AARHUS UNIVERSITET

Software Engineering and Architecture

HotCiv

- Source:
  - Microsoft: Testing for Continuous Delivery with VS 2012

Traditional vs. Agile Development

Traditional

Components are developed separately. Manager likes to say "complete each component then move on"

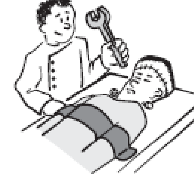
Eye done

Ear done

Leg done

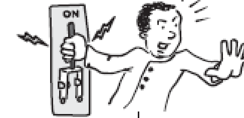
Hand done

Integration late in project



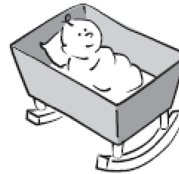
Unexpected behaviors discovered very late

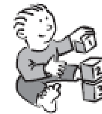
System testing at end

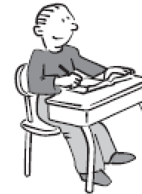


Schedule

Agile







Simple end-to-end functionality is achieved at an early stage

Regular system testing throughout

Each component gets gradually richer features and behavior High confidence in product from an early stage

HotCiv = Agile development

- Iterations of
  - product development
  - **learning increments**
- Context
  - Civilization type games
  - Pretty old-school GUI ☺

Getting Started

- Read Chapter 36.1 + 36.2
- Download hotciv-tdd-start.zip and unzip it!
- **Read the java file: Game.java**
  - **Much of the specification is in there**
- Source code
  - your variant is already initially package'ified
    - hotciv.framework etc. packages
    - split into a *production code (main)* and *test code (test)* source tree
  - your variant is already fully Gradlified'ified
    - 'gradle test' – find details in build/tests folder

Demo

- [Demo]

```
csdev@m31: ~/proj/frsproject/hotciv-tdd-start
File Edit Tabs Help
:processResources UP-TO-DATE
:classes
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test

BUILD SUCCESSFUL

Total time: 2.455 secs
csdev@m31:~/proj/frsproject/hotciv-tdd-start$ gradle test jacocoTestReport
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:jacocoTestReport

BUILD SUCCESSFUL

Total time: 2.073 secs
csdev@m31:~/proj/frsproject/hotciv-tdd-start$
```

reports

File Edit View Bookmarks Go Tools Help

/home/csdev/proj/frsproject/hotciv-tdd-start/build/reports

Places

Home Folder
Desktop
Trash Can
Applications
Floppy Disk
Documents
Music

jacoco
tests

Iteration 1

Implement 50% – 70% of AlphaCiv **using TDD!**

- *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.

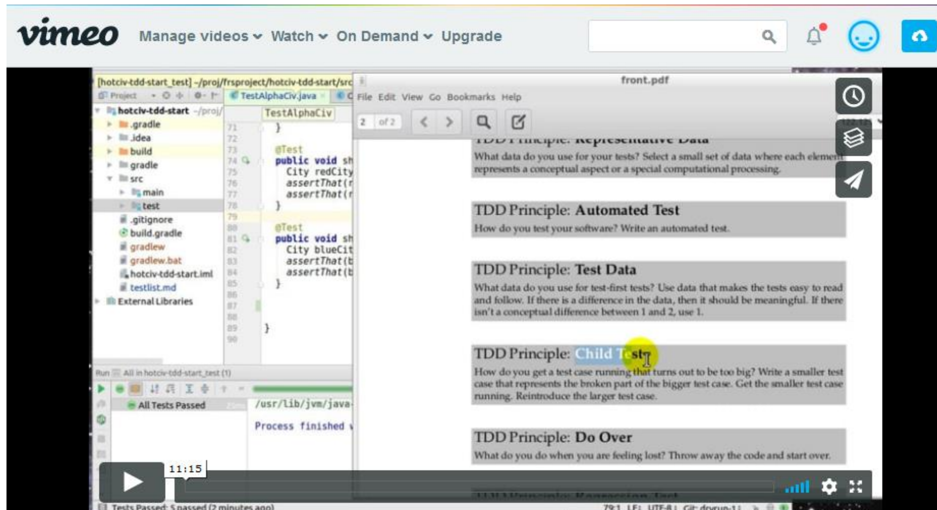
Iteration 2:
Implement ~100% of AlphaCiv using TDD (and transfer to Git)

Iteration 3..10:
Keep using TDD!!!

AARHUS UNIVERSITET

... And report on process (=skills)

- Using screen casts! And deliver your backlog and code



Demo af TDD Iteration Screencast

1 week ago | More

Henrik Bærbak Christensen

Settings | Review page

See all video stats | View Stats

Only visible to you

Related Videos

Autoplay next video

Demo af TDD Iter...

Well structured but not polished screen-cast!

See the advice on the exercise page...

Well-structured, not Polished!

- Screen-casts introduced because
  - TDD is process, better use video instead of reports
  - *To lower your workload!*
    - *10 minute screencast takes 10 minutes*
      - *Plus the two rehearsals that failed after 5 minutes ☺*
- Morale:
  - *Do not spend 4 hours doing story boards, 2 more hours retaking the screencast until it gets **just right**, and 1 more hour to make subtitles!*

... Evaluation

- Refer to the *Evaluation Criteria* to see what Teaching Assistants are supposed to look for…

Evaluation criteria

Your submission is evaluated against the learning goals and adherence to the submission guidelines. The grading is explained in Grading Guidelines.

Learning Goal	Assessment parameters
Submission	Code must compile; 'ant review' must pass; required artefacts (screencasts, backlog, test list) must all be present. The quality of screencast (video, audio) is OK.
TDD Process	Code is written 'test-first'; TDD Rhythm is followed (and referred to in screencast); TDD Principles are applied (and referred to); TDD values are used (and referred to); code is cleaned up (and referred to). The submission must be resubmitted in case you fail in obeying these requirements.
Test Code	The test code is simple (no complex constructs, basically only assignments, simple private method calls, very basic for loop). The test code reflects using the TDD principles (Isolated Test, Evident Test, Evident Data, etc.). The the production code is fullyt covered by test code (JaCoCo coverage is 'green' for all production code).
Production Code	Missing features are described in the backlog! There is no Fake-it code (or it is adequately described in the backlog). The AlphaCiv requirements are mostly obeyed ('simplest interpretation' is fine in case of ambiguities; minor deviations acceptable). The production code has been 'cleaned up' to avoid duplicated code. The production code follows guide lines from Barnes and Koelling (No sideeffects/mustation in accessor methods; source code follows Java conventions; identifiers/names make sense for the abstractions they model;no use of 'magic constants'; no test code has been put in the 'src' tree an vice versa; there are understandable overview comments in the code).

AARHUS UNIVERSITET

Feedback…

- For each learning goal
  - Grade
    - 0, 4, 7, 10
  - Argument
    - *Why a 4?*
- Improvement focus
  - Priority learning focus
- Score
  - Accumulated over 10 iter.
  - Part of final exam grade

A	B	C
		SWEA / Group DA1-7
		Iteration 1 Mandatory: TDD of AlphaCiv
Guidelines	**Pass**	**Argument**
Submission	**Yes**	ok
Learning Goal	**Grade**	**Argument**
TDD Process	4	Fine use of 'isolated test' and you do 'test first' but that is basically all you refer to of tdd principles in the screencast. You fail to mention 'assert first', 'evident test' and many of the other principles, and your process therefore appears less structured and systematic.
Test Code	4	The code does also not really reflect the principles being used - test method 'shouldVerifyALot()' is a prominent example. There are quite a few magic constants around so data is not evident.
Production Code	7	Nice to see missing features mentioned in the backlog. There is quite a bit of duplication in method 'move' which indicate that you have forgotten the 'refactoring' step to clean up.
Improvement Focus		
1st		Apply the TDD principles more rigourously, and remember to refer to them in screen casts
2nd		Remember to clean up the code
3rd		Quite a few identifiers have somewhat misleading names ("count" is used as a sum???). You will be required to fix these things once we get to the code quality exercise.
Score	**15**	
Average	5,00	**Accepted**
Legend	***Grade***	***Assesses the coverage of the Learning goals and the number of errors***
	0	Unacceptable
	4	Adequate (Minor coverage, and several significant errors)
	7	Good (Good coverage with some errors)
	10	Excellent (Very good coverage with minor or no errors)

The Agile backlog...

This is an **architecture** course!

- Clean code that works
  - (65% functionality with clean code **is much better than** 100% functionality with unclean code!)

And remember, **it is perfectly OK not to implement all functional requirements as long as you ensure your TDD process is as good as possible and that your code is "clean code that works".** This is an exercise to train your TDD programming skills, not an industry job to produce an AlphaCiv game product! Report missing features in the backlog, and get them done in the next iteration.

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

... 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
   * position and its "move" is intact (it can be moved to another
   * position.)
   */
  public boolean moveUnit( Position from, Position to );
```

Structure (static view)

GUI

hotciv.framework

«interface»
Player

owns

owns

«interface»
City

«interface»
Unit

0..1

0..1

1

«interface»
Tile

world grid

* {16x16}

«interface»
Game

Position

row, column: int

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 should
    - **Avoid to** add mutator methods to the interfaces!
    - 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

- MoveCount?
  - Distance is measured in ‘move count’ (no euclidian here!)
    - If chariot has move count = 2; and you move one tile, its move count is 1, and so on!
    - To move a chariot 2 tiles, you invoke moveUnit() twice!
      - The old Civilization way – not using the mouse and no routing…
- Treasury?
  - ‘Production’ means two things in the spec ☺
    - city.getTreasury() = count of ‘money’ that city owns right now
      - Gets converted to unit once enough ‘money’ has been earned
- No World abstraction?
  - See where your TDD moves you…

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
    - ‘red has 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’
  - Do **not** code all features as tests and **then** start to implement ☺

Doing TDD

- You have to constantly refactor
  - to make your code clean and abstract
  - students tend to forget => junk pile of special cases ☹
- *Fix your broken windows*
  - *Or the building will become unattractive...*
- *The total cost of owning a mess*

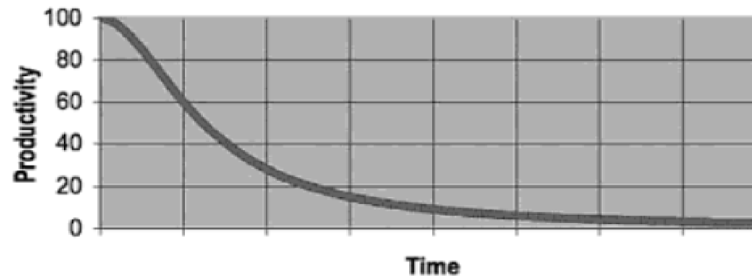
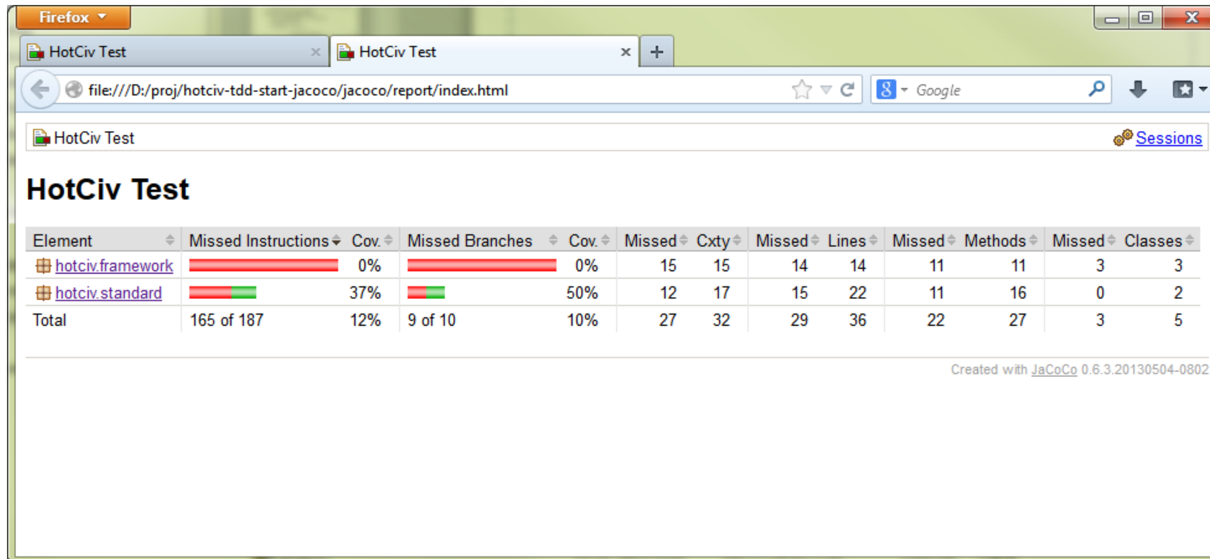


Figure 1-1
Productivity vs. time

Help to 'score' your tests...

- Code coverage
  - How much of your production code is exercised by your tests?
- 'gradle test jacocoTestReport' (only if tests pass!)
  - jacoco/report/index.html



HotCiv Test

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
hotciv.framework		0%		0%	15	15	14	14	11	11	3	3
hotciv.standard		37%		50%	12	17	15	22	11	16	0	2
Total	165 of 187	12%	9 of 10	10%	27	32	29	36	22	27	3	5

Created with JaCoCo 0.6.3.20130504-0802

Use The StudyCafe

- We will generally man the StudyCafe most of the official hours, so please use it.