AARHUS UNIVERSITET

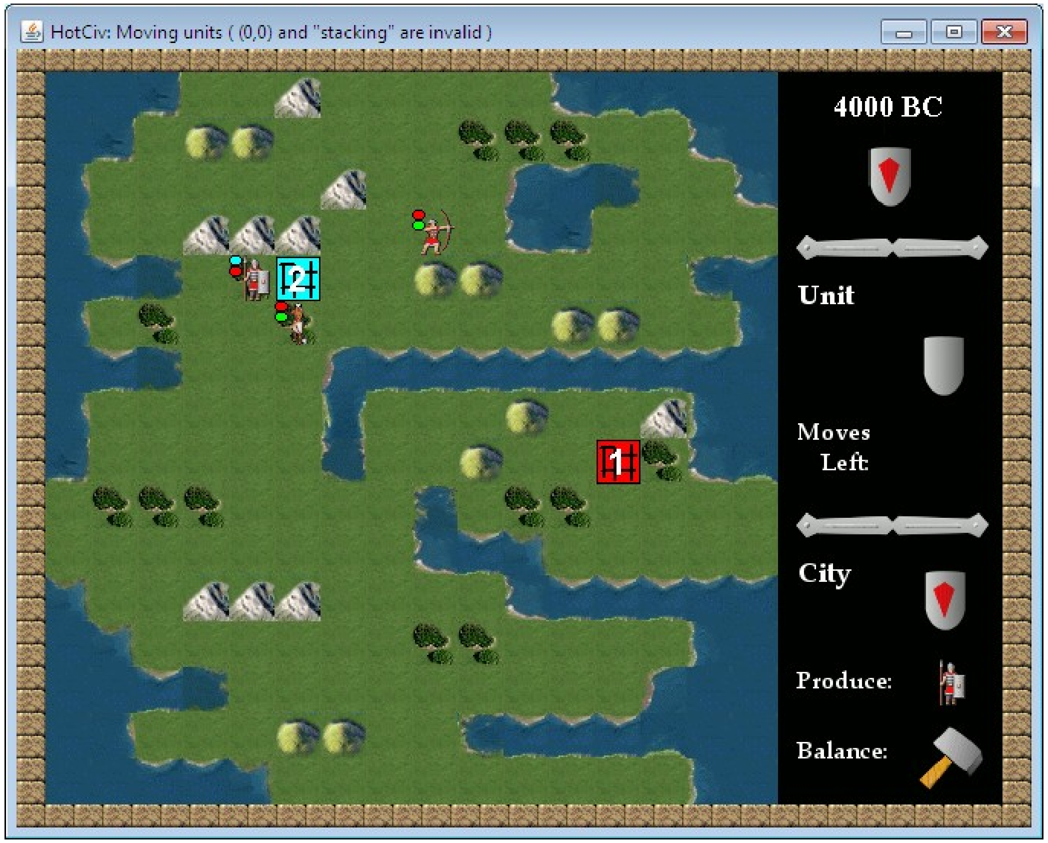
HotCiv Project

Starting up!

HotCiv = Agile development

Iterations of

- product development
- learning increments



Getting Started

Read Chapter 36.1 + 36.2

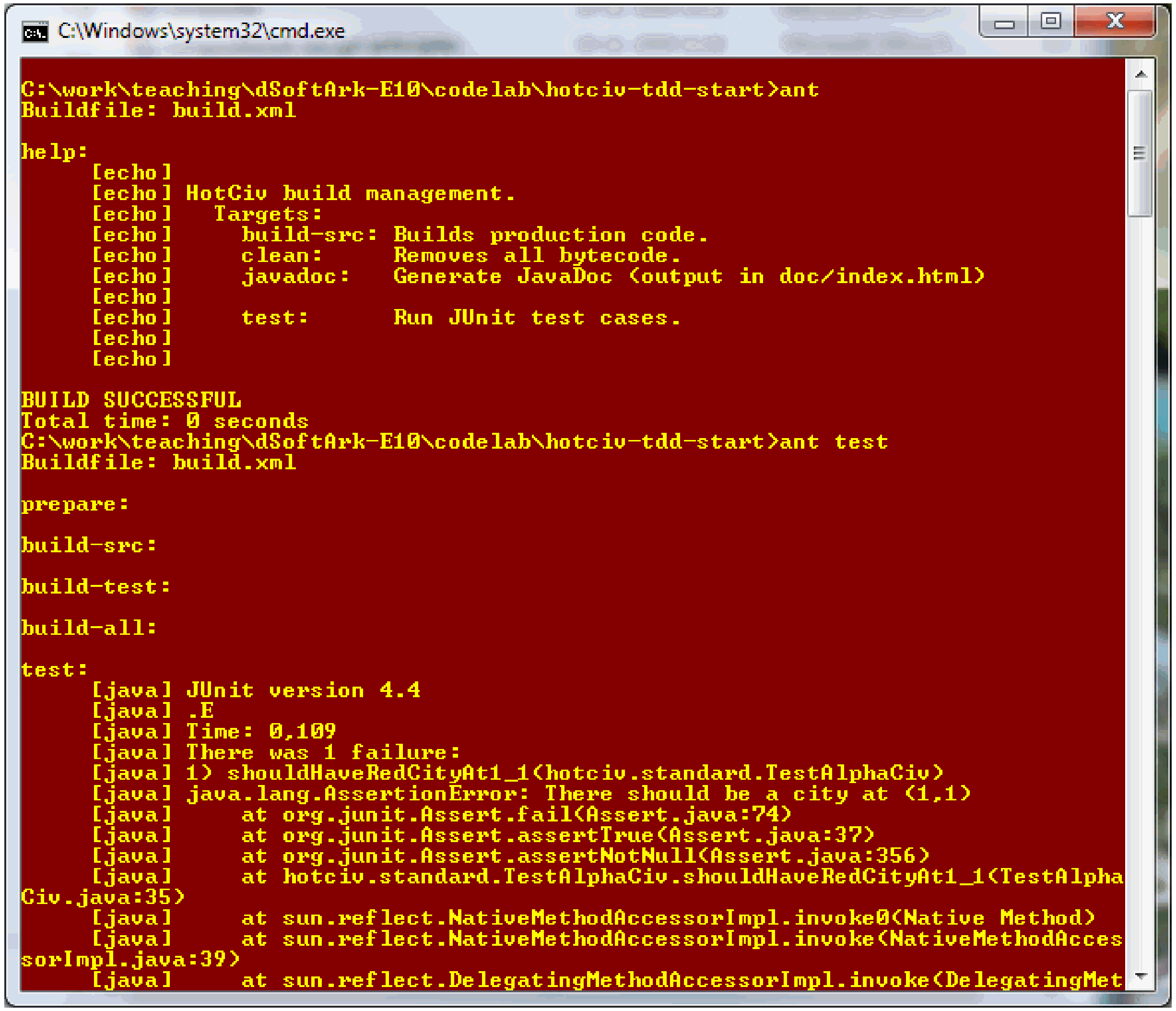
Download hotciv-tdd-start.zip and unzip it!

Read the java file: Game.java

Source code

- your variant is already fully package'ified
  - hotciv.framework etc. packages
  - split into a production code and test code source tree
- your variant is already fully Ant'ified
  - 'ant test' runs out-of-the-box

[Demo]



```
C:\work\teaching\dSoftArk-E10\codelab\hotciv-tdd-start>ant
Buildfile: build.xml

help:
     [echo]
     [echo] HotCiv build management.
     [echo]   Targets:
     [echo]     build-src: Builds production code.
     [echo]     clean:     Removes all bytecode.
     [echo]     javadoc:   Generate JavaDoc (output in doc/index.html)
     [echo]
     [echo]     test:      Run JUnit test cases.
     [echo]
     [echo]

BUILD SUCCESSFUL
Total time: 0 seconds
C:\work\teaching\dSoftArk-E10\codelab\hotciv-tdd-start>ant test
Buildfile: build.xml

prepare:

build-src:

build-test:

build-all:

test:
     [java] JUnit version 4.4
     [java] .E
     [java] Time: 0,109
     [java] There was 1 failure:
     [java] 1) shouldHaveRedCityAt1_1(hotciv.standard.TestAlphaCiv)
     [java] java.lang.AssertionError: There should be a city at (1,1)
     [java]     at org.junit.Assert.fail(Assert.java:74)
     [java]     at org.junit.Assert.assertTrue(Assert.java:37)
     [java]     at org.junit.Assert.assertNotNull(Assert.java:356)
     [java]     at hotciv.standard.TestAlphaCiv.shouldHaveRedCityAt1_1(TestAlpha
Civ.java:35)
     [java]     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     [java]     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
sorImpl.java:39)
     [java]     at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMet
```

Iteration 1

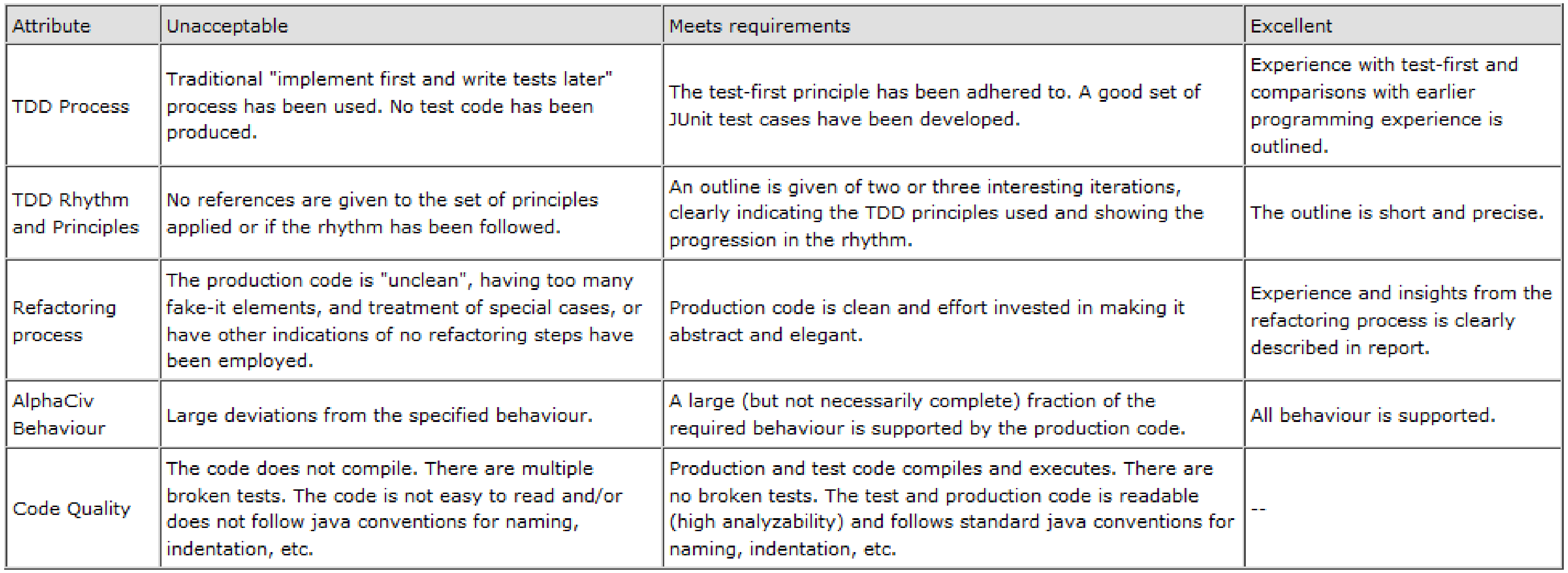
Implement AlphaCiv

- *Players.* There are exactly two players, Red and Blue.
- *World Layout.* The world looks exactly like shown in figure 36.2. That is the layout of terrain is fixed in every game, all tiles are of type "plains" *except* for tile(1,0) = Ocean, tile (0,1) = Hills, tile (2,2) = Mountains.
- *Units.* Only one unit is allowed on a tile at any time. Red has initially one archer at (2,0), Blue has one legion at (3,2), and Red a settler at (4,3).
- *Attacking.* Attacks are resolved like this: The attacking unit always wins no matter what the defensive or attacking strengths are of either units.
- *Unit actions.* No associated actions are supported by any unit. Specifically, the settler's action does nothing.
- *Cities.* The player may select to produce either archers, legions, or settlers. Cities do not grow but stay at population size 1. Cities produce 6 production per round which is a fixed setup. Red has a city at position (1,1) while blue has one at position (4,1).
- *Unit Production.* When a city has accumulated enough production it produces the unit selected for production, and the unit's cost is deducted from the city's treasury of production. The unit is placed on the city tile if no other unit is present, otherwise it is placed on the first non-occupied adjacent tile, starting from the tile just north of the city and moving clockwise.
- *Aging.* The game starts at age 4000 BC, and each round advances the game age 100 years.
- *Winning.* Red wins in year 3000 BC.

AARHUS UNIVERSITET

... And write a report on it...

Refer to the ”rubrics” to see what I deem important...

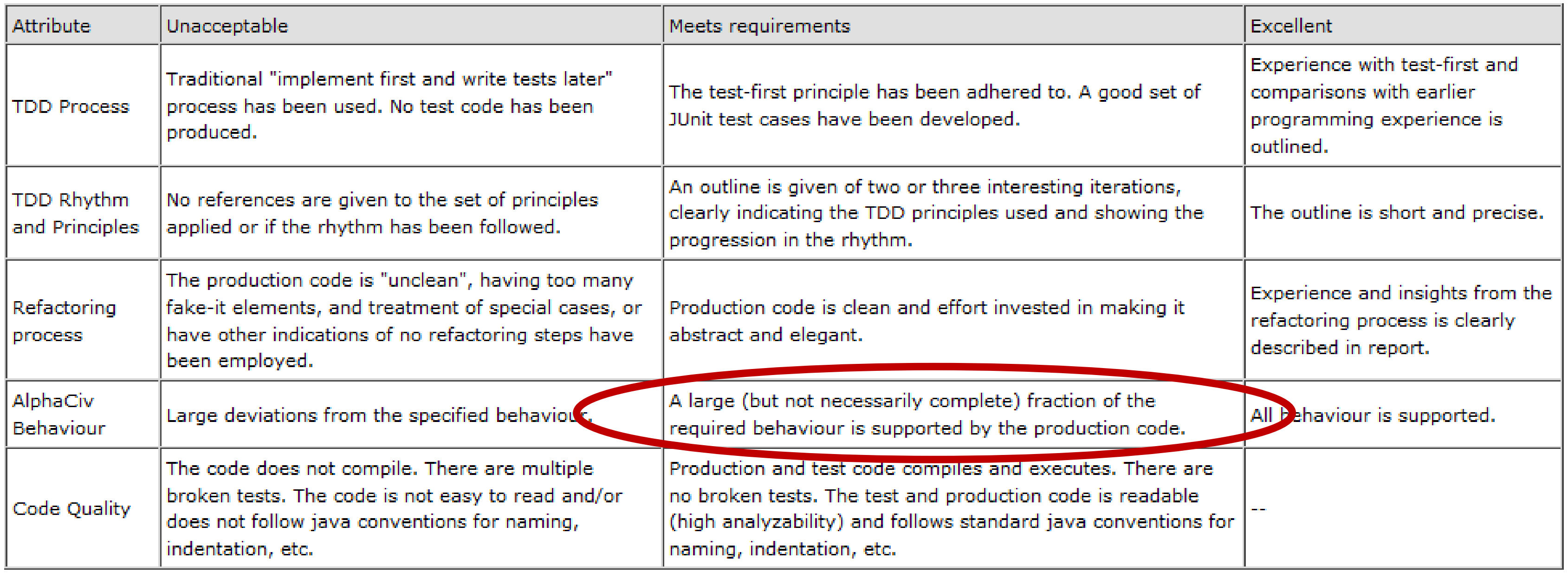


Attribute	Unacceptable	Meets requirements	Excellent
TDD Process	Traditional "implement first and write tests later" process has been used. No test code has been produced.	The test-first principle has been adhered to. A good set of JUnit test cases have been developed.	Experience with test-first and comparisons with earlier programming experience is outlined.
TDD Rhythm and Principles	No references are given to the set of principles applied or if the rhythm has been followed.	An outline is given of two or three interesting iterations, clearly indicating the TDD principles used and showing the progression in the rhythm.	The outline is short and precise.
Refactoring process	The production code is "unclean", having too many fake-it elements, and treatment of special cases, or have other indications of no refactoring steps have been employed.	Production code is clean and effort invested in making it abstract and elegant.	Experience and insights from the refactoring process is clearly described in report.
AlphaCiv Behaviour	Large deviations from the specified behaviour.	A large (but not necessarily complete) fraction of the required behaviour is supported by the production code.	All behaviour is supported.
Code Quality	The code does not compile. There are multiple broken tests. The code is not easy to read and/or does not follow java conventions for naming, indentation, etc.	Production and test code compiles and executes. There are no broken tests. The test and production code is readable (high analyzability) and follows standard java conventions for naming, indentation, etc.	--

The Agile backlog...

This is an **architecture** course!

- Clean code that works (75% clean > 100% unclean!)



Attribute	Unacceptable	Meets requirements	Excellent
TDD Process	Traditional "implement first and write tests later" process has been used. No test code has been produced.	The test-first principle has been adhered to. A good set of JUnit test cases have been developed.	Experience with test-first and comparisons with earlier programming experience is outlined.
TDD Rhythm and Principles	No references are given to the set of principles applied or if the rhythm has been followed.	An outline is given of two or three interesting iterations, clearly indicating the TDD principles used and showing the progression in the rhythm.	The outline is short and precise.
Refactoring process	The production code is "unclean", having too many fake-it elements, and treatment of special cases, or have other indications of no refactoring steps have been employed.	Production code is clean and effort invested in making it abstract and elegant.	Experience and insights from the refactoring process is clearly described in report.
AlphaCiv Behaviour	Large deviations from the specified behaviour.	A large (but not necessarily complete) fraction of the required behaviour is supported by the production code.	All behaviour is supported.
Code Quality	The code does not compile. There are multiple broken tests. The code is not easy to read and/or does not follow java conventions for naming, indentation, etc.	Production and test code compiles and executes. There are no broken tests. The test and production code is readable (high analyzability) and follows standard java conventions for naming, indentation, etc.	--

- You may fill in more behaviour in following iterations...

... And hand it in...

... Using course admin

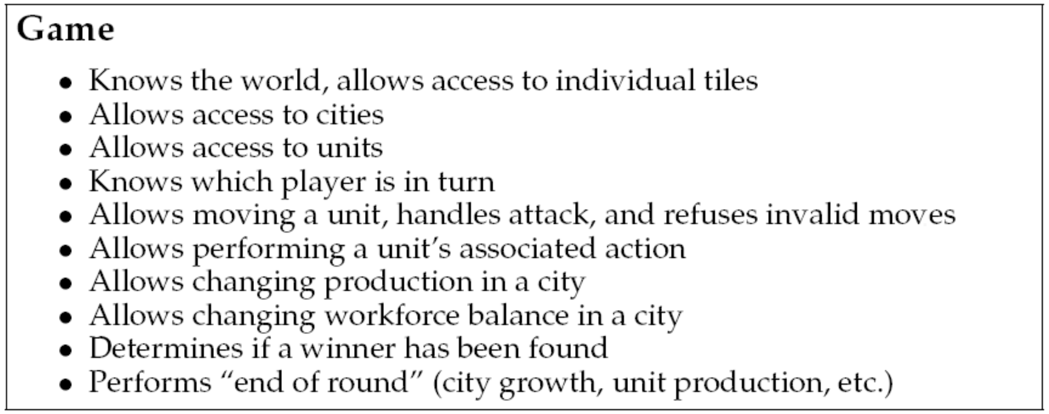
Two files

- R.pdf: Report in pdf format
- C.zip: Zip with ALL code
  - TA should ”unzip; ant test;” and it should work...

... In details...

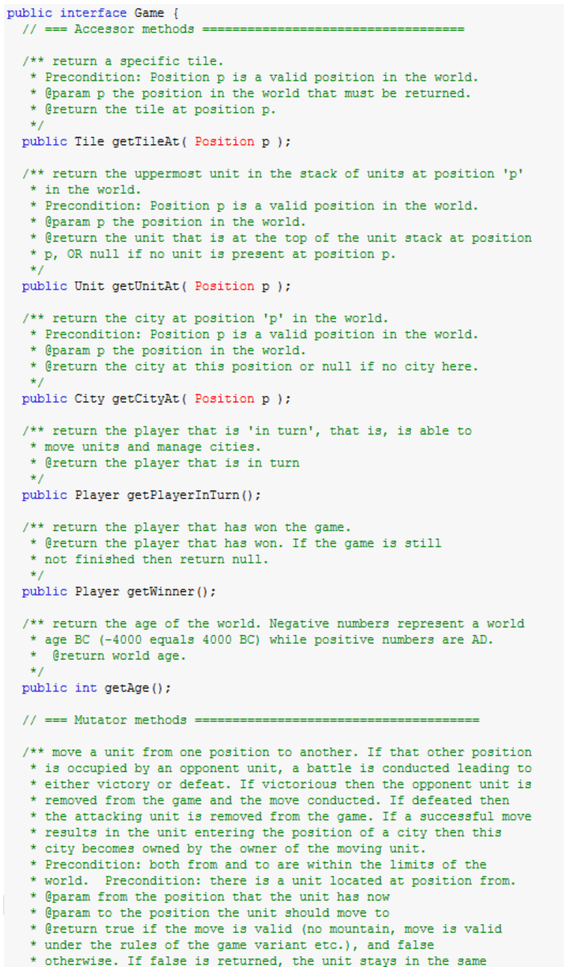
… at the code level

This boils down to implementing *relevant* methods for the Game interface:



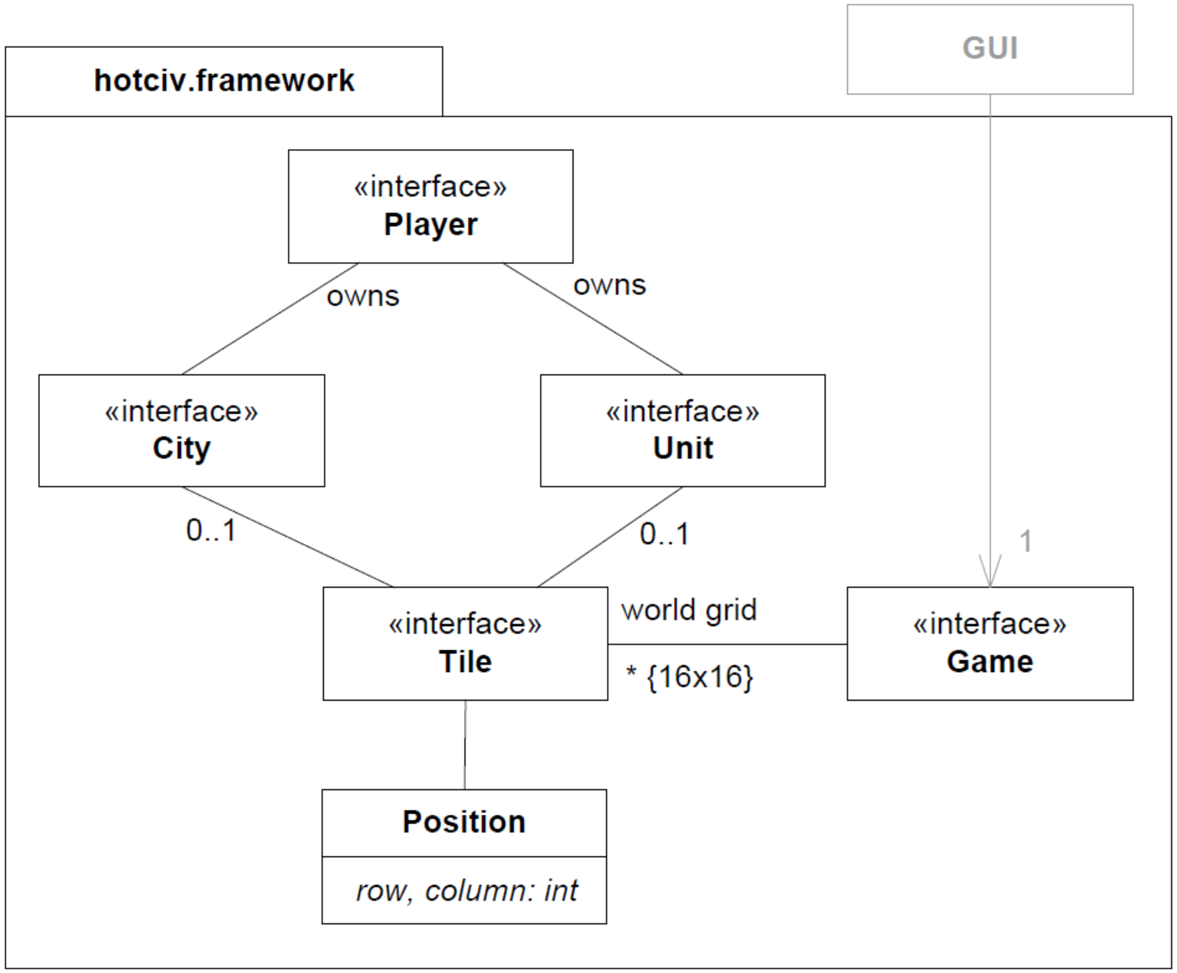
Game

- 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.)



```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.
   * @return the player that has won. If the game is still
   * not finished then return null.
   */
  public Player getWinner();

  /** return the age of the world. Negative numbers represent a world
   * age BC (-4000 equals 4000 BC) while positive numbers are AD.
   *  @return world age.
   */
  public int getAge();

  // === Mutator methods ======================================

  /** move a unit from one position to another. If that other position
   * is occupied by an opponent unit, a battle is conducted leading to
   * either victory or defeat. If victorious then the opponent unit is
   * removed from the game and the move conducted. If defeated then
   * the attacking unit is removed from the game. If a successful move
   * results in the unit entering the position of a city then this
   * city becomes owned by the owner of the moving unit.
   * Precondition: both from and to are within the limits of the
   * world.  Precondition: there is a unit located at position from.
   * @param from the position that the unit has now
   * @param to the position the unit should move to
   * @return true if the move is valid (no mountain, move is valid
   * under the rules of the game variant etc.), and false
   * otherwise. If false is returned, the unit stays in the same
```

Structure (static view)



Dynamics

Generally, consider that the GUI *only* mutate the game's state by using the 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(), …

– Unit, Tile, City's methods…

Some Design Decisions

- Keep interfaces intact!
  - Otherwise the GUI will have trouble interfacing your HotCiv
- Read-only interfaces (Unit, City, …)
  - You may
    - Add mutator methods to the interface (beware – allows state changes not going through the Game interface!)
    - Add mutator methods to StandardX
      - Will require quite a bit of casting
- String base types
  - Enumerations would give better reliability (compiler check) but would delimit future variants ability to add more e.g. more unit types.
- Preconditions
  - Many game methods require e.g. valid positions. **This means you should not make tests for invalid positions!**

Some Design Decisions

No World abstraction?

- My TDD did not need it so far! *Simplicity-the art of maximizing the work not done!*
- Introduce it if you find your code maintainability improves.

Doing TDD

- TDD is about being ’lazy’
  - Do not code in anticipation of need, only when need arise! ***Simplicity – maximize work not done!***
- Morale:
  - Make it as simple as possible!!! Code as little as possible!!!
  - Translate the specs into minimal set of test cases. Make the test cases drive the minimal amount of code.
  - Do **not** design the *swiss army knife*
  - Make the code **clean!!!**

Doing TDD

Experience from earlier years

- Test list is a **test** list
  - 'setup world' = feature; not a test
  - 'city at (1,1)' = test; not a feature
  - morale: write test lists, not feature lists
- Thinking implementation is bad…
  - think "how does my test case look"; not "how do I implement this"
- Thinking too much ahead
  - do not foresee problems that never arise
  - pick "one step tests"
  - be prepared for 'do over'

Doing TDD

You have to constantly refactor

- to make your code clean and abstract
- students tend to forget => junk pile of special cases ☹