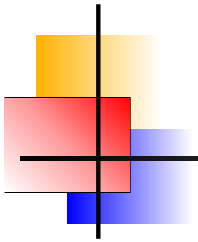


# Programming Project:
# Software Development Tools

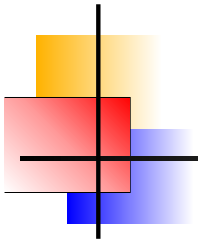
Guillermo Román Díez

groman@fi.upm.es

E.T.S. Ingenieros en Informática

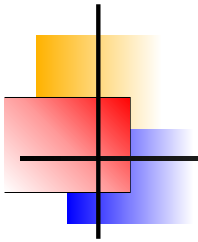
Universidad Politécnica de Madrid

2019-2020

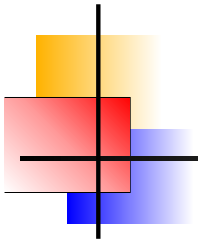


# Software development tools

- Software developers working in teams are continuously writing new source code and changing existing source code
  - Multiple developers work simultaneously with the same files (with some cautions)
  - All changes in the code must the traceable
  - To do so, programmers use **version control systems**
- Software development includes a sequence of actions that are performed repetitively: compile, test, deploy...
  - Doing this activities manually are a waste of time
  - The automation of this activities is crucial
  - To do so, programmers use **build tools**

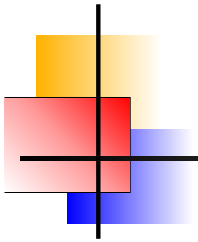


# Version Control Systems



# Version Control Systems

- **Version Control Systems (VCS)** are tools that records changes to a file, or set of files, over time so that you can recall specific versions later
  - Note that VCS are not only used for programming
- The use of version control systems has many advantages in software development
  - Allows multiple people working simultaneously in the same files
  - Is a repository that backups the history of the project, tracing who, when, why has done a change
  - Helps to solve possible *conflicts* in the content of the files/directories
  - Are useful to work in different projects with the same base code

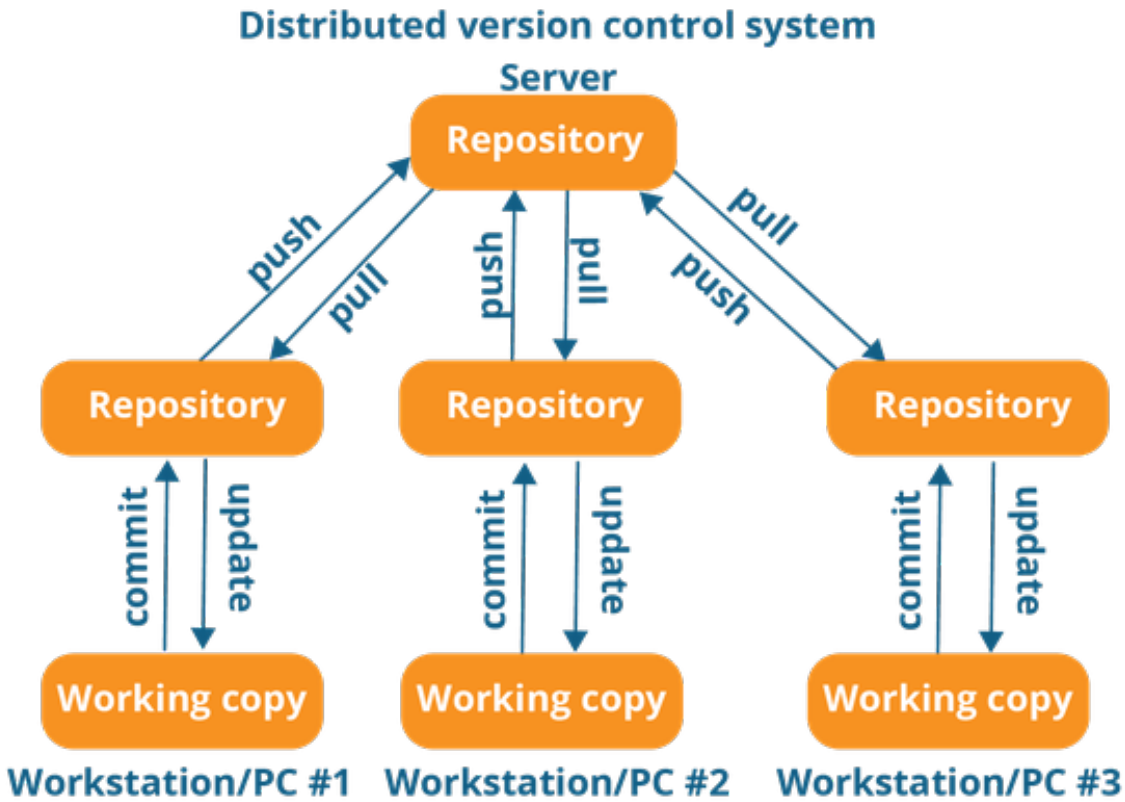


# Version Control Systems: Types

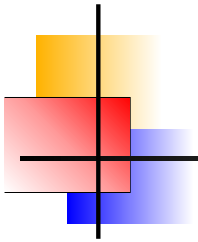
- We can find two different kinds of VCS
  - **Centralized**: CVS, Subversion, ...
    - Each repository has one centralized server which stores the history of the files/directories of the project
    - Users has its own local copy of the project
    - Users can commit changes to the server
    - Users can update their copy with the changes done by others
    - Most operations require connect with the server
  - **Distributed**: GIT, Mercurial, ...
    - Every user has their own repository that has the entire commit history of the project
    - Changes are recorded on your local repository
    - Changes can be pushed to a centralized repository
    - Many operations can be done without communicating with a central server

# Version Control Systems: GIT

- In this course we will use **GIT** [1]
- In GIT, every user has a *clone* of the whole repository
  - With full information of the history of the project

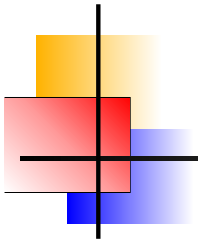



[1] `https://git-scm.com/`



# GIT COMMANDS

| | |
|---|---|
| `git clone [url]` | Clones a repository from `url` into a local directory |
| `git status` | Lists all new or modified files to be committed or staged |
| `git diff` | Shows file differences not staged yet |
| `git add [file]` | Snapshots the file in preparation for versioning and stages the file for commit |
| `git checkout [file]` | Update modified files to a version found in the repository |
| `git reset [file]` | Unstages the file, but preserve its contents |
| `git commit -m 'message'` | Records file permanently in local version history. Committed changes are *staged* |



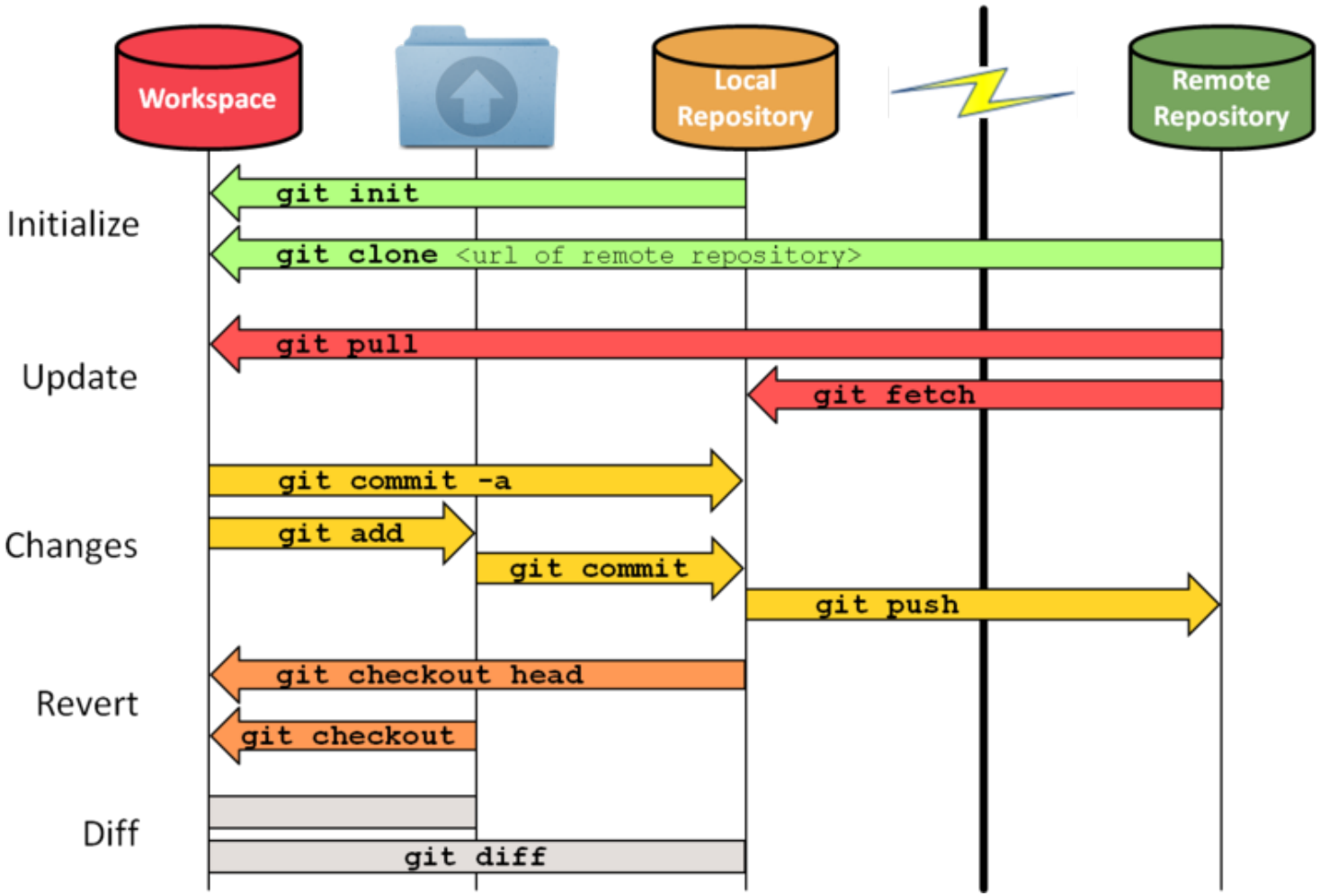
# GIT Commands

| | |
|---|---|
| `git show [rev:file]` | Show a commited revisions of file |
| `git rm [file]` | Deletes the file from the working directory and stages the deletion |
| `git mv [from] [to]` | Changes the file name and prepares it for commit |
| `git push` | Uploads all local branch commits (staged) to the remote server |
| `git pull` | Downloads and incorporates the changes to the local copy of the repo |

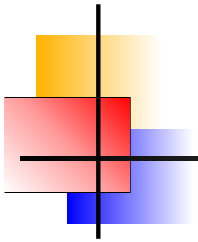
Note: Documentation and other GIT commands can be found at

`https://git-scm.com/docs`

# GIT COMMANDS GRAPHICALLY[2]



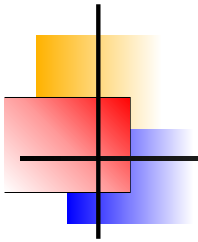

[2]Picture taken from:
https://sselab.de/lab2/public/wiki/sselab/index.php?title=Git



# GIT Good Practices

*Question*

Which files should be under version control?

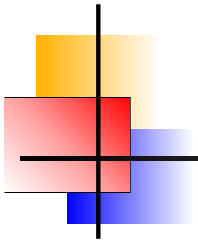


# Git Good Practices

*Question*

Which files should be under version control?

Source code, resources, build files, tests, ...



# Git Good Practices

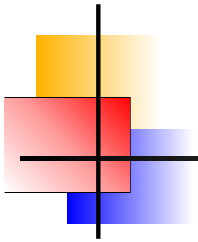
*Question*

Which files should be under version control?

Source code, resources, build files, tests, ...

*Question*

Which files should not be under version control?



# GIT Good Practices

*Question*

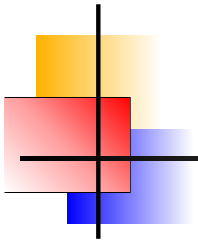
Which files should be under version control?

Source code, resources, build files, tests, ...

*Question*

Which files should not be under version control?

Automatically generated files: .class, tmp, javadoc, ...



# GIT Good Practices

*Question*

Which files should be under version control?

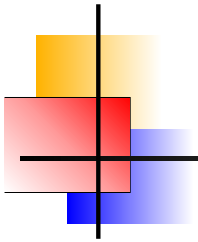
Source code, resources, build files, tests, ...

*Question*

Which files should not be under version control?

Automatically generated files: .class, tmp, javadoc, ...

- Never push a version of the source code that do not compile
- Avoid pushing a version of the source code that do not pass the tests

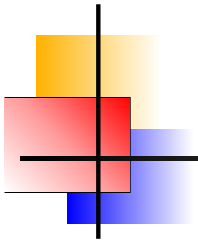


# GitLab Server

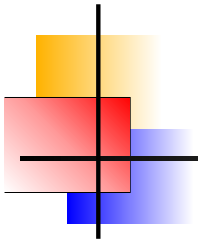
- There are different possibilities for hosting GIT repositories
  - Have your own server and access it via `ssh` or `https`
  - Use web applications like GitHub[3] or GitLab[4]
- `GitLab` has a powerful web interface including a lot features useful for software development
  - Users management
  - Consult the repository files and history
  - Show the documentation of the project
  - Send notifications by email with the changes in the repository
  - Issues and milestones managing
  - Continuous Integration
- In the course, we work with our **own installation of GitLab**

`http://costa.ls.fi.upm.es/gitlab`

[3] https://www.github.com
[4] https://www.gitlab.com

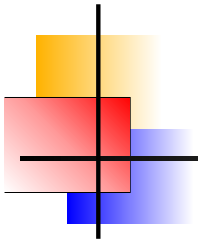


# Build Tools



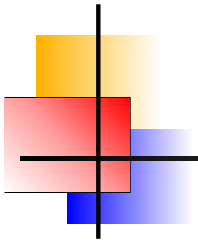
# Build Tools: Why use them

- Programming time is full of highly **repetitive** actions
  - Compile the code, run the tests, deploy the applications, generate the documentation, ...
  - This activities takes a significant percentage of the programming time
  - Build tools are used for **automate** these activities for optimizing the programming time
- Repetitive **tasks** follow a **concrete order**
  - We cannot run the application without compiling it
  - We should not prepare the distribution of the application without compiling and testing it
  - Build tools forces us to follow this order in a strict way



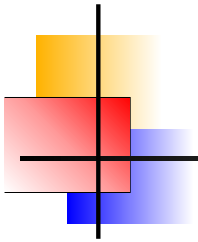
# Build Tools: Why use them

- Most projects use **external libraries**
  - These libraries are not always easy to include in the project (it depends on the IDE, command line, ...)
  - Programmers could be using different non-compatible versions of the libraries
  - The use of a *dependency system* significantly reduces the number of conflicts produced by the libraries used
- **Prepare the code** for compiling and testing takes time and it is an error-prone task
  - The package and deployment of the application is done in multiple environments (development, pre-production, production, QA,...)
  - Package and deploy the application should not be done manually to avoid inconsistencies is the process
  - Build tools can do these activities in an automatic way
- Build tools are also used by **Continuous Integration tools** to automate the integration



# Build Tools: Some tools

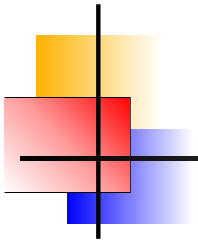
- **Make** is the most popular in UNIX environments
  - Perfectly fits with C and C++ programs
  - Is used in Linux packages for compiling and installing them using the source code
- **Ant** was very popular some years ago for Java programs
  - The configuration file is in `XML` format
  - It does not have a dependency system and requires manually define the dependencies between tasks
- **Maven** is currently the most used for Java projects
  - The configuration file is also in `XML` format
  - Includes a very powerful dependencies systems
- **Gradle** is winning popularity
  - The configuration file is in a specific format (Groovy based)
  - Builds the project in an incremental way



# Build Tools: Maven

- **Apache Maven**[5] is a tool for building and managing Java-based projects
- Allows a project to build using its **project object model (POM)** in XML format which includes:
  - How to package the project (.jar, .war, ...)
  - Version information
  - The dependencies (libraries) used in the project
- Includes a dependency manager that includes automatic updating and dependency closures (also known as transitive dependencies)
  - Automatically downloads the files and used them for compiling, testing, ...
- Is supported by the most popular Java IDE's (Eclipse, IntelliJ IDEA, ...)

[5] https://maven.apache.org/



# Maven: Creating a project

- To create a project some values need to be set
- `groupId`: It helps to identify your project uniquely across all projects
  - It follows the package name conventions

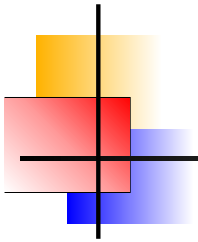
```
es.upm.pproject
```

- `artifactId`: Is the *identifier* of the project
  - It will be the name of the distribution file
  - The artifactId is added to the base package name

```
myproject
```

- `version`: To identify the different versions of the project
  - This value will change along the project evolution
- The base package name for your application will be:

```
es.upm.pproject.myproject
```

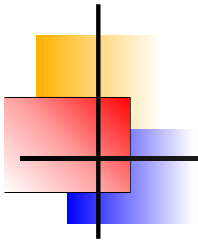


# Maven: Creating a project

- A project can be created by using the command line

```
mvn -B archetype:generate
    -DarchetypeGroupId=org.apache.maven.archetypes
    -DgroupId=MYGROUPID
    -DartifactId=ARTIFACTID
```

- Or we can use Eclipse
  - File / New / Maven (Maven Project)

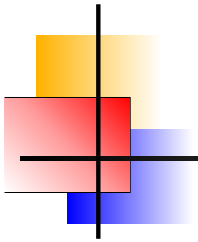


# Maven: Directories layout

- A Maven (Java) project has the following layout

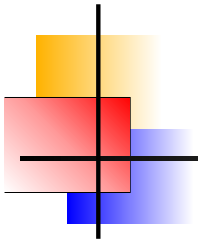
```
src/
  main/
    java/PACKAGE/      - Source code folder
      App.java
    resources/         - Application resources
  test/
    java/PACKAGE/      - Test files (JUnit)
      AppTest.java
    resources/         - Testing resources
target/                - Output of the build
pom.xml
```

- All application files are in `main/java/PACKAGE`
- The test files are in `test/java/PACKAGE`
- The `target` directory is used to house all output of the build



# Maven: POM structure

```
<project ... >
  <modelVersion>4.0.0</modelVersion>          - POM version
  <url>http://maven.apache.org</url>
  <groupId>MYGROUPID</groupId>                - Package
  <artifactId>ARTIFACTID</artifactId>         - App Identifier
  <packaging>jar</packaging>                  - Dist file format
  <version>1.0-SNAPSHOT</version>             - Version
  <name>My pretty app</name>                  - Readable App Name
  <dependencies>                              - Libraries used
    <dependency>
      <groupId>junit</groupId>                - For testing
      <artifactId>junit</artifactId>
      <version>5.20</version>                 - Library Version
      <scope>test</scope>                     - When it is used
    </dependency>
    <!-- More dependencies... -->
  </dependencies>
</project>
```



# Maven: Basic Maven commands

```
mvn COMMAND
```

where `COMMAND` can be:

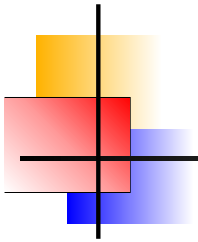
*validate* - validate the project is correct and all necessary information is available

*compile* - compile the source code of the project

*test* - test the compiled source code using a suitable unit testing framework (e.g. JUnit)

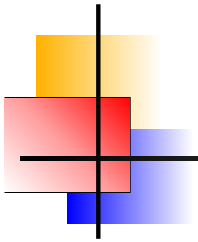
*package* - take the compiled code and package it in its distributable format (e.g. JAR file)

*clean* - Removes the `target` directory



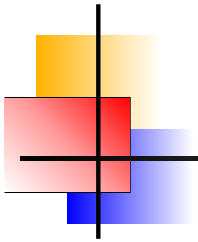
# Creating a Maven Project + GitLab Repo

1. Create a Maven project



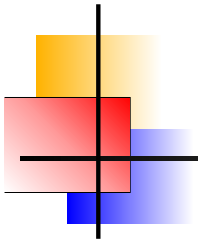
# Creating a Maven Project + GitLab Repo

1. Create a Maven project
2. Create an empty GitLab project and copy the URL of the project



# Creating a Maven Project + GitLab Repo

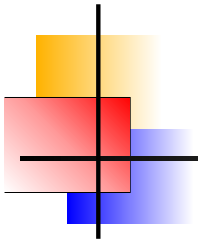
1. Create a Maven project
2. Create an empty GitLab project and copy the URL of the project
3. Enter in the project directory



# Creating a Maven Project + GitLab Repo

1. Create a Maven project
2. Create an empty GitLab project and copy the URL of the project
3. Enter in the project directory
4. Initialize a Git repository and set as remote the URL of the project

```
git init
git remote add origin [URL]
```



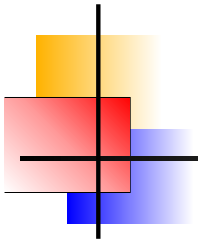
# Creating a Maven Project + GitLab Repo

1. Create a Maven project
2. Create an empty GitLab project and copy the URL of the project
3. Enter in the project directory
4. Initialize a Git repository and set as remote the URL of the project

```
git init
git remote add origin [URL]
```

5. Connect the repo to the remote server and push the files

```
git add .
git commit -m "Initial commit"
git push -u origin master
```



# Creating a Maven Project + GitLab Repo

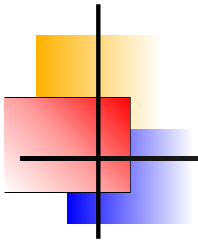
1. Create a Maven project
2. Create an empty GitLab project and copy the URL of the project
3. Enter in the project directory
4. Initialize a Git repository and set as remote the URL of the project

```
git init
git remote add origin [URL]
```

5. Connect the repo to the remote server and push the files

```
git add .
git commit -m "Initial commit"
git push -u origin master
```

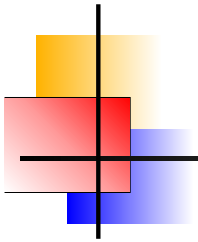
6. Create a `.gitignore` file with content `target` to ignore automatically generated files



# Development tools

- Tools will help during the software development process and its use is widely extended in software companies
- In the subject the use of these tools is compulsory. Your projects must use:
  - **GIT** as version control system
  - **Maven** as build tool
- These tools can be used in *Eclipse*[6]
  - Recent versions of Eclipse include a plugin to work with GIT
  - A Maven Project can be *imported* in Eclipse (File, Import, Existing Maven Project)

[6] `http://www.eclipse.org`



# Further Reading

- Documentation about GIT can be found at:

  `https://git-scm.com/`

- The official book of GIT can be downloaded here:

  `https://git-scm.com/book/en/v2`

- Documentation about Maven can be found at:

  `https://maven.apache.org/`