineering by Frederick Brooks Jr. Anniversary Edition Paperback, 322 pages Published by Addison-Wesley Pub Co in July 1995. (First edition 1975.)

Lessons from ‘The Mythical Man-Month’ (1)

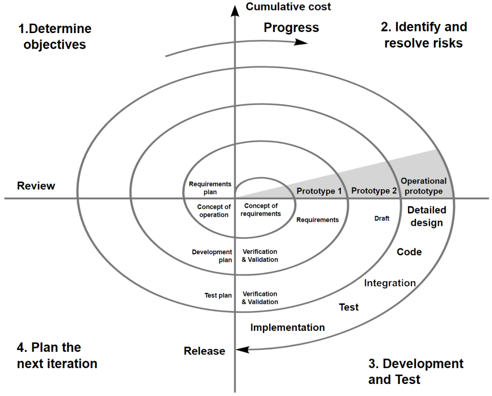
Brooks was an architect on the IBM OS/360 – delivered epically late in the 1970s.

- **Brooks’s law**: ‘*Adding manpower to a late software project makes it later*’. The communication channels increase as $\mathcal{O}(n^2)$ in the number n of developers.
- **No silver bullet**: No single development method, language or tool can reduce software dev. by an order of magnitude, since they may only reduce incidental not intrinsic complexity.
- **The second-system effect**: Be aware of the second version of a system, since the architects will try to include all options they did not include in the first.

Lessons from ‘The Mythical Man-Month’ (2)

- **The pilot system**: when designing a new system, plan for an initial pilot (prototype) which will help you learn. Throw it away, and deliver the second one.
- **Project estimation**: programming products take much longer to build, than internal tools; a lot of time is taken by communications, ‘stand-ups’ and ‘all-hands’.
- **The surgical team**: Build small teams around and supporting experienced programmers.
- **Code freeze**: freeze features at some point, and only increase the quality of code, until next release.

Spiral model to manage risk.



Boehm “A Spiral Model of Software Development and Enhancement”, ACM SIGSOFT Software Engineering Notes, ACM, 11(4):14-24, August 1986

Agile development and practices.

The Manifesto for Agile Software Development (2001).

Radical: Re-focus on **software as working code**: ‘The need for an **alternative to documentation driven, heavyweight** software development processes.’

Why the crisis? Traditional software **project failures** in 1995:

- 16.2% – Project Success: on time, budget, features.
- 52.7% – Project Challenged: overrun time, budget, fewer features.
- 31.1% – Project Impaired: canceled at some point.

(Standish Group, CHAOS Report, 1995)

The Manifesto (1–6).

- Our highest priority is to satisfy the customer through **early and continuous delivery of valuable software**.
- Welcome **changing requirements, even late** in development.
- Deliver **working software** frequently, from **a couple of weeks to a couple of months**, with a preference to the shorter timescale.
- **Business people** and **developers must work together** daily throughout the project.
- Build projects around **motivated individuals**. Give them the support they need, and **trust them** to get the job done.
- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

The Manifesto (7–12).

- **Working software is the primary measure of progress**.
- Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a **constant pace indefinitely**.
- Continuous attention to **technical excellence and good design enhances agility**.
- Simplicity–the art of **maximizing the amount of work not done**–is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**.
- At regular intervals, the team **reflects on how to become more effective**, then tunes and adjusts its behavior accordingly.

Composition of an Agile team

- **The product manager / owner** – Maintain and promote the product vision. Part management, part people coordination, part marketing.
- **On-site customers** – team members representing customers, continuously capturing and discussing requirements, and discussing with programmers.
  Can be business people. Include 2 per 3 programmers!
- **Programmers** – Bulk of the team, no distinction between 'architects', 'developers', 'testers'. Include one senior, hands-on, person and overall teams of 4–9 programmers. Although some members may have a focus.
- **Domain experts, Interaction Designers, Security, Design, and Business Analysts** – experts in their respective domains, are on-call to help on-demand. Work with both users and team.

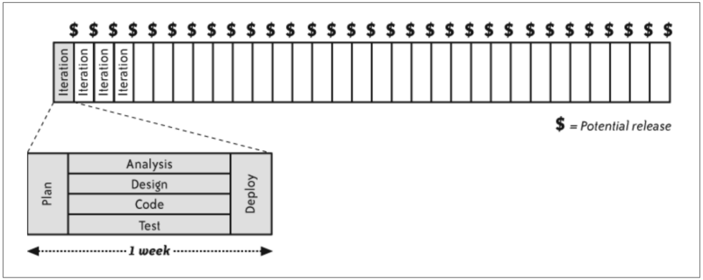
Ratio of on-site customers to programmers

Do you think the ratio is too high or too low?

What processes do user representatives speed up?

The rhythm of an Agile project – Sprints

"You can eliminate requirements, design, and testing phases as well as the formal documents that go with them."



Agile (XP) Lifecycle. Sprints are typically 1-2 weeks.

The Art of Agile Development by James Shore. Wiley 2007.

User Stories

Stories represent **self-contained, individual elements** of the project:

- Individual features.
- Typically represent a few days of work.
- Customer-centric, in terms of business results.
- No implementation details, or full requirements / specifications.

Good user stories have the form:

As a *<type of user>* I want *<some goal>* so that *<some reason>*.

User Stories

E.g. a high level user story example for a laptop backup product:

- *As a user, I can backup my entire hard drive.*

So called “epic” user stories are usually split into smaller user stories before they are worked on:

- *As a power user, I can specify files or folders to backup based on file size, date created and date modified.*
- *As a user, I can indicate folders not to backup so that my backup drive isn’t filled up with things I don’t need saved*

Backlogs: Product, Release & Sprint

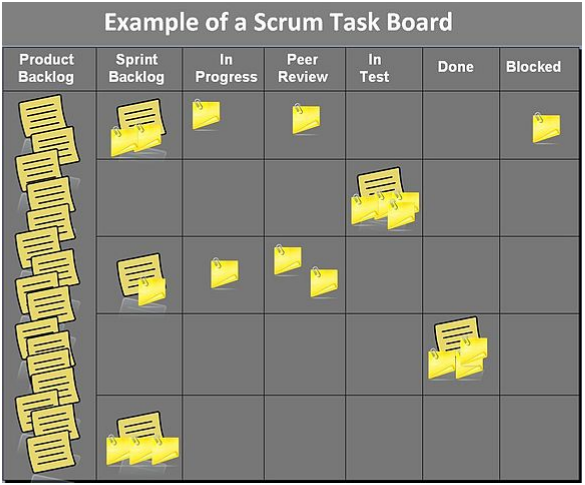
All stories are stored in the **product backlog**:

- When a new release is planned / discussed with customers some stories are moved to the **release backlog**. They are augmented by **acceptance criteria**.
- At the start of every sprint, some stories are moved into the **sprint backlog**, along with noted on **how to design & implement** them.

Programmers are asked to 'cost stories' in 'days', as stories progress through backlogs:

- Programmers are bad at estimating, but their estimates are **off by a fixed multiplicative constant**.
- **Velocity**: the number of story 'days' the project executes (Done Done!) in a day. Base **project estimation** on velocity.

Backlogs & stories: just a bunch of post-its or cards



'Kanban' board. From
`https://dzone.com/articles/product-backlogs-practice/`

On-line tools for Agile project management.

A number of tools can **replace physical post-it notes**:

- Trello – `http://trello.com`.
- Github projects – see ‘Projects’ tab on your project page.

Related but different tool: **Issue / Bug tracker.**

- Allow users, internal and external developers to **report issues**.
- Allows **discussion** on the issue and the development of a **minimal test case to reproduce the error**.
- Allows for **triage**, and **assignment** of issues to team members.
- Can be ‘abused’ to report feature requests, and simulate a story board.

Agile practices: Pair programming

Two programmers work together on a **single workstation**.

- One **driver / codes** the other **navigator / reviews & thinks**.
- They **switch roles** frequently.

Why pair?

- Navigator has opportunity to think about **strategic design** choices. Driver can focus on **code tactics**. Higher quality code.
- Reinforces, and shares **good programming habits**. Continuous testing and refinement result from **peer pressure**.
- Resilient to **interruptions**. When interrupted one person can handle interrupt, while other codes.
- Its **more fun** to not work alone. Can take **physical breaks**.

Focus on sustaining, rather than peaks of high production.

Pair programming or code reviews?

What do the practices have in common?

What are their advantages and disadvantages?

Agile practices: Documentation & Agile Modeling.

Keep the non-code project documentation to a minimum.

- **Document continuously**: **Minimal** stories, design notes and meeting notes.
- **Document late**: Do now write anything that will not be read or acted upon. Minimize speculation.
- **Single Source Information**: All project **documentation** has to be **kept under version control**.
- **Prefer Executable specifications**: Write as **customer tests** that can execute against the code. (TDD at the requirements level.)

All code and docs are kept in version control.

Agile practices: Distributed Version Control.

Use Distributed Version Control for **fearless refactoring** and **collaborative coding**:

- Tool of choice: `git` and `github`.
- Code lives in **repositories**, **remote** and **local**.
- A repository can have multiple **branches**.
- You may **branch** and **merge**, or **commit** code to a branch.

Learn git! `https://git-scm.com/documentation/external-links`

Gitflow illustrations by Vincent Driessen `http://nvie.com`.

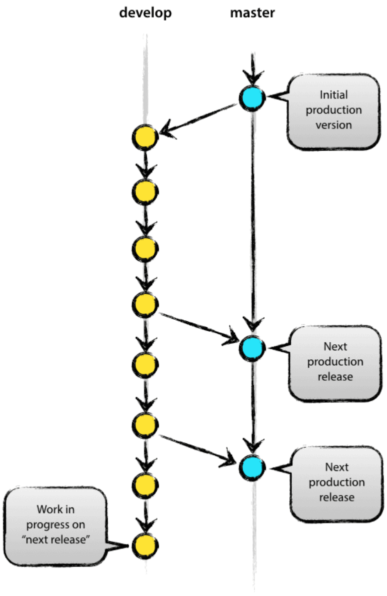
Agile practices: Gitflow & version control.

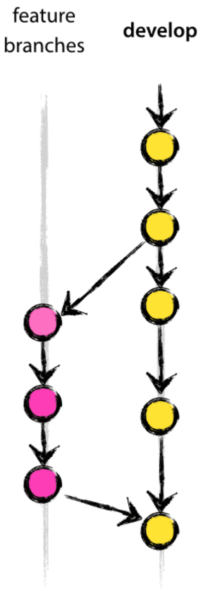
A **branch** in a git repository contains a ‘variant’ of your repository. You can **merge** a branch into another, after a number of **commits**.

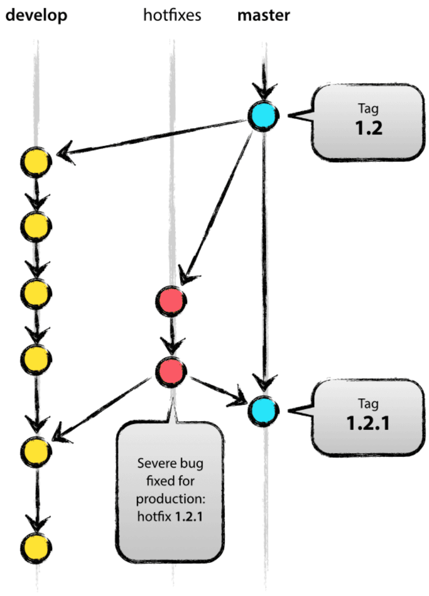
Gitflow basic organization for a repository:

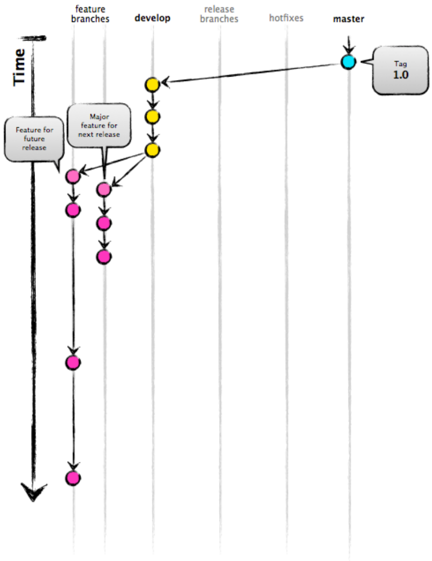
- The **master** branch only contains tagged releases.
- A **develop** branch is used to continuously update the software.
- New features are developed in separate **feature** branches.
- A **release branch** captures the features for a new release, and from there on only accepts **bug fixes**.
- Eventually a release branch is merged into master to make a new release.
- **Hotfixes** are applied to master and merged into develop.

Good programmers use it even when they work on their own!

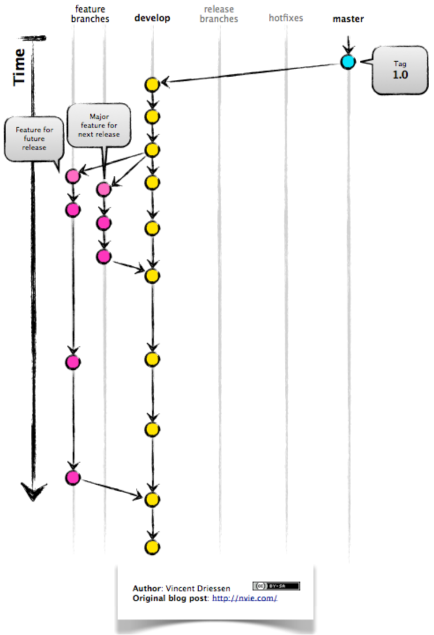

develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"


feature branches
develop

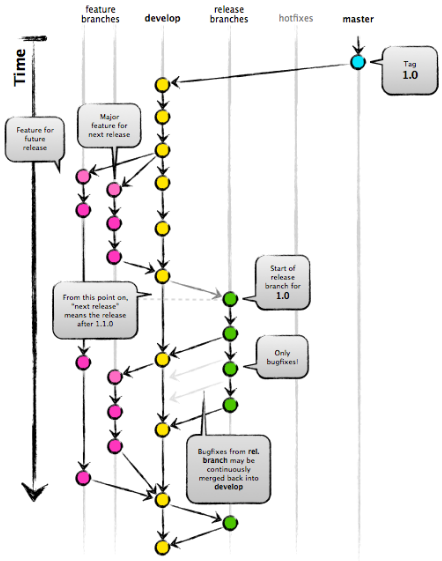

develop
hotfixes
master
Tag 1.2
Tag 1.2.1
Severe bug fixed for production: hotfix 1.2.1



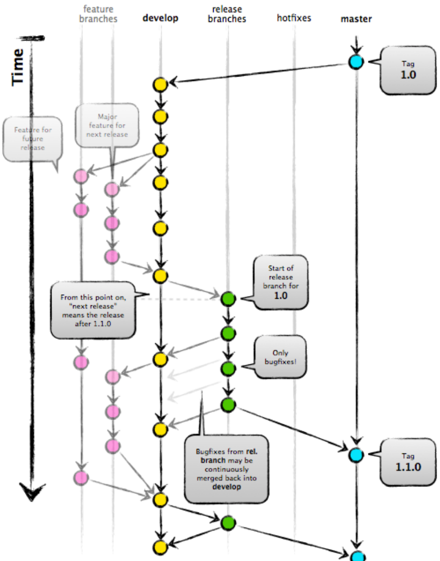
Author: Vincent Driessen
Original blog post: http://nvie.com/



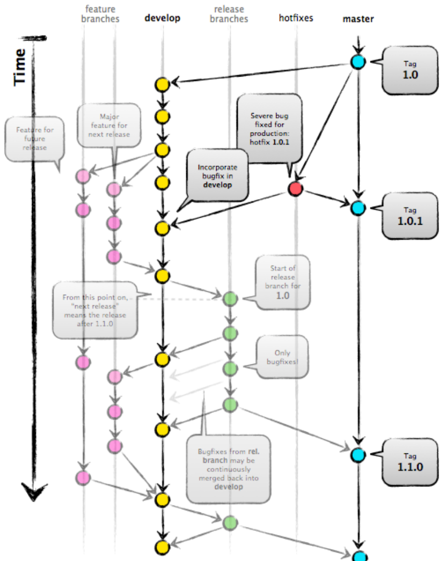
feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Author: Vincent Driessen
Original blog post: http://nvie.com/


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop

Author: Vincent Driessen
Original blog post: http://nvie.com/


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0

Author: Vincent Driessen
Original blog post: http://nvie.com/



feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0

Author: Vincent Driessen
Original blog post: http://nvie.com/

Why different feature branches?

Why develop different user stories on different feature branches?

What is the cost of diverging too much from the develop branch?

How do you ensure you do not diverge too much from the develop branch?

Agile practices:‘Done Done’.

Each feature branch represents a user story, and it is merged into develop when it is ‘**done done**.’:

- Fully Designed, Coded, Integrated (from UI to Database), Tested (unit, integration, and customer tests finished).
- Builds, Installs (incl. auto install & devops), Migrates past state.
- Fixed all bugs, Reviewed by customers and accepted as finished.

From working software to working software

Executing each user story until it is ‘done done’ takes working software and produces working software, end-to-end. This way users can see progress and provide early feedback.

Agile practices:Continuous Integration.

The 'build' engineer

Merging branches that are a lot of different commits apart, is time consuming. At traditional software firms this is the job of the 'build' engineer. 'Integration' and testing of different branches can take months.

Continuous Integration: be technically ready to release, even if you are not functionally ready.

- Create automated pipeline for testing integration, packaging, installers, deployment in servers, and data migration.
- Run those tests on every commit to develop branch.

Why testing continuously?

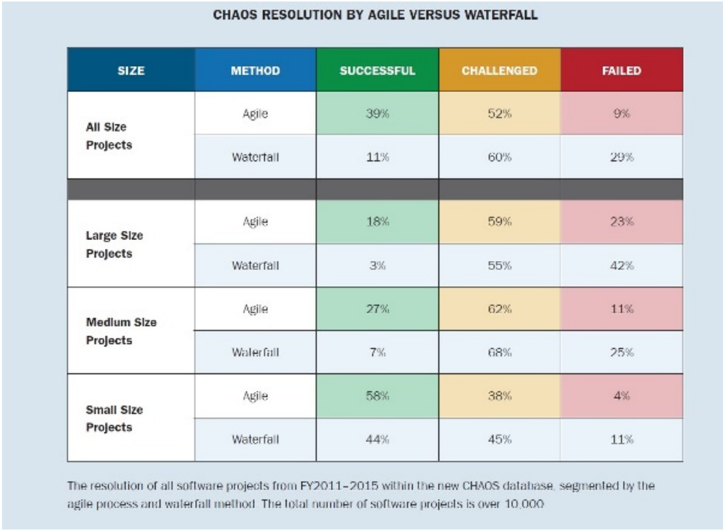
Integration, Install, Deployment, Migration: even when the features are not entirely finished or there?

Other Agile practices

Other tricks of the trade:

- Test driven development.
- Agile testing
  (specification by example.)
- Refactoring.
- Cross-functional teams.
- Information Radiators.
- Planning poker.
- Scrum events
  (planning, daily Stand-up meetings).
- Timeboxing.

Conclusion: ‘No silver bullet’.



CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

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

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

from `https://www.infoq.com/articles/standish-chaos-2015`