Software Engineering and Cloud Computing
WASP PhD course
Spring 2017
Revised: 2017-02-13

Agile Development Tools

Görel Hedin
Dept of Computer Science, Lund University

Agenda

- Agile development
- Version control
- Code review
- Testing
- Continuous integration

- Software Tool Chain Assignment

The Tool Chain Assignment

4 x 4 hours

Iteration 1 **(register your team and url)**

- First Iteration (minimal executable product)
- Java, Ant, Git, GitHub

Iteration 2 **(Deadline: March 8)**

- Test-Driven Development, Centralized workflow
- JUnit

Iteration 3

- Feature branch workflow
- Code Review (GitHub)

Iteration 4 **(Deadline: March 15)**

- Continuous Integration
- Travis
- Review another team's product

Agile Development

Growing software

- Always releaseable
  - Builds and tests correctly
  - Clean code (technical debt if not)
- Work in small steps
- Collective ownership

Agile methods

XP

- Iteration
- Stories
- Customer
- Test-driven dev.
- Pair programming
- Refactoring
- Simple Design
- …

Scrum

- Sprint
- Backlog items
- Product owner
- Self-organizing teams
- …

Lean

- Empower the team
- Eliminate waste
- Decide late
- …

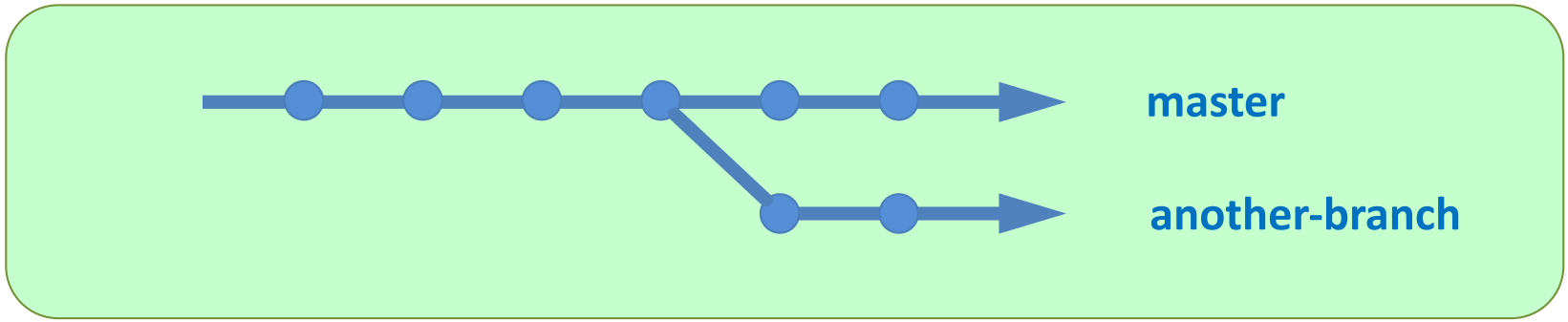
Working in small steps

Product

- Stories (features of value to end user)
  - Tasks (parts of a story)
    - Tests

When to commit new version?
Could be several times per story. As soon as a new small part is completed, and the code is clean, and tests still work.

Version control

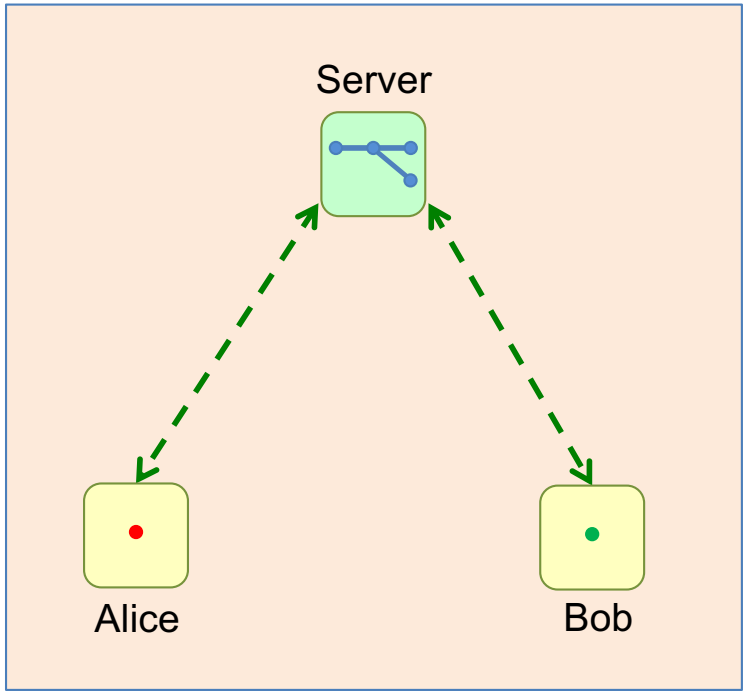


Repository with commits and branches

Version Control Systems

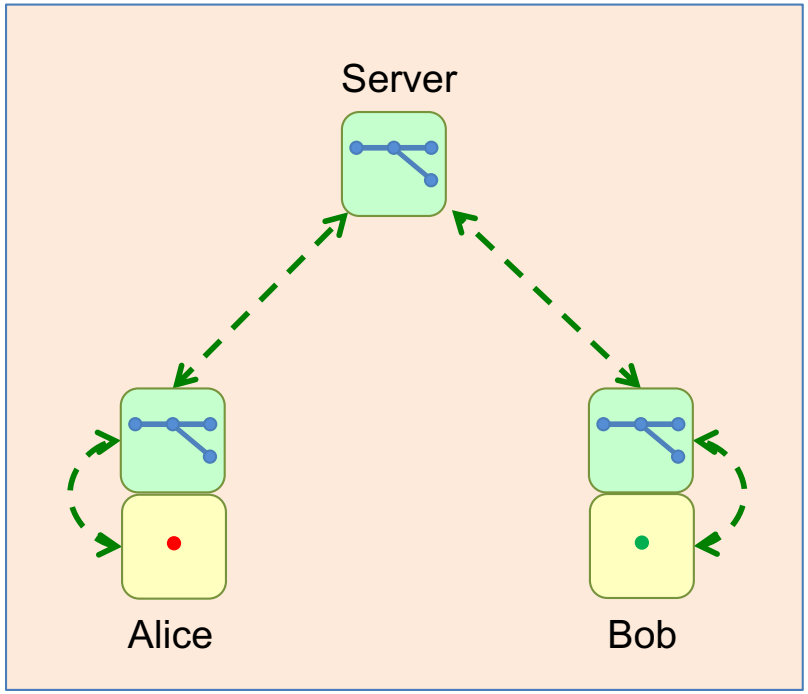
Centralized

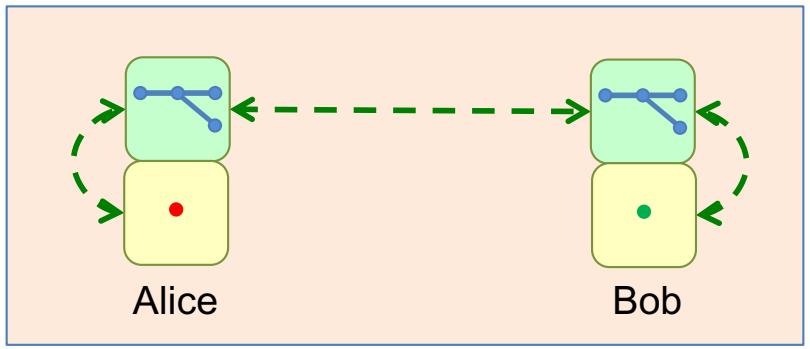
CVS, Subversion, …



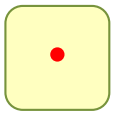
Distributed

Git, Mercurial, Bazaar, …





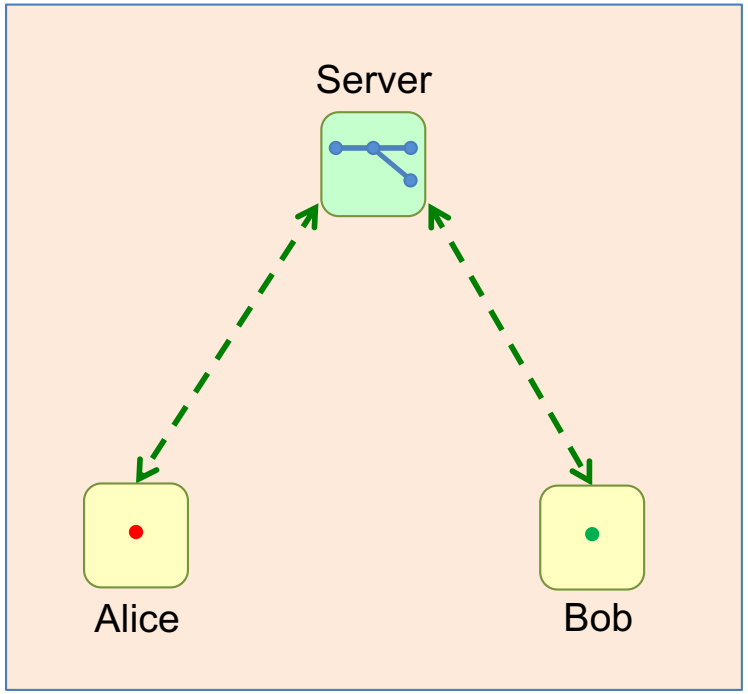
Repository: all commits



Workspace: commit in progress

Comparison

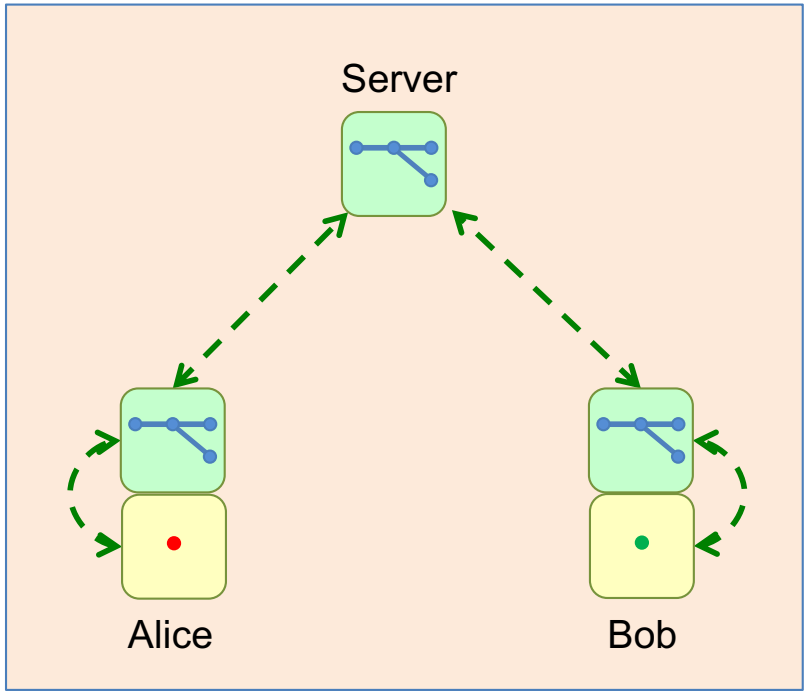
Centralized



Much simpler

Can use very large repositories
- can checkout submodules

Distributed



Very powerful

- Work offline
- Contribute through ”pull requests”
- Code review on branches

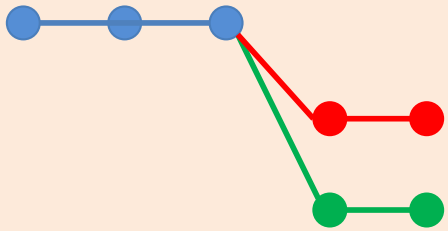
Split into many small repositories

Version control workflows

Centralized

Commit directly to the master branch

Feature branches



Create a branch for each new feature.
Code review and integrate with master when finished.

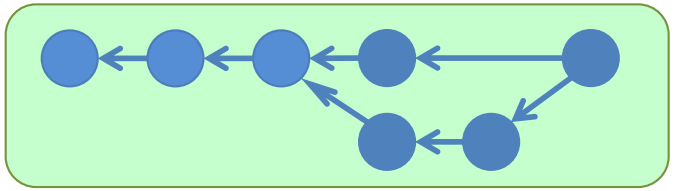
Gitflow

Only releases on the master branch
Developer branch for integrating features.
Bugfix branches ...

Forking workflow

"Fork" (copy) repo to your own server account.
Do "pull request" to ask owner to integrate your changes. For open-source collaboration.

Commit hashes



A git commit is considered consisting of its file contents + all its predecessors.

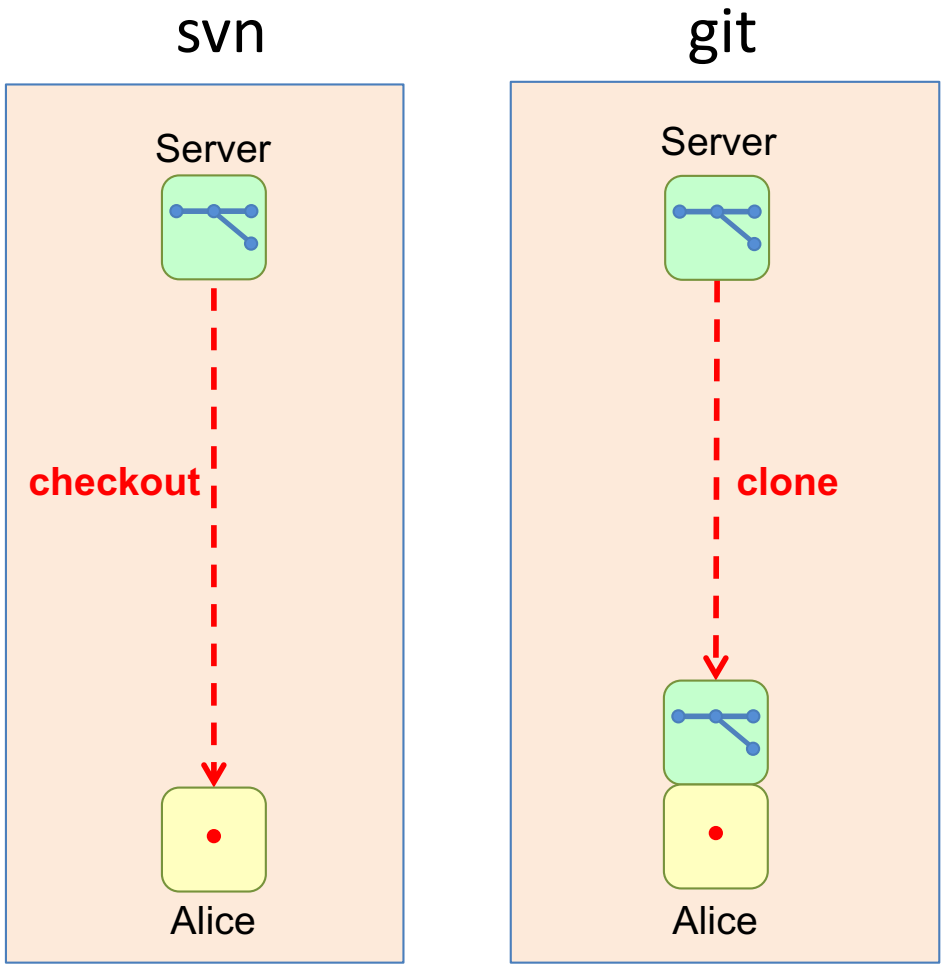
A commit is uniquely identified by a hash[1]. For example:
fbe3a4f0b3b44e916463a91ad8c4aebf411cad6c

Usually abbreviated to the first 7 digits:
fbe3a4f

Computed by a SHA-1 hash over the contents + the hashes of its predecessors.

[1]Unique for all practical purposes.

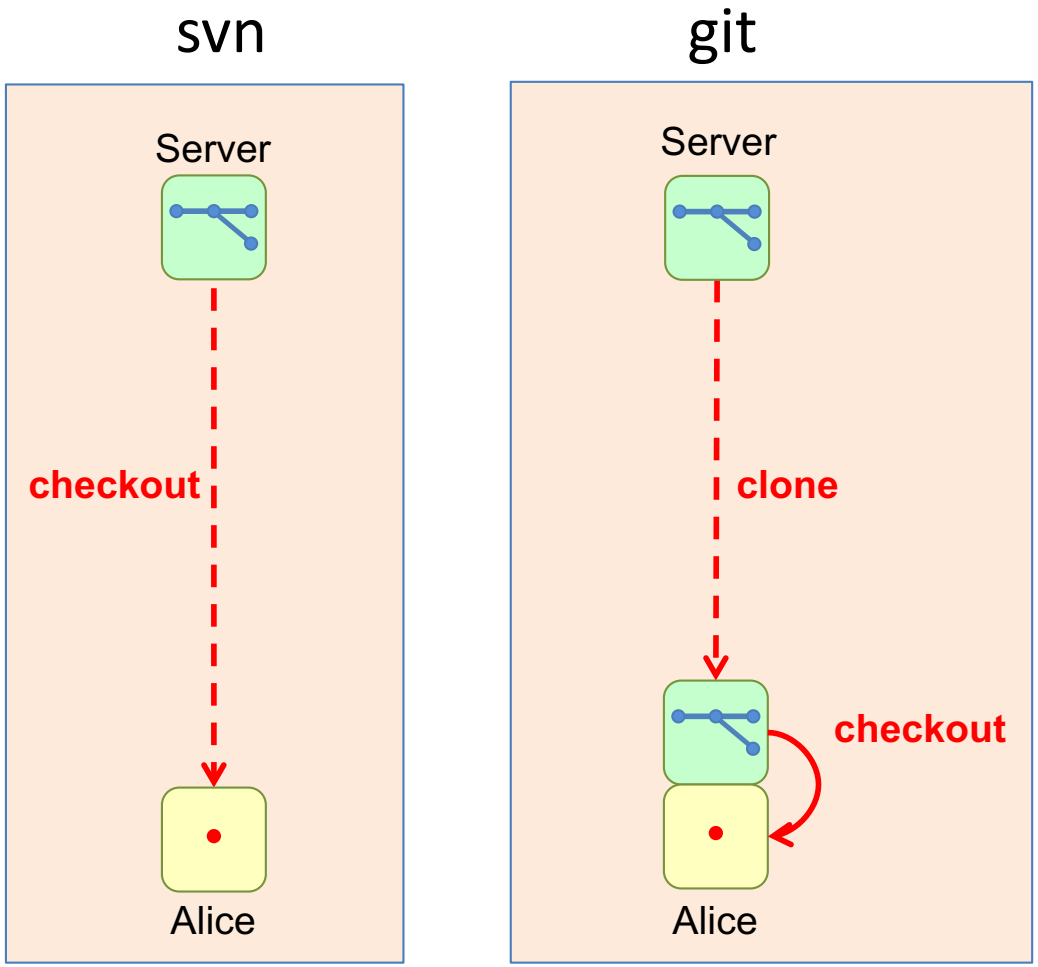
Creating local workspace/repo



clone: creates a local repo and a workspace

```
> git clone https://server/name/repo.git
```

Checkout

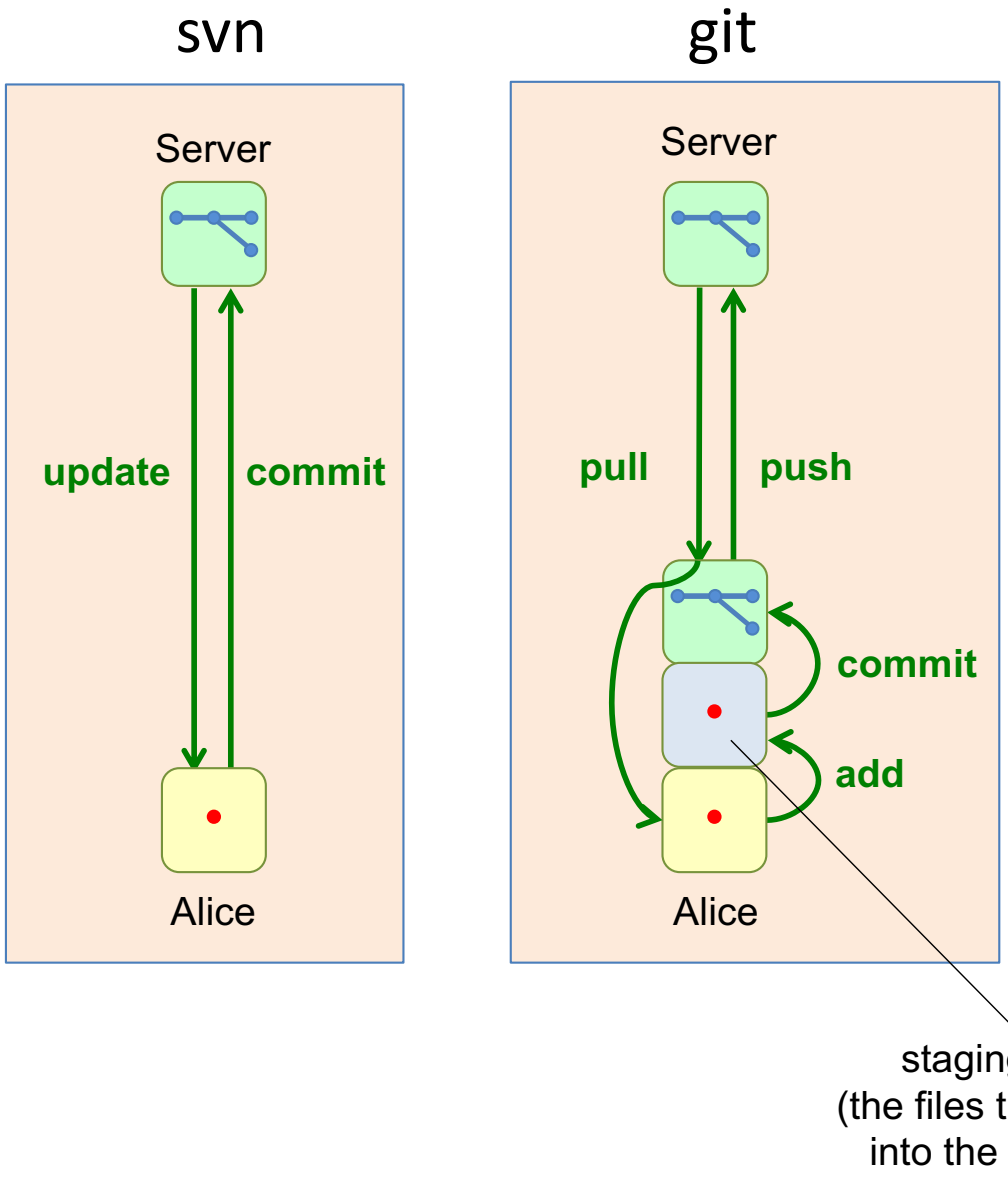


checkout:
Switch to (the latest commit in) a branch

(similar to svn)

```
> git checkout a-branch
```

Basic commands



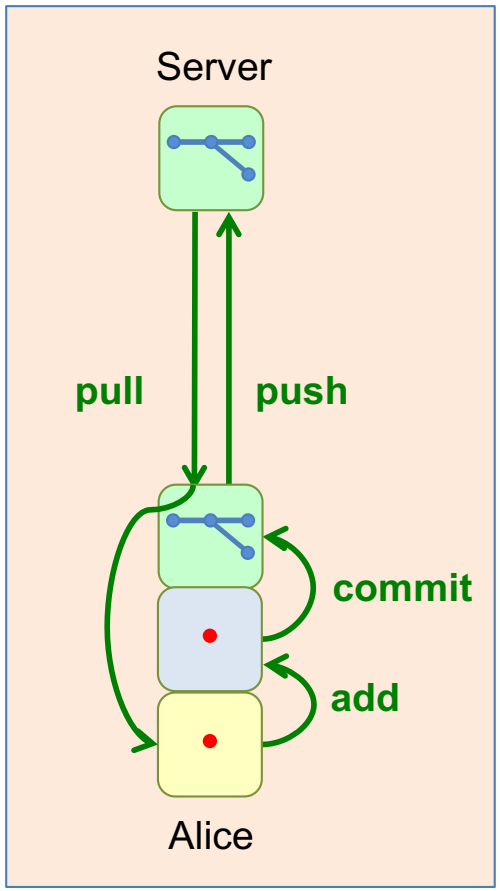
pull: pulls new commits from the server to the local repo and merges them into the workspace

add: copies changed/added files to the staging area

commit: creates a new commit in the local repository

push: pushes new commits from the local repo to the server

Basic recipe for centralized flow



Go to the local repository directory

```
> cd repo
```

...Do new work. Check that it works. ...

Before committing, add any new files to the staging area

```
> git add newfiles
```

Commit the changes locally (the –a option will add any changed files to the staging area first)

```
> git commit -am "Did ..., see #37"
```

... Possibly do more work and more local commits ...

Before pushing, pull the latest changes from the server and merge them

```
> git pull
```

... Merge conflicts? Resolve them! ...

Push the new commits up to the server

```
> git push
```

Before starting new work, check that the current state works.

Pull here, to build the new work on the latest version.

Alternative for more incremental merge and linear commit chain:

```
> git pull --rebase
```

Pull and Push often, to minimize merge conflicts.

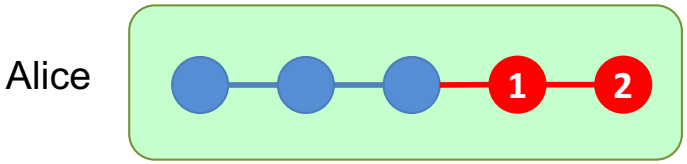
Example automerged file

```
let  a = b
  b = 3.0
  c = 9.0
  a * c
end
```

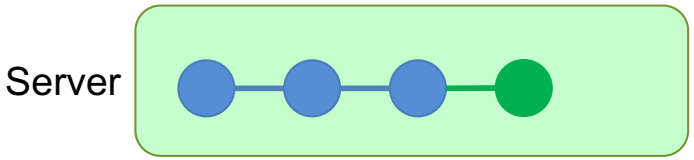
```
let  a = b
  b = 3.0
  c = 10.0
  a * c
end
```

```
let  a = b
  b = 3.0
<<<<<<< HEAD
  c = 9.0
=======
  c = 10.0
>>>>>>> ee18486cd395afa36ed2259b6933bfa91bd41ab4in
  a * c
end
```

Merge or rebase?



Alice has done two local commits.



Meanwhile, Bob has pushed a new commit.

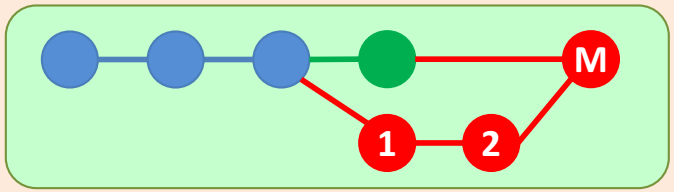
Alice does ordinary merge:

Alice gets Bob's commit with auto merge:

```
> git pull
```

Alice has to create a new commit for the merged version

```
> git commit -am "..."
```



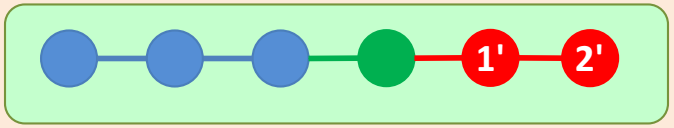
Alice merges with "rebase":

Alice gets Bob's commit with rebase:

```
> git pull --rebase
```

Alice'c commits are changed to be based on Bob's commit, instead of the original one. If there are merge conflicts, Alice will deal with each of her commits at a time, continuing with:

```
> git rebase --continue
```



`pull --rebase` *changes* existing commits. Should only be done on non-public commits!
Rebase results in a cleaner version history.

Some more git commands

See the status of the workspace. For example, which files are not staged.

```
> git status
```

Look at the latest local commits

```
> git log
```

Look at difference between commits

```
> git diff
```

Temporarily move all workspace changes to a "stash"

```
> git stash
```

Move back the changes to the workspace

```
> git stash apply
```

List all tags

```
> git tag
```

Fetch is like Pull, but it only updates the local repo, not the workspace

```
> git fetch
```

Avoid committing generated files

Use the **.gitignore** file for this:

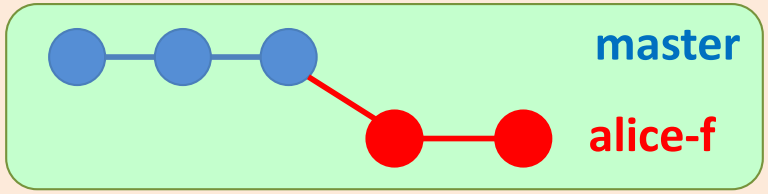
```
> cd repo
> more .gitignore
```

```
# java files
*.class
*.jar

# Mac OS files
.DS_Store

# TeX files
*.pdf
*.log
*.aux
*.bbl
...
```

Feature branch workflow

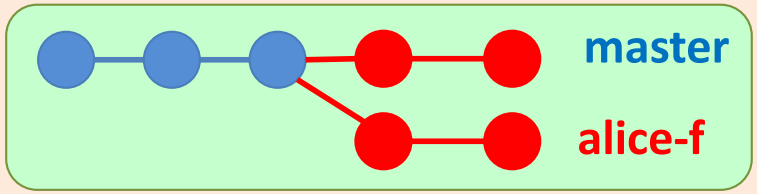
Server

1) Alice pulls down the latest changes on the master branch.

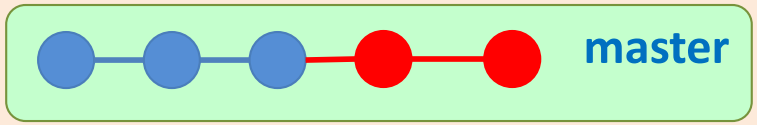


2) Alice creates a local feature branch, performs a number of commits, then pushes the feature branch to the server.

3) Alice goes to the server (GitHub) and asks someone on the team to code review the feature: she does a "pull request".



4) After discussion and possible additional changes, Alice or her team mate merges the branch into to the master



5) The branch is deleted, both on the server and locally.

Basic recipe for feature branch

Create a local branch for the new feature and switch to it

```
> git checkout -b alice-f
```

Short for

```
> git branch alice-f
> git checkout alice-f
```

... implement the new feature ...

Commit one or more times to the local branch

```
> git commit -am "..."
```

Before pushing up the feature, sync with the server: Go to the master branch and pull the newest commits from the server. Then go to the feature branch and rebase the work there on the new commits from master.

```
> git checkout master
> git pull --rebase
> git checkout alice-f
> git rebase master
```

Or merge if you prefer that

```
> git merge master
```

Now, the local feature branch can be pushed up to the server.
("origin" refers to the repo at the server)

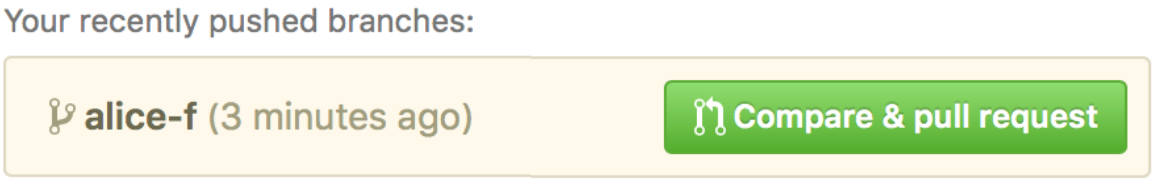
```
> git push --set-upstream origin alice-f
```

Go to GitHub and start the code review. The merge to the master branch can be done at GitHub.

Remains to delete the local branch

```
> git branch -d alice-f
```

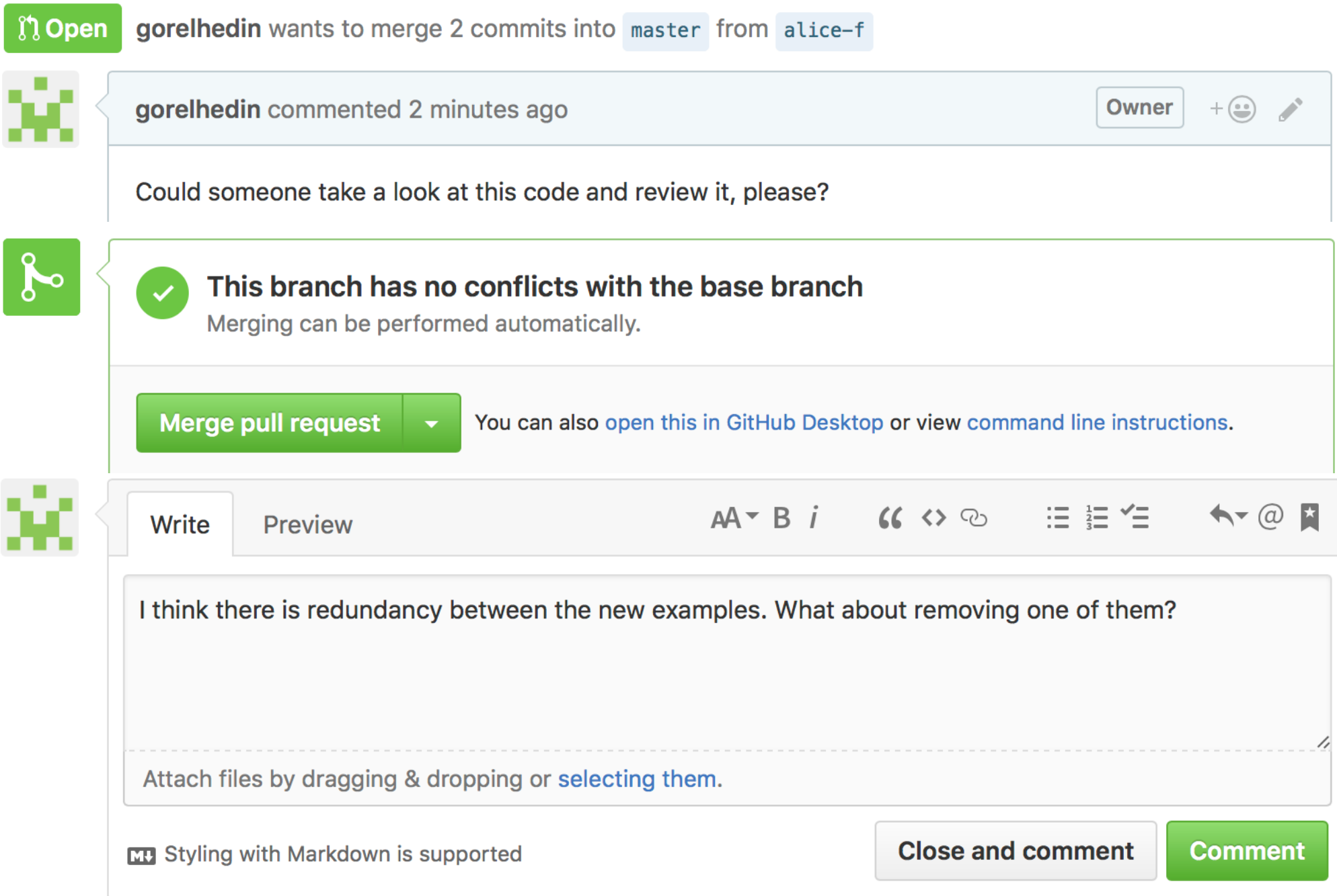
Code Review on GitHub



Create a "pull request" to ask others on the team to inspect the changes and discuss the code before merging into the master branch.

Many other Git hosting services have similar support (BitBucket, GitLab, ...)
Other alternatives: Gerrit.

Example code review



Why do code review?

- Peer pressure to keep high coding standard
- Someone else might know the code better
- Generally, spread knowledge
- New member on the team
- Several companies require it

What to look for when reviewing

- Follows basic coding conventions?
- Fits or improves the current design?
- Clean code? Easy to understand? Good names?

- Sufficient amount of tests?
- Solves the problem?

Automated code review

- Static analysis to look for problems in the code
  - dead code
  - duplicated code
  - complex expressions
  - null pointer exceptions
  - synchronization problems
  - McCabe Cyclomatic Complexity (CC)

- Example free tools for Java
  - FindBugs
  - PMD
  - CheckStyle
  - IntelliJ IDEA
  - Eclipse

Automated tests

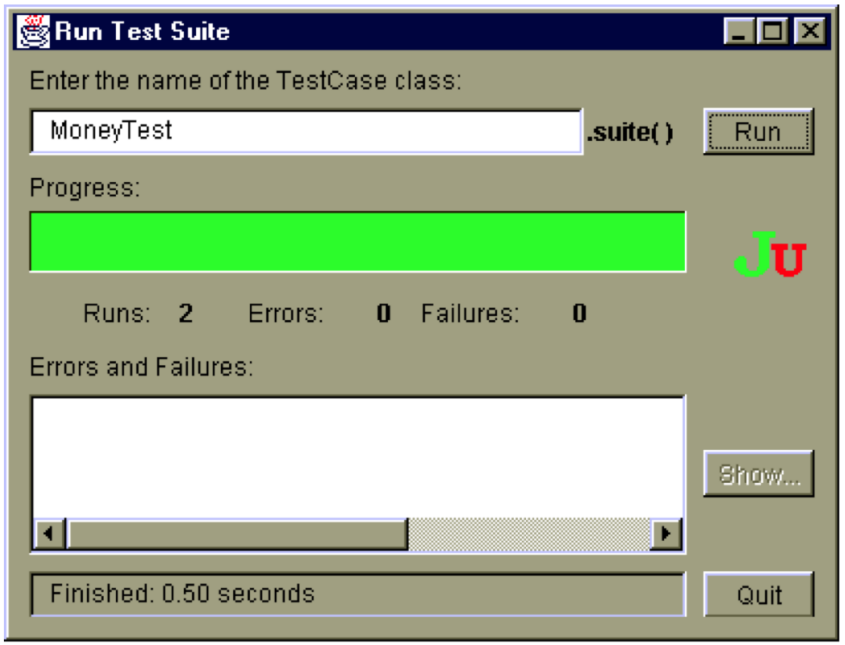
xUnit

Note!
It's for testing at any level.
Not just unit testing!

Simple test automation framework

- JUnit – for Java
- sUnit – for Smalltalk
- cppUnit – for C++
- pyUnit – for Python
- …

Integration in most IDEs.



"Keep the bar green"

Or run it from the command line:

```
> ant test
Buildfile: .../build.xml
...
test:
  [junit] Testsuite: tests.ParseTests
  [junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0,03
  [junit] Testsuite: tests.TestNameAnalysis
  [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0,0
BUILD SUCCESSFUL
```

Example JUnit test

```
import org.junit.*;
static import org.junit.Assert.*;

public class BankAccountTest {
    @Test public void testEmptyAccount() {
        BankAccount a = new BankAccount();
        assertEquals(0, a.balance());
    }
}
```

- Declare a method annotated with **@Test**.
- Call an **assert** method to check a result.

JUnit will find and run all such methods.

How JUnit works (simplified)

JUnit uses reflection (accesses classes at runtime)
Given a number of Java classes

- Find all methods **T.m** annotated with **@Test**
- For each such method:
  - create an instance of **T**
  - call the **@Before** method for **T**, if there is one.
  - call **m**
  - call the **@After** method for **T**, if there is one.

Testing exceptions

```
@Test(expected = EmptyStackException.class)
public void testPopOfEmptyStack()
    throws  EmptyStackException {
    new Stack().pop();
}
```

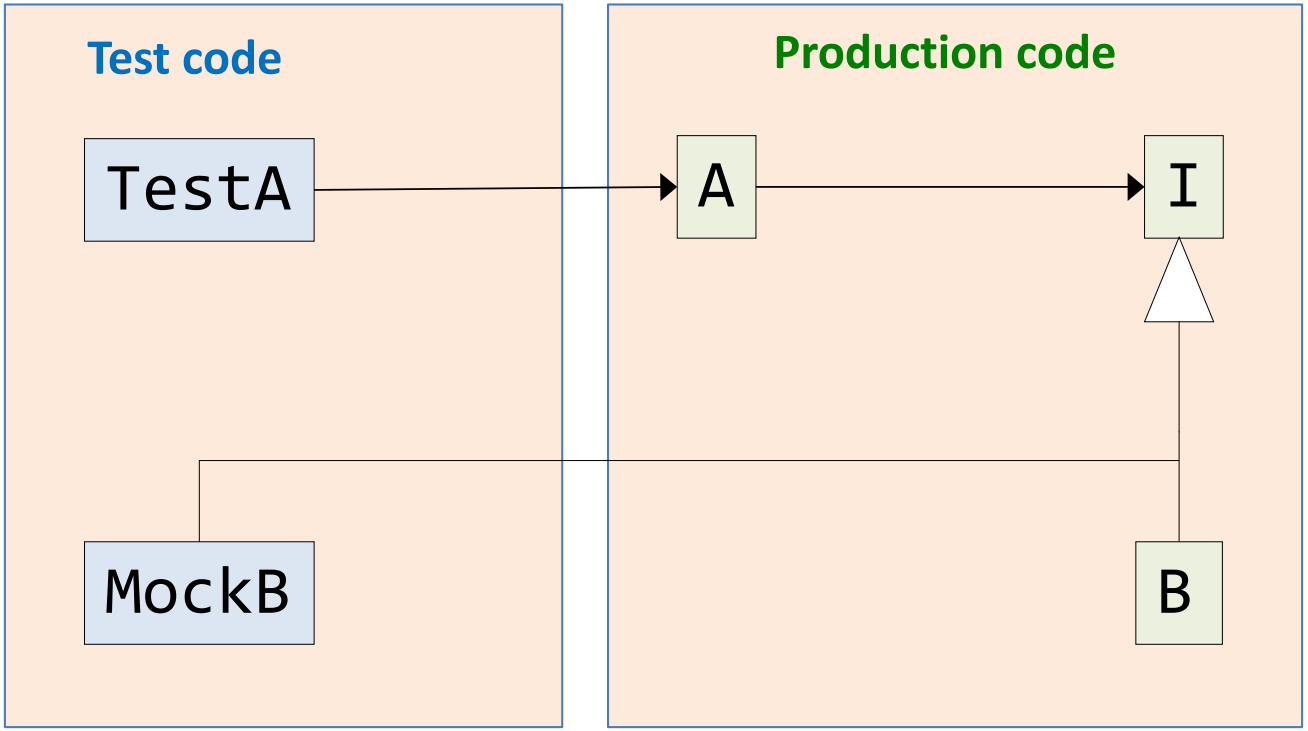
- Test that the code actually throws the right kind of exception.

Fixtures

```
public class TestStack {
    Stack s = new Stack();
    @Before public void setUp() {
        s.push(314);
        s.push(17);
    }
    @Test public void testPop() {
        assertEquals(17, s.pop());
    }
    @Test public void testDepth() {
        assertEquals(2, s.depth());
    }
}
```

- One instance of TestStack is created for *each* test method run.
- The **@Before** method is run before the test method.
  It sets up the "fixture" - the data structure to test.
- Helps isolating tests from each other.

Mock objects



How can we test A without testing B?

- Create a "Mock object" to replace B in the test.
- Implement MockB so it fits the test.

Test Driven Development (TDD)

Small steps

- Write a test. See it fail.
- Make it work.
- Make the code clean (refactor).

A design technique, more than a test technique

- Focus on *what* before *how*.
- Use the API, before implementing it.

Behavior Driven Development (BDD)

A variant on TDD

- Focus on the desired behavior
- Use special notation to write the tests
- Readable tests. Readable error messages.

Example BDD library in JUnit

Hamcrest (matchers) library in JUnit

```
assertThat(x, is(5));

assertThat(x, is(not(4));

assertThat(responseString,
    either(containsString(”color”)).
    or(containsString(”colour”);

assertThat(myQueue, hasItem(”3”));
```

Example test error:

```
junit.framework.AssertionFailedError:
Expected: (a string containing "color" or a
string containing "colour")
but got : hello world
```

More JUnit functionality

- Parameterized tests
  - run the same test for a set of parameters, e.g., input/output pairs
- Property-based testing
  - express assertions ("properties"/"theories"/"laws") that should hold for any test data satisfying certain conditions
- Random testing (like Haskell QuickCheck)
  - Supply theories with random generated data that fulfills the conditions

Read more:

- JUnit documentation: http://junit.org/junit4/
- K. Claessen and J. Hughes: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP 2000. ACM.
- D. Saff, M. Boshernitsan, M. D. Ernst. Theories in Practice: Easy-to-Write Specifications that Catch Bugs. MIT-CSAIL-TR-2008-002.

Automated testing in specific areas

- Graphical user interfaces
  - use Model-View pattern to isolate logic from GUI (test logic using ordinary tests)
  - Capture/Playback tests: Record GUI events. Playback during test.
  - Monkey tests: simulate random events to provoke failures.
- Concurrent and distributed systems
  - property-based random testing
  - mock objects to test parts in isolation
- Performance
  - Benchmarks regularly to avoid degrading performance.

Test coverage tools

To find untested code. Typically focuses on instruction or branch coverage.

For Java:

- JaCoCo – Java Code Coverage Library
- Runs all tests, instruments byte code on the fly, at class loading.
- Can generate various reports.

For other languages

- Coverage.py
- OpenCppCoverage
- …

Build tools

compile
generate executable (jar, exe, ...)
generate documentation
run tests
automatically download dependent libraries
...

Popular build tools

- Make, 1977
- Ant (+ Ivy) (simple, a bit bulky, XML)
- Maven (powerful, complex, XML)
- Gradle (powerful, concise)

Simple Ant file for JUnit testing

```
<project default="build">
    <property name="cp.lib"
        value="lib/hamcrest-core-1.3.jar:lib/junit-4.12.jar" />

    <target name="build">
        <mkdir dir="ant-bin" />
        <javac
            srcdir="src/main/java:src/test/java"
            destdir="ant-bin"
            classpath="${cp.lib}"
        />
    </target>

    <target name="test" depends="build">
        <junit fork="true" failureproperty="test.failed">
            <classpath .../>
            <formatter type="brief" usefile="false" />
            <batchtest .../>
        </junit>
        <fail message="Tests failed" if="test.failed" />
    </target>
</project>
```

Actions call *Tasks* in tools like javac, junit, etc.

Build *Targets* can depend on each other

Continuous Integration Servers (CI)

trigger automated actions from events (commits, timers, ...)

For example:

- check that a new commit builds and tests
- check that downstream projects build after a commit
- run long-running tests nightly
- ...

Many different tools. For example:

- Travis: Online. Very simple.
- Hudson/Jenkins. Open source. Very powerful.

Travis: simple on-line CI server

- Free for public GitHub repositories
- Runs tests on every commit
- Can send email on failures

Hudson/Jenkins
powerful open source tool

- For building, deploying, automating in general
- Hundreds of plugins
- Jobs based on events or timers
- Check out many repositories for a single job, store results, etc.

Summary

- Work in small steps
- Version control, workflows
- Code review
- Automated tests
- Continuous integration

The Tool Chain Assignment

4 x 4 hours

Iteration 1 **(register your team and url)**

- First Iteration (minimal executable product)
- Java, Ant, Git, GitHub

Iteration 2 **(Deadline: March 8)**

- Test-Driven Development, Centralized workflow
- JUnit

Iteration 3

- Feature branch workflow
- Code Review (GitHub)

Iteration 4 **(Deadline: March 15)**

- Continuous Integration
- Travis
- Review another team's product