Reality Check

L09 PA1308

Mikael Svahnberg[1]

[1]Mikael.Svahnberg@bth.se
School of Computing
Blekinge Institute of Technology

October 1, 2012

Caveat

- I will *not* be able to tell you anything specific about any company's architectures.
- To be on the safe side, I will not even mention the companies by name or domain.
- This presentation is based on a merge of experiences gained at several companies.
  - All companies may not be exposed to all challenges.
  - Most are exposed to most, however.

Development Scale I

- Rich hardware environment; many nodes, of many different types of hardware.
- Rich software environment; many collaborating and/or communicating software systems
- Large software systems; several million lines of code per system, at the smallest.

Strategy: Modular design . . .

- in order to manage the size.
  (Concerns for the Module view)
- where the effect of different hardware is localized
  (Execution view $\rightarrow$ Module view)
- where the effect of interacting software is localized
  (Execution view $\rightarrow$ Module view)

Development Scale II

- *Many* developers
  - In multiple sites, in multiple countries

Further complications:

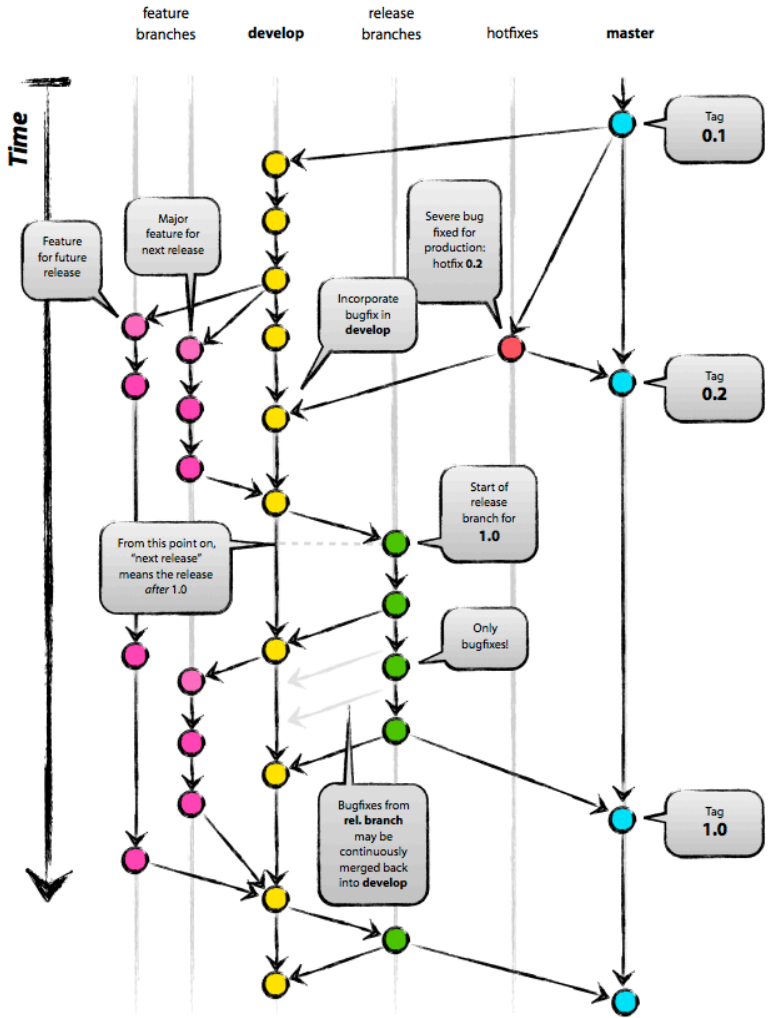
- One feature may span several software systems, and/or several modules, and/or several development teams.
- In order to implement one feature, changes may be required in another part of the system (not owned by you), so you need to deal with change requests.

Strategies:

- Modular design, to keep development teams small.
- Static code ownership vs rotating code ownership?
- Independent development (scrum teams)
  - Localise changes (including well-defined interfaces).
  - Distributed (but synchronised) backlog, or other mechanisms to

Concurrent Development

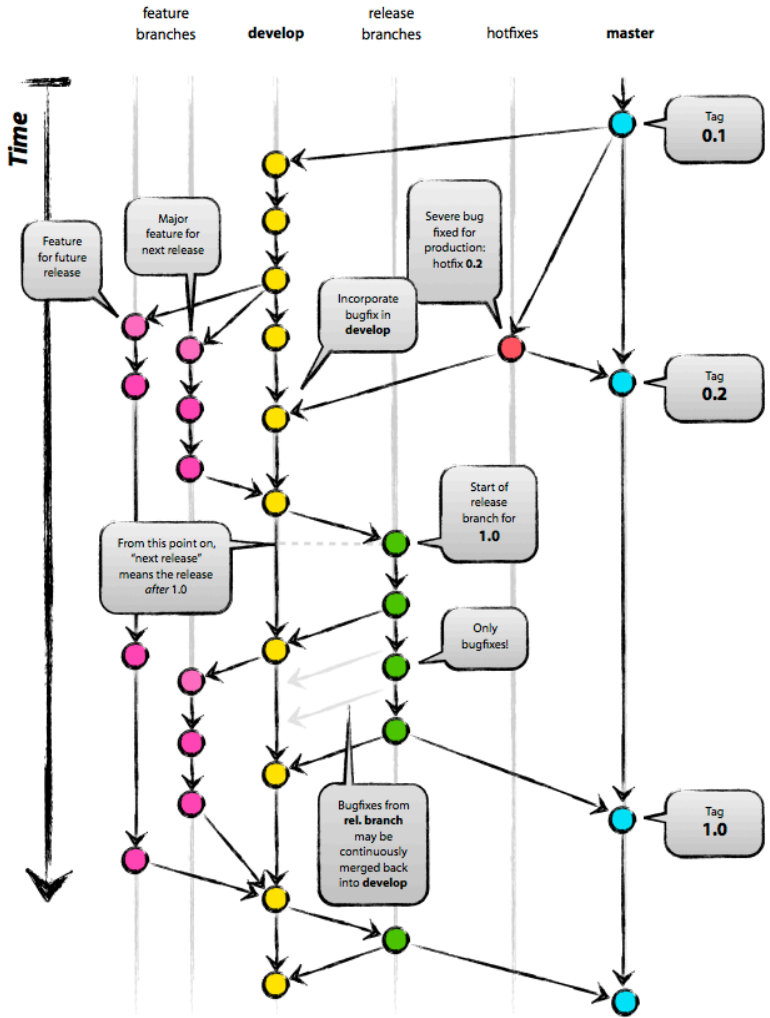
GitFlow



- `http://nvie.com/posts/a-successful-git-branching-model/`
- How can the *architecture* support development of many features concurrently?
- How can the *architecture* support merging of features and releases?

Customer Adaptations I

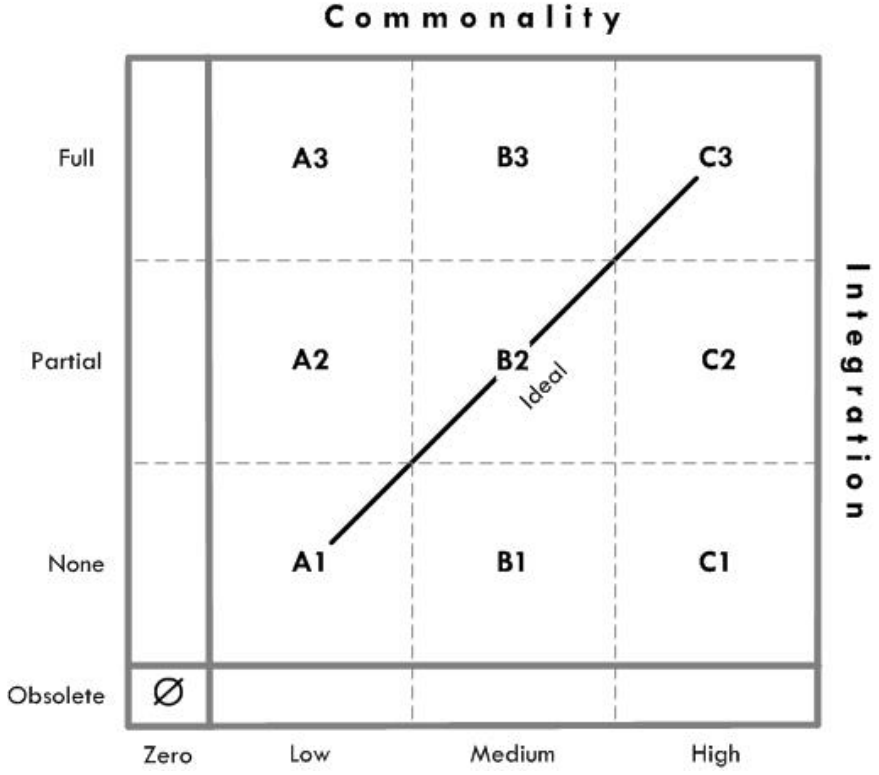
GitFlow



- Customer adaptations creates yet another CVS branch.
- Unlike feature development, customer adaptations may incur modifications in the entire system.
- When should you integrate a product customisation?

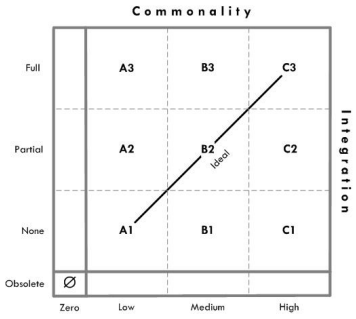
Customer Adaptations I

Product Customisations



- Customer adaptations creates yet another CVS branch.
- Unlike feature development, customer adaptations may incur modifications in the entire system.
- When should you integrate a product customisation?

Customer Adaptations II



- Position within the PC framework determines your choice of integration level
- Most loosely coupled: Keep the PC as a separate CVS branch
- Most tightly coupled: Integrate functionality in main line (at next merge point)
- If you have many PC's of low commonality in the same part of the architecture, you can create a *hotspot*.
  - A hotspot localises a particular type of change to a small set of components.
  - May introduce e.g. scripting languages to deal with the hostpot.
  - In other words, you *defer binding* of a variation point.
- Confounding factor: *build flags*.

Build Flags

- Probably the most common way to deal with variability
- In its simplest form it is a `#define` - statement
- There are, however, several challenges:

- Overuse of build flags: Every PC, almost every feature gets a build flag.

Example

An informal count at a company revealed that you had uptowards 32000 available configurations of the system. It is unknown which flags that are actually set together.

- One flag sets many variation points: One flag may be used in many places in the code.
- Include-or-exclude when set: Is the flag used with `#ifdef` or `#ifndef` or a combination?

Build Flags

- Probably the most common way to deal with variability
- In its simplest form it is a `#define` - statement
- There are, however, several challenges:

- Overuse of build flags: Every PC, almost every feature gets a build flag.

Example

An informal count at a company revealed that you had uptowards 32000 available configurations of the system. It is unknown which flags that are actually set together.

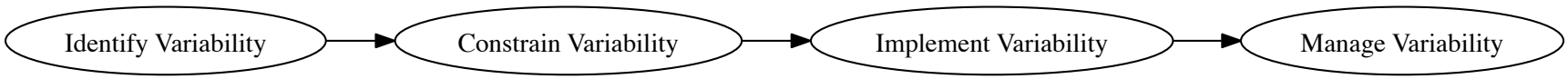
- One flag sets many variation points: One flag may be used in many places in the code.
- Include-or-exclude when set: Is the flag used with `#ifdef` or `#ifndef` or a combination?

Build Flags

- Probably the most common way to deal with variability
- In its simplest form it is a `#define` - statement
- There are, however, several challenges:
  - Overuse of build flags: Every PC, almost every feature gets a build flag.
  - One flag sets many variation points: One flag may be used in many places in the code.
  - Include-or-exclude when set: Is the flag used with `#ifdef` or `#ifndef` or a combination?
- May also impact architecture: include or exclude components, include or exclude interactions between components.

Variability I

- Actually, build flags and script languages are just two ways of dealing with variability[1]
- Generic workflow:



- States of a variant feature:

- Thus, a variant feature can be *introduced* at one time, *populated* at another, and *bound* at a third.

[1] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability

Variability II

- Introduction time: Architecture design, detailed design, implementation, compilation, linking
- Population time: Depends on the variation point.
- Binding time: Product architecture derivation (e.g. CVS checkout), Compilation, Linking, Runtime.

→ Depending on your choice of variability mechanism you may need more or less support from your architecture.

- Are your components of the right size to facilitate your variability mechanisms?
- Are the interactions between components such that you can reduce the number of variation points? Especially to variant components.
- ...

Reuse

- How does your architecture (and organisation) support reuse?
- How are variation points managed to enable reuse of components?
- Special case of reuse: Software Product Lines

Summary

- The architecture is not developed alone.
- Especially organisational factors influence your architectural choices.
  - Number of developers
  - Conway's law[2]
  - Size of features
  - Code branching
  - Number of customers, and customer adaptations
  - Choices of variability mechanisms
  - Company policies regarding reuse of software artefacts.

[2] M. Conway. How do committees invent. *Datamation*, 14(4):28?31, 1968.