



COMP2059 Developing Maintainable Software

LECTURE 05 – CODING & DOCUMENTATION, REPOSITORY TOOLS AND SOFTWARE TESTING FOR MAINTAINABLE SOFTWARE

Boon Giin Lee (Bryan)



Learning Outcomes

- By the end of today lecture, students should be able to
- Learn auto-generation of standard documentation for understanding the software.
- Familiar with automation of code build/compilation – build files for custom compilation of code.
- Revision with Git repository tools and beyond with concept of Container for fabricating computing environment.
- Software testing approach with TDD for developing maintainable software.
- Client vs Business Analyst vs Developer.
 - <https://www.youtube.com/watch?v=yYrkV5ManUo>



Documenting Code

JAVADOC AND BEYOND



Go Beyond Manual Code Comments

- You have been instructed on how to comment your code manually.
 - Hopefully, you understand why this is a good idea.
- Code comments are **essential** for maintenance as it is the key to having another person to be able to **understand** what you have done.
- **Auto documentation** allows for:
 - Consistent formatting and structuring of comments.
 - Widespread readability.
 - You don't have to do it all yourself.
- Examples include Doxygen and Javadocs.
 - Doxygen can be used for C++, with modified version used for Scala and C#.
 - Can also be used in conjunction with the Python live editor.
 - Javadocs are for Java, Hadooc is for Haskell.

IntelliJ/Eclipse – Helpful IDEs for Comments



- Many of today's tools are built into Eclipse/IntelliJ.
- It is useful beyond providing us with a nice interface, project viewer, and telling us of precompile errors, missing libraries and spelling mistakes.
- It is great for helping us create **maintainable code** including in-built testing help.
- Auto generation of self documenting code with the help of a tool called **Javadoc**.
 - This comes with the JDK and requires you to tag your code with special comments.
 - Sometimes it will even add it for you without you noticing.
- Has special syntax to differentiate it from regular comments.
- `/** ... */` as opposed to `/* ... */`.



Useful Javadoc Tags

- Syntax: `@<tag>`
- Really useful for generating comments which are quite tedious to write in full the whole time and then to have to “grep” or “sed” using regexs.
- It generates an easy-to-use HTML based outputs as living document.
- Updated each time during compilation if Javadoc is on the compilation path.
- Some popular tags:
 - `@param`: To explain a parameter.
 - `@return`: To annotate a return value.
 - `@throw/@exception`: For error handling.
 - `@deprecated`: Bits of the code no longer use.
 - `{@code }`: Put syntax in documentation.



Javadoc Tags

- Tags are keywords that do special things.

Writing Doc Comments

Format of a Doc Comment

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags. In this example, the block tags are `@param`, `@return`, and `@see`.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```



Build Files

MAKEFILES



Makefiles are Magic

- Old people like me who grew up on C and C++, it was all about the makefiles.
- A way of **automating the compilation process** in a world before Eclipse did everything for you.
- Used to develop in environments like Gvim, Emacs, and Vi.
 - Visual Basic was for L05serZ.
- Really useful if there are lots of classes and libraries.
- Also used for **installing new packages**.

```
$ make clean all install  
  
hellomake: hellomake.c hellofunc.c gcc -o  
hellomake hellomake.c hellofunc.c -I.  
  
ccc -o hellomake hellomake.c hellofunc.c -I.
```

Build Files: Beyond the Scope of a Basic Package

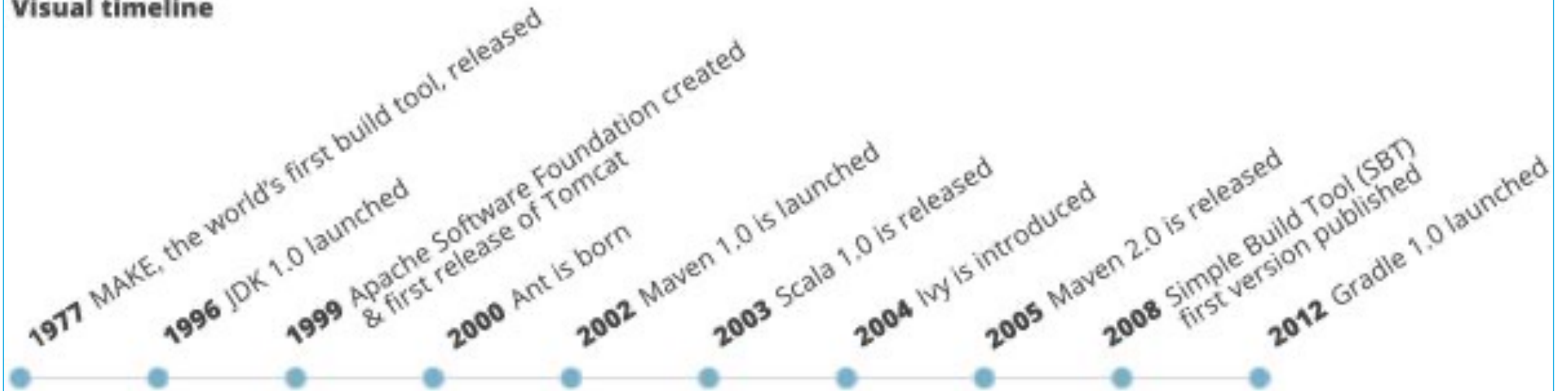


- Known as **build scripts**.
- A set of instructions for how the project should be “compiled” or built.
- IntelliJ/Eclipse handles some of these **automatically**.
- Have not used many external resources in examples so far.
- But, as project get bigger or more complex, or rely on more external sources, custom build scripts are needed.
- **Capabilities**: Can build in dependencies, package files, run tests, deploy software etc.



THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)

Visual timeline



Build Files: Beyond the Scope of a Basic Package



- Both Eclipse and IntelliJ can cope with both writing loads of classes and own packages.
- Build files are like a super amazing versions of makefiles which are used by old people who work with things like C and C++.
- They are massively useful when having lots of extra things like
 - Game engines.
 - Package dependencies.
 - Test harnesses including JUnit.
- As soon as you use code from **other developers**, a Build File is needed.



Build Files are Helpful for Maintenance

- It can be helpful in **ensuring code portability**.
 - It means that peer/teammate doesn't need to have exactly the same coding environment as you.
 - Eliminates class path problems if the build file is constructed.
- They can get really **complicated**: teams often have a member who works only on a build file.
- They ensure consistent compilation and eliminates “classpath” problems.
- It allows you to integrate:
 - Testing frameworks.
 - Automated quality metrics.
 - Deployment in unknown environments.



Common Frameworks for Build Files

- There are three main frameworks used by developers:
 1. Ant – <https://ant.apache.org>
 2. Maven – <https://maven.apache.org>
 3. Gradle – <https://gradle.org/>
- Next, let's have a look in detail at Ant (one of the oldest and widely used).
- Compare and contrast these tools.
- Note: although there is inbuilt support in IDEs like IntelliJ and Eclipse for these tools, many developers prefer to run them through command line.
 - For both brevity and simplicity.



Apache Ant

- “Ant” is a standard tool for Build Files for Java – “**Another Neat Tool**”.
- It is very similar to the principles of “make”, but it is only optimized and constructed with Java and not C in mind.
- Make is not quite as good with Java due to things like Java Virtual Machine (JVM).
- Ant is now about 20 years old.
- Contains 1 project and multiple targets (contains tasks).
- A **task** is a piece of code that can be executed.
- XML based-in the build file with special tags.





Anatomy of an Ant Build File

- Contains: Project and Target.
- Constructed: Quite strict XML.
- Example: like “make clean”.


```
<target name = "clean" description = "clean up">  
    <!-- delete the $(build) and $(dist) directory trees -->  
    <delete dir = "$(build)"> </delete>  
    <delete dir = "$(dist)"> </delete>  
</target>
```




Other Interesting Ant Facts

- It is the **most complete tool** for Java builds, but this can get tedious.
- It is great at **automating repetitive tasks**.
 - Can be used to write a custom framework for testing.
- It has a massive list of pre-defined tasks yet provides the facility to create own custom tasks.
- Can be run using command line but is also integrated into many IDEs.
 - IntelliJ/Eclipse can generate these for you!
- **However**: it is a bit deprecated these days: most prefer Maven, as in Ant, you need to specify everything explicitly.



- 
- ```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="zoo" default="all" basedir=".">
3
```



# Anatomy of Build File: Targets

- Target is a build module in Ant.
- Each target contain task(s) for Ant to do.
- One must be a project default.

```
<target name="register.custom.compilers">
 <taskdef name="javac2" classname="com.intellij.ant.Javac2" classpathref="javac2.classpath"/>
 <taskdef name="instrumentIdeaExtensions" classname="com.intellij.ant.InstrumentIdeaExtensions"
</target>
```

```
<target name="compile.module.zooproject" depends="compile.module.zooproject.production,compile.modu 81 ^ v
<target name="compile.module.zooproject.production" depends="register.custom.compilers" description="Compil
 <mkdir dir="${zooproject.output.dir}"/>
 <javac2 destdir="${zooproject.output.dir}" debug="${compiler.debug}" nowarn="${compiler.generate.no.warni
```



# Anatomy of Build File: Tasks

- Each target comprises one or more tasks.
- Task is a **piece of executable Java code** (e.g., javac, jar etc.).
- Tasks do the actual “build” work in Ant.
- Ant has core (built-in) tasks and the ability to create own tasks.

```
<target name="compile.module.zooproject.production" depends="register.custom.compilers" description="Comp
<mkdir dir="${zooproject.output.dir}"/>
<javac2 destdir="${zooproject.output.dir}" debug="${compiler.debug}" nowarn="${compiler.generate.no.war
 <compilerarg line="${compiler.args.zooproject}"/>
 <bootclasspath refid="zooproject.module.bootclasspath"/>
 <classpath refid="zooproject.module.production.classpath"/>
 <src refid="zooproject.module.sourcepath"/>
 <patternset refid="excluded.from.compilation.zooproject"/>
</javac2>
```

The screenshot shows the IntelliJ IDEA interface with the 'Build' menu open. The 'Generate Ant Build...' option is highlighted. A dialog box titled 'Generate Ant Build' is displayed, showing options for generating a single file or multiple files, and checkboxes for enabling UI forms compilation, using JDK definitions, inline runtime classpaths, and using the current IDEA instance for the idea.home property. A large red 'DEPRECATED!' watermark is overlaid on the dialog.

**Build Menu Options:**

- Build Project (Ctrl+F9)
- Build Module 'ZooProject'
- Rebuild (Ctrl+Shift+F9)
- Recompile (Ctrl+Shift+F9)
- Rebuild Project
- Build Artifacts...
- Generate Ant Build...**
- Analyze APK...
- zooproject
- Run Ant Target

**Generate Ant Build Dialog Options:**

- ☐ Generate multiple-file ant build (requires ant 1.6 or later to execute)
- ☒ Generate single file ant build
- ☒ Backup previously generated files
- ☐ Overwrite previously generated files
- Output file name: zooproject
- ☒ Enable UI forms compilation (requires "javac2" ant task from IDEA distribution)
- ☒ Use JDK definitions from project files
- ☒ Inline runtime classpaths
- ☒ Use current IDEA instance for idea.home property

**Ant View:**

- zooproject
  - all
  - build.modules
  - clean
  - clean.module.zooproject
  - compile.module.zooproject

The screenshot shows the IntelliJ IDEA IDE interface. The main editor displays the `zooproject.xml` file. The Project tool window on the left shows the project structure, with `zooproject.xml` highlighted. The Ant tool window on the right shows the build targets for the project.

**Project Structure (Left):**

- 1: Project
  - ZooProject
    - .idea
    - out
    - src
      - com.ae2dms.zooproject
      - com.ae2dms.zooproject.animal
      - com.ae2dms.zooproject.db
      - com.ae2dms.zooproject.employ
      - com.ae2dms.zooproject.food
      - com.ae2dms.zooproject.misc
      - com.ae2dms.zooproject.visitor
      - com.ae2dms.zooproject.zoo
    - ZooProject.iml
    - zooobject.properties
    - zooobject.xml
- 2: Structure
  - External Libraries
  - Scratches and Consoles
- 3: Favorites

**Main Editor (Center):**

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="zooproject" default="all">

 <property file="zooproject.properties"/>
 <!-- Uncomment the following property if no tests compilation is needed -->
 <!--
 <property name="skip.tests" value="true"/>
 -->

 <!-- Compiler options -->

 <property name="compiler.debug" value="on"/>
 <property name="compiler.generate.no.warnings" value="off"/>
 <property name="compiler.args" value=""/>
 <property name="compiler.max.memory" value="700m"/>
 <patternset id="ignored.files">
 <exclude name="**/*.hprof/**"/>
 <exclude name="**/*.pyc/**"/>
 <exclude name="**/*.pyo/**"/>
 <exclude name="**/*.rhc/**"/>
 </patternset>
</project>
```

**Ant Tool Window (Right):**

- zooproject
  - all
  - build.modules
  - clean
  - clean.module.zooproject
  - compile.module.zooproject
  - compile.module.zooproject.production
  - compile.module.zooproject.tests
  - init
  - register.custom.compilers

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help ZooProject - zooproject.xml

ZooProject > zooproject.xml

Project > zooproject.xml

1: Project

- ZooProject C:\Users\leebg\IdeaProjec
  - .idea
  - out
  - src
    - com.ae2dms.zooproject
    - com.ae2dms.zooproject.animal
    - com.ae2dms.zooproject.db
    - com.ae2dms.zooproject.employ
    - com.ae2dms.zooproject.food
    - com.ae2dms.zooproject.misc
    - com.ae2dms.zooproject.visitor
    - com.ae2dms.zooproject.zoo
  - ZooProject.iml
  - zooproject.properties
  - zooproject.xml

2: Structure

- External Libraries
- Scratches and Consoles

3: Favorites

```
229 <target name="clean.module.zooproject" description="cleanup" >
230 <delete dir="${zooproject.output.dir}"/>
231 <delete dir="${zooproject.testoutput.dir}"/>
232 </target>
233
234 <target name="init" description="Build initialization">
235 <!-- Perform any build initialization in this target -->
236 </target>
237
238 <target name="clean" depends="clean.module.zooproject" description="c
239
240 <target name="build.modules" depends="init, clean, compile.module.zoo
241
242 <target name="all" depends="build.modules" description="build all"/>
243 </project>
```

Ant

- zooproject
  - all
  - build.modules
  - clean
  - clean.module
  - compile.mod
  - compile.mod
  - compile.mod
  - init
  - register.custo

Run Target  
Create Meta Target  
Create Run Configuration  
Remove Meta Target  
Jump to Source  
Execute on  
Assign Shortcut...  
Properties

project > target

Event Log

238:17 CRLF UTF-8 2 spaces\*





# Generate Ant Build File

---

- <https://www.jetbrains.com/help/idea/generating-ant-build-file.html>
- \*\*Required Plugin: Ant Build Generation - <https://www.jetbrains.com/help/idea/managing-plugins.html>
- Some useful Ant Tasks - <https://ant.apache.org/manual/tasklist.html>





File Edit View Navigat

File

- New
- Open...
- Open Recent
- Close Project
- Settings...
- Project Structure...
- File Properties
- Save All
- Reload All from Disk
- Invalidate Caches / Restart
- Manage IDE Settings
- New Projects Settings
- Export
- Print...
- Add to Favorites
- Power Save Mode
- Exit

Project Structure

Scratches and Console

TODO Problems

Edit application settings

Settings

Appearance & Behavior

Keymap

Editor

Plugins

Version Control

Build, Execution, Deployment

Languages & Frameworks

Tools

Plugins

Search in Marketplace

Ant Build Generation

Search Results (4)

Sort By: Relevance

Ant Build Generation  
1.3K

ANTLR v4 grammar plugin  
294.1K 4.50

Jar Tool  
15.5K 4.13

hSenid Mobile TAP IDE  
987

Click "Install"

Click "Search in Marketplace"

Ant Build Generation

1.3K JetBrains

Build 202.0 Sep 08, 2020

Plugin homepage

Supports generating an Ant build file out of an IntelliJ IDEA project

Size: 18.9K

DEPRECATED!

OK Cancel Apply



# Maven: Less Tedious than Ant

- Released in 2004, also by Apache with the goal to make it better than Ant!
- It's more of a pro-choice issue like Eclipse vs IntelliJ or Emacs vs Vim or Diet Coke vs Pepsi Max.
- Maven's strength is that it manages for you the use of dependencies, know where to find source files etc.
  - The dependencies have to do it manually at Ant, but Ant remains more flexibility.
- “Convention over Configuration”.
  - Maven relies on conventions and provides predefined commands as goals.
- Could be more complex than Ant sometimes!



# Maven: Project Object Model (POM)



- XML representation of a Maven project held in a file named `pom.xml`.

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
6.
7. <!-- The Basics -->
8. <groupId>...</groupId>
9. <artifactId>...</artifactId>
10. <version>...</version>
11. <packaging>...</packaging>
12. <dependencies>...</dependencies>
13. <parent>...</parent>
14. <dependencyManagement>...</dependencyManagement>
15. <modules>...</modules>
16. <properties>...</properties>
17.
18. <!-- Build Settings -->
19. <build>...</build>
```

*Object model version*

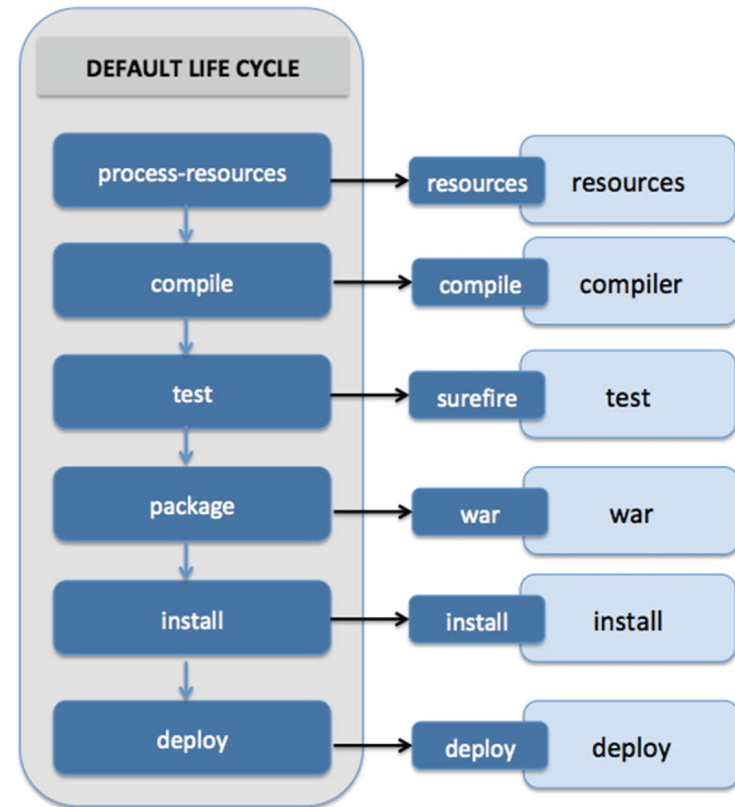
*Group / organization ID*  
*Project ID*  
*Project version*  
*Packaging type*  
*Dependencies*

For more details,  
see [https://maven.apache.org/pom.html#What is the POM](https://maven.apache.org/pom.html#What%20is%20the%20POM)



# Maven Phase – Build Lifecycle

- Validate: Check if all information necessary for the build is available.
- Compile: Source code compilation.
- Test-compile: Compile the test source code.
- Test: Run unit tests.
- Package: Compile source code into distributable format (jar, war, ...).
- Integration-test: Process and deploy the package if needed to run integration tests.
- Install: Package installation to a local repository.
- Deploy: Copy the package to the remote repository.



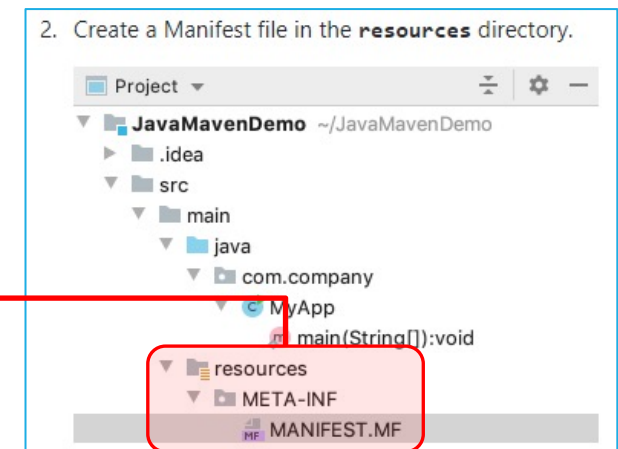
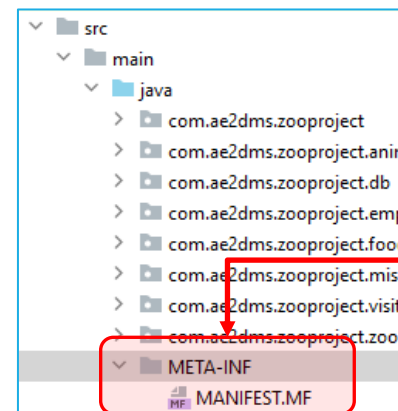
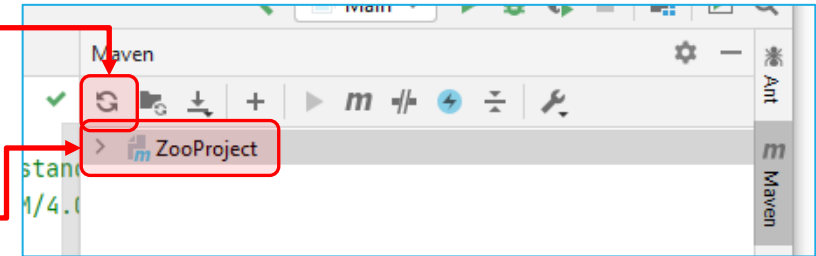


# Create Jar File with Maven

- <https://www.jetbrains.com/help/idea/convert-a-regular-project-into-a-maven-project.html>

- Troubleshooting:

- Click “Reload All Maven Projects” if shows “Unknown”.
- **NOTE:** The Manifest file is created in src.main.java.META-INF by default, **NOT** in the resources folder as stated in the guideline.

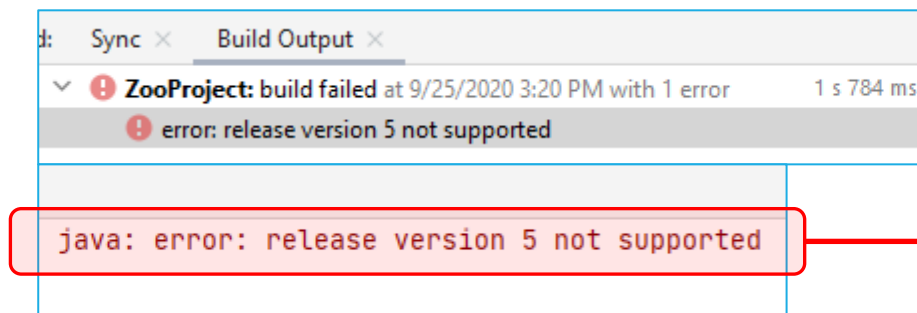


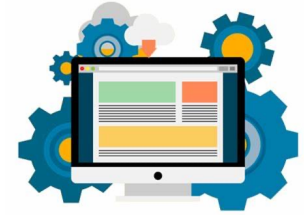


# Create Jar File with Maven

## ○ Troubleshooting:

- Add the following code to `pom.xml` if shows “java: error: release version X not supported”, then click “Reload All Maven Projects” again.





# Create Jar File with Maven

- Troubleshooting:
  - Remember to change the location of manifest file in step 3.

3. In your POM specify the [Manifest file](#) ↗ information, so you can use Maven to generate an executable **jar** file.

```
× pom.xml
<modelVersion>4.0.0</modelVersion>
<groupId>groupId</groupId>
<artifactId>JavaMavenDemo</artifactId>
<version>1.0-SNAPSHOT</version>
<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <configuration>
 <archive>
 <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
 </archive>
 </configuration>
 </plugin>
 </plugins>
</build>
</project>
```

*src/main/java/META-INF/MANIFEST.MF*



# Gradle: A Contemporary Approach (2012?)



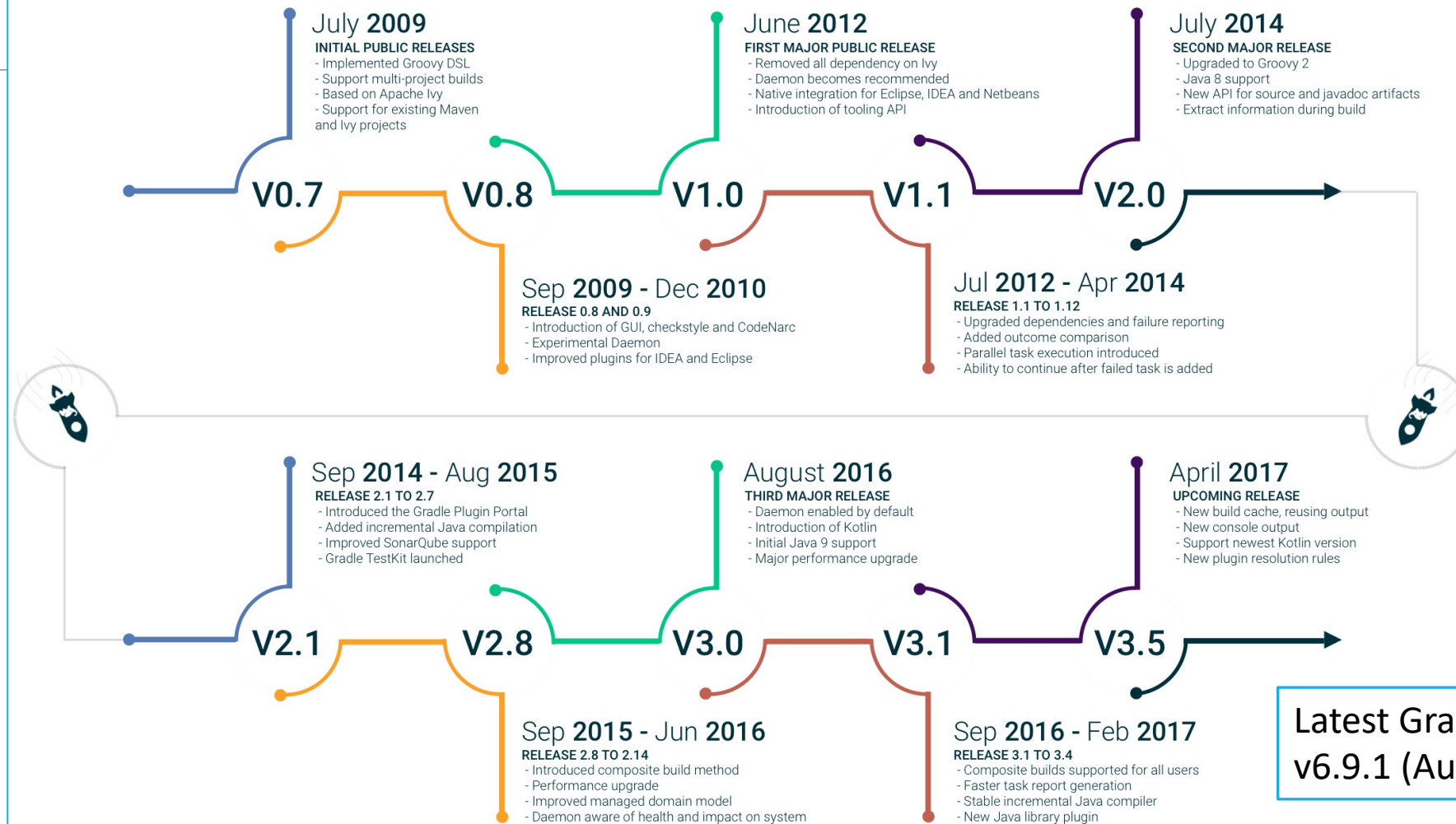
- Released in 2012 (most recent compared to Ant and Maven).
- Tried to take best parts of Ant and Maven.
- Dispenses with XML in favor of a domain specific language “groovy”.
- Based on a programming language, therefore can implement control flow.
- Although it uses groovy itself, it can act as a build manager for any language.
- **Declarative**: plug-ins add functionality.
- More cleanly accomplishes required tasks of a typical development project, from compilation through testing and deployment.
- It's the official build system for Android.







# Evolution of Gradle



Latest Gradle version –  
v6.9.1 (August 2021).



# Gradle Build Cycle

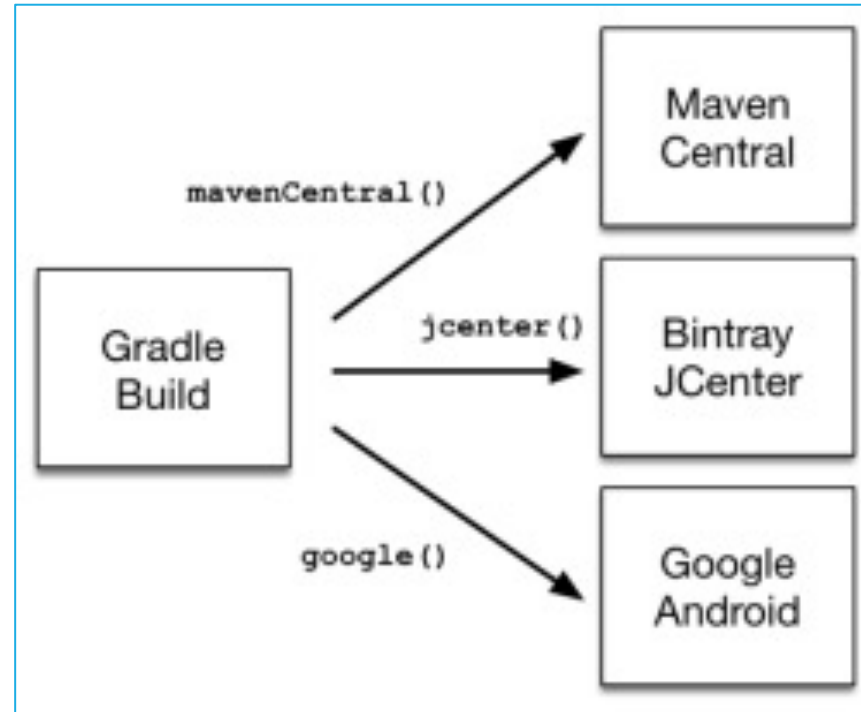
---

- It launches as a new JVM process.
  - It parses the `gradle.properties` file and configures Gradle accordingly.
- Next, it creates a Settings instance for the build.
- Then, it evaluates the `settings.gradle` file against the Settings object.
- It creates a hierarchy of Projects, based on the configured Settings object.
- Finally, it executes each `build.gradle` against its project.
  - These files must be precisely named.
  - Automatic setup using a Gradle Wrapper.



# Example:

- Integration with public and open source to link to available sources and working with dependencies.



# Build Files Tutorial Videos – *for the brave and challenge*

---



- Tutorial series on Gradle.
  - Designed for the more advanced level.
  - Will use **Maven in the Computing Lab session**.



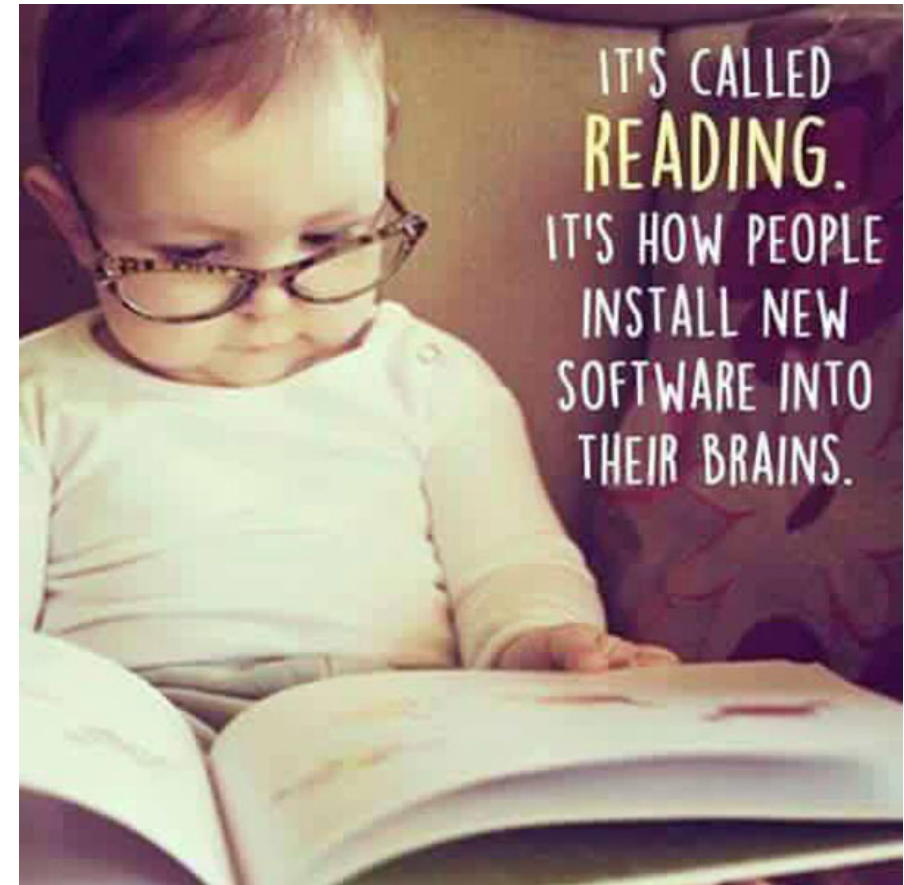
# Build Files Tutorial Videos – *for the brave and challenge*

- Gradle Beginner Tutorials (4 videos)
  - <https://www.youtube.com/playlist?list=PLhW3qG5bs-L8kzOvEjaOyUs2LqHZ3fz4X>
- How to create Gradle Java Project using IntelliJ IDEA?
  - [https://youtu.be/Y8\\_rQ20C1CI](https://youtu.be/Y8_rQ20C1CI)
- Gradle Tutorial – Why You Should Use it and How to Get Started
  - <https://youtu.be/ojx49J1JCdQ>
- Working with Gradle in IntelliJ IDEA (Java SDK 1.8)
  - <https://youtu.be/JwPYjnhah3g>



# Reading Homework

- Interesting blog articles for you to read by comparing the two:
  - Maven vs Ant: [https://www.adam-bien.com/roller/abien/entry/maven\\_vs\\_ant](https://www.adam-bien.com/roller/abien/entry/maven_vs_ant)
  - Ant vs Maven vs Gradle: <https://www.baeldung.com/ant-maven-gradle>





# Repository Tools

---

CONTROL SOURCE



# GIT and Repository Tools

---

- Git is a (free and open source!) distributed version control system.
- Designed originally for command line use.
- But there are various GUI clients available.
- And web front ends for management, such as GitLab and GitHub.
  - E.g., <https://csprojects.nottingham.edu.cn>





# Git Resources for Homework

---

- Get familiar with Git.
- Resources Git book available for free:
  - Pro Git by Chacon and Straub: <https://git-scm.com/book/en/v2>
- Tutorial:
  - <https://git-scm.com/docs/gittutorial>
- Cheat sheets:
  - [http://rogerdudler.github.io/git-guide/files/git\\_cheat\\_sheet.pdf](http://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf)





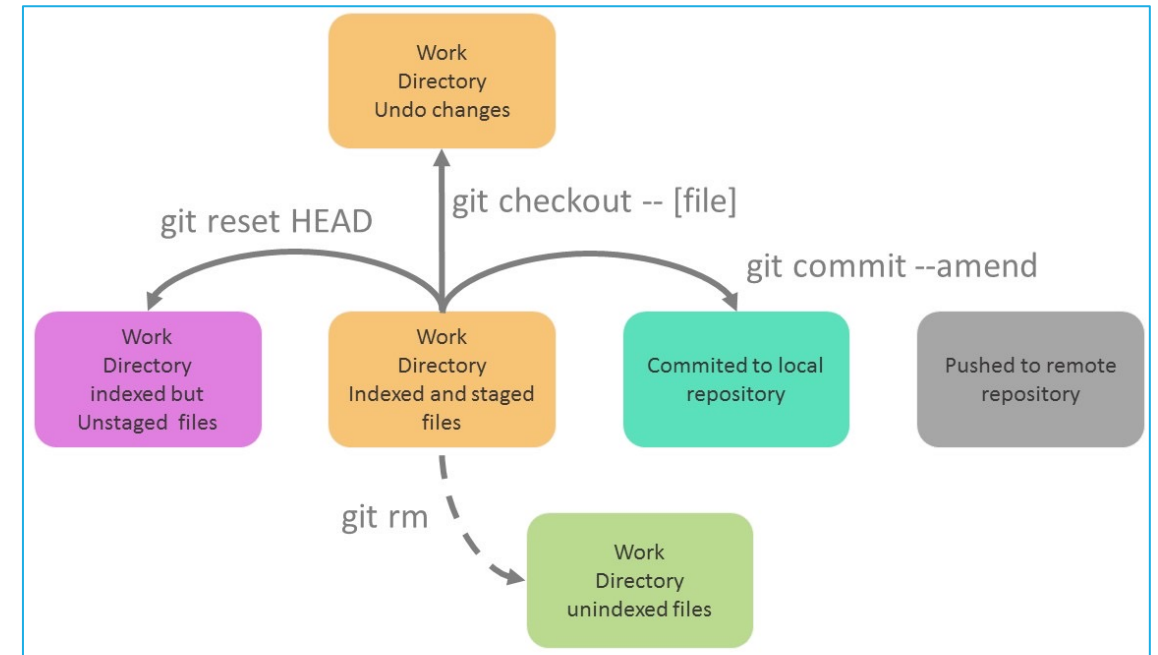
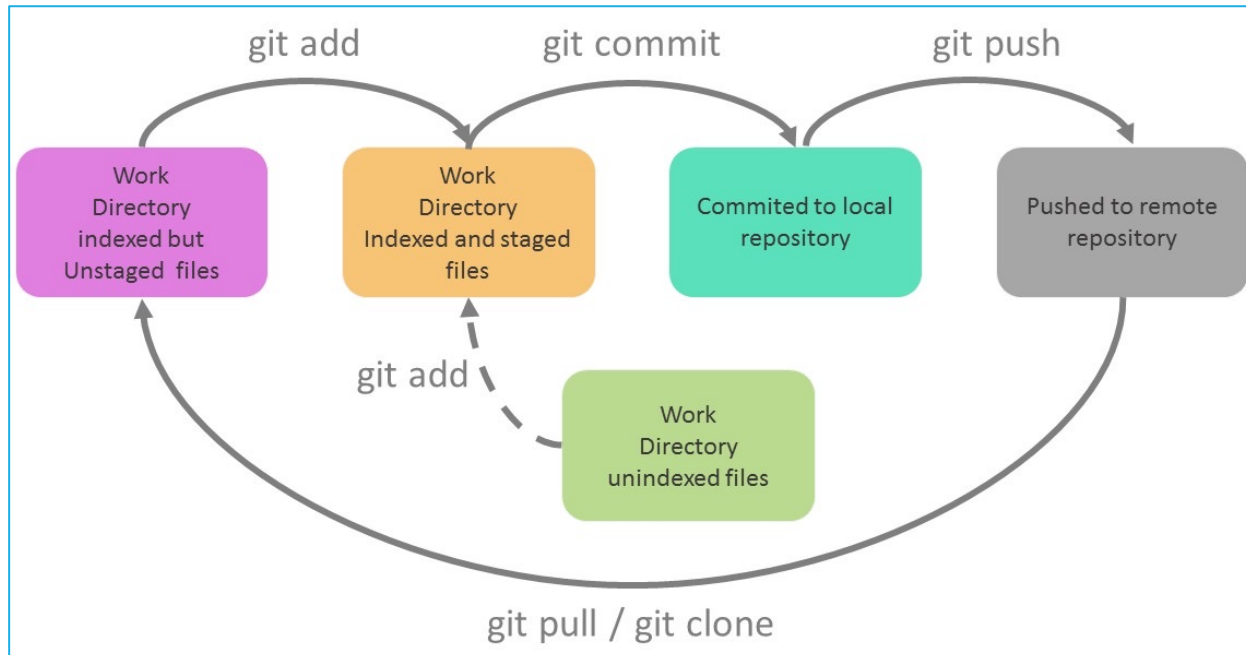
# Git

---

- You have a folder assigned as your project folder.
- You instruct Git that these are the files you want to **keep track** of.
- Build a new one; and then Git allows for the creation of history.
  - Like loading a saved game in GTA, went to casino, lost the bet, reset until win the bet = \$\$\$\$.
- Git allows you to:
  - Create repository.
  - Commit code with initial version.
  - Commits transmit back and forth.
  - Clone or revert and manage history.
- Work on your own, but what if you need to work with your friends?

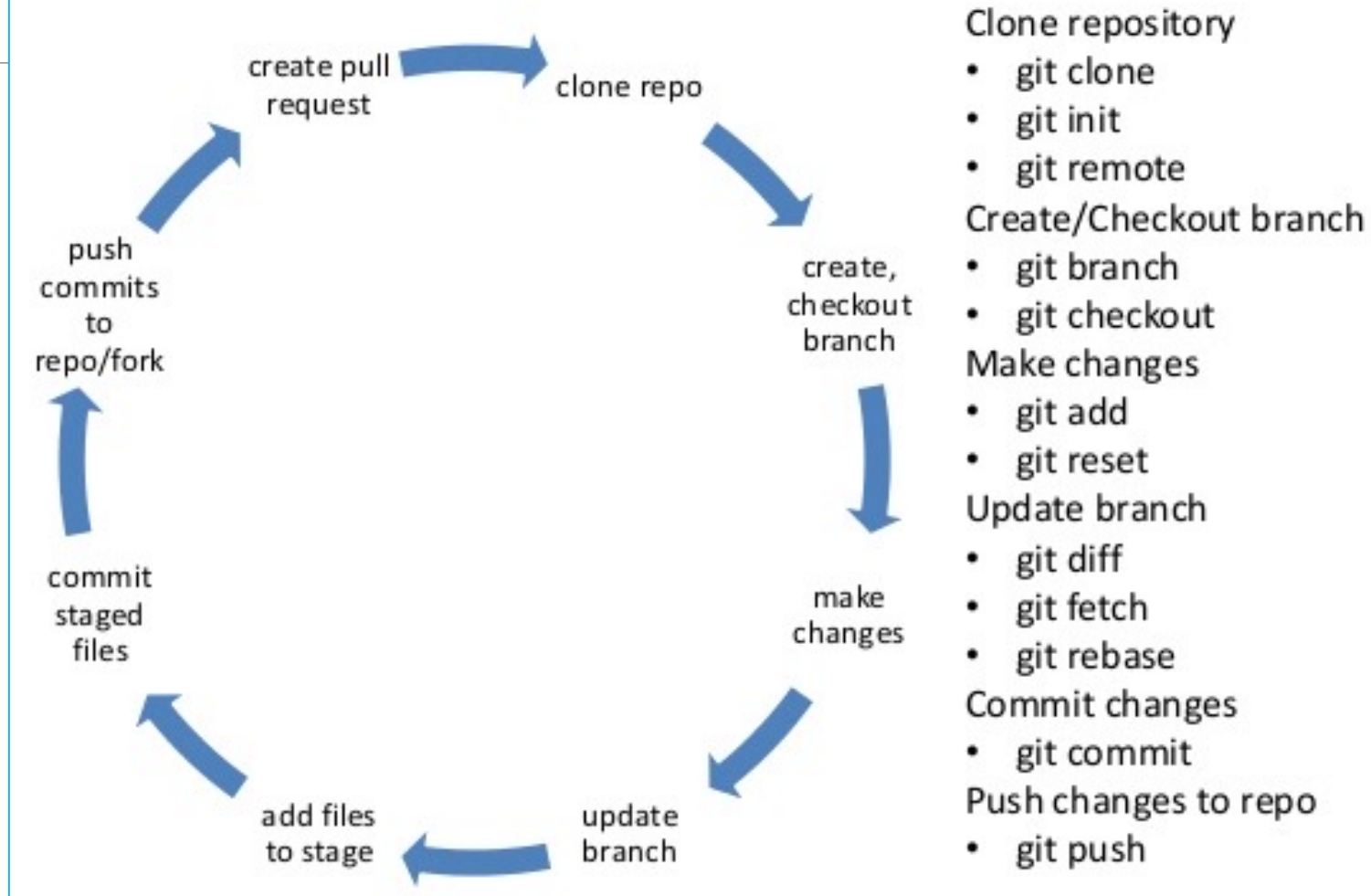


# Git Work Cycle





# Git Work Flow Cycle





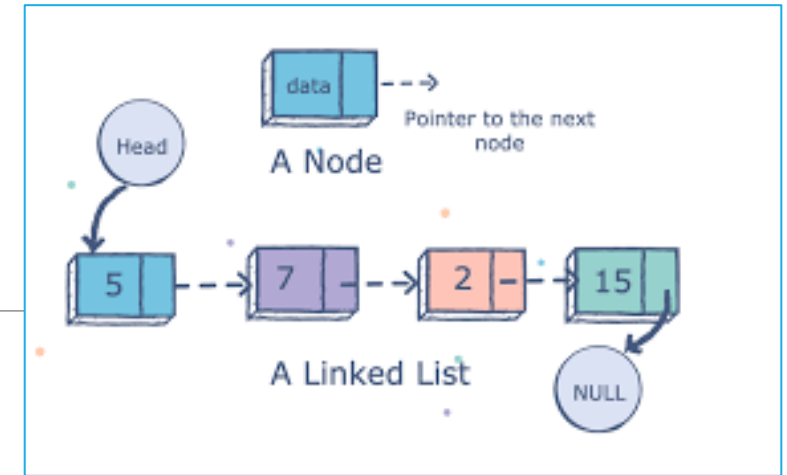
# Multi-User Management with Git

---

- In the old days “three ways merge”.
- You can your peers want to create a commit which includes the changes of both parties.
- If you have edited different parts of the file, Git will know what to do, usually.
- If you have edited the **SAME** part of the file, a good tool will show you both modifications and then you choose which one to go with.
- Forking and branches ...
- Interesting features like “cherry picking” version.

# Branching in Git

- Each commit is a node in a **linked list** on disk.
- Branch is the pointer to that node.
  - If you want to force the branch back, you can lose the old node.
- History tree is preserved by the data structure retaining its linked list integrity.
- Server have their own copies of branches.
  - Pushing or pulling can result in moving your own or someone-else's
- Best thing:
  - Tiny itty-bitty branches so to avoid egomaniacs working on the master branch.
  - Our server is configured to provide protection of the master (can be turned off).

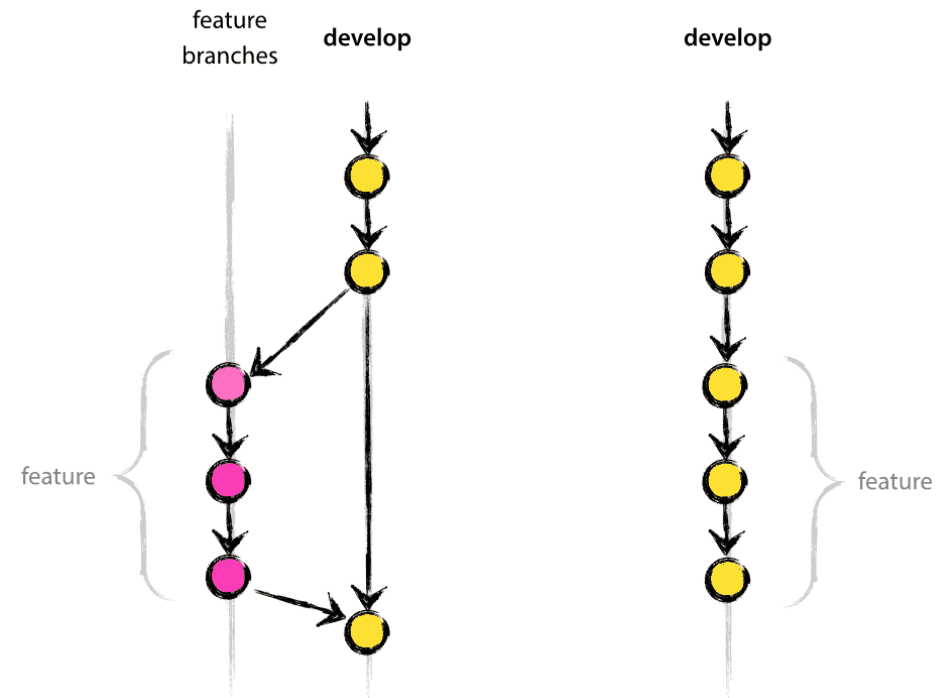




# Branching Models

## ○ Feature Branches

- The target release for the feature may be unknown at the start of development.
- A feature branch exists while the feature is in development.
- The branch will eventually either:
  - Be merged into the develop branch (if the feature is successful)
  - Be discarded (if the feature is not needed or is unsuccessful).
- Feature branches usually exist in developer repositories, not in the main repository (origin).





# Branching Models

---

## ○ Release Branches

- These branches allow for:
  - Minor bug fixes.
  - Adjustments like version numbers and build dates.
  - Last-minute changes before release.
- The **develop branch** remains available for future feature development while the release branch is prepared.
- A **release branch** is created when the develop branch almost reflects the desired release state.
- Features for future releases must wait until the release branch is created.
- The **version number** is assigned when the release branch is started, following project versioning rules.

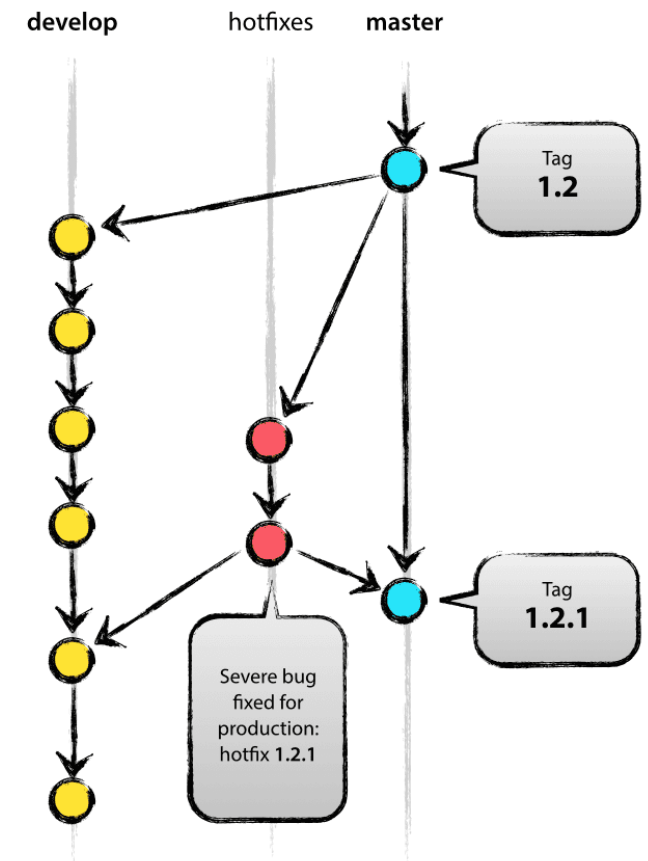


# Branching Models



## ○ Hotfix Branches

- Created for **unplanned** production releases to fix critical issues in a live version.
- They are similar to release branches but arise from urgent bug fixes.
- A hotfix branch is branched from the **master branch**, specifically from the tag marking the current production version.
- This allows regular development work on the **develop branch** to continue uninterrupted while the hotfix is addressed.





# Branch vs Tag

---

- **Branch**

- An active line of development, with the most recent commit referred to as the **tip** of the branch.
- The **branch head** references the tip of the branch and moves forward as development progresses.
- A git repository can track multiple branches, but the working tree is associated with only one active (checked out) branch at a time.
- **HEAD** points to the current branch.

- **Tag**

- Points to a specific commit or object and is not moved by future commits, commonly used to mark significant points in the commit history.



# Tools for Using Git

---

- Command line tools, built into clients, IDEs, browsers, pretty much anything, they are CLIENTS.
- The servers run another instance of Git but often provide a **basic interface** for configuration.
- Example: GitKraken
  - Usually silently download the changes for you whenever it sees them.
  - But doesn't update any of the work for you.
  - <https://www.gitkraken.com/>
- GitHub like setups have fancy interfaces which allow you to do stuff like review other peoples changes before you accept them into master.
  - Still plain old Git under the hood.



# Alternatives to Git

---

- Trade war between Git, Mercurial and BitKeeper, it got nasty!
  - Bitkeeper won, but no one really cares much about them anymore.
- For your own GitHub.
  - Gogs: Fast and good.
  - GitBucket: Also good, pretty but not as fast.
  - Gitea: Community based development of Gogs.
  - GitLab: Lots of features, large overhead.
- No set rules for which one to use, its generally good though to make sure you are happy with the reliability!

# Git's Fun Other Commands and Additions



- Great for group projects (especially when members go missing/off script) – perfectly suitable in this COVID-19 situation.
- `Git blame <filename>`
  - Show what revision and author last modified each line of a file.
  - <https://git-scm.com/docs/git-blame>
- `Git bisect`
  - Text based wizard to find out a bad git commit file.
  - <https://git-scm.com/docs/git-bisect>
- Needing to clean up the repository with the BFG repository cleaner as an alternative to `git-filter-branch`.
  - <https://rtyley.github.io/bfg-repo-cleaner/>

# Self Study - Homework

- Set up a Git repository with IntelliJ IDEA
  - <https://www.jetbrains.com/help/idea/set-up-a-git-repository.html#clone-repo>
- GitLab IntelliJ
  - [https://youtu.be/KZwUAbyYC\\_M](https://youtu.be/KZwUAbyYC_M)
- IntelliJ IDEA: Adding a Project to Git and GitHub
  - <https://youtu.be/mf2-MO10VXY>

## ▼ Git

### Set up a Git repository

Sync with a remote Git repository  
(fetch, pull, update)

Commit and push changes to Git  
repository

Investigate changes in Git  
repository

Manage Git branches

Apply changes from one Git branch  
to another

Resolve Git conflicts

Use Git to work on several features  
simultaneously

Undo changes in Git repository

Use tags to mark specific Git  
commits

Edit Git project history



# Container Systems

---

BEYOND THE VIRTUAL MACHINE WITH DOCKER –  
SHIP YOUR COMPUTER



# Meet Docker

---

- In Build Files, we discussed the need for portability.
- “Oh no! But the coursework worked on my machine”.
  - “Where is your machine?”
    - “Oh, it meant my friend’s machine”.
  - Docker can help us with that.
  - “If it works on your machine, we can ship your machine”.
- Using containers, you can effectively fake a replica of a particular computing environment.
- Essentially a \*Virtual Machine (\*VM) for a platform environment.
- Centered around an inheritance hierarchy.



# Once Upon a Time ... (Example with Python)

- There were two software, which needed Python v2.7 and another which needed Python v3.8.6.
- Both were clever enough to find the wrong version and tried to use it.
  - Haiyaaa!!! Of course, this was a disaster and crashed.
- We created a container for one, and a separate container for the other.
- They both lived happily ever after, not knowing that they were sharing the same computer.
- The partitioning into these virtual containers means you can even be as drastic as delete entire codebase.
- It even runs nicely on a Raspberry Pi (RPi)!



**Container**



# Hardware/Software Requirements for Docker



- Any \*nix (Unix, Linux, BSD, RPi) will run Docker if the packages are integrated into your package manager.
- Windows 10 Professional/Enterprise Editions are OK with Docker.
  - Windows 10 Home Edition refuses to work as it wants you to upgrade to the pro version.
  - For older versions of windows, it is recommended that you run a virtual machine with Linux and run the Docker inside the VM.
  - Or you can plug in a RPi for about ¥125 for a RPi Zero W.



# Self Study Docker Resources

- Docker using IntelliJ (Tutorial)
  - <https://www.jetbrains.com/help/idea/docker.html#run-containers>
- Docker with IntelliJ
  - <https://youtu.be/h8QMB0EZMK8>
- Docker Containers & Java: What I wish I Had Been Told
  - <https://youtu.be/d7ajT14ENKk>





# Testing

---



# Testing Your Code is the Most Important Process

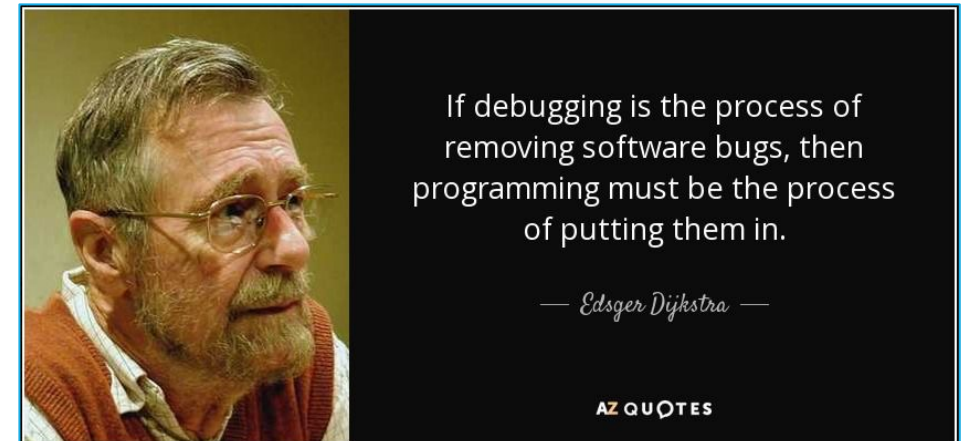


- Software testing is the foundation on which software development is based on.
- Little to no point in writing code if you are not going to test it.
- Many different **aspects** of testing exist:
  - Developing a testing strategy.
  - Unit testing, regression testing
  - Test Driven Development (TDD) with JUnit.
  - Property based testing.
- Testing and debugging are often used synonymously but they are distinct.
  - Testing shows the code functions as expected.
  - Debugging allows you to identify what is causing a test to fail.



# Famous Quotes on Testing

- “Debugging is what you do when you know your program is broken. Testing is a determined, systematic attempt to break a program that you think is working.”
  - Brian Kernighan and Rob Pike – The Practice of Programming.
  
- “Testing can demonstrate the presence of bugs but not their absence.”
  - Edsgar Dijkstra







# Coding is Never a Perfect Process

---

- We all like to think that the code we write is perfect in construction and so that we have no need for systematic and extensive coverage.
- This assumption of belief can lead to catastrophic consequences.
  - Buffer overflow exploits and many other security flaws.
  - Challenger disaster.
  - Radiation poisoning due to decimal point error.
- Testing can start as small as checking a value has been copied correctly or as large as using a completely **automated continuous integration** server.
- In essence, test, test your tests, and test your tests testing.

# Coding Best Practice to Minimize Errors



- The more automation use in development process, the fewer errors are likely to be introduced in the software.
- If code writing feels mechanical, then it should be mechanized.
  - **Auto generation from frameworks** like visual paradigm or auto class method stub generation in IntelliJ (<https://www.jetbrains.com/help/idea/generating-code.html>).
  - Using regular expression to specify patterns of text and variable sets.
  - Using auto complete tools where available to avoid simple typing errors and mistakes.
- In general, if a generator is correct, specification is covered, the interpreter/compiler is correct, so too will be the resulting program.
  - Sometimes though, can't make these assumptions without testing.





# Test Your Code as You Write

---

- The biggest difference between a person who has learnt to code versus a seasoned developer is in the mindset regarding testing.
- The **earlier a problem is found, the easier it is to fix**.
  - Repeatable errors and bugs are usually OK to track down.
  - Use assistance from a debugger, putting breakpoints and watches on variables (<https://www.jetbrains.com/help/idea/using-breakpoints.html#set-breakpoints>).
- Transient or intermittent faults are an absolute nightmare.
  - Ensuring effective testing coverage minimizes the chance of this scenario.
- Professional developers write code with testing at the forefront of their minds.

# “BUT ....”



- **“I am confident that my code is correct!”**
- Even the simplest of operations should be tested as they are written.
- As a result, more complex bugs do not get the chance to occur.
- It is a grave mistake to make that just because something compiles or IntelliJ says its OK, it does not mean that it does not need to be tested.
- Initially, let's look at Boundary Test.



# Test Code at Its Boundaries

---

- Boundary condition testing is the **absolute minimum amount of testing** that you should perform.
- It involves probing the natural boundaries within a small unit of your code.
- Based on the assumption that most errors will occur at the boundaries.
- Examples:
  - Non-existent or empty input.
  - Single item input.
  - Completely full array.
  - While loop conditions and sentinels.
  - Logical statements.



# What Can Go Wrong?

---

```
public double average(double[] gradesArray, int n) {
 int i;
 double sum = 0.0;

 for(i = 0; i < n; i++)
 sum += gradesArray[i];

 return sum / n;
}
```



# Things that Will Go Wrong

- So,  $n$  can represent the full set of integers: it's a good idea to test the boundary of the smallest value which is 0.
- An array with no elements is meaningful to the function but a value of  $n = 0$  to calculate an average is not.
- **Question: What would this function do?**
  - Continue with division by zero and return not a number (NaN)?
  - Abort the program?
  - Complain to the programmer?
  - Return an erroneous random value?
  - Simply return the integer 0?
- `return n <= 0 ? 0.0 : sum / n; ??`



# The Answer is ...

---

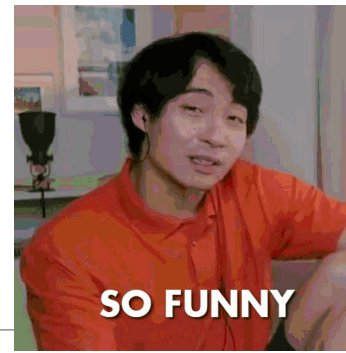
- The answer is NEVER ... to do anything.
- In the old days, would print stuff to standard error stream and save this a log file so one could read the failure conditions.
- There are modern, less archaic strategies.

# Using Assertions in Pre/Post Condition Testing



- Java, C++, Python etc. have an assertion testing facility to add pre and post condition to ensure that these assertions are validated.
- **Assertions** are used to abort the program when a condition fails.
- Example: could have added `assert (n > 0)` to previous function to avoid the division by zero.
- Rarely see this style of coding in student coursework (other than FSE)!
  - Have you been taught this? If so, why are you not using it?
  - If not, learn how to use it!

# Program Defensively: “It’ll Never Happen!”



- We didn’t think that Trump would be President of USA or that Brexit would be happening either, just make you think eh!
- No matter how you think through how a program will be used, you need to account for unlikely or rare situations.
  - This is another fault I see frequently in labs and in student coursework!
- Example:
  - “A user would never enter a negative value for their age.”
  - “This file will always be created by other class that I have written.”
- Also, as software is maintained or changed, it can cause changes to values, files, data, class structure, available methods, that you can’t always predict.
- The process of performing these simple tests on small units of code is called ...



# Program Defensively: “It’ll Never Happen!”

---



- Never underestimate the users!



# ... Unit Testing

---

- This can be **WHITE BOX** or **BLACK BOX** testing.
- Question: If you know how to do this, why have so many forgotten to do this while developing the ZooApp in our computing labs?
- When writing code and managing your build, you should always know what the expected output is of every method, the size of every buffer and the type of every variable.
- Testing can ensure you have coded what you intended to code if you write the tests first, even if you do this on paper.



# Unit Testing with Help from JUnit

---

- When code gets very complex, will need to employ some help from a library that allows us to build a test harness/scaffold.
- Today will look at **JUnit** as it works nicely with the Java based principles that are teaching in this module.
- In real life, the business analyst or product owner will provide some sample input and output to an application.
- JUnit is not part of the standard JDK but provides an interface or API to a testing library.
- It allows for extensive test coverage.



# Adding JUnit to Project

---

- Simply add junit.jar to classpath or add it to xml in Maven build file.
  - JUnit is already supported in IntelliJ IDEA, Eclipse and NetBeans.
  - Prepare for Testing in IntelliJ: <https://www.jetbrains.com/help/idea/testing.html>
- Can use a Test-Driven Development (TDD) approach of writing tests in JUnit BEFORE write the code.
  - Tests all fail at first as there isn't any code.
  - Once have added all the codes, and they pass all the tests, all done!
- Can achieve exceptional coverage of your code using this method.
- Remember, must run all tests every time of compilation, so it is put in a separate testing folder.



# The Best Way

---

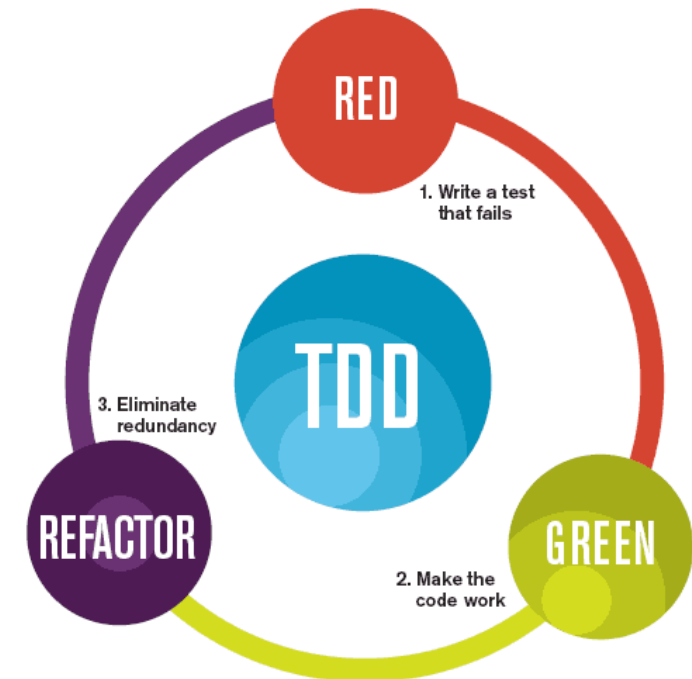
- The best way to understand JUnit, is to use JUnit.
- See what Heng Yu has developed in the Computing Lab.





# Test-Driven Development (TDD)

- Not actually about testing or development – Fowler, 1999.
  - Gaining full coverage.
  - Developing a sustainable way through refactoring.
- RED, GREEN, REFACTOR.
  - Write the test first.
    - The code fails.
  - Write the code.
    - Repeat until the tests are passed.
  - Refactor the code.
    - Increase the maintainability and readability.
    - Preserve the functionality.
  - Agile in Practice: Test Driven Development (<https://youtu.be/uGaNkTahrIw>)
- Tutorial: Test Driven Development in IntelliJ
  - <https://www.jetbrains.com/help/idea/tdd-with-intellij-idea.html>



The mantra of Test-Driven Development (TDD) is "red, green, refactor."



# Regression Testing

---



# Regression Testing

---

- This is a **major concept** in Developing Maintainable Software.
- When make changes to codebase, test the new code added AND the whole rest of all the codes in the application.
- Modification may have broken the existing code.
  - Never just test the new fix that has been implemented.
- This requires extensive, systematic and automated methods for testing programs.
- Otherwise, the complexity of the testing becomes far too big to manage.



# Regression Testing with Test Scaffolds



- Can do a DIY small scale approach.
- A **test scaffold** to interface and run hundreds, thousands, millions of tests EVERY TIME THE CODE COMPILES.
- Couldn't do this without build files which help to construct scaffolds.
  - Many JUnit tutorials require one first to understand how to use Maven.
- Tools and framework exist to aid this automation.
- Big projects with many people use **continuous integration servers**.
- Test suits or scaffolds rely on the same principles as writing with minimum errors: “if it can be mechanized then it should.”



# DIY Approaches with UNIX Tools

---

- Much of the code develop for coursework or projects is not large enough in scope to warrant a large test scaffold.
- There are some simple techniques to use that can help to do this effectively and quickly.
- Classic UNIX tools can help: `wc` `cmp` `diff` `grep` `freq`.
- Create a log file where the results of the tests are stored.
- Use UNIX tools to compare the test logs before and after changes were made to the code.



---

```
for i in ka_data.* // loop through test logs
do
 old_ka $i > out1 // run old version
 new_ka $i > out2 // run new version
 if !cmp -s out1 out2 // compare logs
 then
echo $1: is_bad // outputs that there's a problem
 diff out1 out2
 wc out1 out2
 fi
done
```

This script only produces any output if there is a problem – it runs silently.

# CI Servers Working with Everything Else!



- Static **analysis** in the Continuous Integration (CI) bit.
- GitLab built in CIS stuff.
- GitLab using YAML.
  - Commit a special file called `.gitlab-ci.yml` in project root.
  - Composed of jobs of work, rules about when they should be run, and scripts of commands.
  - Set up Runners in GitLab CI to action the jobs in the script.
- Make stuff work with Javadoc, JUnit, Test hardness, build files, CI server scripts.



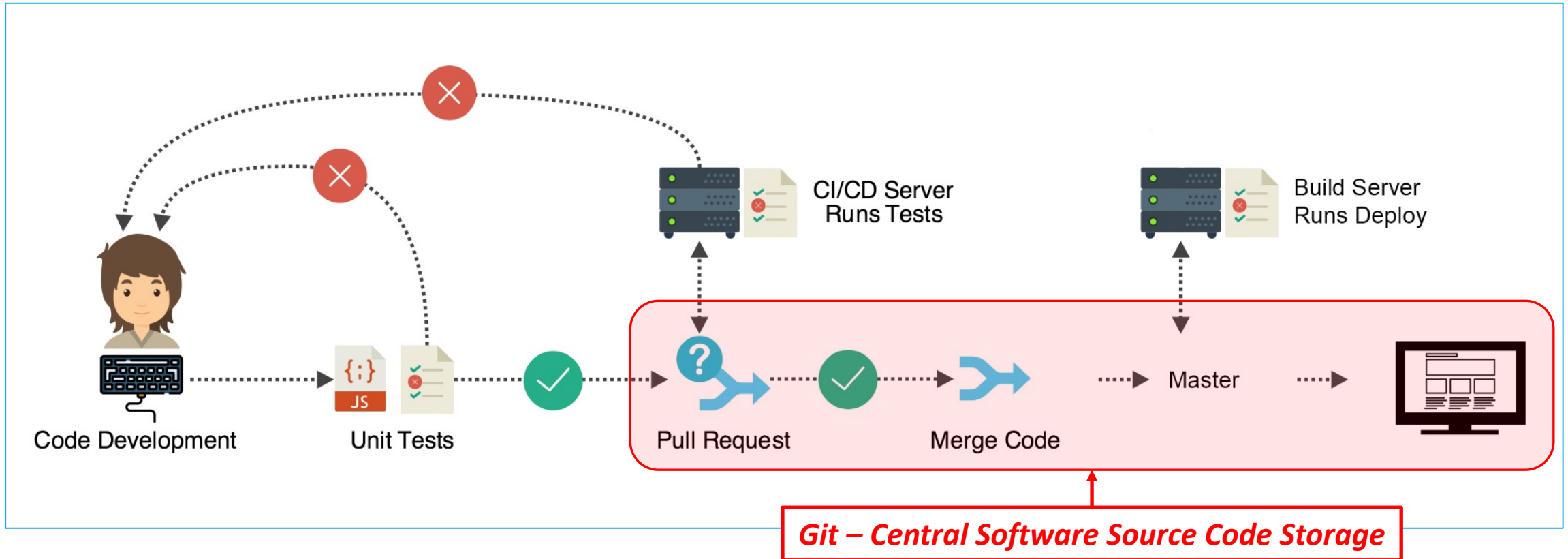
# CI Servers

---

- It is **helpful to do regression testing** if maintaining code.
- Want to do very regular (or “continuous”) testing of the complete code.
- To do this, use the “central store” of the source code normally held by a team.
- Often a dedicated manager of the CI server.
- CI servers sit on top of an existing repository (such as Git repository).
  - E.g., Jenkins: <https://www.jenkins.io/>



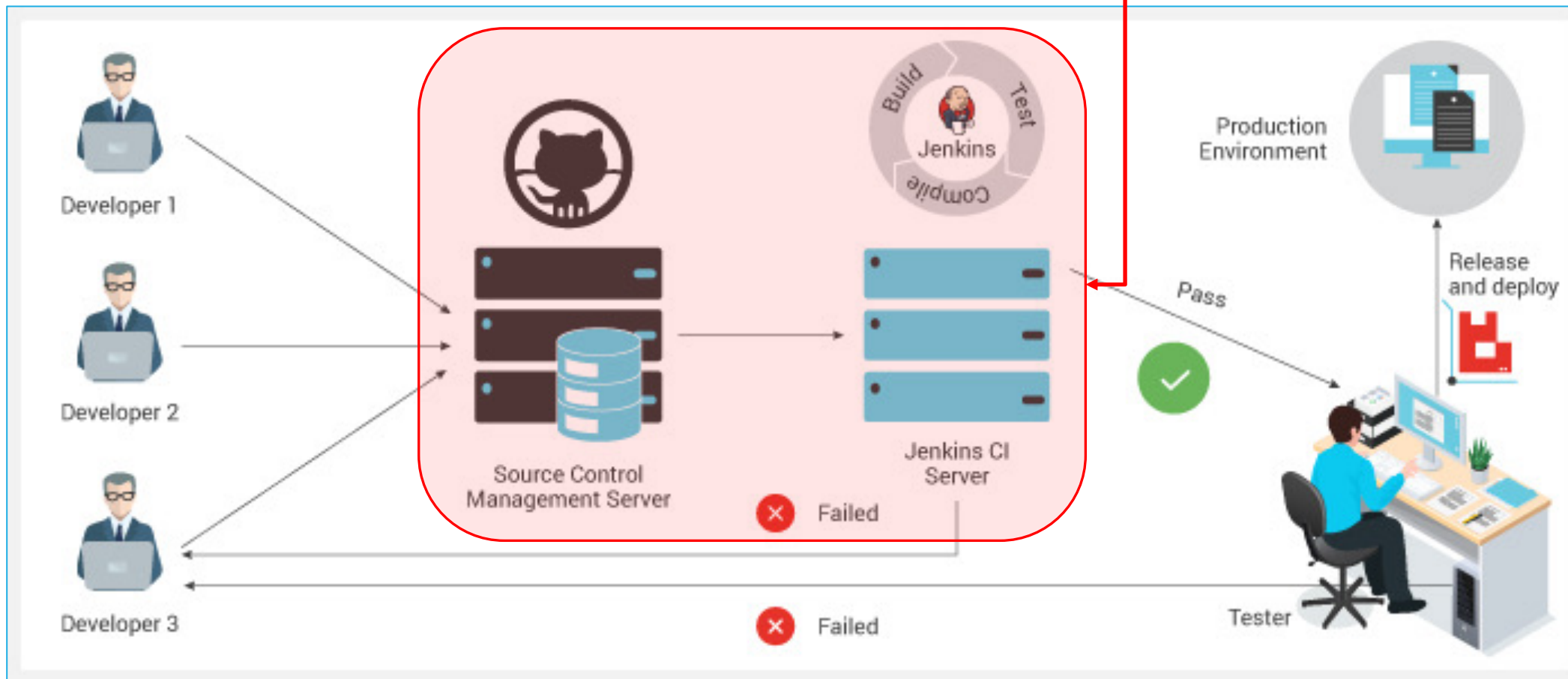
# CI Servers





# Jenkins CI Server

*Watch repository  
for changes*





# Minimising Risk with CI

---

- Common **practice** in software maintenance is to
  1. Branch a project.
  2. Develop and test a new feature.
  3. Attempt to integrate completed features into a project.
- Know from looking at regression testing that this is **a risky strategy** – why?
- CI is a “live” process of compiling the **whole project** using your test harness and build files **each time** you make a modification.
  - Not at the point where the new feature is completed!

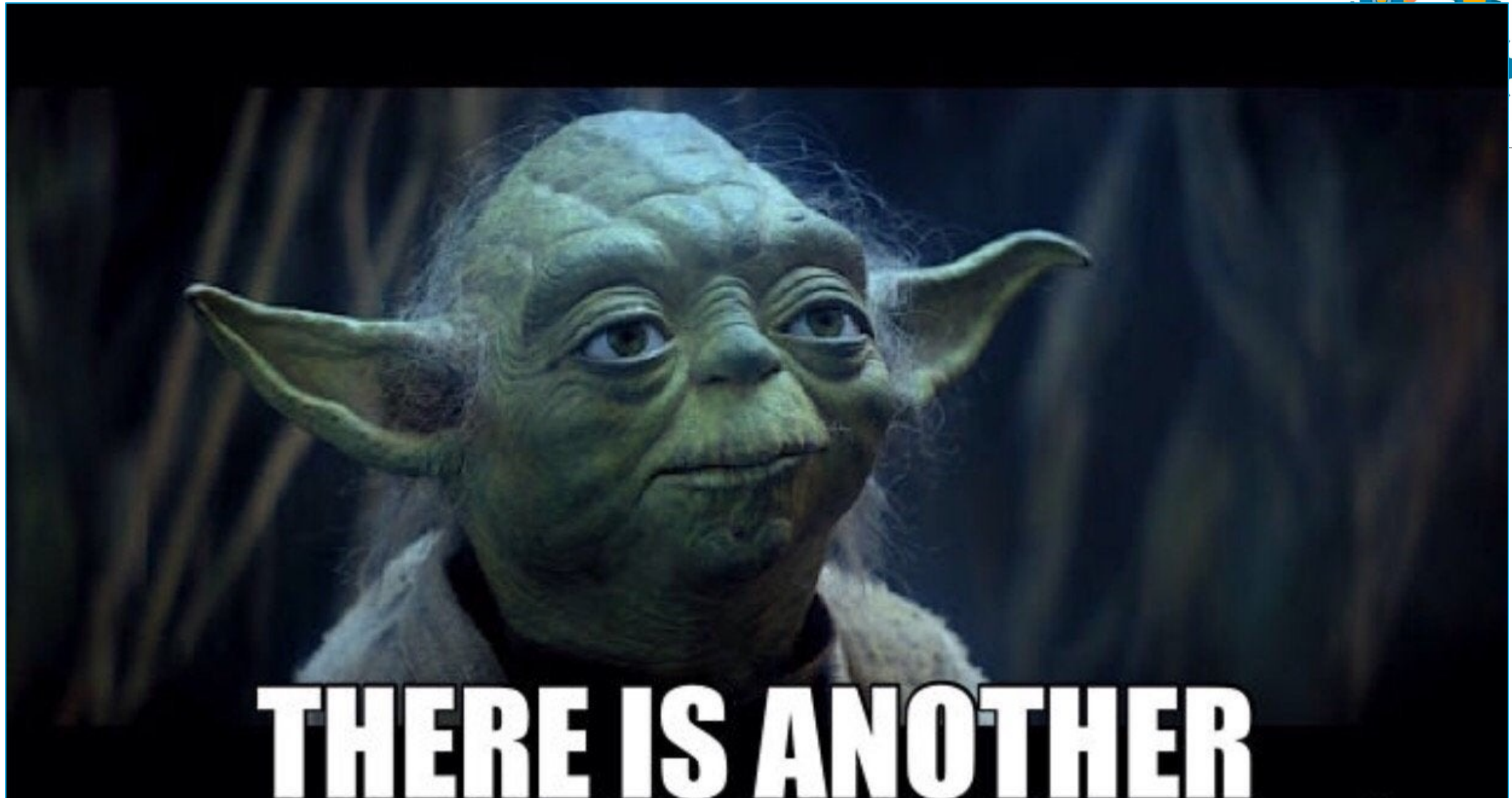




# Minimising Risk with CI

---

- Dev Leaders Compare Continuous Delivery vs. Continuous Deployment vs. Continuous Integration
  - <https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/>
- What is Continuous Integration?
  - <https://youtu.be/1er2cjUq1UI>
- CI Server Integration in IntelliJ (under development)
  - <https://www.jetbrains.com/help/space/ci-server-integration.html>
- Team City: Powerful CI by JetBrains
  - <https://www.jetbrains.com/teamcity/>
  - CI server integration with IntelliJ: <https://www.jetbrains.com/help/upsources/ci-server-integration.html>





# Property Based Testing: A New Hope

---

- A long time ago in A Haskell Galaxy, far far away.
- Wanted to prove properties of **basic functional** operation.
- Ensures that a program satisfied specified properties across a huge variety of input.
- Sometimes, can't think of every permutation of an input for unit testing.
- In TDD, it is possible to gain maximum coverage over all statements but not over all conditions and inputs.



# PBT Process

---

- E.g., Heart rate monitor software which checks the range of heart rate and sounds an alarm if the values are out of range.
  - With unit testing, it could be extremely difficult to check every single value in the range for this type of input.
  - Yes, it is a Safety Critical System and so this should be done.
- PBT specifies the outputs and the range of inputs:
  1. Define the **behaviors** that want to be tested.
  2. Define the **range of data** to check against each outcome.
  3. Generate test datasets which can be totally random to completely specified.
  4. Write and run the tests.
- The tool QuickCheck is used to assist developers in this task.
  - And its related tool smallCheck.

# QuickCheck for Property Based Testing



- PBT is almost synonymous with QuickCheck which is a popular tool.
- It started out life just for functional languages like Haskell and Erlang.
  - Has been commercially extended for use in other languages, also gaining popularity.
- Provides an elegant framework for supporting up to **brute force style** stress testing of program properties.
  - E.g., has a list which has been inverted and reinverted – returned the original list?
  - Do recursive functions return the correct output.
- Learning to use QuickCheck in our labs is unfortunately out of scope, but main idea is to let you be aware of its power and existence.

# QuickCheck



- QuickCheck
  - <https://hackage.haskell.org/package/QuickCheck>
  
- Code Checking Automation
  - <https://youtu.be/AfaNEebCDos>
  
- Property Based Testing
  - <https://youtu.be/7kB6JaSH9p8>



# Testing Summary

---

- Testing is a crucial part of both creating maintainable software and software maintenance.
- Testing must be an ongoing and integral part of software development, no matter what the scale of the development.
- Very basic techniques can be very effective.
- Unit testing can be scaffolded using JUnit for Java.
- Test Driven Development (TDD) is where JUnit tests are written before code functionality.
- QuickCheck and Property Based Testing (PBT) can be used to verify properties and test unthinkable amounts of scenarios.

# Put Your Mind in Maintenance Mode

