Lecture 05

ECE 1145: Software Construction and Evolution

Branching

Build Management (CH 6)

Project Tips

Announcements

- Iteration 1 due Sept. 19
  - Utility.java, TestIterators.java
  - http://hamcrest.org/JavaHamcrest/tutorial
- Iteration 2 will be posted next week, due Sept. 26
  - Judge how much work to leave for Iteration 2 on AlphaCiv
- Resources
  - Branching models
    - https://nvie.com/posts/a-successful-git-branching-model/
    - https://guides.github.com/introduction/flow/
- Engineering Virtual Career Fair Wed. Sept. 29, register via Handshake for appointments from 12 – 4pm
  - Resume events: https://www.studentaffairs.pitt.edu/cdpa/events/

Questions for Today

How do we track and maintain software releases?

How do we ensure repeatable software builds?

Branching

Suppose we release a version of our PayStation to AlphaTown.

Then, we start rewriting part of the AlphaTown code to support BetaTown.

We're in the middle of major refactoring when…

AlphaTown calls to report a serious bug, they need a fix **yesterday**!

But the code base is currently broken and in no state to be released!

What can we do?

- Panic?

Branching

Suppose we release a version of our PayStation to AlphaTown.

Then, we start rewriting part of the AlphaTown code to support BetaTown.

We're in the middle of major refactoring when…

AlphaTown calls to report a serious bug, they need a fix **yesterday**!

But the code base is currently broken and in no state to be released!

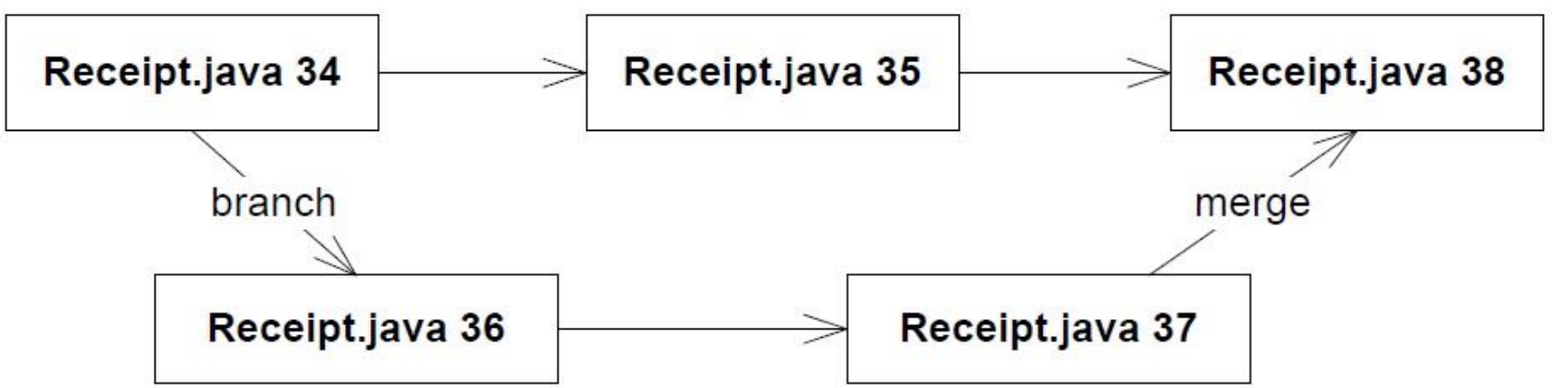
What can we do? → Avoid this with version control!

- Panic?

Branching

Definition: Branch

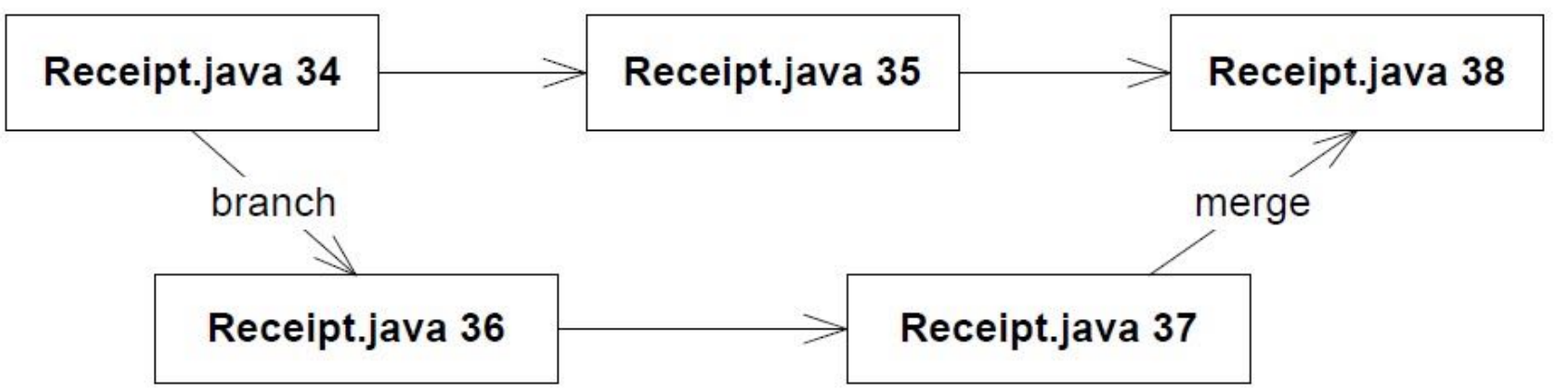
A branch is a point in the version graph where a version is ancestor to two or more descendant versions.



Branching

Definition: Branch

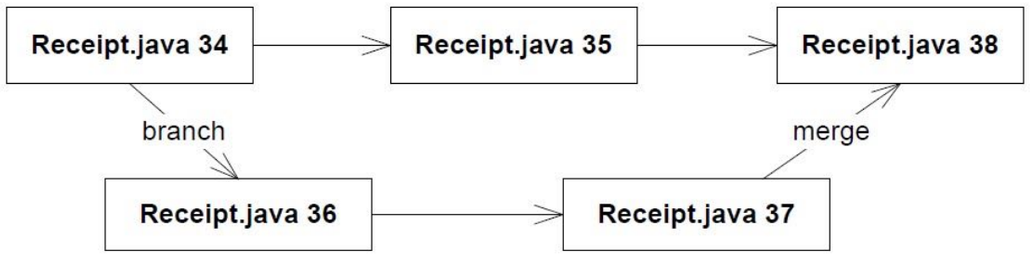
A branch is a point in the version graph where a version is ancestor to two or more descendant versions.



Git uses SHA-1 hashes for version/commit IDs

→ Why not sequential numbering?

Branching



git checkout -b <new branch name>

- Create and checkout a new branch off of the current one
- Any changes and commits will be on the new branch

git merge <branch name>

- Merge all commits from the specified branch into the current branch

git checkout <branch name>

- Checkout the specified branch (when branch already exists)

git branch -a

- Show all branches *including* all on the origin that you do not have currently

Branching

Some versions have a special meaning, e.g., **Release**

How can we manage releases?

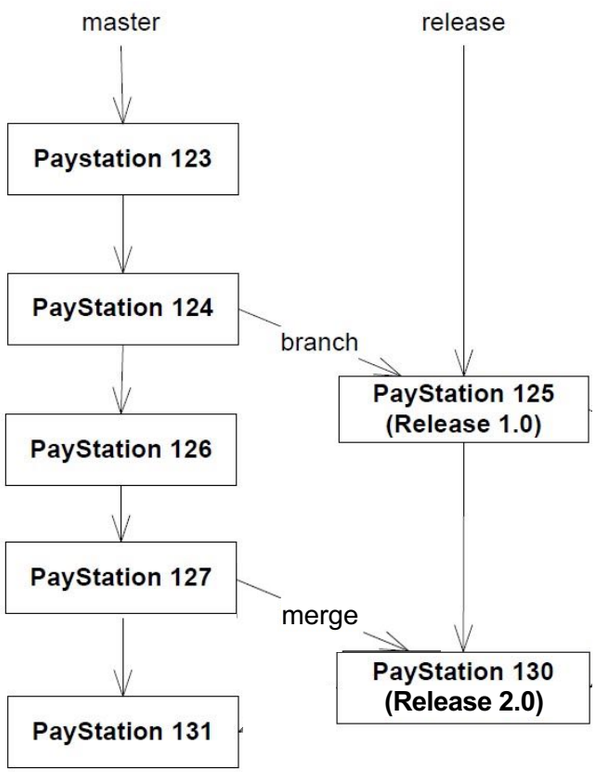
- ‘Tag’ a commit with a release number (e.g., “v1.0”)
- Make a new branch for the release
  - Name it ‘Release-AlphaTown-V1.7.4’
- Merge into a dedicated ‘release’ branch and tag the commit

Releases should always reside on a branch! **Why?**

Branching Models: Single Release Branch

Single Release Branch

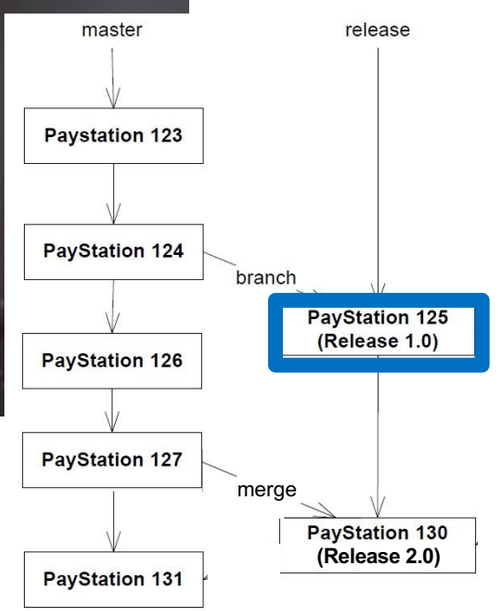
- Dedicated branch for releases
- Hotfixing must be done on separate branch and merged back



Branching Models: Single Release Branch

Release 1: All releases are on the release branch; use ‘tag’ to id it

```
csdev@m31:~/proj/single-release-branch$ git branch release
csdev@m31:~/proj/single-release-branch$ git checkout release
Switched to branch 'release'
csdev@m31:~/proj/single-release-branch$ git tag -a Release1.0 -m "Release1.0 for AlphaTown"
csdev@m31:~/proj/single-release-branch$ git tag
Release1.0
csdev@m31:~/proj/single-release-branch$ git show Release1.0
tag Release1.0
Tagger: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:56:39 2017 +0200

Release1.0 for AlphaTown

commit 6a3c5cb9c20db46a761794cf38d65e1a136e905f
Author: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:53:12 2017 +0200

    Cancel feature implemented.
```

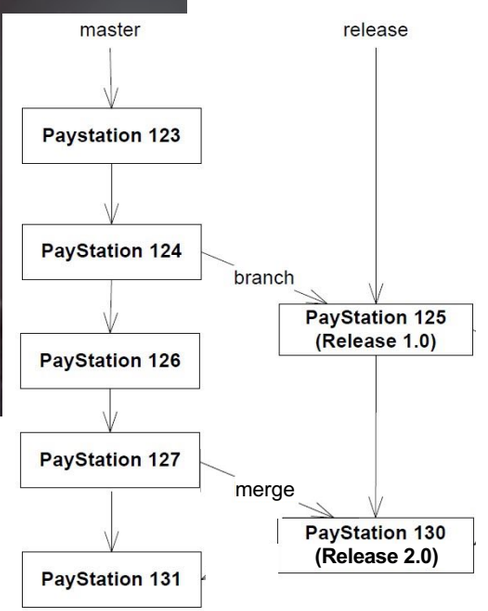


Branching Models: Single Release Branch

Release 1: All releases are on the release branch; use ‘tag’ to id it

git checkout –b release

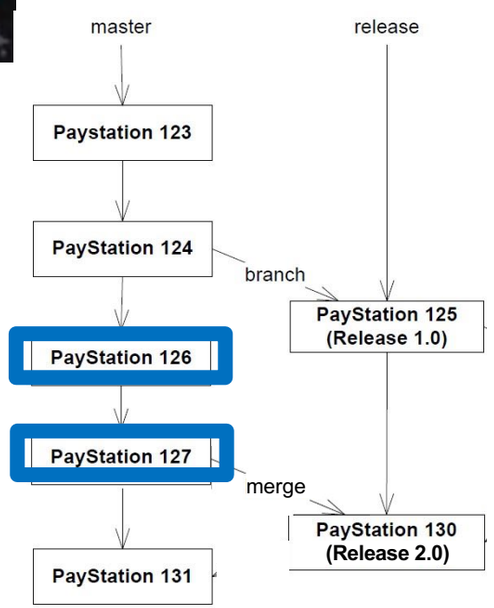
```
csdev@m31:~/proj/single-release-branch$ git branch release
csdev@m31:~/proj/single-release-branch$ git checkout release
Switched to branch 'release'
csdev@m31:~/proj/single-release-branch$ git tag -a Release1.0 -m "Release1.0 for AlphaTown"
csdev@m31:~/proj/single-release-branch$ git tag
Release1.0
csdev@m31:~/proj/single-release-branch$ git show Release1.0
tag Release1.0
Tagger: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:56:39 2017 +0200

Release1.0 for AlphaTown

commit 6a3c5cb9c20db46a761794cf38d65e1a136e905f
Author: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:53:12 2017 +0200

    Cancel feature implemented.
```



Branching Models: Single Release Branch

Refactor architecture to support BetaTown

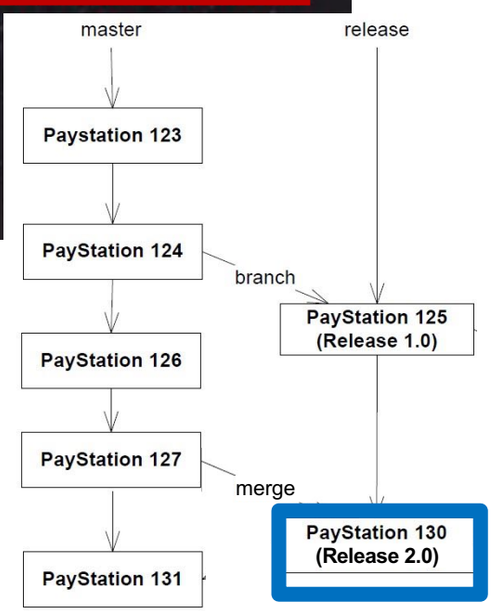
```
csdev@m31:~/proj/single-release-branch$ git checkout master
Switched to branch 'master'
csdev@m31:~/proj/single-release-branch$ git commit -a -m "major refactoring"
[master 4001ece] major refactoring
 1 file changed, 2 insertions(+)
csdev@m31:~/proj/single-release-branch$ git commit -a -m "BetaTown now working"
[master 02c58f8] BetaTown now working
```



Branching Models: Single Release Branch

Release 2: Checkout the release branch; merge master into it; tag

```
csdev@m31: ~/proj/single-release-branch
File  Edit  Tabs  Help
csdev@m31:~/proj/single-release-branch$ git checkout release
Switched to branch 'release'
csdev@m31:~/proj/single-release-branch$ git merge master
Updating 6a3c5cb..02c58f8
Fast-forward
 PayStation.java | 4 ++++
 1 file changed, 4 insertions(+)
csdev@m31:~/proj/single-release-branch$ git tag -a Release2.0 -m "Release2.0 for BetaTown and AlphaTown"
csdev@m31:~/proj/single-release-branch$ git tag
Release1.0
Release2.0
csdev@m31:~/proj/single-release-branch$ git checkout master
Switched to branch 'master'
csdev@m31:~/proj/single-release-branch$
```

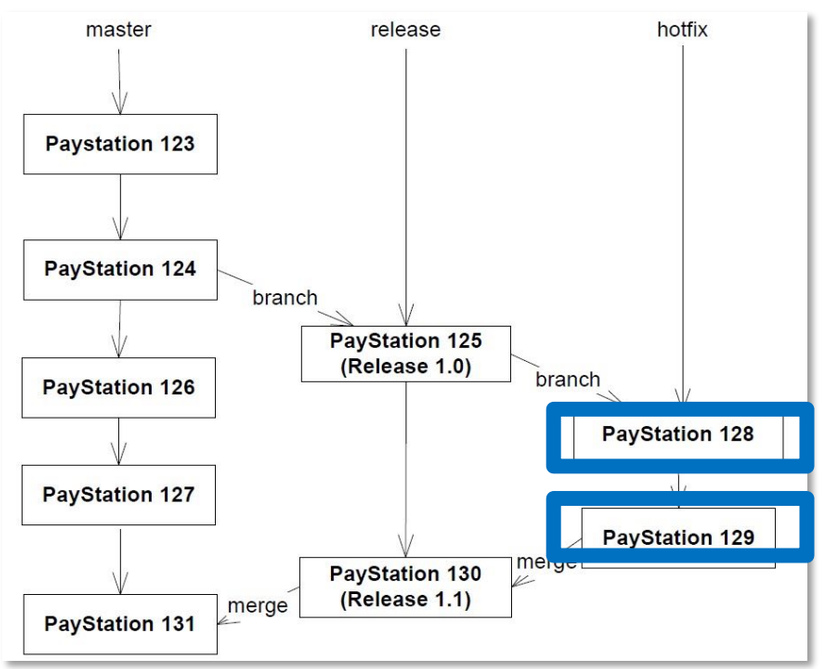


Branching Models: Single Release Branch

Fix something in the release

```
csdev@m51:~/proj/paystation-e19$ git checkout release
Switched to branch 'release'
csdev@m51:~/proj/paystation-e19$ git checkout -b hotfix
Switched to a new branch 'hotfix'
```

```
csdev@m51:~/proj/paystation-e19$ git commit -a -m "Hotfix: Updated commenting per request by customer!"
[hotfix 8a77a7f] Hotfix: Updated commenting per request by customer!
 1 file changed, 2 insertions(+)
```



Branching Models: Single Release Branch

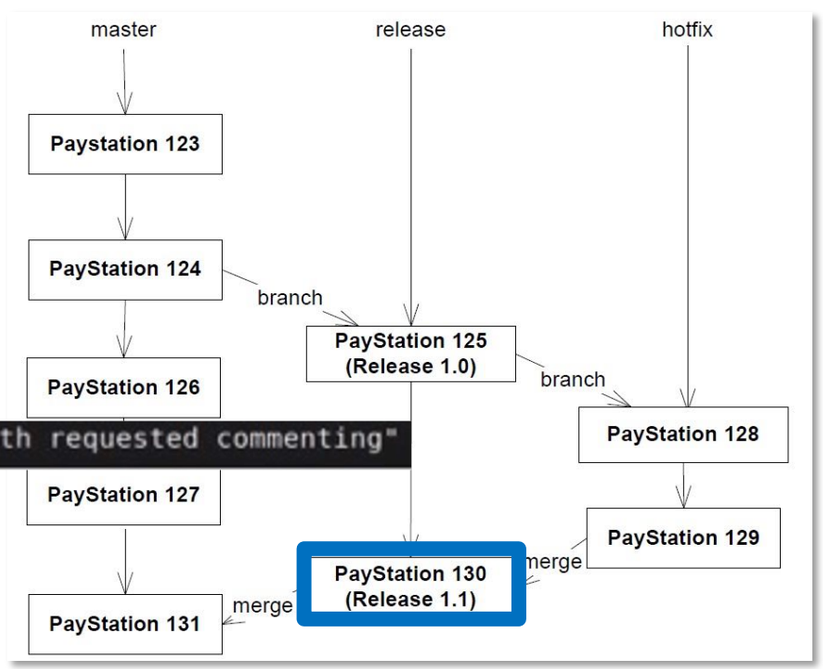
Fix something in the release

```
csdev@m51:~/proj/paystation-e19$ git checkout release
Switched to branch 'release'
csdev@m51:~/proj/paystation-e19$ git checkout -b hotfix
Switched to a new branch 'hotfix'
```

```
csdev@m51:~/proj/paystation-e19$ git commit -a -m "Hotfix: Updated commenting per request by customer!"
[hotfix 8a77a7f] Hotfix: Updated commenting per request by customer!
 1 file changed, 2 insertions(+)
```

```
csdev@m51:~/proj/paystation-e19$ git checkout release
Switched to branch 'release'
csdev@m51:~/proj/paystation-e19$ git merge hotfix
Updating a89b773..8a77a7f
Fast-forward
 src/main/java/paystation/domain/StandardPayStation.java | 2 ++
 1 file changed, 2 insertions(+)
```

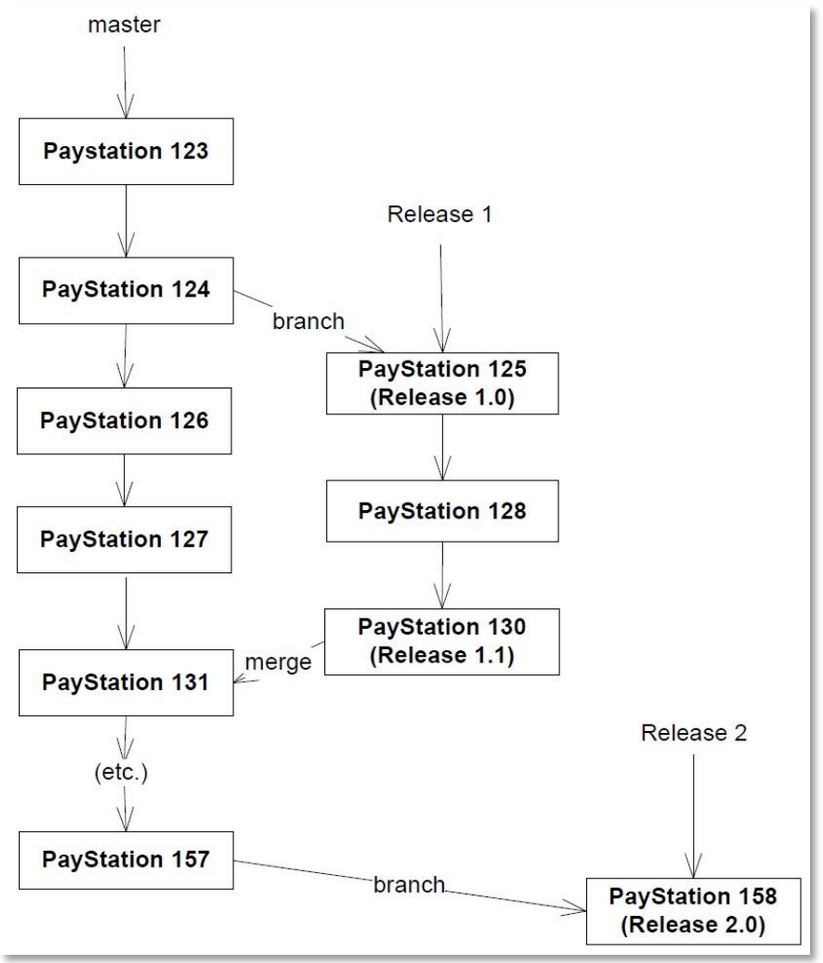
```
csdev@m51:~/proj/paystation-e19$ git tag -a "Release1.0.1" -m "Release 1.0.1 with requested commenting"
```



Branching Models: Major Release Branches

Major Release Branches

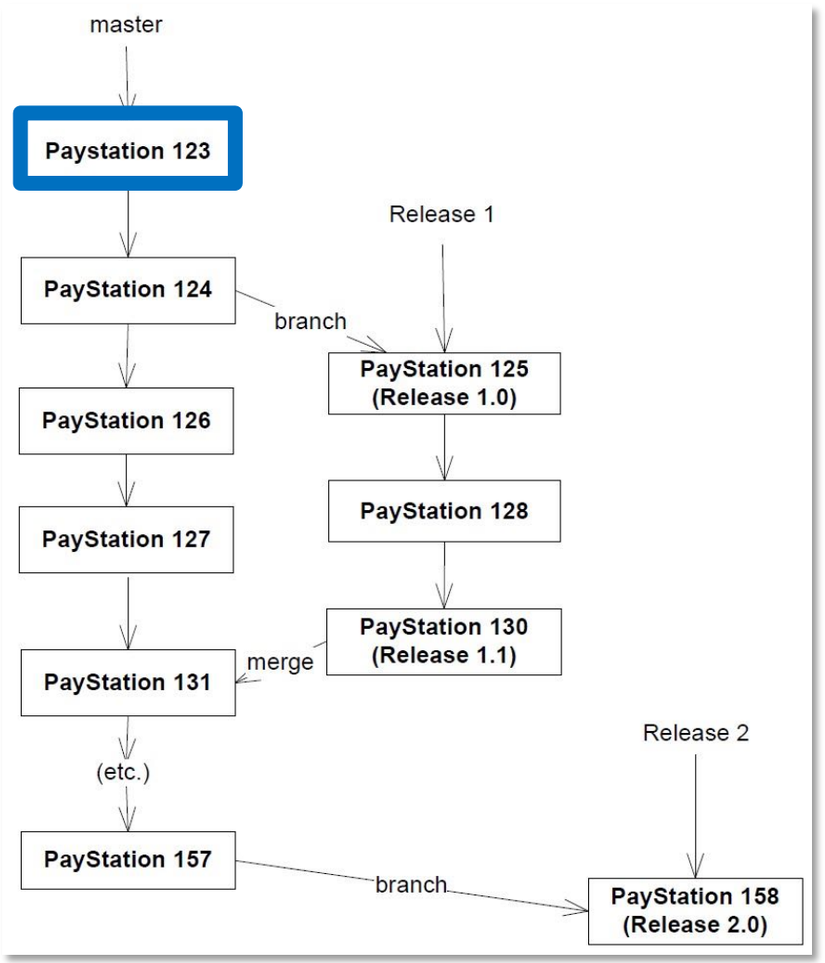
- Each major release gives rise to new branch



Branching Models: Major Release Branches

Working on ‘master’, close to release

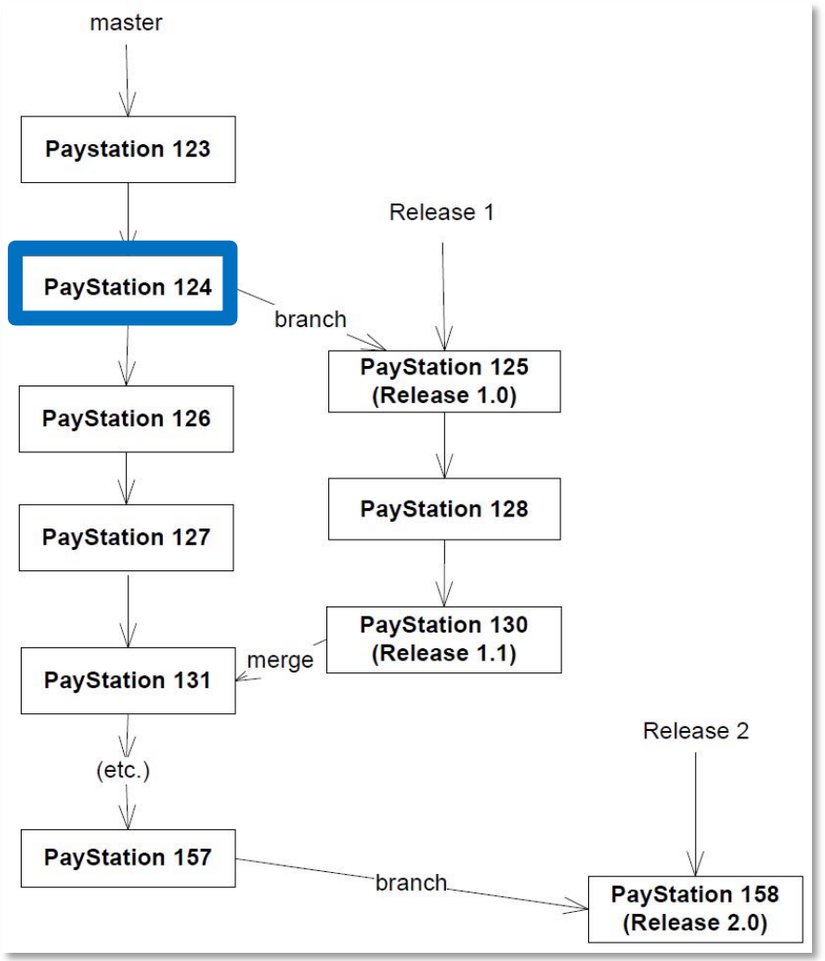
```
csdev@m31:~/proj/major-release-branch$ git log -1
commit f5efa17feb4fcf365aad5186ee5198aboee3eb3b
Author: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:32:42 2017 +0200

    Receipt functionality implemented. Cancel is missing
csdev@m31:~/proj/major-release-branch$ git status
On branch master
nothing to commit, working directory clean
csdev@m31:~/proj/major-release-branch$
```



Branching Models: Major Release Branches

Commit the final code

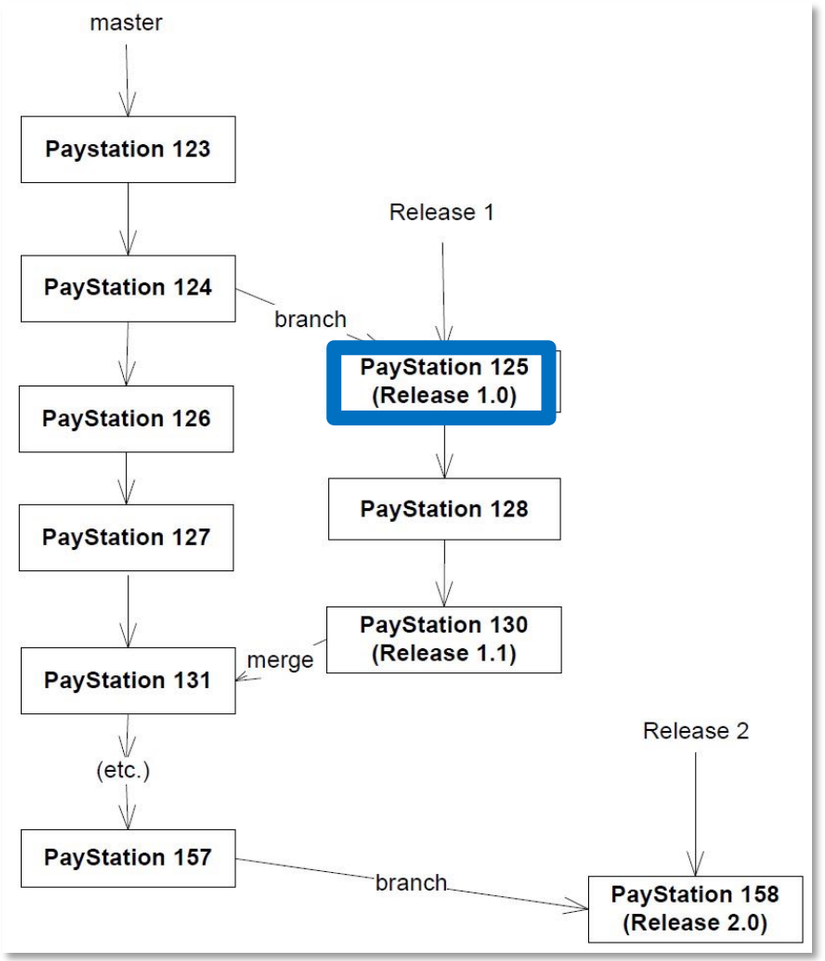
```
commit 021972c7ef561c28742f99214fc8eca3a5018475
Author: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:36:01 2017 +0200

    AlphaTown, now passes all test cases

commit f5efa17feb4fcf365aad5186ee5198ab6ee3e636
Author: baerbak <hbc@cs.au.dk>
Date:   Wed Jul 12 13:32:42 2017 +0200

    Receipt functionality implemented. Cancel is missing
csdev@m31:~/proj/major-release-branch$
```



Branching Models: Major Release Branches

Release 1:
'branch' and potentially 'checkout'

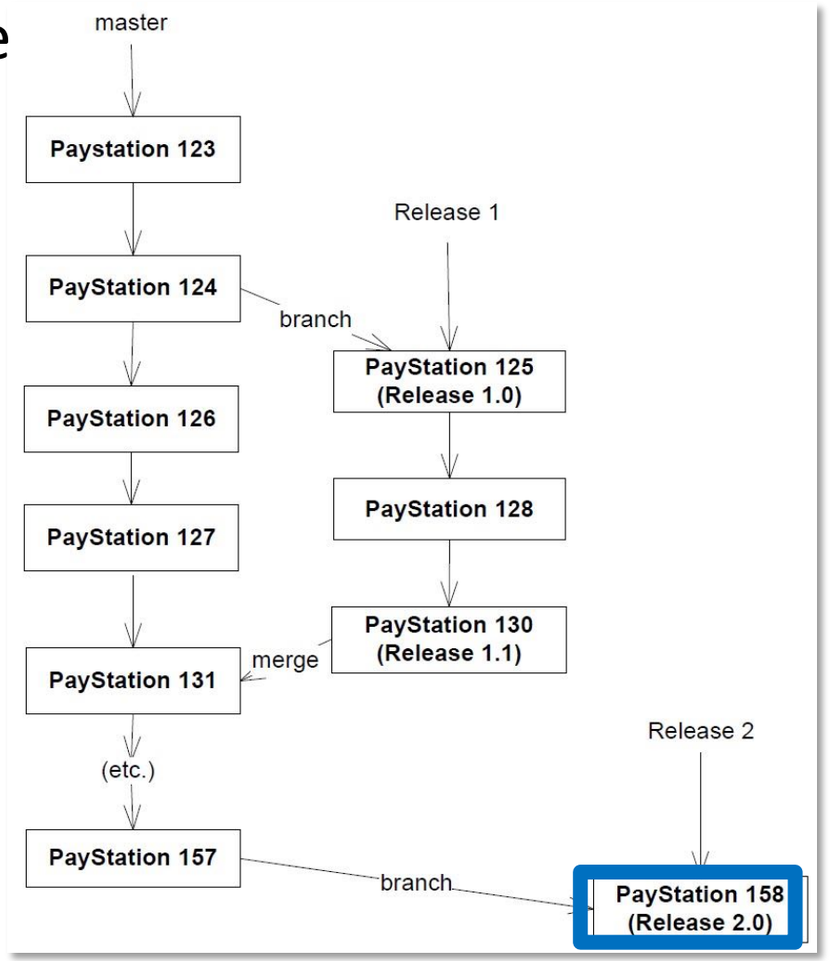
```
csdev@m31:~/proj/major-release-branch$ git branch Release1.0
csdev@m31:~/proj/major-release-branch$ git status
On branch master
nothing to commit, working directory clean
csdev@m31:~/proj/major-release-branch$ git checkout Release1.0
Switched to branch 'Release1.0'
csdev@m31:~/proj/major-release-branch$
```



Branching Models: Major Release Branches

Make a *new branch* for every major release

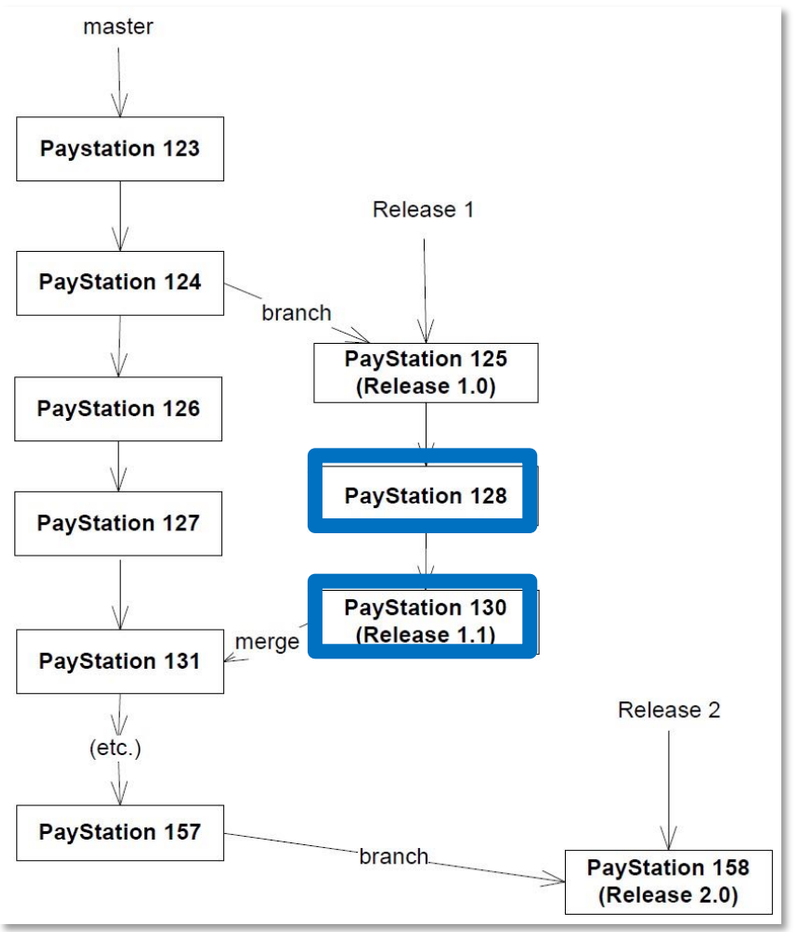
```
csdev@m31:~/proj/major-release-branch$ git branch Release2.0
csdev@m31:~/proj/major-release-branch$ git branch
  Release1.0
  Release2.0
* master
csdev@m31:~/proj/major-release-branch$
```



Branching Models: Major Release Branches

Hotfixing (e.g. Release 1)

- Checkout that release branch, make changes, commit
- Tag it with Release1.1
- Merge hotfix into master before making further changes



Branching Models

Single Release Branch:

Pros

- Get the latest release with ‘git checkout release’
- Fewer branches (only ‘release’ and ‘hotfix’ used for releases)

Cons

- You need a separate branch for hotfixing
  - Branch/Merge over into hotfix, make changes, merge back
- Hotfixing release 1 after release 2 **will** require a new branch
  - Or you will mix the two!

Major Release Branches:

Pro

- Each major release has a branch, allows hotfixing each release with ease

Con

- Many releases means many branches to maintain

Branching Models

Maintaining multiple concurrent releases is **expensive**

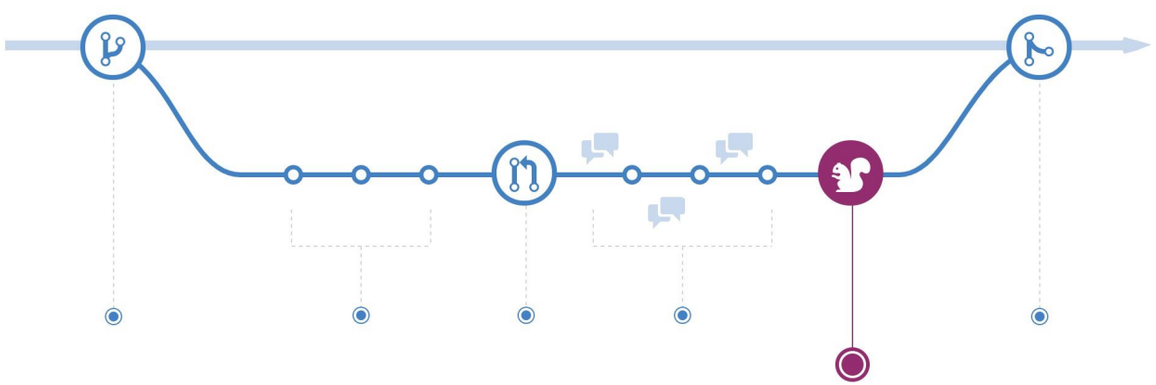
- Windows 7, Windows 8.1, Windows 10, Win Server …

Many companies adopt the **single release** model

- End support for previous versions, force update to latest version

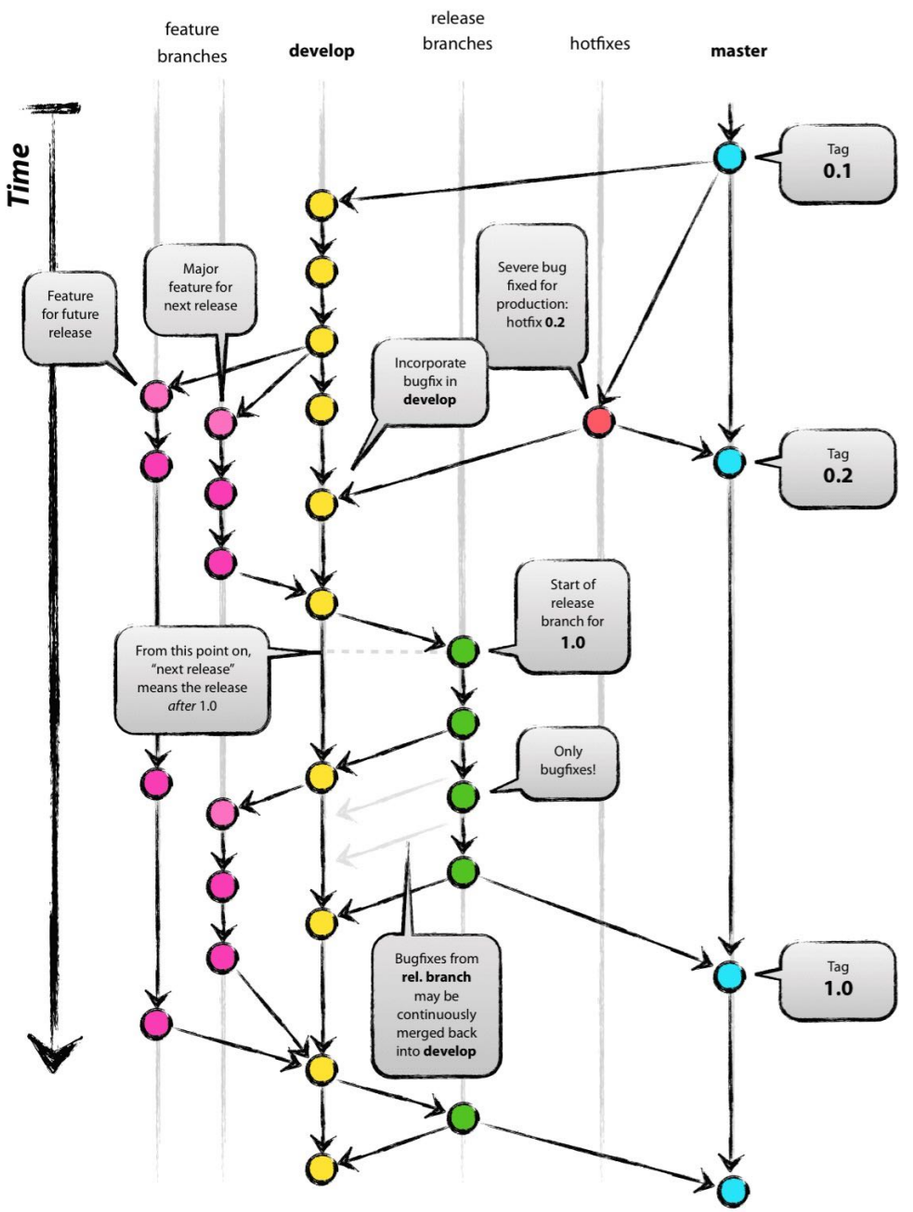
Branching Models

GitHub Flow



https://guides.github.com/introduction/flow/

Git Flow



https://nvie.com/posts/a-successful-git-branching-model/

Branching Models: Recommendations

Keep it simple

Git defaults to the branch called ‘main’ – use it as your development branch

Create and work on feature/refactoring branches to preserve a stable version on main (e.g., an iteration ready to submit)

- Easy to ‘Do Over’, just abandon that branch, tag it with ‘end of line’/’bad idea’
- If the idea was good, merge it back into ‘main’

You ‘release’ every time you hand in a project iteration

- Single release branch – or one branch per release model

Build Management

As a project gets more complicated, how do we ensure we build the same way each time?

→ Automated build management

Build Management

As a project gets more complicated, how do we ensure we build the same way each time?

→ Automated build management

Definition: **Build management**

The process of managing and constructing an executable software system from its parts in a reliable and cost-efficient way.

Build Management

As a project gets more complicated, how do we ensure we build the same way each time?

→ Automated build management

Definition: **Build management**

The process of managing and constructing an executable software system from its parts in a reliable and cost-efficient way.

Definition: **Build description**

A description of the goals and means for managing and constructing an executable software system. A build description states *targets*, *dependencies*, *procedures*, and *properties*.

Description/Script

Build Management: Script

Targets: tasks/goals, like “compile all source code files”

Build Management: Script

Targets: tasks/goals, like “compile all source code files”

Dependencies: tasks can depend on other tasks (e.g., you must compile all source code before executing, so the execution target **depends on** the compilation target); or external libraries required for the build

Build Management: Script

Targets: tasks/goals, like “compile all source code files”

Dependencies: tasks can depend on other tasks (e.g., you must compile all source code before executing, so the execution target **depends on** the compilation target); or external libraries required for the build

Procedures: associated with the targets; describe how to meet the goal of the target

- For example, the compile goal must have an associated procedure that describes the steps necessary to compile all source files (e.g., call javac on all files)

Build Management: Script

Targets: tasks/goals, like “compile all source code files”

Dependencies: tasks can depend on other tasks (e.g., you must compile all source code before executing, so the execution target **depends on** the compilation target); or external libraries required for the build

Procedures: associated with the targets; describe how to meet the goal of the target

- For example, the compile goal must have an associated procedure that describes the steps necessary to compile all source files (e.g., call javac on all files)

Properties: variables and constants that you can assign and use in your procedures; used to improve readability in the build script

Build Management

Example: Makefiles for C/C++

- Specify compiler, flags
- Target: build executable(s)
- Properties: Makefile variables
- Define commands, organize files, etc.

```
> gcc -o hellomake hellomake.c hellofunc.c -I
```

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)
ODIR=obj
LDIR =../lib
LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

$(ODIR)/%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)
```

https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Build Management

Example: Makefiles for C/C++

- Specify compiler, flags
- Target: build executable(s)
- Properties: Makefile variables
- Define commands, organize files, etc.

(CMake is compiler/platform independent)

Makefile created with CMake

```
#=============================================================================
# Target rules for targets named submission

# Build rule for target.
submission: cmake_check_build_system
	$(MAKE) $(MAKESILENT) -f CMakeFiles/Makefile2 submission
.PHONY : submission

# fast build rule for target.
submission/fast:
	$(MAKE) $(MAKESILENT) -f CMakeFiles/submission.dir/build.make CMakeFiles/submission.dir/build
.PHONY : submission/fast

#=============================================================================
# Target rules for targets named bitset-tests

# Build rule for target.
bitset-tests: cmake_check_build_system
	$(MAKE) $(MAKESILENT) -f CMakeFiles/Makefile2 bitset-tests
.PHONY : bitset-tests

# fast build rule for target.
bitset-tests/fast:
	$(MAKE) $(MAKESILENT) -f CMakeFiles/bitset-tests.dir/build.make CMakeFiles/bitset-tests.dir/build
.PHONY : bitset-tests/fast

bitset.o: bitset.cpp.o

.PHONY : bitset.o

# target to build an object file
bitset.cpp.o:
	$(MAKE) $(MAKESILENT) -f CMakeFiles/bitset-tests.dir/build.make CMakeFiles/bitset-tests.dir/bitset.cpp.o
.PHONY : bitset.cpp.o

bitset.i: bitset.cpp.i

.PHONY : bitset.i

# target to preprocess a source file
bitset.cpp.i:
	$(MAKE) $(MAKESILENT) -f CMakeFiles/bitset-tests.dir/build.make CMakeFiles/bitset-tests.dir/bitset.cpp.i
.PHONY : bitset.cpp.i

bitset.s: bitset.cpp.s
```

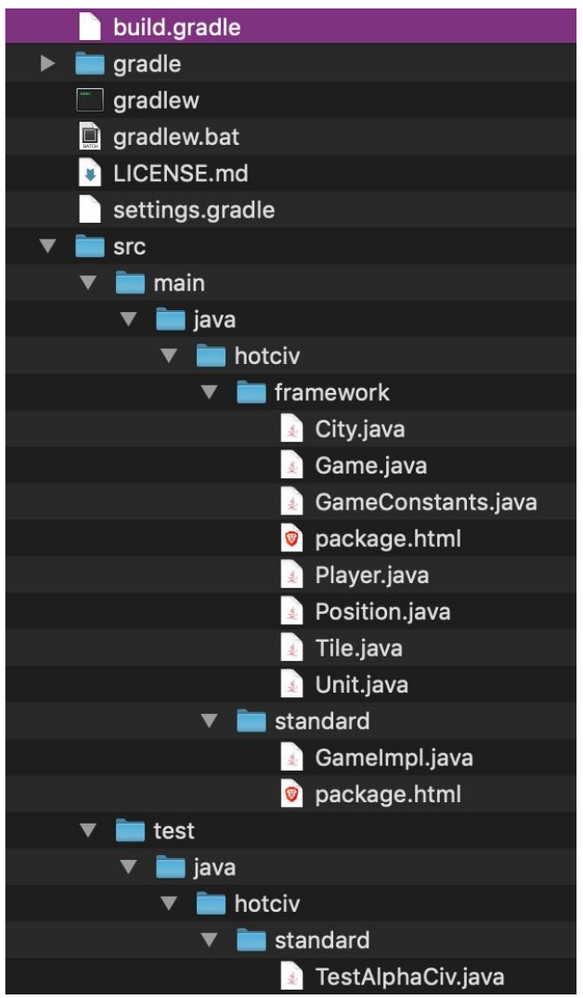
Build Management: Gradle

Gradle is a **convention-based** build management tool

- Java (Kotlin, Groovy, Scala) , C/C++, JavaScript

Conventions:

- A set of **tasks** (targets) are defined (build, test, …)
- Specified folder **hierarchy and naming**
- The ‘build script’ is in **build.gradle** in the project root



Build Management: Gradle

The simplest build.gradle file for a Java project contains one line:

```
apply plugin: 'java'
```

Now, we can do basic (Java) predefined build management tasks, like:

```
gradle test
```

- Compiles all **production** code, all **test** code, and executes all Junit code in the 'test' source tree

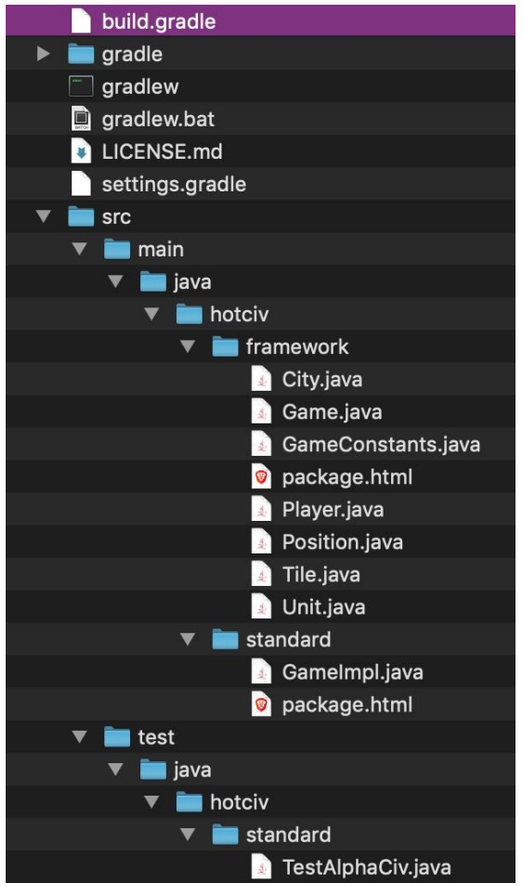
```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
}
```

Build Management: Gradle

How does it work?

Convention

- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production code
  - **src/test/java/HERE** ← Test code

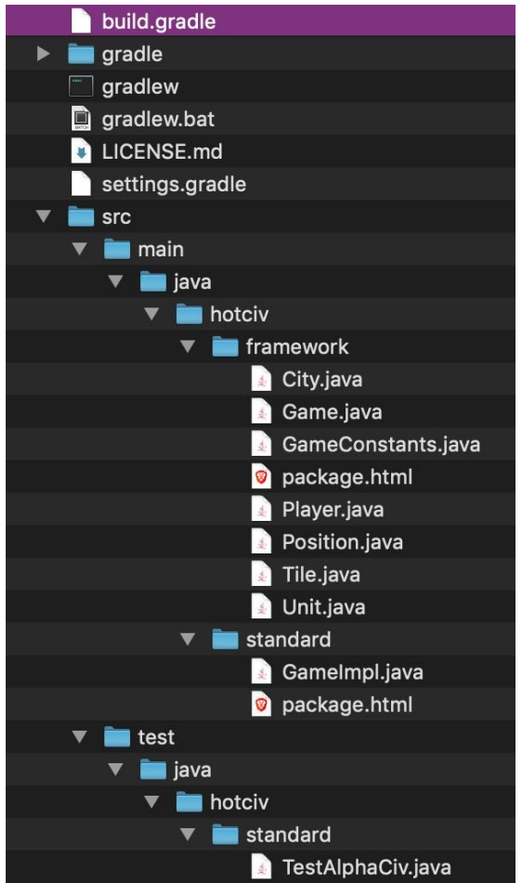


Build Management: Gradle

How does it work?

Convention

- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production code
  - **src/test/java/HERE** ← Test code
- Predefined tasks
  - Like ‘build’, ‘test’, …



Build Management: Gradle

How does it work?

Convention

- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production
  - **src/test/java/HERE** ← Test code
- Predefined tasks
  - Like ‘build’, ‘test’, …
  - ‘gradle tasks’ will display all known tasks/targets

```
------------------------------------------------------------
Tasks runnable from root project
------------------------------------------------------------

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend
on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root projec
t 'starter-code'.
components - Displays the components produced by root project 'starter-code'. [i
ncubating]
dependencies - Displays all dependencies declared in root project 'starter-code'
.
dependencyInsight - Displays the insight into a specific dependency in root proj
ect 'starter-code'.
dependentComponents - Displays the dependent components of components in root pr
oject 'starter-code'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'starter-code'. [incuba
ting]
```

Build Management: Gradle

How does it work?

Convention

- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production code
  - **src/test/java/HERE** ← Test code
- Predefined tasks
  - Like ‘build’, ‘test’, …

Plugins

- Define tasks, conventions
  https://docs.gradle.org/current/userguide/plugin_reference.html

```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
}
```

Build Management: Gradle

How does it work?

Convention

- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production code
  - **src/test/java/HERE** ← Test code
- Predefined tasks
  - Like ‘build’, ‘test’, …

Plugins

- Define tasks, conventions
  https://docs.gradle.org/current/userguide/plugin_reference.html

Custom tasks in Groovy or Kotlin, if needed
https://docs.gradle.org/current/userguide/tutorial_using_tasks.html

(not necessary for this class)

```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}
```

```
tasks.register('hello') {
    doLast {
        println 'Hello world!'
    }
}
```

```
nit:junit:4.12'
g.hamcrest:hamcrest-library:1.3'
```

Dependency Management: Gradle

Gradle is also a **dependency-management tool.**

Ex: hamcrest

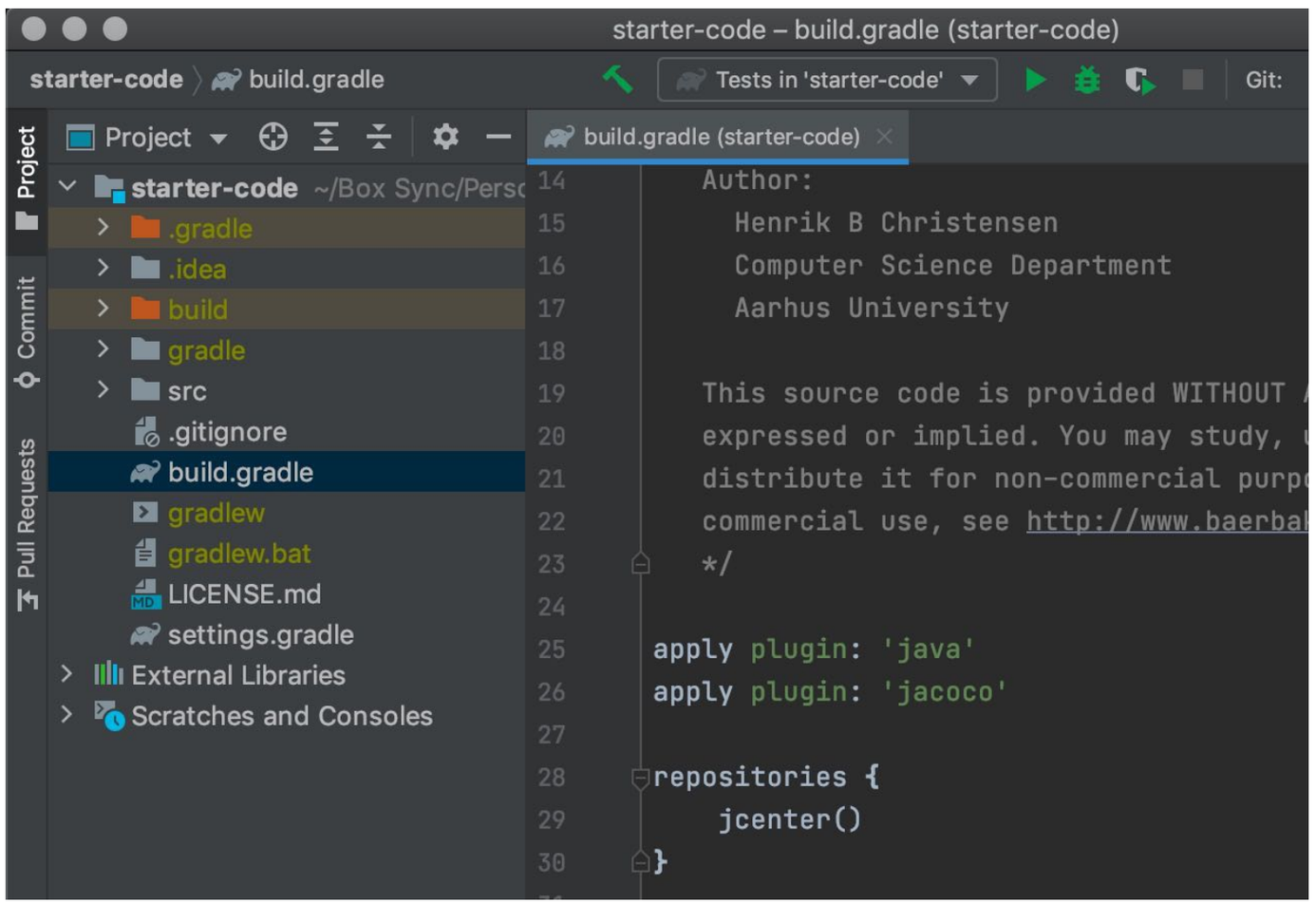
```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
}
```

Gradle will download ‘org.hamcrest….:1.3’ from JCenter and set the classpath accordingly

Build Management: Gradle + IntelliJ

Integration with IntelliJ

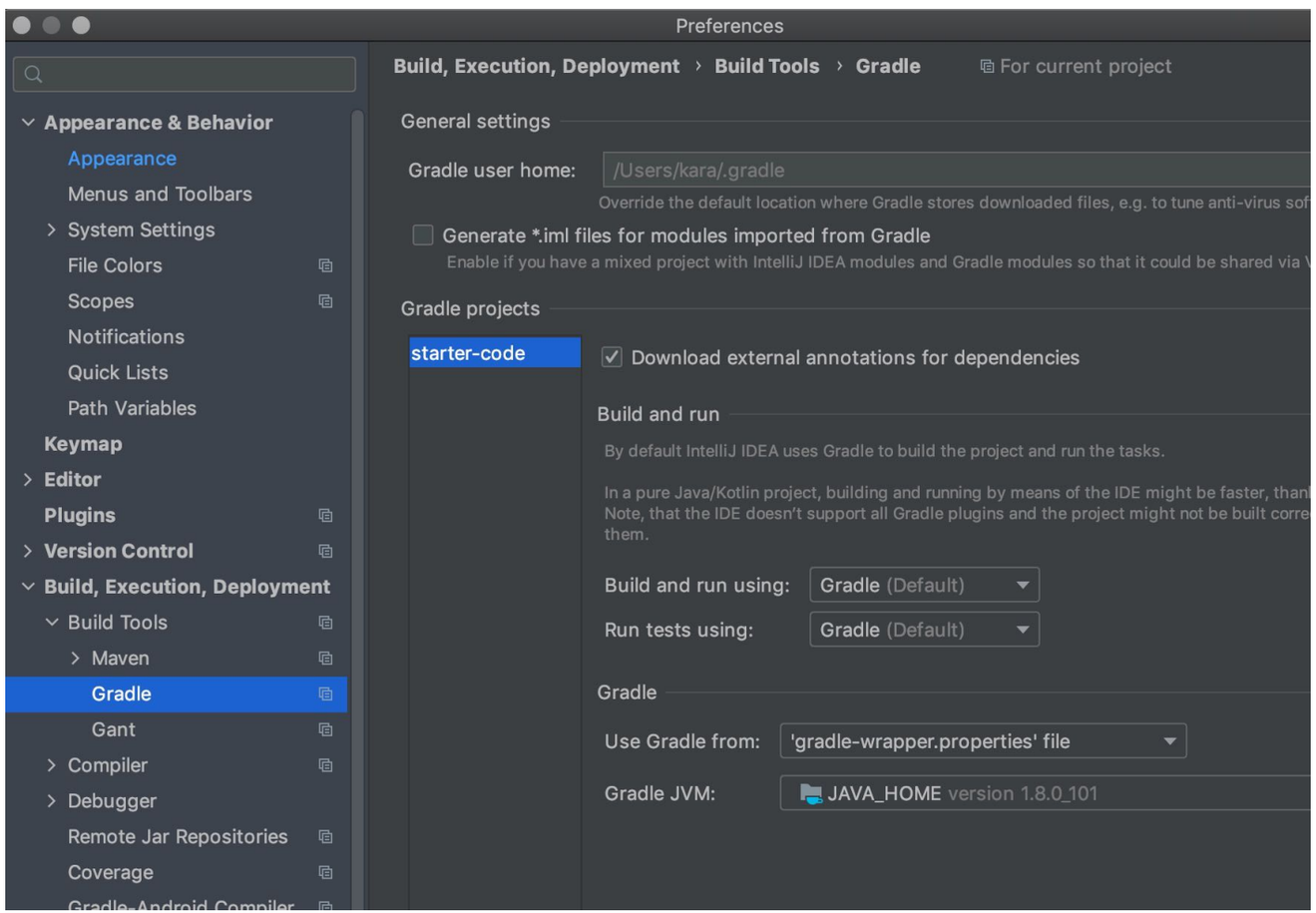
- Create new gradle projects
- Open existing gradle projects



Build Management: Gradle + IntelliJ

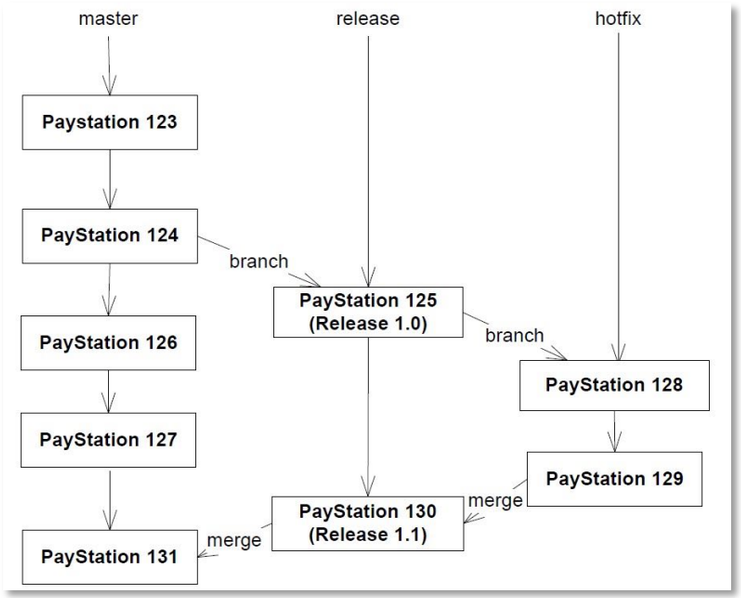
Integration with IntelliJ

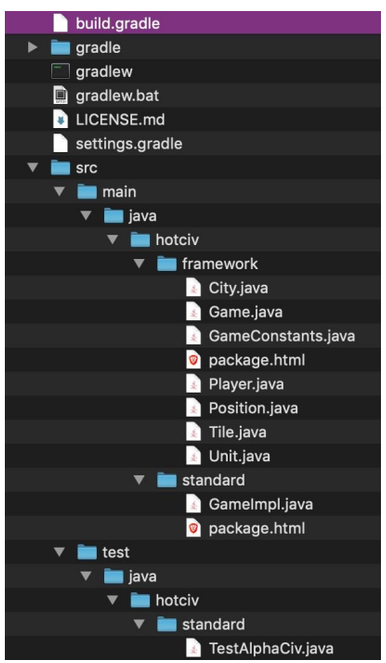
- Create new gradle projects
- Open existing gradle projects
- Recognizes build.gradle and gradle project structure



Summary

- Branches enable working in parallel (features, releases, etc.)
- Build management (e.g., Gradle) enables consistent compilation, dependency management, definition of tasks





build.gradle

```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
}
```

HotCiv Project Tips

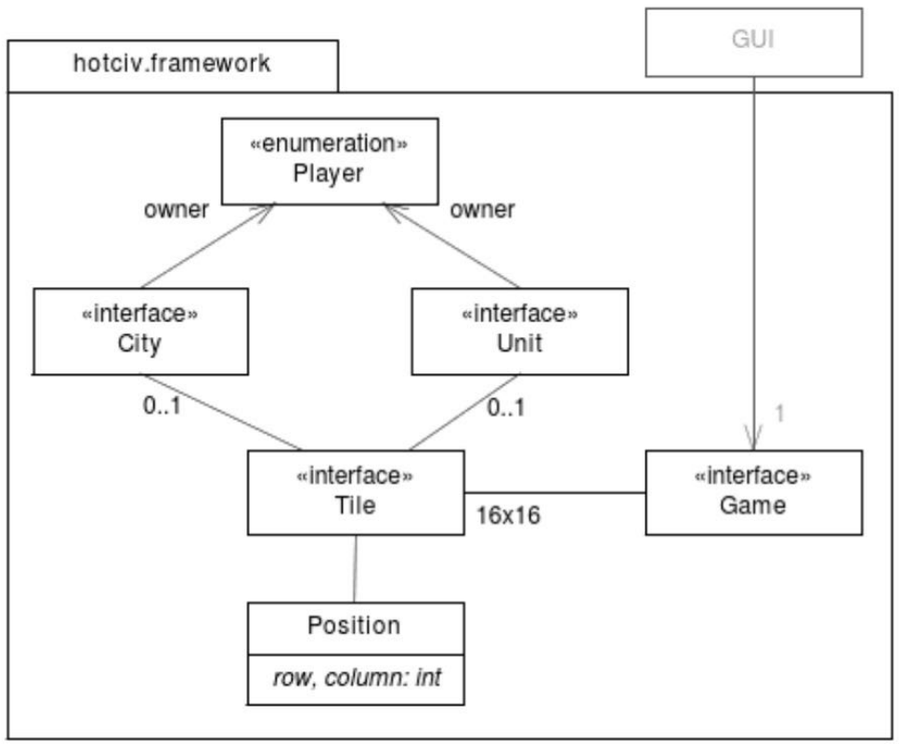


Figure 36.3: HotCiv central abstractions.

TestAlphaCiv.java

```java
public class TestAlphaCiv {
  private Game game;

  /** Fixture for alphaciv testing. */
  @Before
  public void setUp() { game = new GameImpl(); }

  // FRS p. 455 states that 'Red is the first player to take a turn'.
  @Test
  public void shouldBeRedAsStartingPlayer() {
    assertThat(game, is(notNullValue()));
    // TODO: reenable the assert below to get started...
    // assertThat(game.getPlayerInTurn(), is(Player.RED));
  }
```

GameImpl.java

```java
public class GameImpl implements Game {
  public Tile getTileAt( Position p ) { return null; }
  public Unit getUnitAt( Position p ) { return null; }
  public City getCityAt( Position p ) { return null; }
  public Player getPlayerInTurn() { return null; }
  public Player getWinner() { return null; }
  public int getAge() { return 0; }
  public boolean moveUnit( Position from, Position to ) { return false; }
  public void endOfTurn() {}
  public void changeWorkForceFocusInCityAt( Position p, String balance ) {}
  public void changeProductionInCityAt( Position p, String unitType ) {}
  public void performUnitActionAt( Position p ) {}
}
```

HotCiv Project Tips

Goal is to build the **backend**, to be controlled by a GUI. Assume that the GUI will only mutate the game's state by using **Game's mutator** methods:

endOfTurn, moveUnit, etc.

And only inspect it using either **Game accessor methods** *or* the **"read-only" interfaces:**

getTileAt(p), getCityAt(p), getPlayerInTurn(), …

HotCiv Project Tips

Keep interfaces intact!

Otherwise the GUI will have trouble interfacing

Read-only interfaces (Unit, City, …)

Don't add mutator methods to the interfaces (no setting)!

Add mutator methods to **implementations** (UnitImpl, etc.)

Preconditions

Many game methods require valid inputs (e.g., positions)

You should not make tests for invalid positions!

Goal is to build the backend, to be controlled by a GUI
Assume that the GUI will only mutate the game's state by using Game's mutator methods

endOfTurn, moveUnit, etc.

And only inspect it using either Game accessor methods *or* the "read-only" interfaces

getTileAt(p), getCityAt(p), getPlayerInTurn(), …

```
/** return a specific tile.
 * Precondition: Position p is a valid position in the world.
 * @param p the position in the world that must be returned.
 * @return the tile at position p.
 */
public Tile getTileAt( Position p );
```

HotCiv Project Tips

String types in GameConstants?

Enums would give better reliability but would limit future variants' ability to add more types (more when we get to frameworks)

MoveCount?

Distance is measured in 'move count' →

If unit has move count = 2; and you move one tile, its move count is 1, and so on!

To move a unit 2 tiles, you invoke moveUnit() **twice**

Treasury?

'Production' means two things!

city.getTreasury() = how much 'money' that city has right now

A unit is produced once enough 'money' has been generated

HotCiv: Test-Driven Development

No “World” abstraction?

Trust the TDD process!

TDD is about being efficient/lazy

Do not code in anticipation of a need! **Maximize work not done!**

Translate the specs into a **minimal** set of test cases.
Make the test cases drive the **minimal** amount of code.
Make it as **simple** as possible!!! Code as little as possible!!!
Write **quality** code

HotCiv: Test-Driven Development

Write test lists, not feature lists:

setup world = feature; not a test
red has city at (1,1) = test

Keep it Test-Driven

Think “what is my test case”; not “how do I implement this”

Don’t think too far ahead

Don’t get distracted by problems that may never arise
Pick “one step tests”
Be prepared for “do over” – better to refactor than to overcomplicate things early

Do not write a ton of tests and **then** start to implement – one test/feature iteration at a time

Check code coverage

How much of your production code is tested by your tests?
Use IntelliJ “Run Tests with coverage”

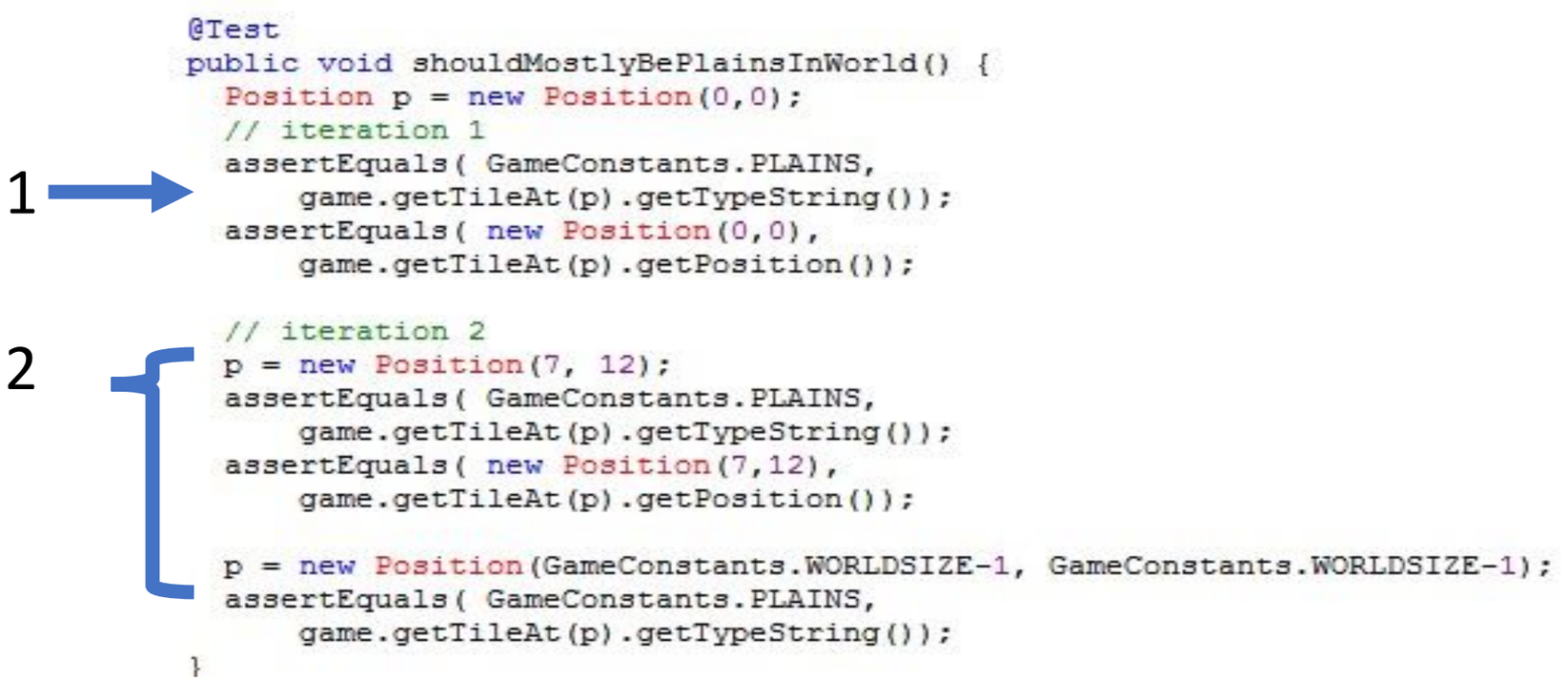
Refactor often!

HotCiv: Test-Driven Development

Example test: Tile types

Iteration 1: Fake it – return plains

Iteration 2: Create data structure, fill **all entries** with plains



HotCiv: Test-Driven Development

Add test cases for differences from the “default” (plains)

- Test that one tile is “Mountains”
  - Write the test case ‘shouldHaveMountainAt2_2()’
  - See it fail
  - Enhance the production code to
    - world[2][2] = new StandardTile(GameConstants.MOUNTAIN);
    - (or world.put(new Position(2,2), new StandardTile…..)
    - (or …)
  - See it pass
- Test that one tile is “Ocean”, etc.

HotCiv: Test-Driven Development

The more your test cases only use the given **Game, City, Unit interface methods**, the more stable your test cases will be against refactoring!

Do this:

- game.getCityAt(p)

Not this:

- ((GameImpl) game).getInternalCityHashmap.get(p)

Try to keep City, Unit as read-only interfaces

→ This is actually the **facade** pattern which we will discuss later

HotCiv: Test-Driven Development

Which data structure(s) should I use?

- Whatever you like!
  - Array of arrays
  - HashMap
  - Same or different data structure for tiles vs. units vs. cities

Don’t be afraid to try something, do over if it doesn’t work

- Create a branch to try something, easy to abandon
- “git stash”

If you have a strong set of test cases and you use the Game, City, Unit interface methods in your tests, you can quickly refactor the backend

HotCiv: Data Structures

Example design decision: City objects are stored in a matrix

- Matrix[4][1] contains city object in world position (4,1)

```
// Demonstration of the sweep template on a matrix
public void sweepMatrix() {
  System.out.println("-- Sweeping a matrix --");
  City[][] matrix = new City[WORLDSIZE][WORLDSIZE];
  matrix[1][1] = new City(); // The red city
  matrix[4][1] = new City(); // The blue city
  // Sweeping a matrix
  for (int row = 0; row < WORLDSIZE; row++) {
    for (int column = 0; column < WORLDSIZE; column++) {
      City element = matrix[row][column];
      if (element != null) {
        // process element
        System.out.println("Processing "+element);
      }
    }
  }
}
```

HotCiv: Data Structures

Example design decision: unfold matrix to a one-dimensional List

- list.get(row*16+col) contains city at (row,col)

```
// Demonstration of the sweep template on a List
public void sweepList() {
  System.out.println("-- Sweeping a List --");
  // We create a List<City> of size 16x16,
  // and then position (3,4) is index (3*16+4)
  List<City> list = new ArrayList<City>(WORLDSIZE * WORLDSIZE);
  // Though the capacity is 64, the size is still 0 so I need to
  for (int i = 0; i < WORLDSIZE * WORLDSIZE; i++) list.add(null);
  int index;
  index = computeIndex(1,1); list.set(index, new City()); // The red city
  index = computeIndex(4,1); list.set(index, new City()); // The blue city

  // Sweeping the List
  for (City element: list) {
    if (element != null) {
      // process element
      System.out.println("Processing "+element);
    }
  }
}
```

HotCiv: Data Structures

Example design decision: City objects are stored in a HashMap

- map.get(new Position(4,1)) contains city object in world position (4,1)

```java
// Demonstration of the sweep template on a Map
// using the Java 7 / classic for iteration
public void sweepMapClassic() {
  System.out.println("-- Sweeping a map / Classic --");
  Map<Position, City> map = new HashMap<>();
  map.put(new Position(1,1), new City()); // The red city
  map.put(new Position(4,1), new City()); // The blue city
  // Sweeping a map
  for (Position p: map.keySet()) {
    City element = map.get(p);
    if (element != null) {
      // process element
      System.out.println("Processing "+element);
    }
  }
  // Note: the 'if' is not necessary as the map only
  // contains the two cities.
}
```

```java
// Demonstration of the sweep template on a Map
// using the Java 8 / stream api
public void sweepMapStream() {
  System.out.println("-- Sweeping a map / Stream --");
  Map<Position, City> map = new HashMap<>();
  map.put(new Position(1,1), new City()); // The red city
  map.put(new Position(4,1), new City()); // The blue city
  // Sweeping a map
  map.keySet()
    .stream()
    .filter(p -> map.get(p) != null)
    .forEach(p -> {
        City element = map.get(p);
        System.out.println("Processing "+element);
      });
  // Note: the 'filter' is actually not necessary here,
  // as the map only contains the two cities
}
```

Or use for (Map.Entry() entry : map.entrySet())

HotCiv: Data Structures

How to fill your data structure?

```
public GameImpl() {
  // Setup the tile, city and unit locations, creating the world
  this.tiles = new TileImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
  this.cities = new CityImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
  this.units = new UnitImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];

  // Runs through the different tiles to give them the right terrain types
  for(int i = 0; i < GameConstants.WORLDSIZE; i++) {
    for(int j = 0; j < GameConstants.WORLDSIZE; j++) {
      if(i == 1 && j == 0) {
        tiles[i][j] = new TileImpl(GameConstants.OCEANS);
      }
      else if(i == 0 && j == 1) {
        tiles[i][j] = new TileImpl(GameConstants.HILLS);
      }
      else if(i == 2 && j == 2) {
        tiles[i][j] = new TileImpl(GameConstants.MOUNTAINS);
      }
      else {
        tiles[i][j] = new TileImpl(GameConstants.PLAINS);
      }
    }
  }
```

```
for(int i = 0; i < GameConstants.WORLDSIZE; i++) {
        for (int j = 0; j < GameConstants.WORLDSIZE; j++) {
                g.createTile(newPosition(i, j), GameConstants.PLAINS);
        }
}
```

```
g.createTile(new Position(0, 1), GameConstants.HILLS);
g.createTile(new Position(1, 0), GameConstants.OCEANS);
g.createTile(new Position(2, 2), GameConstants.MOUNTAINS);
```

HotCiv: Data Structures

How to fill your data structure?

```
public GameImpl() {
  // Setup the tile, city and unit locations, creating the world
  this.tiles = new TileImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
  this.cities = new CityImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
  this.units = new UnitImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];

  // Runs through the different tiles to give them the right terrain types
  for(int i = 0; i < GameConstants.WORLDSIZE; i++) {
    for(int j = 0; j < GameConstants.WORLDSIZE; j++) {
      if(i == 1 && j == 0) {
        tiles[i][j] = new TileImpl(GameConstants.OCEANS);
      }
      else if(i == 0 && j == 1) {
        tiles[i][j] = new TileImpl(GameConstants.HILLS);
      }
      else if(i == 2 && j == 2) {
        tiles[i][j] = new TileImpl(GameConstants.MOUNTAINS);
      }
      else {
        tiles[i][j] = new TileImpl(GameConstants.PLAINS);
      }
    }
  }
```

Next time:

Our first design pattern!

```
for(int i = 0; i < GameConstants.WORLDSIZE; i++) {
        for (int j = 0; j < GameConstants.WORLDSIZE; j++) {
                g.createTile(newPosition(i, j), GameConstants.PLAINS);
        }
}
```

```
g.createTile(new Position(0, 1), GameConstants.HILLS);
g.createTile(new Position(1, 0), GameConstants.OCEANS);
g.createTile(new Position(2, 2), GameConstants.MOUNTAINS);
```