Lecture 02

ECE 1145: Software Construction and Evolution

Maintainability (CH 1, 3, 10)

Tools and Environment Setup

Course Project Overview (CH 36.1, 36.2)

Announcements

- Complete CATME survey by Sept. 5 11:59 PM
  - Contact me if you did not receive a login email
- Iteration 0 report due Sept. 12 11:59 PM
  - Individual report submissions, team GitHub page
- No class next Monday Sept. 6 (Labor Day)

Questions for Today

What are qualities of maintainable software?

What are qualities of flexible software?

How can we write maintainable, flexible software?

Recall: Code Quality

- **ISO 9126:** 1991 - 2011
- **ISO 25010:** 2011 - Present

Flexibility and Maintainability

Maintenance and **Evolution** both refer to making changes to an existing system

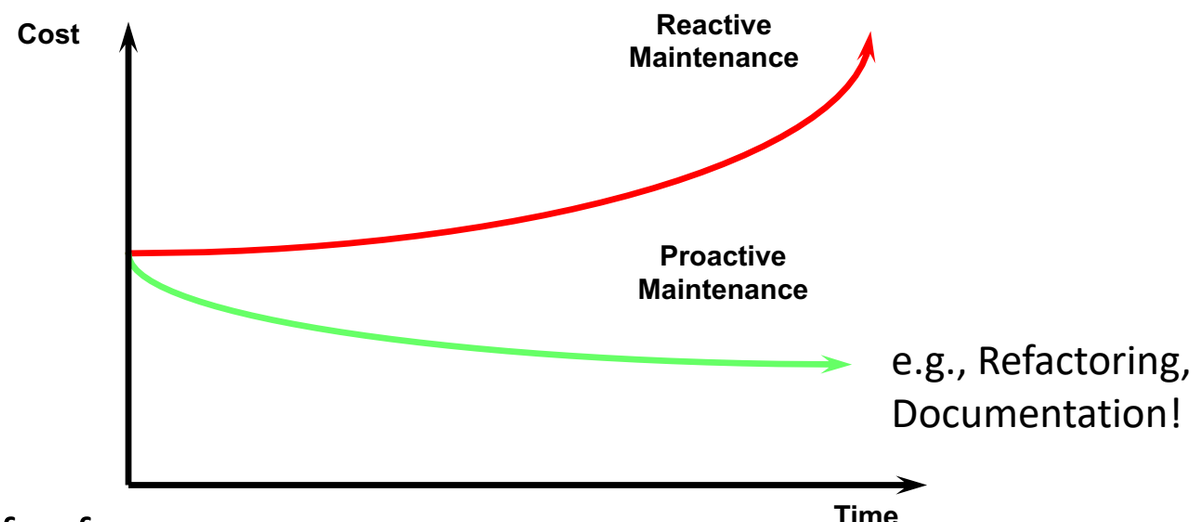
- **Maintenance** often refers to fixing bugs or porting a system to a new platform
- **Evolution** is making enhancements to existing software when the specifications or technology changes

Flexibility and Maintainability

Seemingly minor changes often turn out to be more extensive than expected…

This leads to:

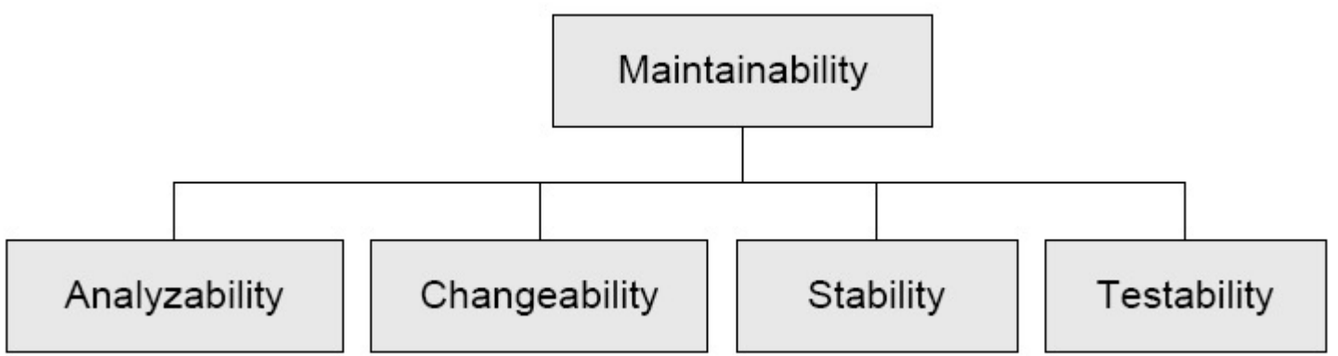
- Incomplete changes (maybe discovered by user...)
- Poorly implemented changes (patches and spaghetti/ravioli)
- Cost estimate errors
- Reduced maintainability and useful life of the software



Maintainability

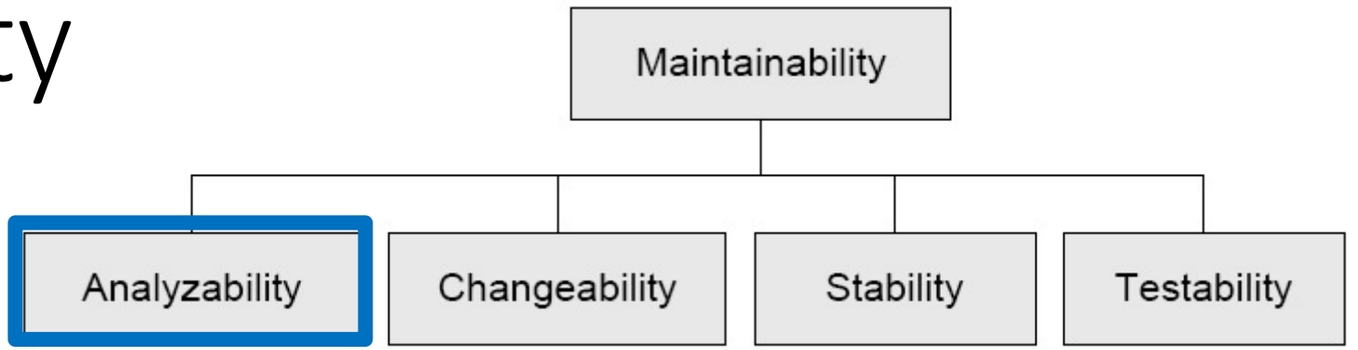
Definition: Maintainability (ISO 9126)

The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.



(Modifiability)

Maintainability

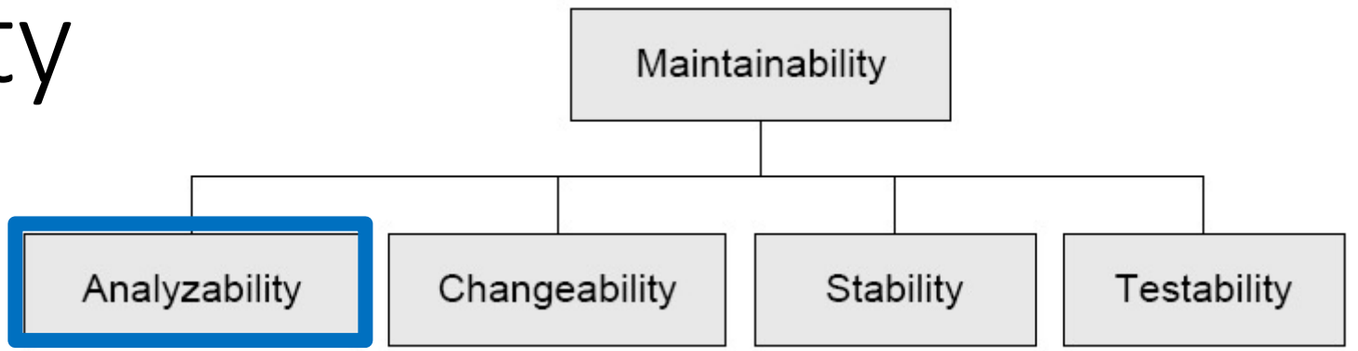


Definition: Analyzability (ISO 9126)

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Can we **understand** the code?

Maintainability



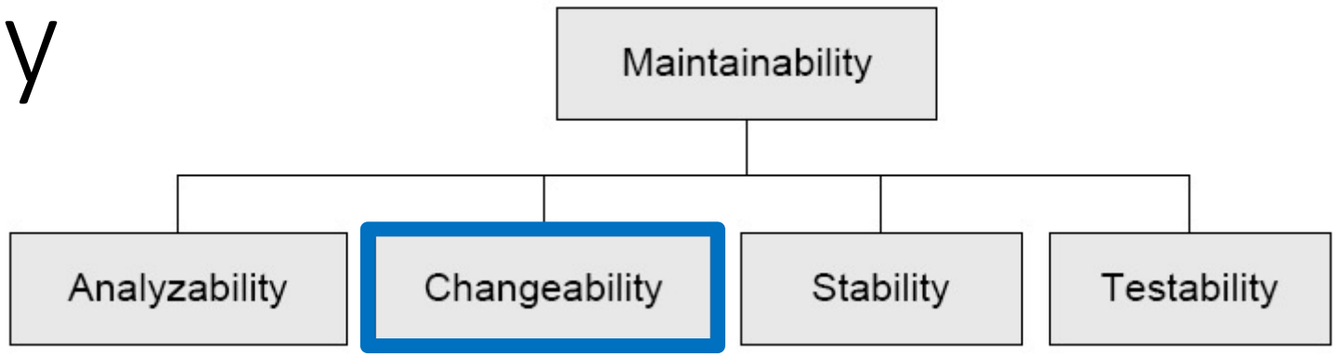
Definition: Analyzability (ISO 9126)

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Can we **understand** the code?

- Indentation
- Naming conventions for classes/methods
- Useful comments
- Descriptive variable names
- Training, e.g., to recognize **design patterns**

Maintainability

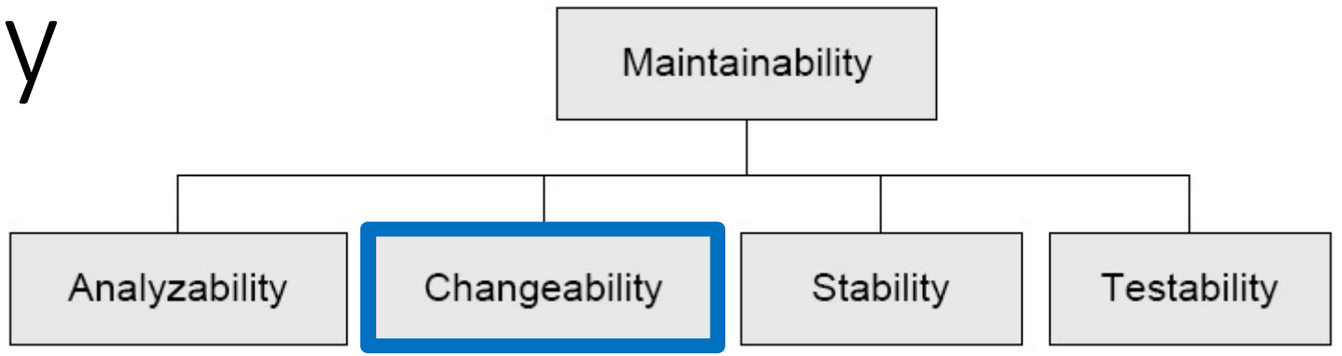


Definition: Changeability (ISO 9126)

The capability of the software product to enable a specified modification to be implemented.

What is the **cost** to modify the code?

Maintainability



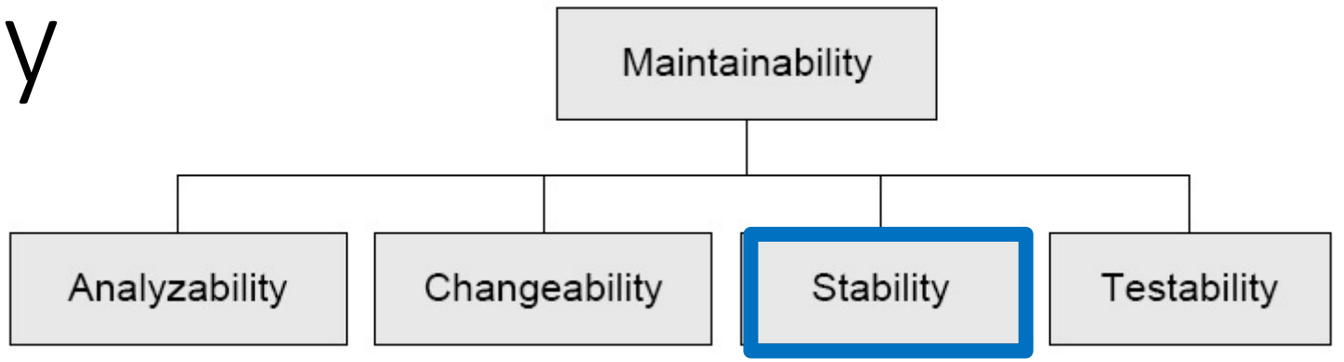
Definition: Changeability (ISO 9126)

The capability of the software product to enable a specified modification to be implemented.

What is the **cost** to modify the code?

- Cost: money, time, personnel, etc.
- Example: Use named variables instead of hardcoded “magic numbers” throughout the code
- Will discuss more with design patterns and framework theory

Maintainability

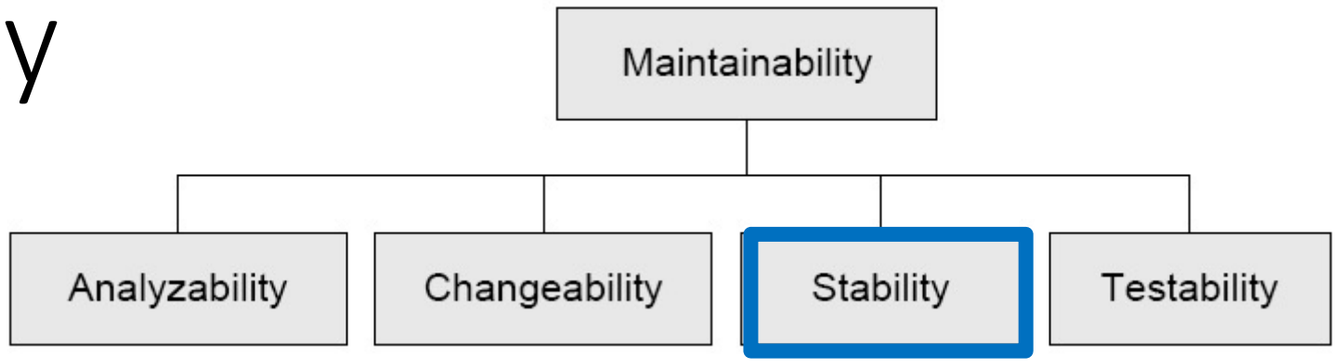


Definition: Stability (ISO 9126)

The capability of the software product to avoid unexpected effects from modifications of the system.

What are **potential (negative) effects** of modifying the code?

Maintainability



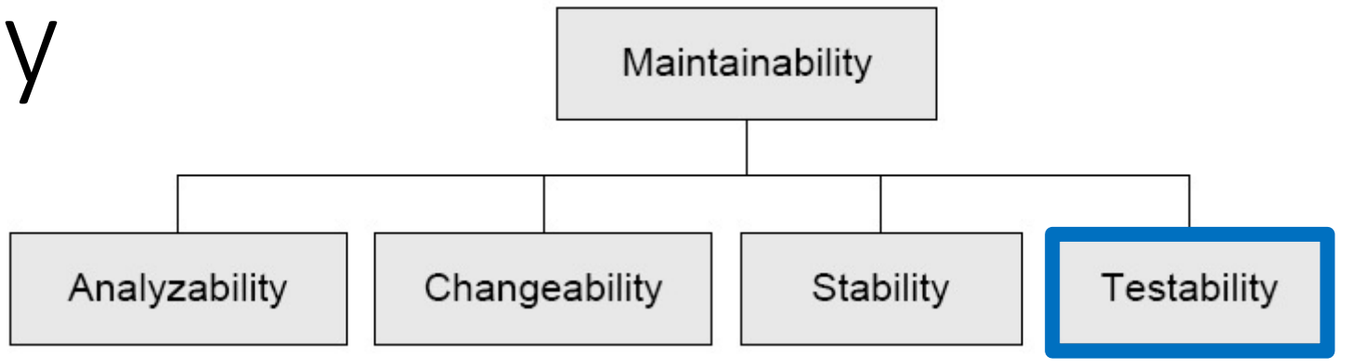
Definition: Stability (ISO 9126)

The capability of the software product to avoid unexpected effects from modifications of the system.

What are **potential (negative) effects** of modifying the code?

- Will discuss more with design patterns, integration testing, and compositional design

Maintainability

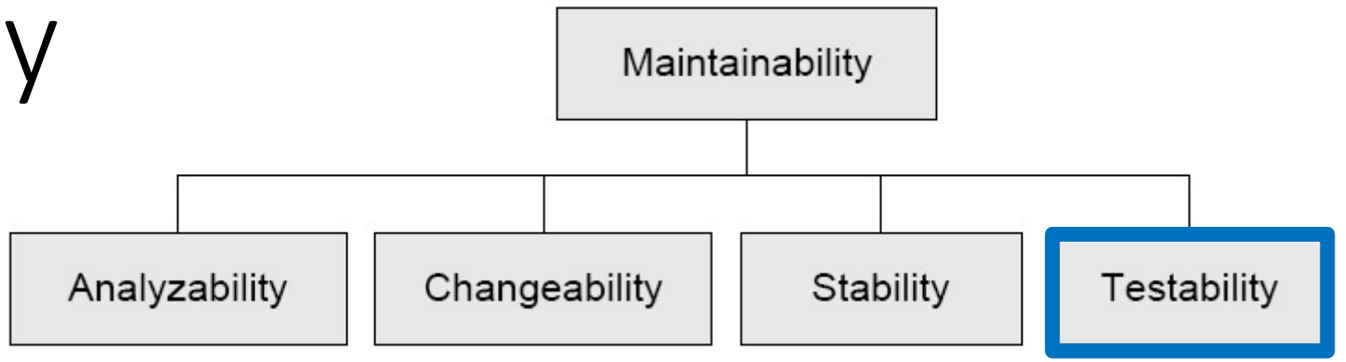


Definition: Testability (ISO 9126)

The capability of the software product to enable a modified system to be validated.

Can we **verify** the code?

Maintainability



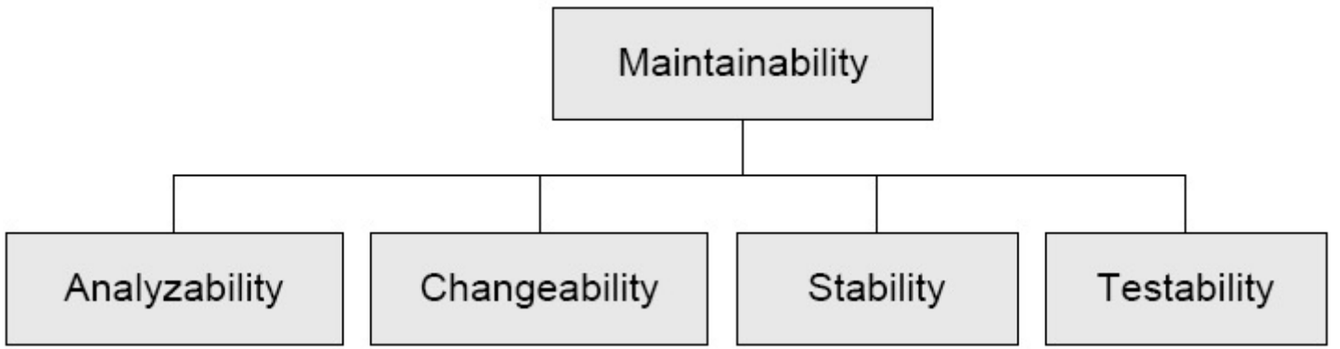
Definition: Testability (ISO 9126)

The capability of the software product to enable a modified system to be validated.

Can we **verify** the code? (with tests)

```
/** A testing tool written from scratch.
*/
public class TestDayOfWeek {
  public static void main(String[] args) {
    // Test that December 25th 2010 is Saturday
    Date d = new Date(2010, 12, 25); // year, month, day of month
    Date.Weekday weekday = d.dayOfWeek();
    if ( weekday == Date.Weekday.SATURDAY ) {
      System.out.println("Test case: Dec 25th 2010: Pass");
    } else {
      System.out.println("Test case: Dec 25th 2010: FAIL");
    }
    // ... fill in more tests
  }
}
```

Flexibility



Definition: Flexibility

The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

- A special case of **changeability**
- Also relates to **stability**

Flexibility

Definition: Coupling

Coupling is a measure of how strongly dependent one software unit is on other software units.

unit = a well delimited piece of code: class, package, module, method, application, etc.

→ **Assign responsibility so coupling is low**

- Local change has no/less impact
- Easier to understand modules in isolation
- Higher probability of reuse with few dependencies

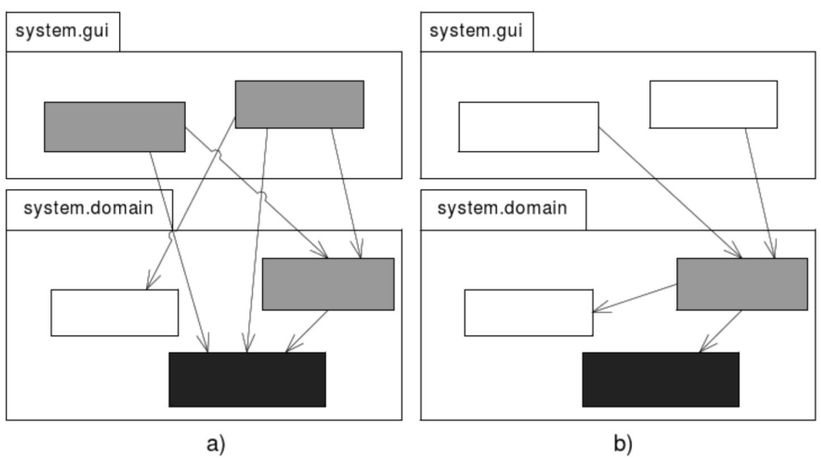


Figure 10.1: Tight (a) and low (b) coupling.

Flexibility

Definition: Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities and provided behaviors of a software unit are.

Example:

- Package *ABCClasses*: Contains all classes whose names begin with either the letters A, B, or C in my flight reservation system.
- Package *SeatBooking*: All classes related to booking a seat on a plane in my flight reservation system.

Flexibility

Definition: Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities and provided behaviors of a software unit are.

Example:

- Package *ABCClasses*: Contains all classes whose names begin with either the letters A, B, or C in my flight reservation system. ← Low cohesion
- Package *SeatBooking*: All classes related to booking a seat on a plane in my flight reservation system. ← High cohesion

→ Assign responsibility so cohesion is high

Flexibility

Maintainable software generally has **weak coupling** and **high cohesion**.

- Weak coupling means one change does not influence all other parts of the software

  → lowering cost of change

- High cohesion means that a change is likely localized in a single subsystem, easier to spot

  → lowing the cost of change

Flexibility

Maintainable software generally has **weak coupling** and **high cohesion**.

- Weak coupling means one change does not influence all other parts of the software

  → lowering cost of change

- High cohesion means that a change is likely localized in a single subsystem, easier to spot

  → lowing the cost of change

We will discuss more when we get to design patterns, compositional design, framework theory

→ **Change by addition, not by modification**

Course Project

In this course, we will practice iterative development of maintainable (and flexible) software.

→ You will code a turn-based strategy game over 8 iterations

→ Iteration 0: Civilization-like games (https://www.freecivweb.org/)

"HotCiv"

Course Project Iterations

0: Development Environment

1: Test-Driven Development (TDD)

2: TDD and Git

3: Strategy Pattern, Refactoring

4: Code Quality (w/ Code Review)

5: Test Stubs, More Patterns

6: Compositional Design

7: Blackbox Testing, Pattern Hunting

8: Frameworks and MiniDraw

HotCiv

- 2-4 players
- Each player has **cities** and **units**
- Cities produce units
- Units **move** and can **attack** opponent units and cities
- Win with the highest score

HotCiv

- 16 x 16 **tiles**
- Terrain types
- Resources: **food** and **production**

HotCiv

- 16 x 16 tiles
- Terrain types
- Resources: food and production

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Food: increases city population
Production: build units in a city

HotCiv

- Units: are produced in cities (if a city has enough **production resources**)
- Each unit has a **production cost**

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

HotCiv

- Units: are produced in cities (if a city has enough production resources)
- Each unit has a production cost
- Units may have **actions**, can move a certain number of tiles (through allowable terrain!)

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

HotCiv

- Move a Unit into another to **battle**!
- Outcome depends on each unit's defense and attack (and some probability)

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

HotCiv

- Players: Red, Blue, Yellow, Green
- Turn based:
  Red → Blue → Yellow → Green
- Each turn, a player can **move** units or **change production** of units in cities
- A player can end their turn at any time

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

HotCiv

- Completing all player turns (Red → Blue → Yellow → Green) completes one round
- Unit movement ability is **restored to maximum** each round
- At the end of a round, each city **collects food and production** according to population
- Then, the world "**ages**" (time moves forward)

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

HotCiv

- Cities produce units according to production resources from surrounding tiles
- Collecting food leads to increase in population (more people = more resources!)

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

HotCiv

- Each city is owned by a player
- A unit in the same tile as a city is "**defending**" the city
- An opposing unit attacking a city must defeat the defending unit to capture the city (the city then changes ownership)

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

HotCiv

- On their turn, a player can choose for each city:
  - Which type of unit the city is producing
  - Whether to focus on production (produce more units) or gathering food (increase population / size of the city)

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

HotCiv

You will build this across 8 iterations throughout the course!

Type	Production	Graphics	Movement
Plains	3 food		Yes
Oceans	1 food		No
Forests	3 production		Yes
Mountains	1 production		No
Hills	2 production		Yes
(City)	1 food + 1 production		Yes

Table 36.1: Data for terrain types.

Type	Cost	Distance	Graphics	Defense	Attack	Action
Archer	10	1		3	2	fortify
Legion	15	1		2	4	none
Settler	30	1		3	0	build city

Table 36.2: Data for unit types.

Iteration 0: The Development Environment

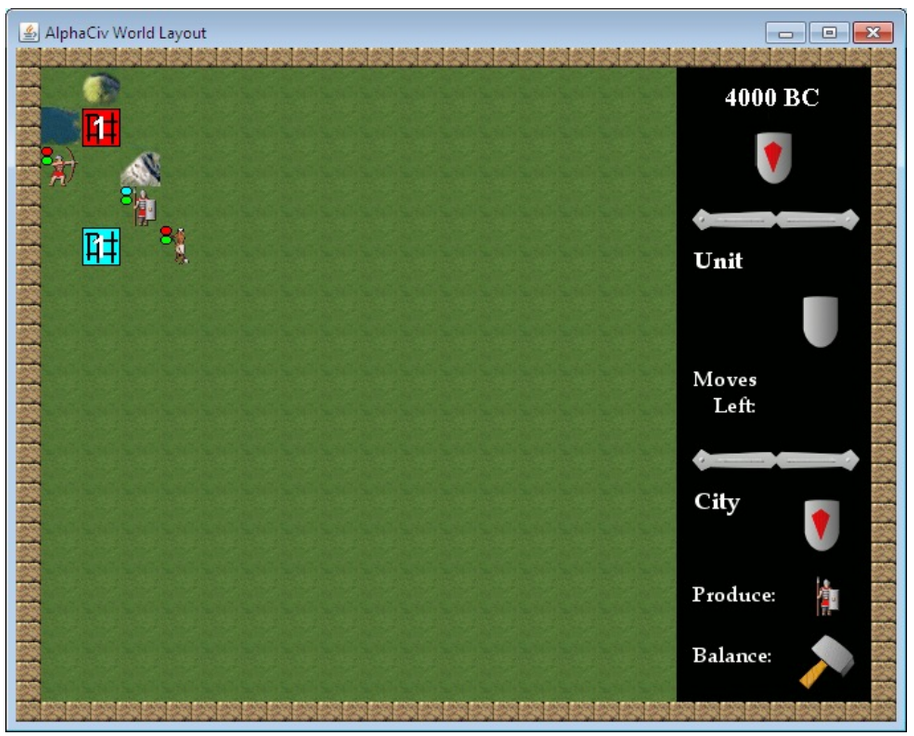
Collect your tools:

- Java
- Gradle
- Git
- IntelliJ
- HotCiv starter code

Iteration 1: Test-Driven Development I

Concepts: Test-Driven Development (more next time)

- Develop a simplified version of HotCiv (”AlphaCiv”) using Test-Driven Development
- Read 36.1 for an overview of HotCiv (mostly what we just covered)
- Read 36.2 for instructions for initial implementation (use the provided hotciv starter files)



Iteration 2: TDD II and Git

Concepts: Configuration Management, Software Deployment

- Use Git for release management, and continue practicing TDD

Iteration 3: Strategy Pattern and Refactoring

Concepts: Design Patterns, Refactoring

- Refactor your HotCiv code base to a Strategy-pattern based architecture, by using your existing test cases to refactor the code base to a new design
- Introduce new variants of the HotCiv game by implementing new Strategy implementations
  - BetaCiv, GammaCiv, DeltaCiv

Iteration 4: Code Quality

Concepts: Coding Standards and Code Review, More Refactoring

- Choose two complex aspects (a complex method, or a feature covered by a small set of methods) from your current HotCiv implementation and for each:
  - Analyze it in terms of “Clean Code” properties
  - Refactor it so it adheres as best as possible to the Clean Code properties

Iteration 5: Test Stubs and More Patterns

Concepts: Integration Testing, Variability Management

- Use test stubs to get "randomness" under automated test control; and apply the State and Abstract Factory patterns

Iteration 6: Compositional Design

Concepts: Compositional Design, Roles and Responsibility

- Apply compositional design principles, especially the ability to support multi-dimensional variance, and compare to parametric and polymorphic designs
- Generalize your HotCiv framework for new unit types

Iteration 7: Blackbox Testing and Pattern Hunting

Concepts: Systematic Testing, Code Coverage

- Use equivalence-class partitioning to develop high quality test cases, and to apply a set of design patterns to your HotCiv design

Iteration 8: Frameworks and MiniDraw

Concepts: Frameworks

- Analyze your current HotCiv system as a framework
- Add a graphical user interface using the MiniDraw framework

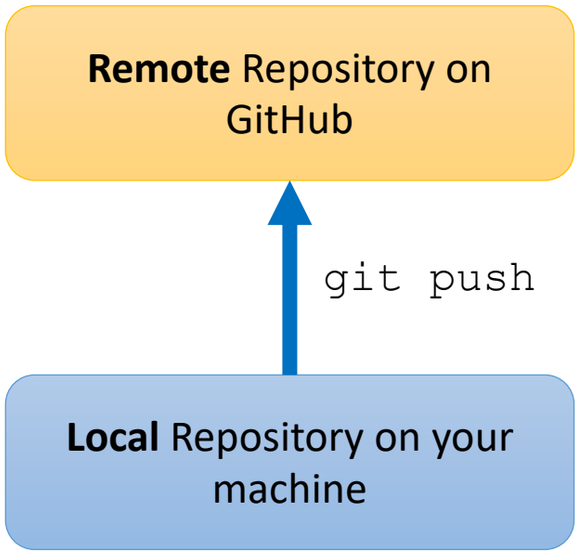
Where to Start?

- Complete Iteration 0: Environment Setup
  - Clone hotciv starter code
- Read 36.1, 36.2
- Read specifications in Game.java

Focus on learning and applying concepts from class

"65% functionality with **quality code** is much better than 100% functionality with bad code!"

Version Control: Git



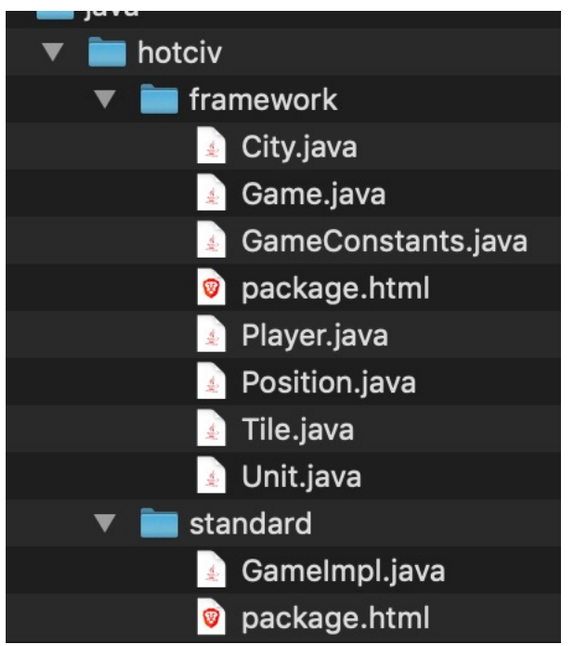
More in a couple weeks

For now, simple workflow:

```
git add <files>
git commit –m “made these changes”
```

```
git status
git add <file names or . for all files in current directory>
git status
git commit -m “<informative message>”
git status
git push
```

Object-Oriented Language: Java




```java
public interface Game {
  // === Accessor methods ==================================

  /** 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 );

  /** return the uppermost unit in the stack of units at position 'p'
   * in the world.
   * Precondition: Position p is a valid position in the world.
   * @param p the position in the world.
   * @return the unit that is at the top of the unit stack at position
   * p, OR null if no unit is present at position p.
   */
  public Unit getUnitAt( Position p );

  /** return the city at position 'p' in the world.
   * Precondition: Position p is a valid position in the world.
   * @param p the position in the world.
   * @return the city at this position or null if no city here.
   */
  public City getCityAt( Position p );

  /** return the player that is 'in turn', that is, is able to
   * move units and manage cities.
   * @return the player that is in turn
   */
  public Player getPlayerInTurn();

  /** return the player that has won the game.
```

Game.java

HotCiv Starter Code

Starter code: Provided interfaces

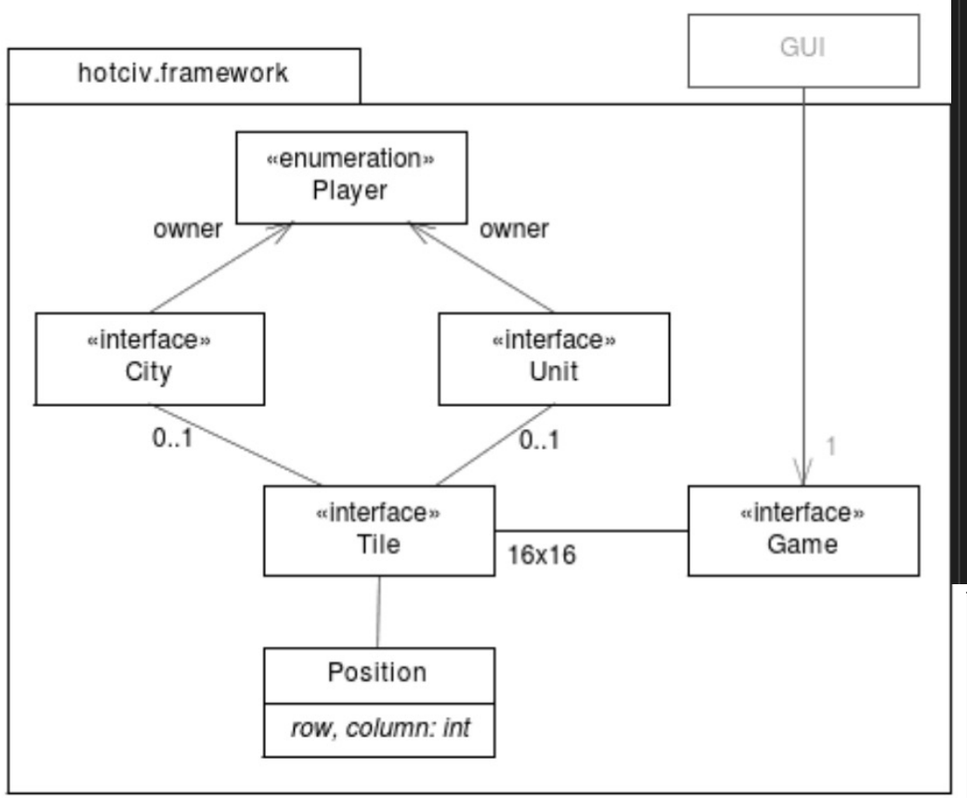


Figure 36.3: HotCiv central abstractions.

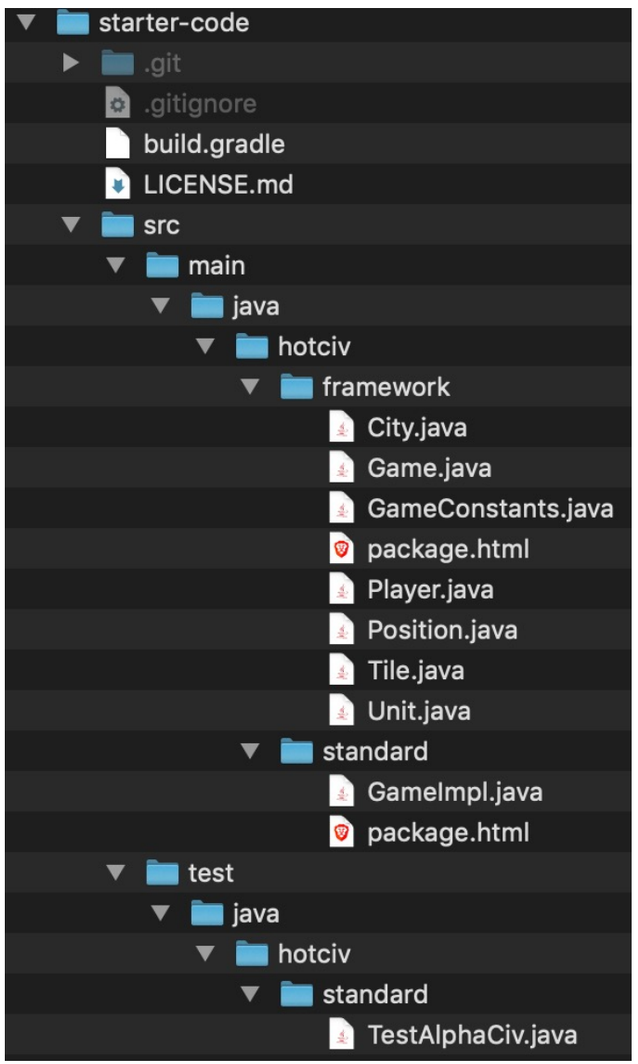
```java
public interface Game {
  // === Accessor methods ================================

  /** 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 );

  /** return the uppermost unit in the stack of units at position 'p'
   * in the world.
   * Precondition: Position p is a valid position in the world.
   * @param p the position in the world.
   * @return the unit that is at the top of the unit stack at position
   * p, OR null if no unit is present at position p.
   */
  public Unit getUnitAt( Position p );

  /** return the city at position 'p' in the world.
   * Precondition: Position p is a valid position in the world.
   * @param p the position in the world.
   * @return the city at this position or null if no city here.
   */
  public City getCityAt( Position p );

  /** return the player that is 'in turn', that is, is able to
   * move units and manage cities.
   * @return the player that is in turn
   */
  public Player getPlayerInTurn();

  /** return the player that has won the game.
```

Game.java

Game

- Knows the world, allows access to individual tiles
- Allows access to cities
- Allows access to units
- Knows which player is in turn
- Allows moving a unit, handles attack, and refuses invalid moves
- Allows performing a unit's associated action
- Allows changing production in a city
- Allows changing workforce balance in a city
- Determines if a winner has been found
- Performs "end of round" (city growth, unit production, etc.)

Build Management: Gradle

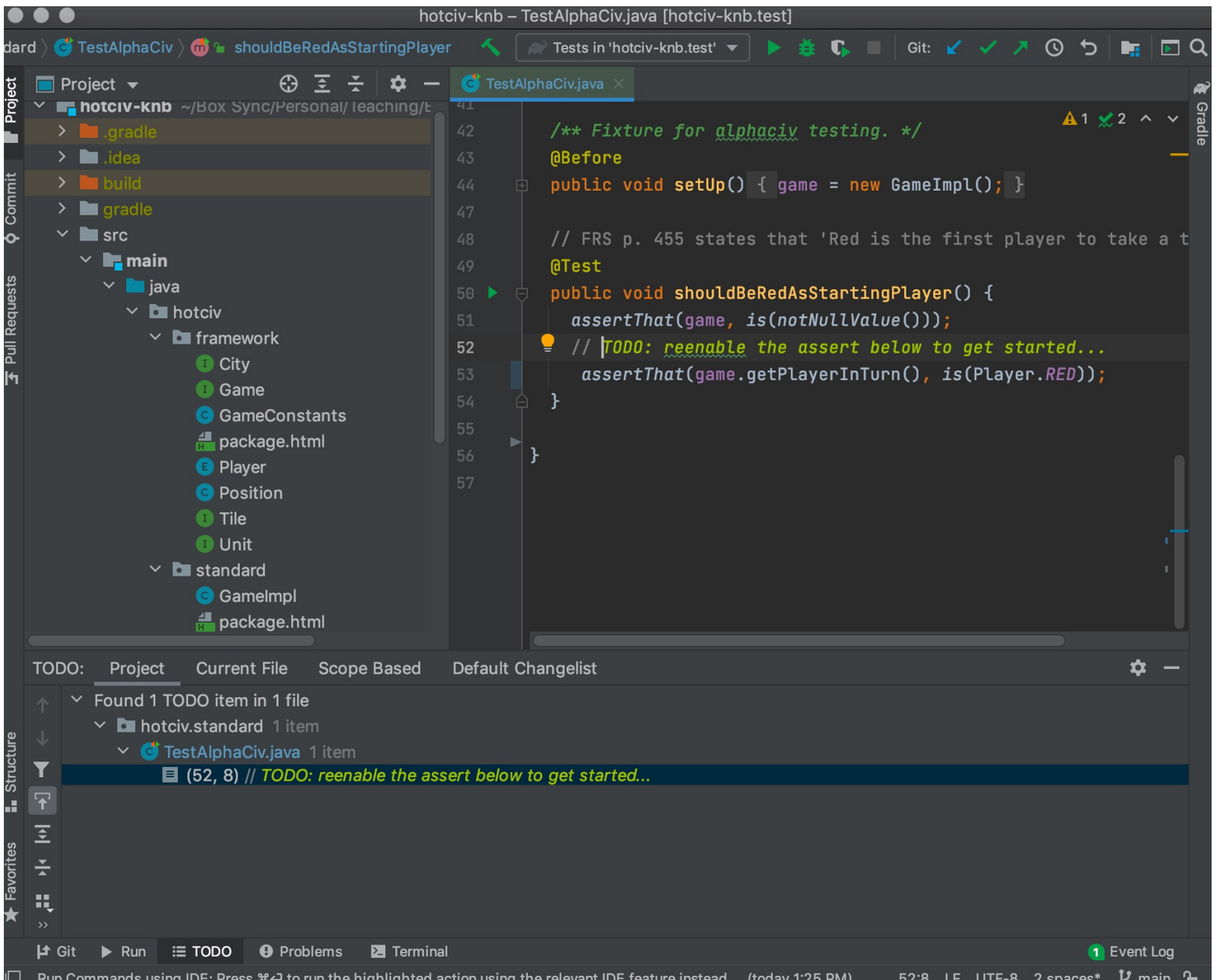


build.gradle

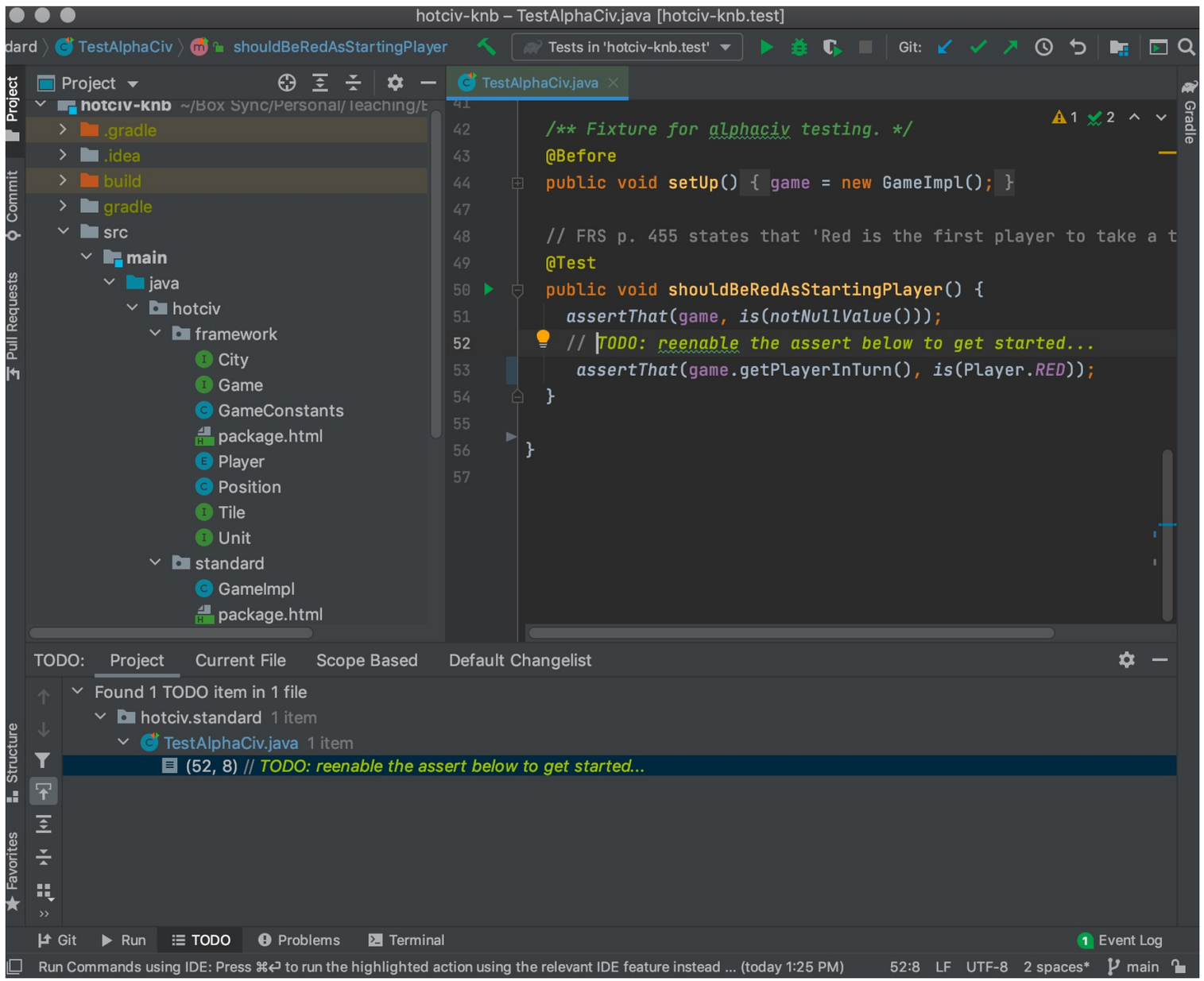
```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

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

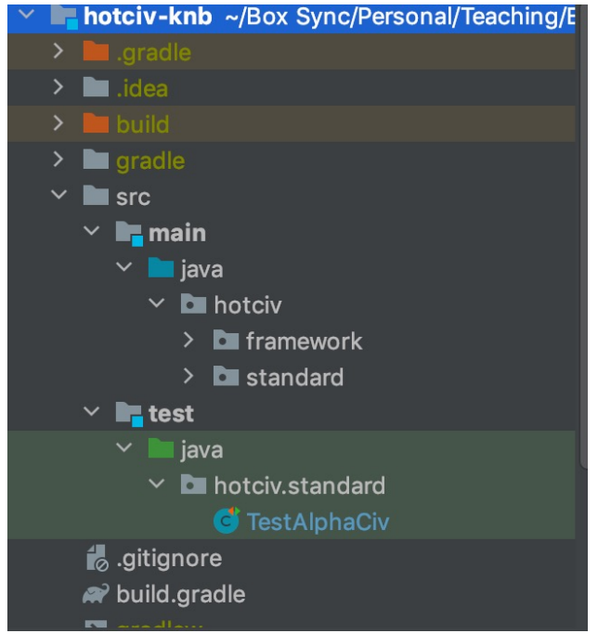
More later

Development: IntelliJ

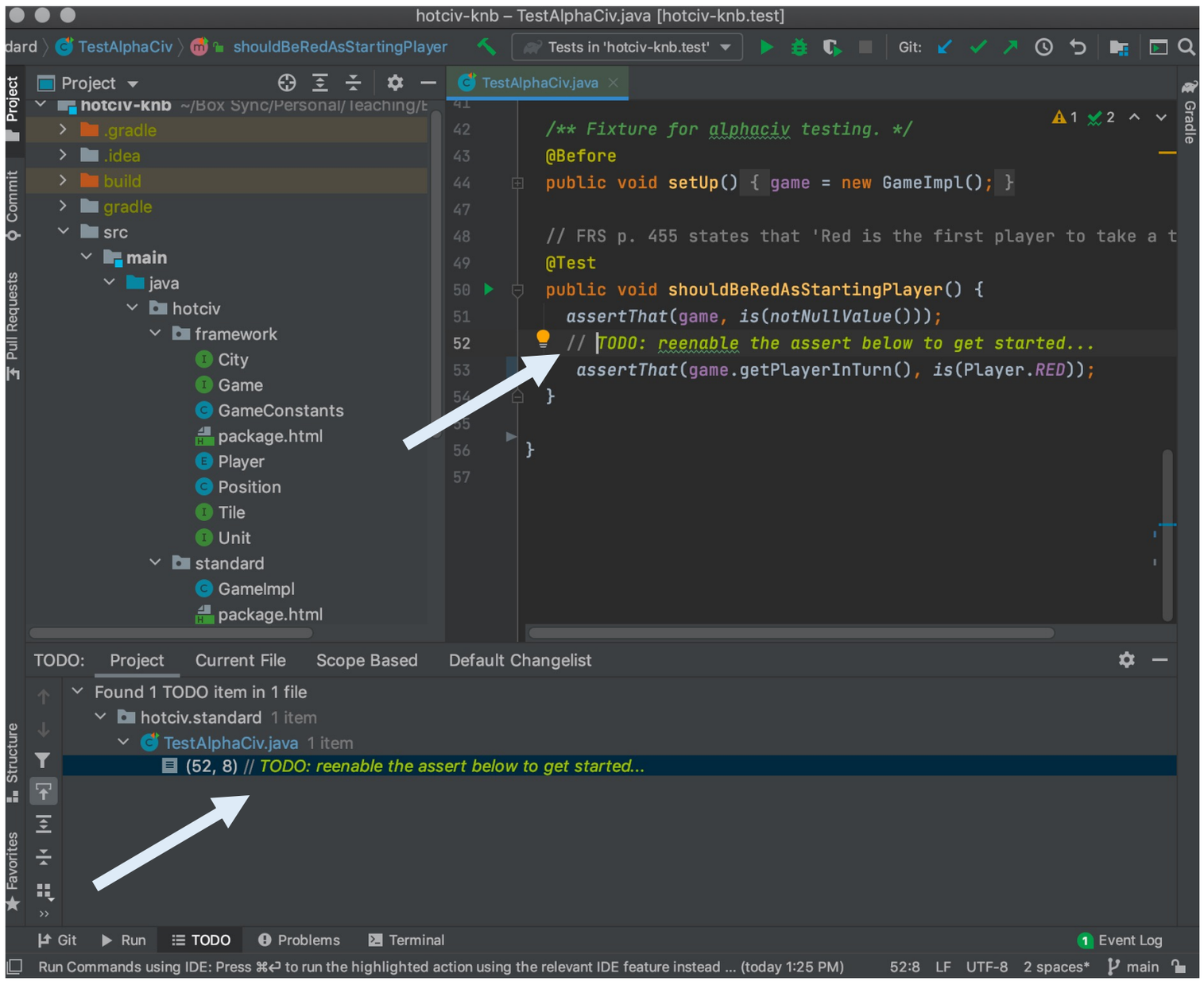


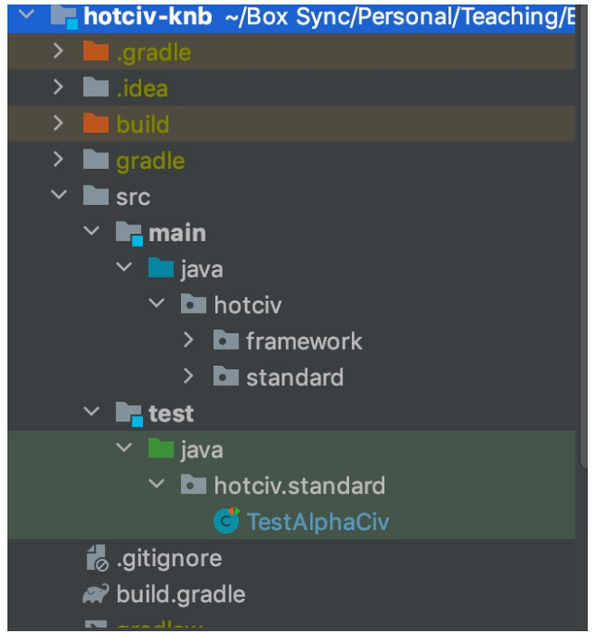
Development: IntelliJ



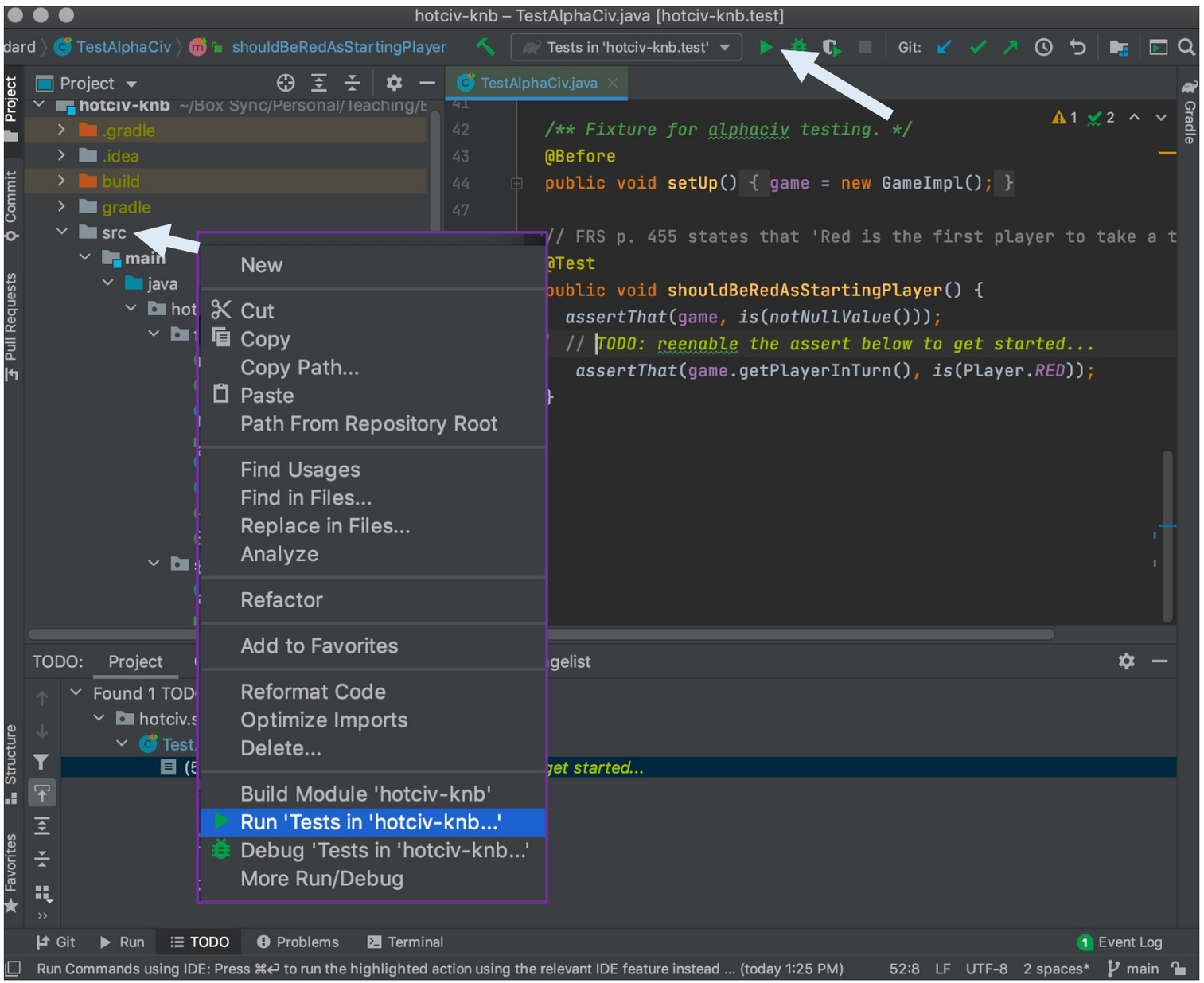


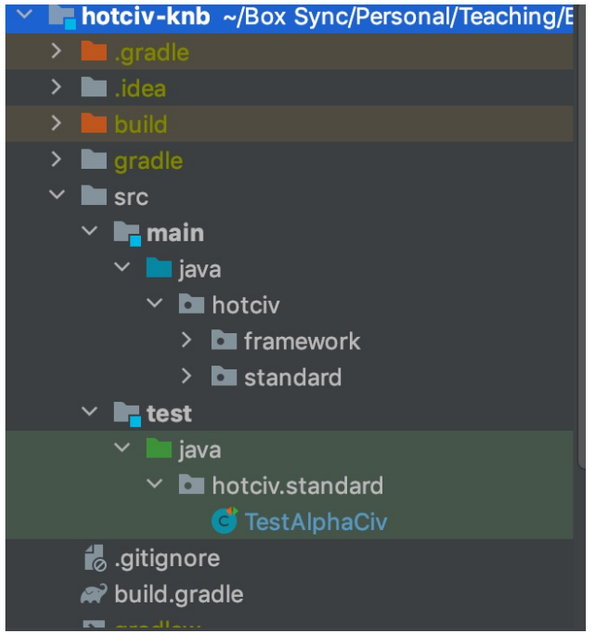
Development: IntelliJ



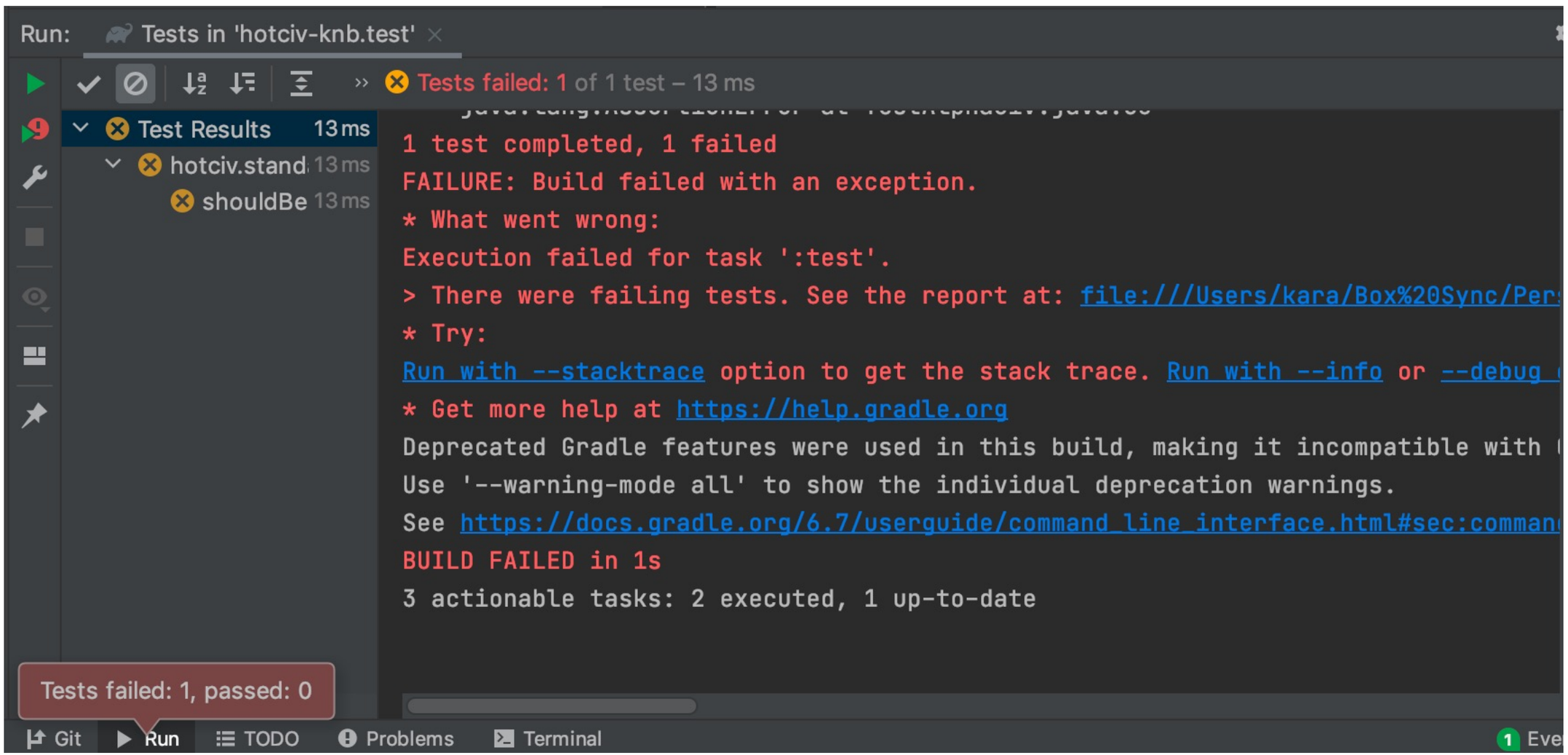


Development: IntelliJ

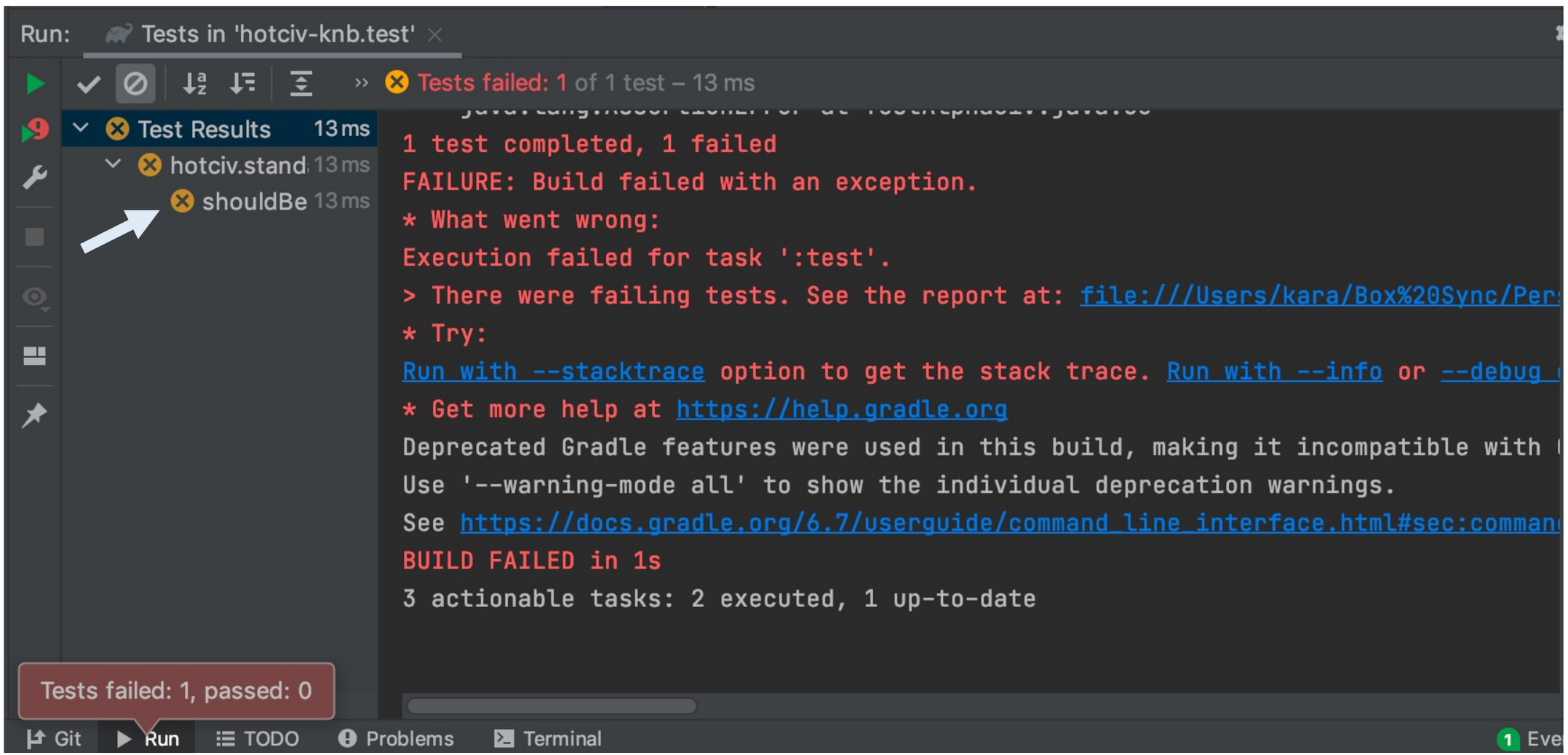




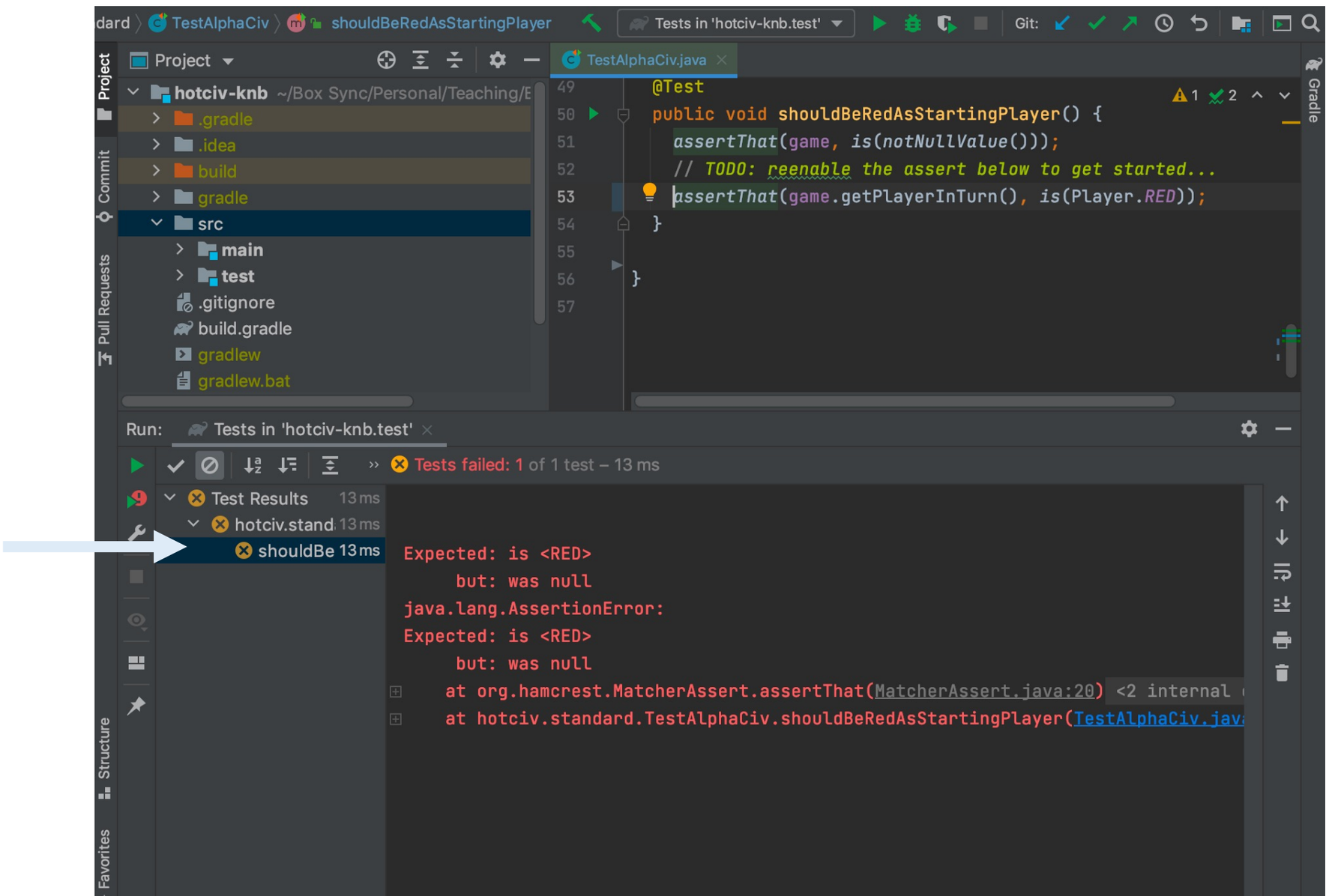
Development: IntelliJ



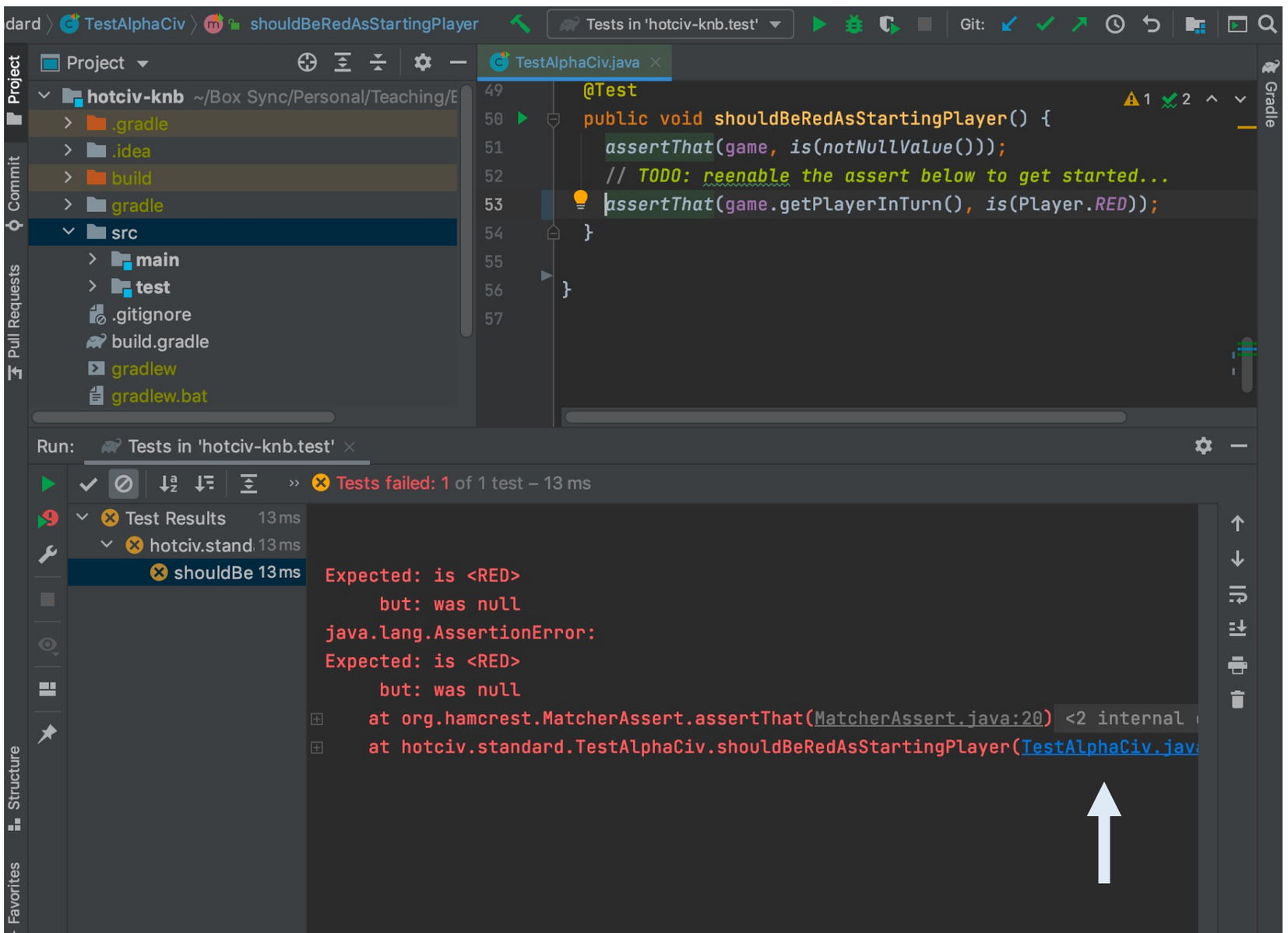
Development: IntelliJ



Development: IntelliJ



Development: IntelliJ



Testing: JUnit / Hamcrest

```java
/** REMOVE ME. Not a test of HotCiv, just an example of what
    matchers the hamcrest library has... */
@Test
public void shouldDefinetelyBeRemoved() {
  // Matching null and not null values
  // 'is' require an exact match
  String s = null;
  assertThat(s, is(nullValue()));
  s = "Ok";
  assertThat(s, is(notNullValue()));
  assertThat(s, is( value: "Ok"));

  // If you only validate substrings, use containsString
  assertThat( actual: "This is a dummy test", containsString( substring: "dummy"));

  // Match contents of Lists
  List<String> l = new ArrayList<String>();
  l.add("Bimse");
  l.add("Bumse");
  // Note - ordering is ignored when matching using hasItems
  assertThat(l, hasItems(new String[] {"Bumse","Bimse"}));

  // Matchers may be combined, like is-not
  assertThat(l.get(0), is(not( value: "Bumse")));
}
}
```

← Annotated method defines a test case, all methods marked constitute a test suite

Use assert methods for comparing output with expected output

Example: Testing

Suppose we are developing a calendar system: the class Date represents … dates!

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *              januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
  */
  public Date(int year, int month, int dayOfMonth) {
```

Example: Testing

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *              januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
   */
  public Date(int year, int month, int dayOfMonth) {
```

Suppose we want to add a method dayOfWeek to calculate the weekday for any given date. How do we make sure it works?

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY,
    SATURDAY, SUNDAY };
/**
* Calculate the weekday that this Date object represents.
* @return the weekday of this date.
*/
public Weekday dayOfWeek() {
```

Example: Testing

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *              januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
   */
  public Date(int year, int month, int dayOfMonth) {
```

Suppose we want to add a method dayOfWeek to calculate the weekday for any given date. How do we make sure it works?

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY,
    SATURDAY, SUNDAY };
/**
* Calculate the weekday that this Date object represents.
* @return the weekday of this date.
*/
public Weekday dayOfWeek() {
```

Definition: Test case

A test case is a definition of input values and expected output values for a unit under test.

```
Date d = new Date(2008, 5, 19); // 19th May 2008
// weekday should be MONDAY
Date.Weekday weekday = d.dayOfWeek();
```

Example: Testing

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *              januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
   */
  public Date(int year, int month, int dayOfMonth) {
```

Unit under test Test inputs Expected output

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

Example: Testing

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *              januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
   */
  public Date(int year, int month, int dayOfMonth) {
```

Unit under test Test inputs Expected output

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

Definition: Test suite

A test suite is a set of test cases.

Example: Testing

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

Manual testing: Compare output to expected output

Automated testing: Verify automatically using a computer program

Example: Testing

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

```
/** A testing tool written from scratch.
*/
public class TestDayOfWeek {
  public static void main(String[] args) {
    // Test that December 25th 2010 is Saturday
    Date d = new Date(2010, 12, 25); // year, month, day of month
    Date.Weekday weekday = d.dayOfWeek();
    if ( weekday == Date.Weekday.SATURDAY ) {
      System.out.println("Test case: Dec 25th 2010: Pass");
    } else {
      System.out.println("Test case: Dec 25th 2010: FAIL");
    }
    // ... fill in more tests
  }
}
```

Example: Testing

Definition: Production code

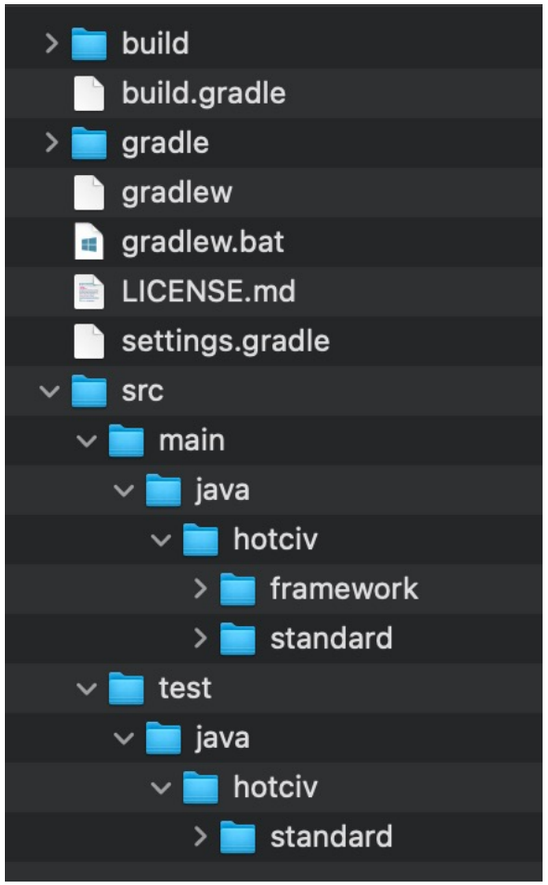
The production code is the code that defines the behavior implementing the software's requirements.

Definition: Test code

The test code is the source code that defines test cases for the production code.

Definition: Regression testing

Regression testing is the repeated execution of test suites to ensure they still pass and the system does not fail after a modification.



Example: Testing

For the sake of example, we ignored the built-in **Date** class – in reality it is best practice to utilize existing software units, because they have (usually) been thoroughly tested

Testing: JUnit

Unit testing tools support executing large test suites, reporting, and finding test cases that fail

Example: JUnit

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

/** Testing dayOfWeek using the JUnit 4.x framework.
*/
public class TestDayOfWeek {

  /**
   * Test that December 25th 2010 is Saturday
   */
  @Test ← Annotated method defines a test case, all methods marked constitute a test suite
  public void shouldGiveSaturdayFor25Dec2010() {
    Date date = new Date( 2010, 12, 25);
    assertEquals( "Dec 25th 2010 is Saturday",
                  Date.Weekday.SATURDAY, date.dayOfWeek() );
  }
}
```

Testing: JUnit

Unit testing tools support executing large test suites, reporting, and finding test cases that fail

Example: JUnit

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

/** Testing dayOfWeek using the JUnit 4.x framework.
*/
public class TestDayOfWeek {

  /**
   * Test that December 25th 2010 is Saturday
   */
  @Test
  public void shouldGiveSaturdayFor25Dec2010() {
    Date date = new Date( 2010, 12, 25);
    assertEquals( "Dec 25th 2010 is Saturday",
                  Date.Weekday.SATURDAY, date.dayOfWeek() );
  }
}
```

^ Use assert methods for comparing output with expected output

Testing: JUnit

assert	Pass if:
assertTrue(boolean b)	expression b is true
assertFalse(boolean b)	expression b is false
assertNull(Object o)	object o is null
assertNotNull(Object o)	object o is not null
assertEquals(double e, double c, double delta)	e and c are equal to within a positive delta
assertEquals(Object[] e, Object[]c)	object arrays are equal

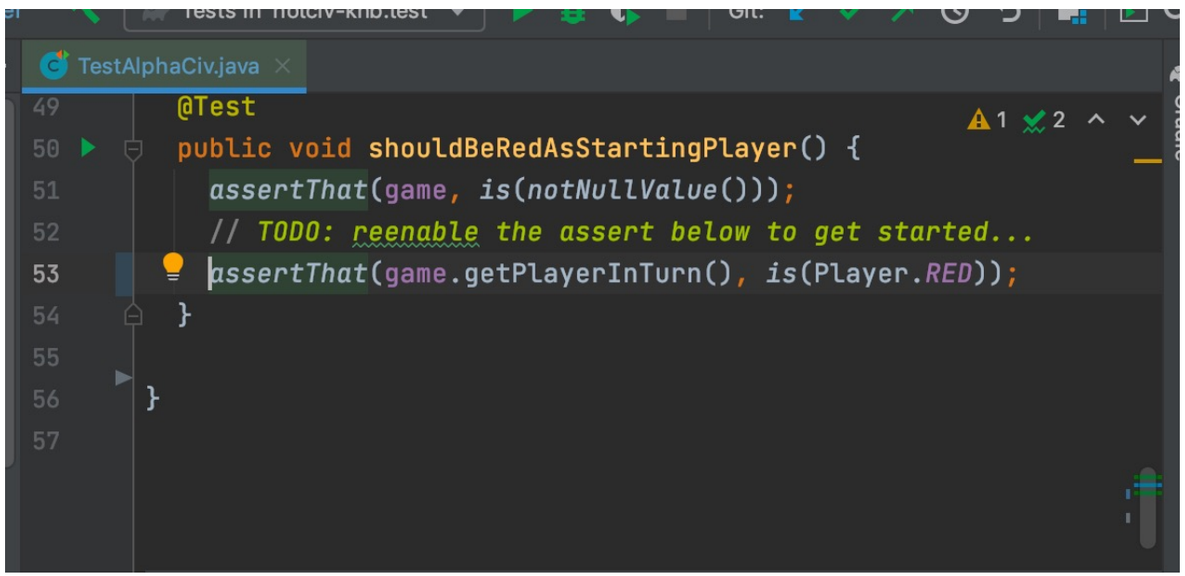
Unit testing tools support executing large test suites, reporting, and finding test cases that fail

Example: JUnit

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

/** Testing dayOfWeek using the JUnit 4.x framework.
*/
public class TestDayOfWeek {

  /**
   * Test that December 25th 2010 is Saturday
   */
  @Test
  public void shouldGiveSaturdayFor25Dec2010() {
    Date date = new Date( 2010, 12, 25);
    assertEquals( "Dec 25th 2010 is Saturday",
                  Date.Weekday.SATURDAY, date.dayOfWeek() );
  }
}
```

^ Use assert methods for comparing output with expected output

Testing: JUnit / Hamcrest



http://hamcrest.org/JavaHamcrest/tutorial

Core

`anything` - always matches, useful if you don't care what the object under test is

`describedAs` - decorator to adding custom failure description

`is` - decorator to improve readability - see "Sugar", below

Logical

`allOf` - matches if all matchers match, short circuits (like Java &&)

`anyOf` - matches if any matchers match, short circuits (like Java ||)

`not` - matches if the wrapped matcher doesn't match and vice versa

Object

`equalTo` - test object equality using Object.equals

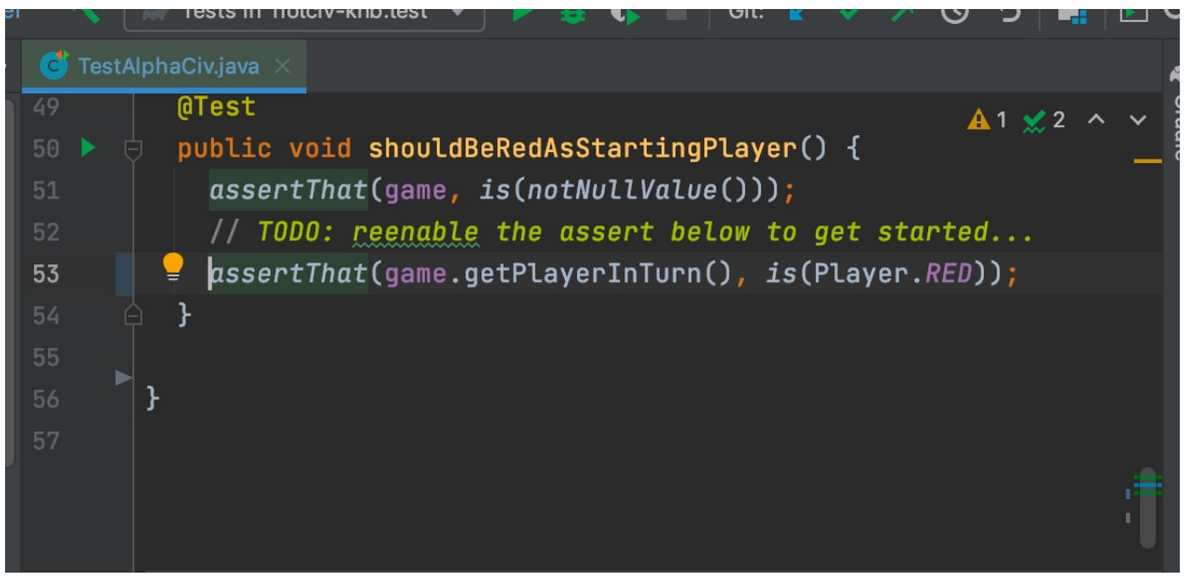
`hasToString` - test Object.toString

`instanceOf`, `isCompatibleType` - test type

`notNullValue`, `nullValue` - test for null

`sameInstance` - test object identity

Testing: JUnit / Hamcrest



Core

`anything` - always matches, useful if you don't care what the object under test is

`describedAs` - decorator to adding custom failure description

`is` - decorator to improve readability - see "Sugar", below

Logical

`allOf` - matches if all matchers match, short circuits (like Java &&)

`anyOf` - matches if any matchers match, short circuits (like Java ||)

`not` - matches if the wrapped matcher doesn't match and vice versa

Object

`equalTo` - test object equality using Object.equals

`hasToString` - test Object.toString

`instanceOf`, `isCompatibleType` - test type

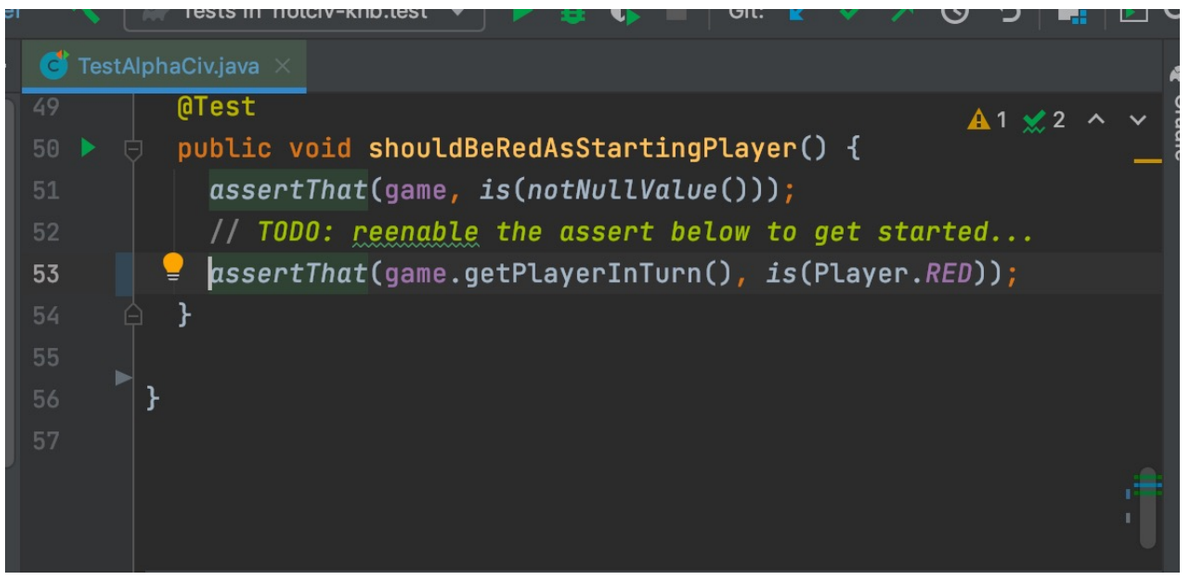
`notNullValue`, `nullValue` - test for null

`sameInstance` - test object identity

http://hamcrest.org/JavaHamcrest/tutorial

Keep test code **simple**, short and easy to read.
Avoid loops and conditions!

Testing: JUnit / Hamcrest



Core

`anything` - always matches, useful if you don't care what the object under test is

`describedAs` - decorator to adding custom failure description

`is` - decorator to improve readability - see "Sugar", below

Logical

`allOf` - matches if all matchers match, short circuits (like Java &&)

`anyOf` - matches if any matchers match, short circuits (like Java ||)

`not` - matches if the wrapped matcher doesn't match and vice versa

Object

`equalTo` - test object equality using Object.equals

`hasToString` - test Object.toString

`instanceOf`, `isCompatibleType` - test type

`notNullValue`, `nullValue` - test for null

`sameInstance` - test object identity

http://hamcrest.org/JavaHamcrest/tutorial

Keep test code **simple**, short and easy to read.

Avoid loops and conditions!

Next time: Test-Driven Development