Managing Data on the World Wide-Web

Development Tools and Methods
Overview

1. Unit Testing and TDD
2. Maven
3. Version Control with Git
Testing – Why?

• When devolving a large program, bugs are expected to be discovered as new functionalities and features are being added and modified.

• Tests guaranty (to some extent)
  – Correctness
  – Robustness

• We benefit from the process: writing good tests forces us to improve our code.
Testing – When?

• When it’s done?
  – Too late, fixing bugs is easier if they are found earlier
  – When is it “done”?

• Very frequently
  – New functionality should be tested as soon as we finish writing it
  – Or is it the other way around? (Test Driven Development – later on)
Unit Testing

• Unit testing is a method by which individual units of code are tested to determine if they work as intended
  – Unit: small testable part of an application (e.g., functions, classes)
• A Unit Test is simple:
  – Only tests one part of the code
  – When it breaks, the bug is easily tracked
  – Smaller, more simple and therefore faster

• A unit test covers a small part of the system
  – We will need a lot of these
Unit Testing with JUnit

- JUnit: A unit testing framework for Java
  - [http://www.junit.org](http://www.junit.org)

- A **Test** is a method

- A **Test Case** is a class which includes tests methods
  - Aggregates tests for a single tested unit

- A **Test Suite** is a collection of Test Cases / Test Suites (or both)

Based on: [Unit Testing, Software Design (236700)](Unit%20Testing%2C%20Software%20Design%20(236700))
JUnit 4.0

- Test methods are annotated with `@Test`

- `@BeforeClass` – runs once before the first test
  - Setting up the stubs objects / DB Connections

- `@AfterClass` – runs once after the last test
  - Closing DB connections

- `@Before method` – runs before every test
  - Preparing data
  - Could be used for advanced construction

- `@After method` – runs after every test
  - cleaning data

Based on: Unit Testing, Software Design (236700)
TDD – a New Way of Thinking

• Test Driven Development (TDD) is a software development technique

• According to the book Test-Driven Development by Example by Kent Beck, TDD centers on two basic rules:
  1. Never write a single line of code unless you have a failing automated test
  2. Eliminate duplication (refactor the code)
1. Add a test
   - Each new feature begins with writing a test
2. Run all tests and see if the new one fails
   - Makes sure that the new test does not mistakenly pass without requiring any new code
3. Write some code
   - At this point, the only purpose of the written code is to pass the test

Based on: [https://en.wikipedia.org/wiki/Test-driven_development](https://en.wikipedia.org/wiki/Test-driven_development)
The TDD Cycle (2)

4. Run Tests
   – The new code meets the test requirements
   – and does not break or any existing features

5. Refactor code
   – The growing code base must be cleaned up
   – Duplication must be removed
   – Improve readability and maintainability

6. Go to 1
The TDD Cycle (3)
Online demonstration:

1. http://www.youtube.com/watch?v=2HP8Z93BwIg
2. http://www.youtube.com/watch?v=HUX0JJFXhLA
3. http://www.youtube.com/watch?v=etpsp8hjlcY
4. http://www.youtube.com/watch?v=1ABI2wOooYo
5. http://www.youtube.com/watch?v=-HjfkoDFtIw
package util;

public class Fraction {
    public Fraction(int num, int denom) {
        this.num = num;
        this.denom = denom;
    }

    @Override
    public boolean equals(Object obj) {
        return false;
    } // TODO: implement

    public int getNum() { return num; }
    public int getDenom() { return denom; }

    private int num;
    private int denom;
}

package util;

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

public class FractionTest {

    @Test
    public void testEquals() {
        Fraction f = new Fraction(1, 2);

        assertEquals(f, f);
        assertEquals(new Fraction(1, 2), f);
        assertFalse(f.equals(new Fraction(3, 2)));
        assertFalse(f.equals(null));
        assertFalse(f.equals("abc"));
    }
}

JUnit Test Results

Finished after 0.024 seconds

- Runs: 1/1
- Errors: 0
- Failures: 1

util.FractionTest (Runner: JUnit 4) (0.002 s)
  testEquals (0.002 s)
@Override
public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Fraction)) {
        return false;
    }

    Fraction that = (Fraction) obj;

    return getNum() == that.getNum() &&
           getDenom() == that.getDenom();
}
import org.junit.Test;

public class FractionTest {

    @Before
    public void connectToDB() {
        :
    }

    @After
    public void disconnect() {
        :
    }

    @Test
    public void testEquals() {
        :
    }
}

References

- Test-Driven Development by Example
  https://en.wikipedia.org/wiki/Test-Driven_Development_by_Example
Overview

1. Unit Testing and TDD
2. Maven
3. Version Control with Git
Building Projects with Maven

• Maven is a powerful build tool for (Java) projects
  – Build tools automate common build tasks (e.g., compiling source code, packaging compiled code into JAR files)

• Also a project management tool
  – Helps in testing, managing dependencies, generating documentation and reports, deploying
Key Features

• Convention over configuration
  – default configuration, project templates (archetypes)

• Dependency management
  – dependencies to other projects/libraries

• Central repository
  – Local and remote

• Extensible via plug-ins
Maven projects are configured using a Project Object Model (POM) - which is represented by a pom.xml file.

A POM provides all the configuration for a single project - e.g., dependencies (external JARs), plugins.

The pom.xml file should be located in the root directory of the project.
Build Lifecycles

• The build process is called a build lifecycle
• A life cycle consists of build phases
• Each build phase consists of goals
• The default lifecycle:
  1. Validate
  2. generate-sources
  3. process-sources
  4. generate-resources
  5. process-resources
  6. compile
  7. process-test-sources
  8. process-test-resources
  9. test-compile
  10. test
  11. package
  12. install
  13. deploy

• Phases can be executed explicitly using the command line
Dependency Management

• Dependencies are external JAR files (Java libraries) used in the project

• If a dependency is not found in the local Maven repository, Maven will download it from a central Maven repository and store it in the local repository
  – The local repository is simply a directory in the user's file system
Plugins

- Plugins are used to specify goals in a phase
  - For example, to compile source code, Maven uses the "compile" goal which specified in the "compiler" plugin

- There are standard plugins for building, testing, source control management, running a web server, generating Eclipse project files, and more
  - Also, custom plugins can be implemented (in Java)

- Plugins can be used from the command line or be added to the Build Lifecycle
Profiles

• Profiles are used to build the project in different ways.

• Each profile defines different build configurations.

• Common use cases:
  – Development/production environment
  – Local/remote server
M2E – Maven and Eclipse

• M2Eclipse provides integration for Maven into Eclipse IDE
  – http://www.eclipse.org/m2e/index.html

• Already included in Eclipse 😊

• Recommended:
  https://youtu.be/yb-VG-s-lfc
Live Demo
Overview

1. Unit Testing and TDD
2. Maven
3. Version Control with Git
A Version Control System (VCS) is a system that records changes to a file (or set of files) over time so that you can recall specific versions later

- If you mess up your code, you can easily recover

It allows you to:

- revert files back to a previous state
- compare changes over time
- see who last modified something that might be causing a problem
- and more

What is “Version control”? 
Centralized Version Control Systems

• A single server contains a central copy of the files
• Programmers *commit* their changes to this central copy
  – Committing: recording your changes in the central copy, making them available to others
• The files, their directory structure and additional metadata, such as the history of changes, are bundled together under the term *repository*
CVCS (cont’d)

• Common CVCS:
  – CVS
  – Subversion (or SVN)

• SVN supplied new features that were missing in CVS
  – In CVS, revisions created by a commit are per file, rather than spanning the entire repository (as in SVN)
  – In SVN, renamed/copied/moved/removed files retain full revision history (in contrast to CVS)
CVCS - Advantages and Drawbacks

• Advantages:
  – Manual management of files is no longer required
  – Recovering from errors is fast and easy
  – Convenient logging system for changes done by others

• Drawbacks:
  – Each commit action is visible to others. Is it a good practice to commit a code in its development stage?
  – What happens if the server goes down or is unreachable?
In Distributed Version Control Systems (DVCS), every developer **clones** a copy of the repository

- This copy has *all* of the metadata of the original (not just the latest snapshot of the files)
- Every clone is a full backup of all the data
Distributed vs. Centralized

• Version control can be done locally without anyone being affected by your changes
  – When the new functionality is robust, then changes can be pushed to the remote repository

• Common operations (such as commits, viewing history, and reverting changes) are fast
  – there is no need to communicate with a central server

• Communication is only necessary when sharing changes among other peers
Git Terminology

- Pulling - incorporating changes from the remote repository into the current *branch* (next slide)
- Pushing - moving your changes to the remote repository
- Fetching – getting changes from the remote repository without merging them with your current branch

- For extra reading: [Fetch and Merge, Don't Pull](#)
Branches

• **Branch** – a line of development
  – Under the hood, branches are just pointers to commits
• **New commits are recorded in the history for the current branch**
• **When you want to add a new feature or fix a bug, you create a new branch to encapsulate your changes**
  – This makes sure that unstable code is never committed to the main code base
The default branch name in Git is **master**

When the new feature is ready, merge the branch into the master branch
• GUIs are intuitive and friendly
• However, to enjoy the full power of Git, it’s recommended to get familiar with the command line tool

https://tortoisegit.org/
The following slides will provide a recommended workflow for working with Git on a shared project.

We will walk through three common operations in Git:

1. Getting others’ recent work from the server
2. Pushing your work to the server
3. Resolving conflicts

In the following, [dev_branch] is the development branch.
1. Make sure you're in [dev_branch]

2. Perform git->commit (to keep last changes)
   – or git->revert (to delete last changes)

3. Perform git->fetch
   – Now your view of origin/master is updated to the server's master
To merge the last changes from the server with yours:

4. Perform git-merge
   – Select remotes/origin/master
   – If there are conflicts, resolve them (explained next)
To only see the last changes from the server (without merging them with yours):

4. Switch to Master

5. Perform git->merge
   - Select remotes/origin/master
   - Don't forget to switch back to [dev_branch]
Pushing Your Recent Work to Server (1)

- Make sure you're in [dev_branch]
- Perform git->commit (to keep last changes)
  - or git->revert (to delete last changes)
- Perform git->fetch
  - Now your view of origin/master is updated to the server's master
- Switch to Master
• Perform git-merge
  – Select remotes/origin/master
  – If there are conflicts, resolve them
• Perform git-merge
  – Select [dev_branch]
  – If there are conflicts, resolve them
• Perform git-push
• Switch back to [dev_branch]
Suppose branch A was merged into branch B and some conflicts were detected

- A notification will be prompted

Conflicts are needed to be resolved in order to complete the merging

1. In the repository’s root folder, select `git->resolve`
   - a list of all conflicts will appear
Resolving Conflicts (2)

2. Select each file at a time

A window will be opened with 3 versions:
   - The “Their” file - the version in A branch
   - The “Mine” file - the version in B branch
   - The “Merged” file - the result; the file that will be in the merged branch

3. Edit the result and save it when finished
   - Select “Marked as resolved”
4. After resolving all files in the list, perform `git-commit`
   - when `git` merges two files and no conflicts are detected, an "auto commit" is applied (this can be avoided by checking "no-commit" in the merge dialog)
   - If conflicts are found, there is no auto-commit. The user should perform commit after resolving all conflicts
References

- Pro Git

- What is Version Control: Centralized vs. DVCS
  http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs/