UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Fabian Gilson

Software Engineering II

SENG301 - Continuous integration

Lecture 5 - Semester 1, 2020

University of Canterbury | Te Whare Wānanga o Waitaha
Computer Science and Software Engineering

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Fowler

Refactoring: Improving the Design of Existing Code – 1999

The integration problem

The integration problem

Process of combining **units** into a **product**

- after you developed your part…
- … and tested on your side

Even if every part is 100% functional, problems may arise

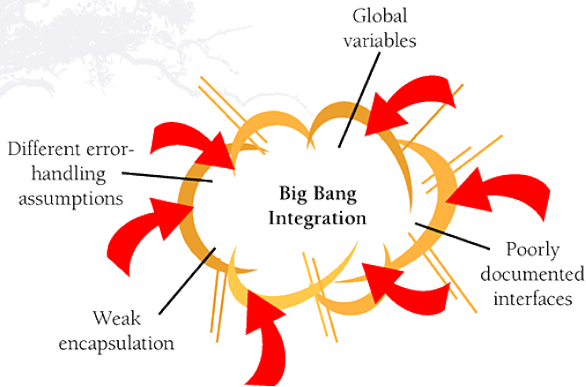
- conflicting (inter-) **dependencies**
- integration contracts (**APIs**) badly specified
- **diagnosis** is harder as more units are combined

Integration may become a nightmare

- if everyone builds things on his side …
- … and waits until the end before merging

Phased integration

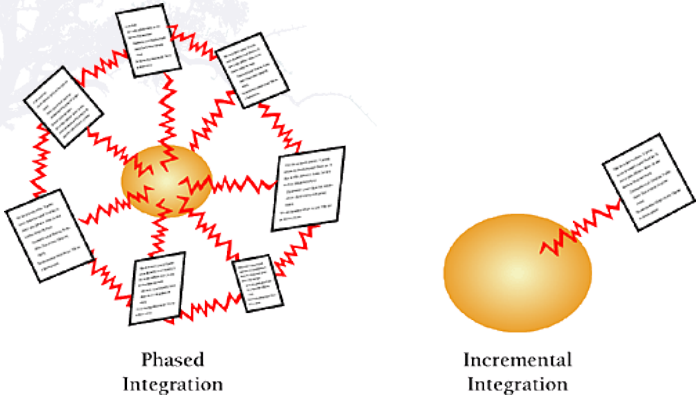
Multiple **variables**, **interfaces** and **technologies**



McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2004

Incremental integration

Smaller units, **one** at a time, **tested** as you go



McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2004


Integrate as you develop

Continuous integration

" Working software is the primary measure of progress. "

[Seventh principle of Agile manifesto]

eXtreme Programming has influenced Agile to focus on code

- iterative development also means **iterative integration**

→ **structured testing** through **gradual integrations**

The basic principles are

- testing a **smaller chunk** of program is easier and faster
- making those tests automated makes them **repeatable**

→ ensure the developed code meets his expected behaviour **at different levels**

Fowler's principles

Martin Fowler defined 11 principles (as part of the eXtreme Programming)

- maintain a **single source repository**
- **automate** the build
- make your **build self-testing**
- everyone **commits** to the mainline **every day**
- every commit should **build the mainline** on an integration machine
- **fix** broken builds **immediately**
- 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** the **deployment** phase

Managing sources

Single source repository

Maintain the code and resources at one single place

- many people are involved (not always at the same geographical place)
- centralised traceability of changes
- newcomer may spawn the project and start working

Tool support for version control

centralized one single master repository, *e.g.*, *subversion*

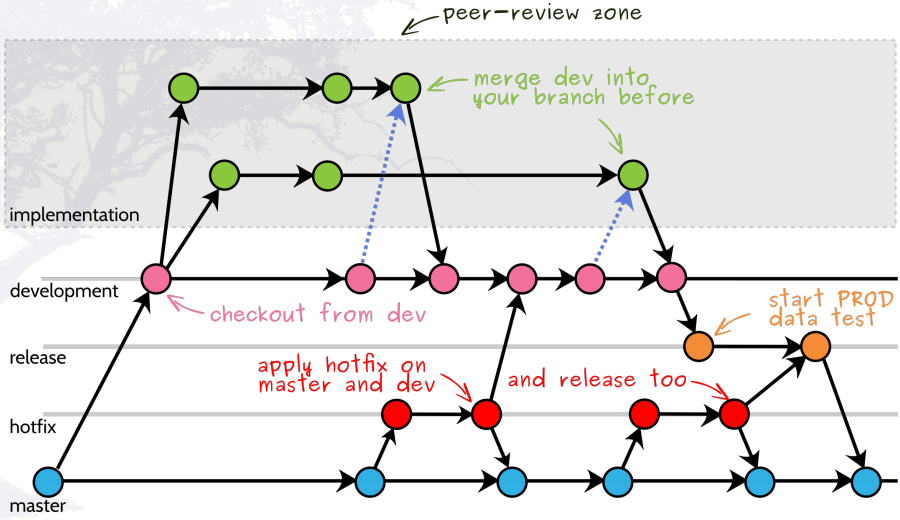
distributed two levels of commits, *e.g.*, *git*

Single repository does not mean single copy, but one **mainline**

master with the production code

branches with your development features, plus **hotfix** (and **release**)

Driessen's branching and merging



Everyday commits

Pushing your work everyday is meant to

- **identify** problems sooner
- lower the occurrences of merge **conflicts**
- ease the **integration** tasks (smaller units to debug)

What if your code does not build at end of day

- push **partial implementations** (remember to not break the *dev* line)
- and/or evaluate if task was not **under-estimated**

Everyday commits have a **motivating asset** for developers

→ you see your **contributions**, so the others too...

Merge requests are another layer of **quality assurance**

→ formally **peer-reviewing**, but do not let them waiting for days...

Let's try some code review

```java
/**
 * This method checks whether an integer is negative or not.
 */
public int isNegative(int a) {
  int result;
  if (a < 0) {
    result = -1;
  } else {
    result = a;
  }
  return result;
}
```

```java
/**
 * This method checks whether the first given int is less
      or equal to second given int
 * @param a an int
 * @param b another int
 * @return true if a <= b
 */
public boolean isLessOrEqual(int a, int b) {
  if (isNegative(a) == -1 && isNegative(b) > 0) {
    return true;
  }
  if (isNegative(a)  == -1 && isNegative(b) == 1) {
    return -a > -b ? true : false;
  }
  if (isNegative(a) > 0 && isNegative(b) == -1) {
    return false;
  }
  // a and b are positive
  return a <= b;
}
```

Merge requests & code reviews

Merge (also called Pull) requests ideally ensure

- the code is **readable**, *e.g.,* proper naming, documentation
- the code meets the expected level of **quality**, *e.g.,* code style
- **tests** are shipped with it
- configuration or migration **scripts** are there, *e.g.,* database
- the code won't cause **security** or **compatibility** problems
- the code does what it is **expected to do**
- **commit message** makes sense

→ Lower the risk to **break something** and increase **cross-functional** knowledge

You are not your code

Be open-minded as a reviewer

- reviewer's **logic** is not the author's logic
- look if **abstractions** can be created / removed
- look for potential **duplications** with existing code base
- take another critical look to potential new **dependencies**
- is there the right number of **unit/acceptance tests**?

Be open to critiques as an author

- comments are **not** made against you
- be open for changes when **responding** to comments

→ Always be **constructive**, **positive** and **compliments** are welcome too!

Managing builds

Build on commit and automate the build

Self-testing builds are meant to keep the *dev* mainline healthy, but

- often conflicts with **time pressure** to deliver
- **setting up** a fully working test environment is time-consuming

Use an external instance that is **accessible to anyone**

→ *a.k.a* an integration server with **automated** builds

Manual tasks are error-prone and time consuming

- automate the build with adequate **scripts**
- plenty of technologies available for many platforms, *i.e* *maven*, *gradle*, *sbt*,...

Automated builds are also designed for newcomers to **start up in minutes**

→ they don't have to struggle with the **technologies** and **dependencies**

Self-testing and fast build

Automating the build with upfront tests enables to

- lower the risk to commit **non-runnable code**
- provide adequate **feedback** to developers

Tests are then fully part of commits and builds

→ unit testing may be accompanied by **functional/acceptance** tests

Subsetting, pipelining and faking

- consider to concentrate on a **subset** of tests
- use **fake stubs** for external services or slow resources (*e.g.,* database)
- **asynchronously run** one or more tests after the CI build

Automated tests are not the panacea, but still valuable **health indicators**

Ease the succession

Access to the latest source and build

As the sources are accessible, the latest executable is accessible

- in a **dedicated repository** everyone knowns about
- this build is **as stable as possible** (maybe not all latests commits)

Like testing on build, this executable should have passed **pipelined tests**

→ kind of **snapshot** of latest stable build of *dev* mainline

Commits/tags must be accompanied by **meaningful comments**

- members see what are the **last changes**
- members may (almost) **take over on failed builds**
- add a calendar of **releases** or **milestones** in your project room

Automated deployment

Requiring many testing environments and *instant start-up*

- enforces **scripting** for building
- encourages to also **script the deployment**

Deploying applications means *rollbacking* too

- what if an **unexpected error** occurred?
- testing even on clone does **not give all guarantees**

Bad deployments are inevitable

- real data and users will sometimes trigger **unexpected behaviour** or **effects**

→ so we always need a **plan B**


TO BE
CONTINUED...

Next episode

Want to know more on automated deployment?

May we apply the same practice for continuous delivery of changes?

But then, what does it mean in terms of task management and planning?

... come next lecture