Best Practice in Developing Java Applications

Package Structure And Naming Conventions. Collaborative Working With Git

Dott. Marco Solinas, PhD

03 Nov 2021

Agenda

Package Structure and Naming Conventions

- Why Should We Care?
- Package Naming Conventions
- Package Structure

Collaborative Working with Git

- What is Git
- Git: Remote, Local, Staging and Stash
- How to start working with github
- Working with Branches
- Pull Requests & Code Review

Package Structure and Naming Conventions

Package Structure and Naming Conventions

Why Should We Care?

Following packaging conventions helps us to improve:

- Readability
- Maintainability
- Scalability

Package Naming Conventions

Being Java a worldwide used language, different developers may use same names for different types. To avoid this:

- Package names in lower case
  - Avid conflicts with class/interface names

```
package figure
public class Figure {…}
```

- Companies use their reverse domain name as package prefix
  - Conflicts within a given company must be addressed with internal conventions
  - Legalizing: if the internet domain name is not valid for packaging, the convention is to use underscores

```
www.iongroup.com  →  com.iongroup.…
www.dii.unipi.it  →  it.unipi.dii.…
```

Domain Name	Package Name Prefix
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

- Packages in the Java language itself begin with `java.` or `javax.`

Source : https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

Package Naming Conventions

A few examples from ION experience:

- [rev internet domain].[division].[lib name]. …
  - com.iongroup.client_tech.cache_lib. …
- [rev internet domain].[solution].[app name]. …
  - com.iongroup.sdp.clientgateway. …
- [rev internet domain].[framework].[module name]. …
  - com.iongroup.ifs.talk. …

Reference package for this seminar examples:

`it.unipi.dii.inginf.lsdb.…`

Package Structure

Applications can be structured by organizing packages:

- By Features

```
it.unipi.dii.inginf.lsdb.library
   .admin
   .login
   .registration
   .user
   .reserve
   .search
   ....
```

- By Layers (and/or By Type)

```
it.unipi.dii.inginf.lsdb.library
   .bean
   .cache
   .config
   .persistence
   .factory
   .model
   .service
      .local
      .remote
   ....
```

Package Structure and Naming Conventions
Package By Features (1/2)

All items related to a single feature into a single package:

- High modularity, minimal coupling
  - A good test: what happens if an entire feature must be deleted?
- Easy to read and understand how features are implemented
  - Also, relations among different features becomes easier to identify
- Package scope should be exploited
  - If a given object is used only within a package, why should it be visible outside it?
- Does NOT mean a given package cannot use items defined in other packages
  - Increase the scope of an item if it is required outside the package it belongs to
  - Extract API if necessary
  - There are programmers who prefer code duplication – consider the rule of three

Package Structure and Naming Conventions

Package By Features (2/2)

All items related to a single feature into a single package:

- At the beginning, it might be difficult to decide which are the main features
- Might be difficult to decide where a new piece of code should be placed
- Package size can grow quickly
  - Consider the extraction of classes sharing the same ‘concern’ (sub features)
- Pay attention to package circular dependencies
  - Not an actual problem per-se, but can decrease the readability of the code base

Package Structure and Naming Conventions

Package By Layers (1/2)

All items related to a single layer into a single package:

- Items doing the same type of job are placed all together
- This is the more ‘natural’ way to organize applications
- Easy to change technology for a single layer
  - Old and new can even coexist (in different packages) while the refactoring is in progress
- Easy to build reusable framework/libraries
  - Identify the commonalities among different items of the same layer/type and extract an abstract framework for that

Package Structure and Naming Conventions

Package By Layers (2/2)

All items related to a single layer into a single package:

- Difficult to understand the implementation of a given feature
  - Tendency to generic, reused and complex code, which is hard to understand,
    and changes can easily break other use cases as the impact of a change is hard to understand
- Low modularity, high package coupling
- Editing a feature will require changes across many packages
  - What about the deletion of an entire feature?
- Difficult to reduce the scope of objects that are supposed to be used in different packages
  - Although, if you can separate API from implementation…

Package Structure and Naming Conventions

Package By Features Then By Layers

Mix-up the two techniques:

- Each feature package is internally organized by layers/types
  - Again: separate API from implementation whenever convenient/possible

- Use your brain: do not stick with a solution if you see you took the wrong path
  - Maybe breaking the initial design can help you to write better code

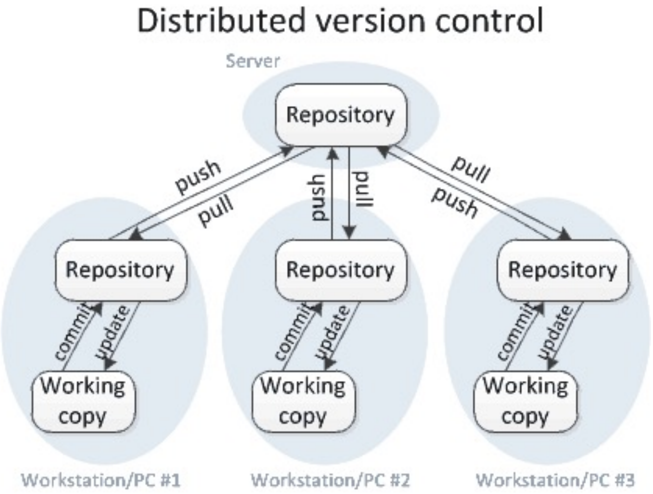
Collaborative Working with Git

Collaborative Working with Git

What is Git

Git is a software that implements a distributed version control system meant to be used for software development:

- Version control systems
  - Software tools that help a software team manage changes to source code over time
  - Main benefits are:
    - History of all changes to every single file
    - Branching (support for non-linear development)
    - Traceability
- Distributed
  - Each developer has a local copy of the full history
- Many other technical advantages…



Collaborative Working with Git

Git: Remote, Local, Staging and Stash

Git has many layers which you can think of as “saving areas”:

- Working directory
  - The actual copy you have access to edit
- Staging area

```
$ git add MyClassFile.java AnotherClass.java
$ git add .
```

  - A sort of buffer of your changes ready for commit
- Local repository

```
$ git commit -m “A commit message”
```

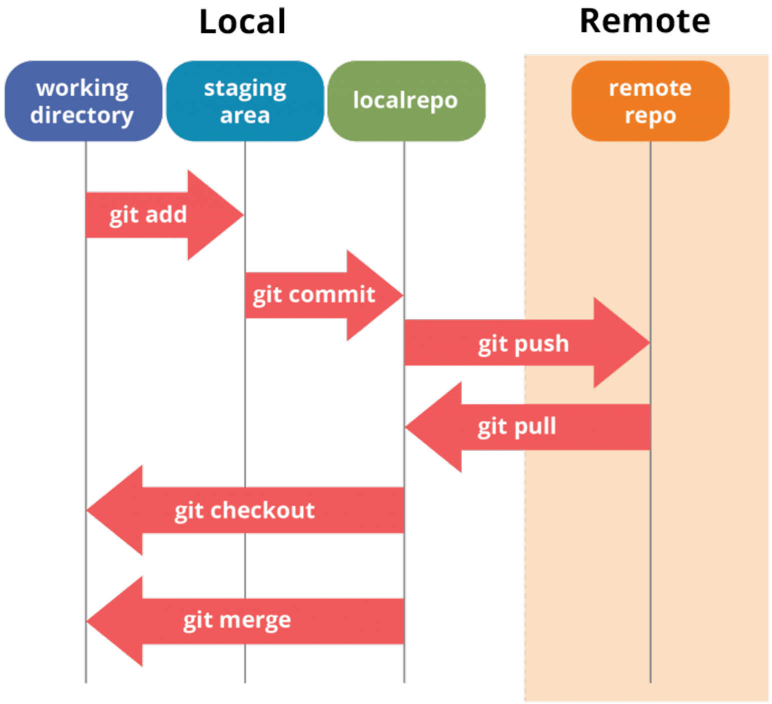
  - A full copy of what is available in the remote
  - Includes all commits and branches created until the last pull
- Remote repository

```
$ git push
```

  - The centralized copy of your project, hosted in a server
- Stash

```
$ git stash
$ git stash pop
```

  - A parallel buffer you can use to temporary save your changes to the working copy
  - Useful when you need to perform operations like merge/rebase from another copy
  - Works like a stack (push and pop commands are available)



Collaborative Working with Git

How to start working with github

Github is a hosting service for software projects based on git.

- **In this scenario**, we assume you already have a project, but it is not hosted

1. Create a new repository in github directly from your profile
2. Open your git bash, and cd into the working directory of you project. Initialize it as a local git repository:

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (master)
$ git init
```

3. Add the files in your new local repository:

```
$ git add .
```

Then commit your changes:

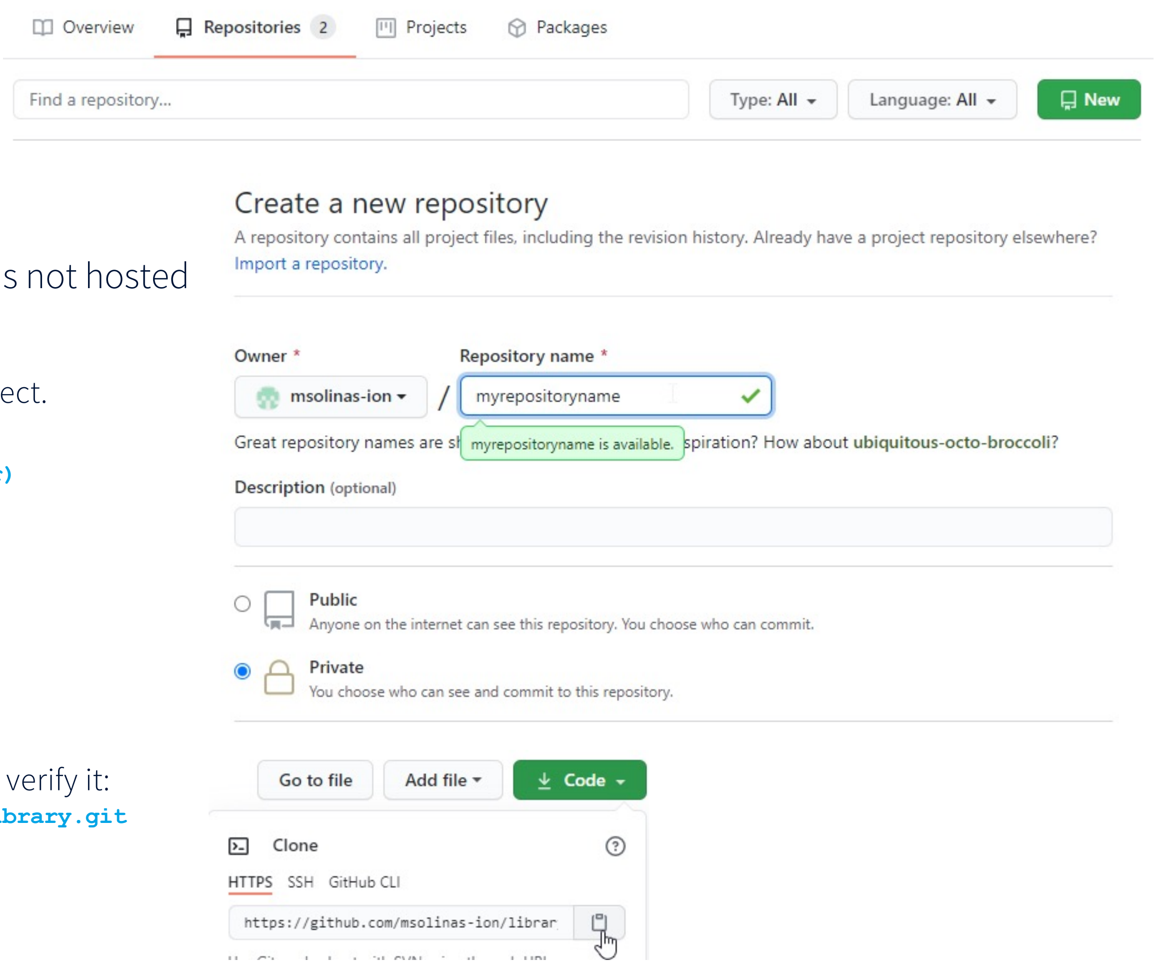
```
$ git commit –m “First upload to github”
```

4. From your github’s repository page, copy the URL for cloning
5. From your git bash, set the remote for your local repository, then verify it:

```
$ git remote add origin https://github.com/msolinas-ion/library.git
$ git remote –v
```

6. Last step, push everything to the remote:

```
$ git push origin master
```



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * msolinas-ion / Repository name * myrepositoryname ✓

Great repository names are sh myrepositoryname is available. spiration? How about **ubiquitous-octo-broccoli**?

Description (optional)

○ Public
Anyone on the internet can see this repository. You choose who can commit.

● Private
You choose who can see and commit to this repository.

https://github.com/msolinas-ion/librar

Source : https://docs.github.com/en/free-pro-team@latest/github/importing-your-projects-to-github/adding-an-existing-project-to-github-using-the-command-line

Collaborative Working with Git

Working with Feature Branches (1/3)

You can think to a branch as a “parallel” code thread that is created from another code thread (typically the master) and has his own commits:

- The checkout command is used with –b option to create a new branch:

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (master)
$ git checkout -b feat/branch1
```

  and alone to switch from an existing branch to another one:

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (feat/branch1)
$ git checkout master
```

- After you create a new branch, you must push it to the remote server

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (feat/branch1)
$ git push --set-upstream origin feat/branch1
```

  at this point, the new branch is an exact copy of the master (local copy!!!)

- A colleague pushed new commits to the master but I’m working on a branch… what should I do? In this case, you have to pull the latest master, then come back to your branch:

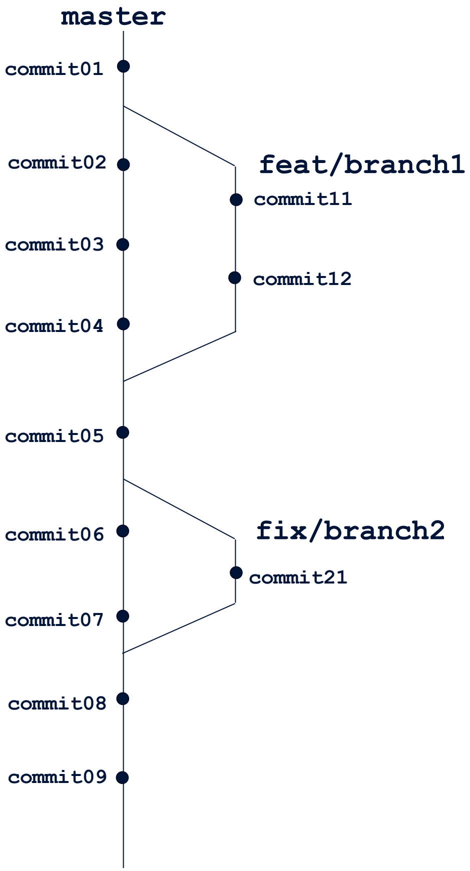
```
$ git checkout master
$ git pull
$ git checkout feat/branch1
```

  then, you can either manually port all those commits with a cherry-pick command:

```
$ git cherry-pick commit02
```

  or you can rebase the history of your branch with respect to latest master:

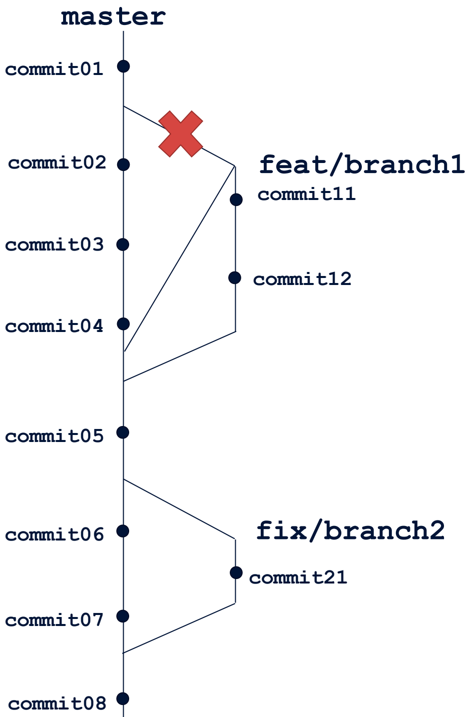
```
$ git rebase master
```



Collaborative Working with Git

Working with Feature Branches (2/3)

You can think to a branch as a “parallel” code thread that is created from another code thread (typically the master) and has his own commits:



- The rebase operation
  - Destroys the local copy your branch
  - Creates a new one from the specified branch (the master, in our example)
  - Applies all your existing commits in the same order you committed them.
  - The result will be equivalent to having checked-out the branch from latest master.
  - After a rebase, your remote copy is no longer valid. You must force a push to update the remote:

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (feat/branch1)
$ git push -f
```

- Mind the conflicts!!!
  - The mergetool option can help:

```
marco.solinas@PISA122 MINGW64 ~/workspace4/library (feat/branch1|REBASE 1/2)
$ git mergetool
```

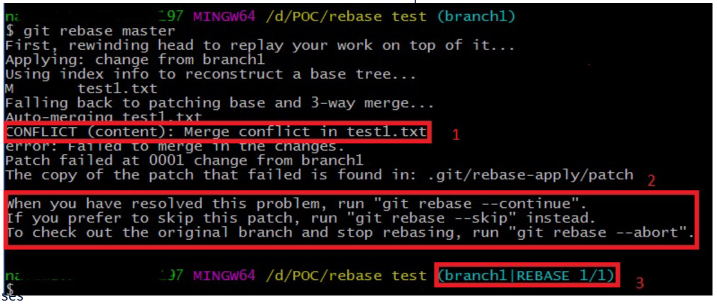
  You have one rebase step for each commits of you branch

  - Once you solved all conflicts of a step, you must tell git to continue with next commit:

```
$ git rebase --continue
```

  - In case you messed-up while resolving conflicts during rebase, you can abort at any time

```
$ git rebase --abort
```



```
na[redacted]197 MINGW64 /d/POC/rebase test (branch1)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: change from branch1
Using index info to reconstruct a base tree...
M       test1.txt
Falling back to patching base and 3-way merge...
Auto-merging test1.txt
CONFLICT (content): Merge conflict in test1.txt
error: Failed to merge in the changes.
Patch failed at 0001 change from branch1
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".

na[redacted]197 MINGW64 /d/POC/rebase test (branch1|REBASE 1/1)
$
```

Collaborative Working with Git

Working with Feature Branches (3/3)

Why this approach is important

- You should always checkout your private branch where you can **commit & push** as often as needed without interfering with other developers
  - Other developers won’t see anything of your activity, even if you push to the remote
  - You can work in isolation, so take your time to extensively test all your changes
  - Whatever is pushed to the remote copy of your branch, won’t be lost (unless the server dies…)
  - Do a rebase from time to time
- You should establish a sort of naming convention for your branches, a widely adopted practice is to use prefixes, e.g.:
  - feat/ OR feature/ when you checkout a branch for implementing a new feature, or extending an existing one

```
$ git checkout -b feat/delete_user_from_admin_ui
```

  - fix/ when your branch is created to resolve a bug in your application

```
$ git checkout -b fix/issue_on_book_loan_when_user_is_admin
```

  - chore/ when you’re doing operations like refactoring or re-engineering

```
$ git checkout -b chore/required_changes_to_support_java11
```

  - release/ if it is a release branch (not exactly a feature branch… but worth to mention ☺)

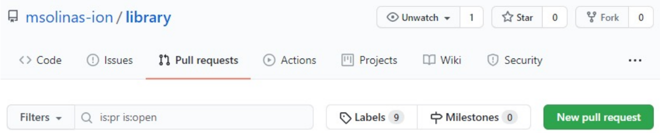
```
$ git checkout -b release/229
```

Collaborative Working with Git

Pull Requests & Code Review

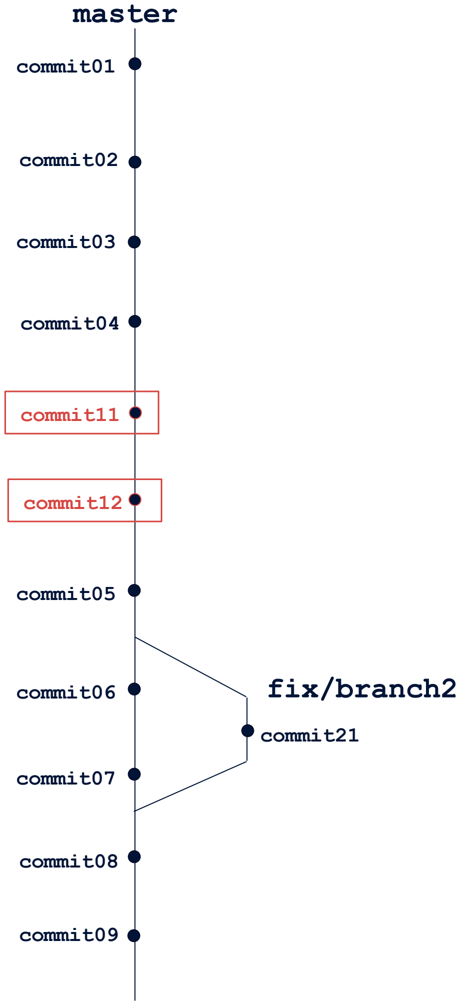
Once you're done with the changes on your feature branch, they can be merged into master:

- From github, you can open a Pull Request (aka Merge Request):



  - Essentially, you ask your colleagues to review your changes before you port them on master
  - Colleagues will review what you've done, and may ask you to do changes before you merge
  - Once you addressed all the comments of your reviewers, you are allowed to merge!
- This is an important step of the development process
  - Your colleagues might be senior devs suggesting you interesting improvements… learn from their comments!
  - Reviewers might see bugs in advance… this will save time later!
  - Reviewers can learn from what you've done… explain them the idea behind your coding choices!
- After a branch is merged in master, you will see your commits directly in master
  - You can delete the branch… do not accumulate branches!

```
$ git branch -D feat/branch1
```



Time for Hands-On!