AARHUS UNIVERSITET

Software Engineering and Architecture

Test-Driven Development

AARHUS UNIVERSITET

What is it?

- ***Test-Driven Development***

- *It is a* ***systematic programming technique***
  - Focus on
    - **Continued Speed**
    - **Reliability**

- It is not a *testing technique, despite the name*
  - Tests are *means*, not *an end in itself*
  - But they are sooo nice to have afterwards…

AARHUS UNIVERSITET

Systematic Programming

- TDD replaces *tacit knowledge* with a formulated process
  - *The mantra*
    - ***Clean code that works***
  - *The values*
    - The **four values** that abstractly define what we want
  - *The rhythm*
    - The **five steps** in each highly focused and fast-paced iteration.
  - *Testing principles*
    - The **testing principles** that defines a term for a specific action to make in each step
    - Form: Name - Problem – Solution

AARHUS UNIVERSITET

The Mantra

Clean code that works

- *To ensure that our software is reliable all the time*
  - *"Clean code **that works**"*
- *To ensure fast development progress*
- *To ensure that we dare restructure our software and its architecture*
  - *"**Clean code** that works"*
  - *Clean = understandable and changeable*

AARHUS UNIVERSITET

The Values

- ***Keep focus***
  - Make one thing only, at a time!
  - *Often*
    - *Fixing this, requires fixing that, hey this could be smarter, fixing it, ohh – I could move that to a library, hum hum, …*

- ***Take small steps***
  - Taking small steps allow you to backtrack easily when the ground becomes slippery
  - *Often*
    - *I can do it by introducing these two classes, hum hum, no no, I need a third, wait…*

AARHUS UNIVERSITET

The Values

- ***Speed***
  - You are what you do! Deliver every 14 days!!! Or two hours…
  - *Often*
    - *After two years, half the system is delivered, but works quite in another way than the user anticipate/wants…*
  - Speed, not by being sloppy but by making less functionality of high quality!
- ***Simplicity***
  - Maximize the amount of work *not done!*
  - *Often*
    - *I can make a wonderful recursive solution parameterized for situations X, Y and Z (that will never ever occur in practice)*

AARHUS UNIVERSITET

The Iteration Skeleton

- Each TDD iteration follows the Rhythm

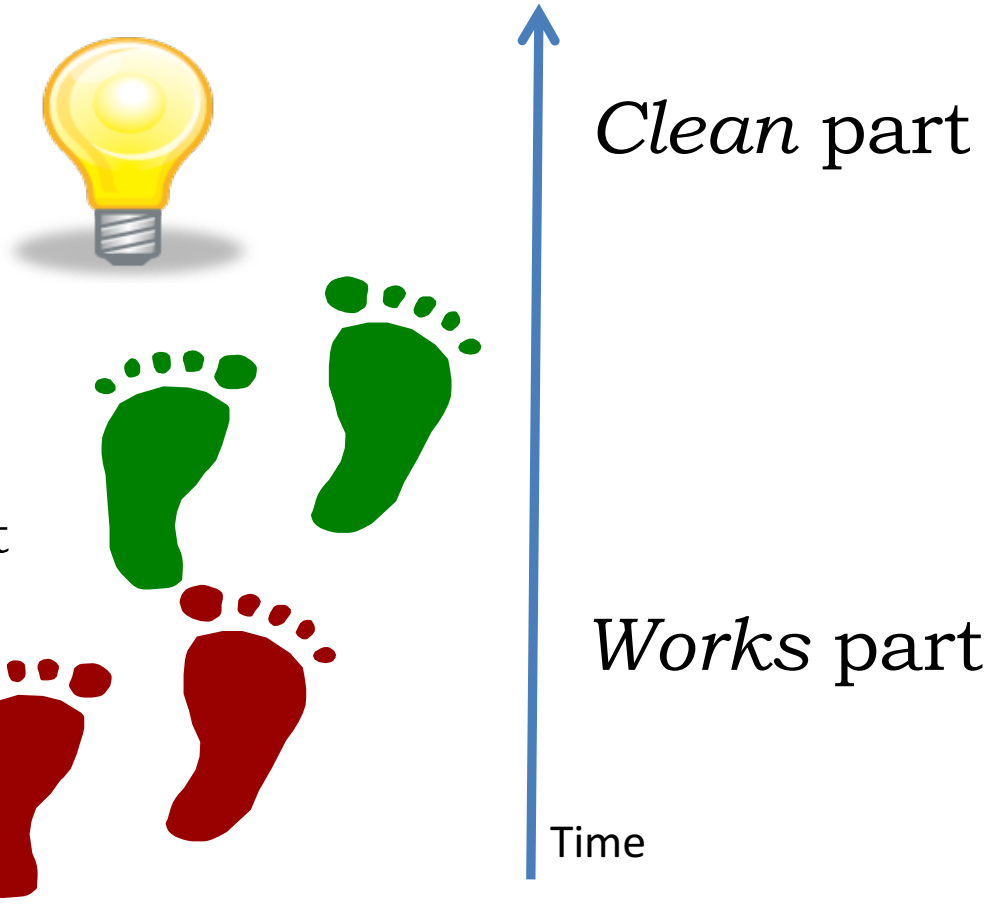
The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

- (6. All tests pass again after refactoring!)

The Rhythm: Red-Green-Refactor

- The Rhythm



The Three Starter Principles

TDD Principle: **Automated Test**

How do you test your software? Write an automated test.

TDD Principle: **Test First**

When should you write your tests? Before you write the code that is to be tested.

TDD Principle: **Test List**

What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

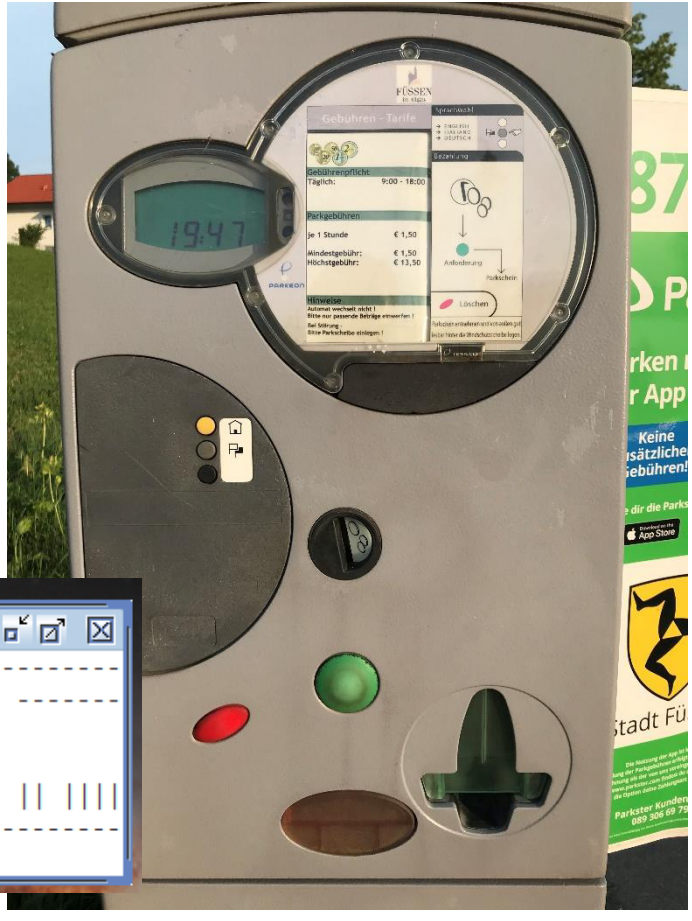
AARHUS UNIVERSITET

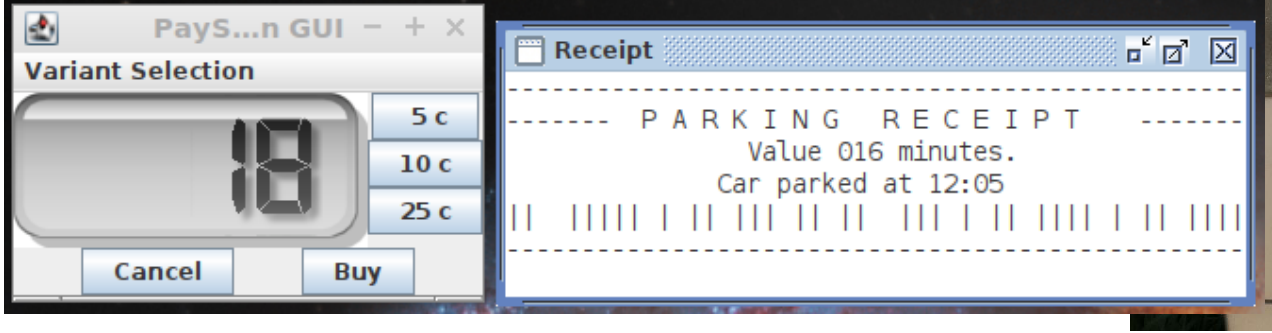
Enough…

Learning by Doing…

You are all employed today

- Welcome to *PayStation Ltd.*
- We will develop the main software to run *coin-operated* pay stations
  - [Demo]



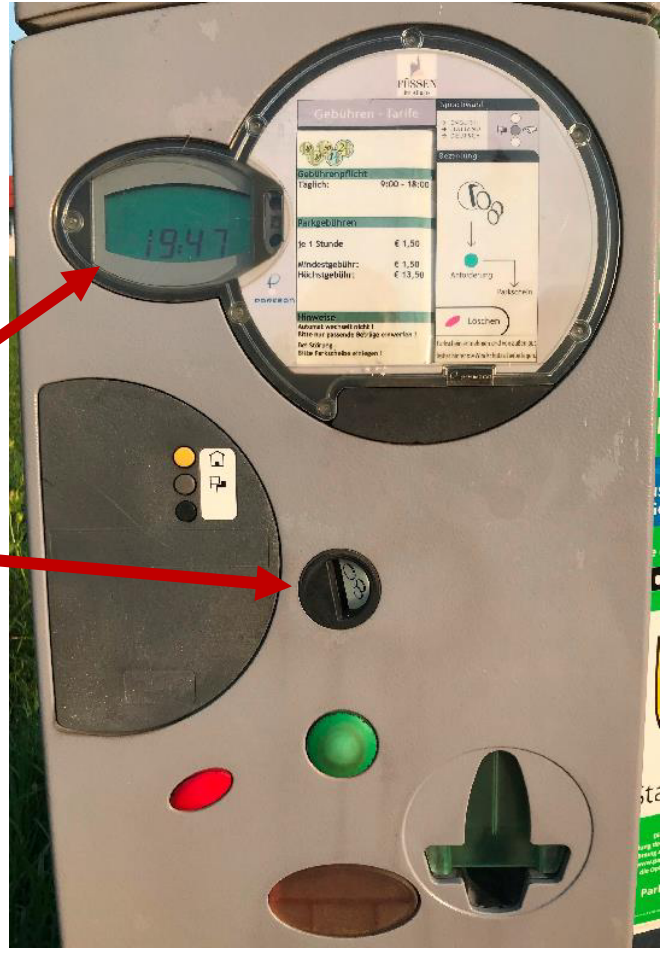


AARHUS UNIVERSITET

Case: Pay Station

- Welcome to *PayStation Ltd.*
- Customer: AlphaTown

- Requirements
  - show time bought on display
  - accept coins for payment
    - 5, 10, 25 cents
  - print parking time receipts
  - US: 2 minutes cost 5 cent
  - handle buy and cancel



Stories

Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

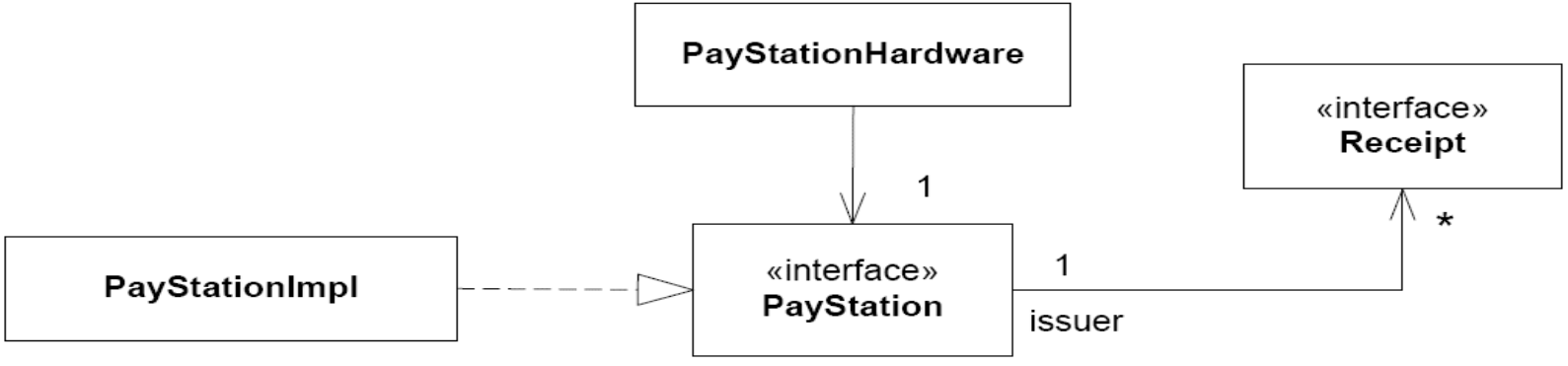
Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

AARHUS UNIVERSITET

Design: Static View

- For our purpose the design is given…
  - Central *interface* **PayStation**



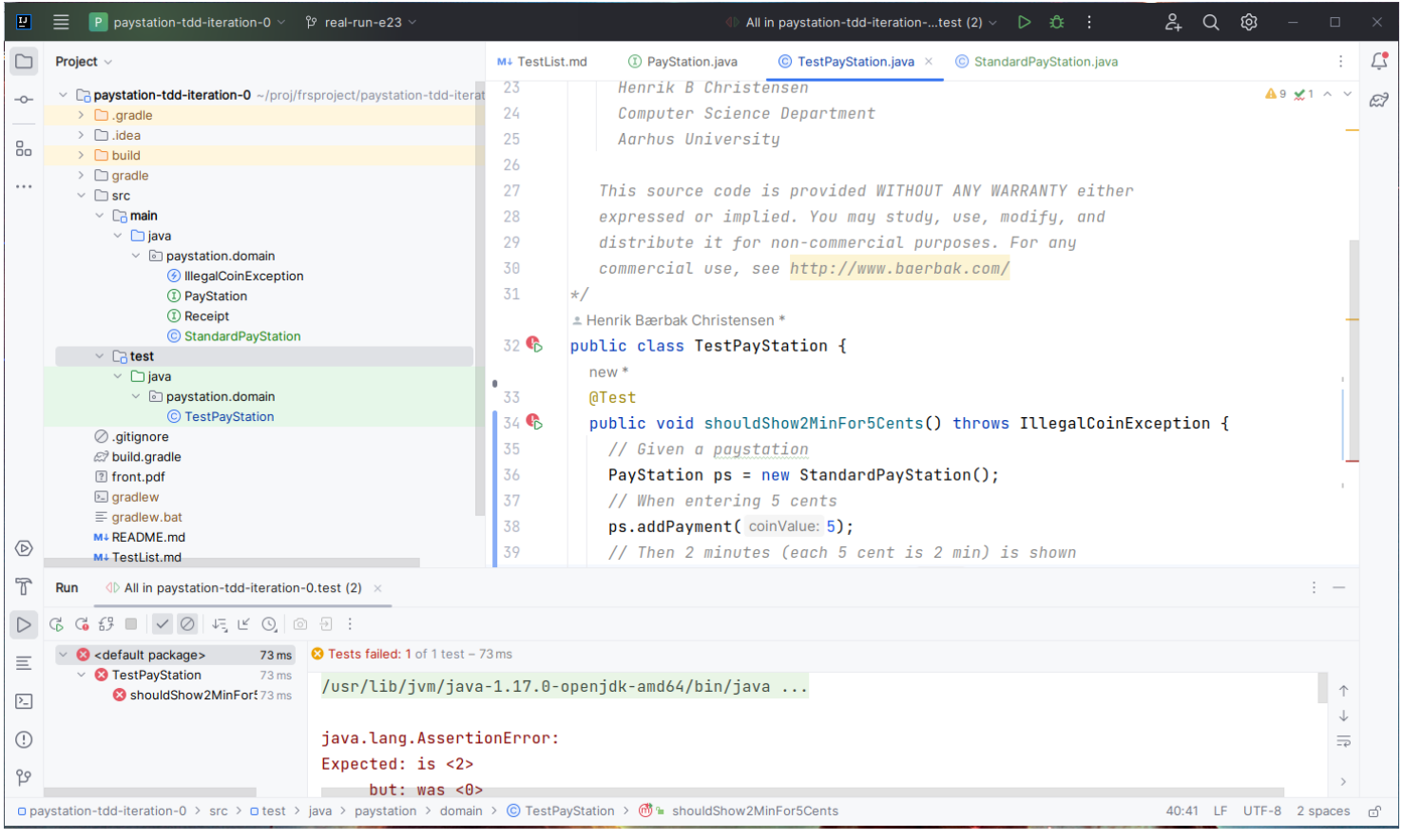
My Testlist

- From the book

* accept 5-cent coin
* 5 cents should give 2 minutes parking time
* reject 17-cent coin
* can read display
* buy for 40 cents produces receipt of 10 min
* cancel resets pay station

AARHUS UNIVERSITET

And on to...

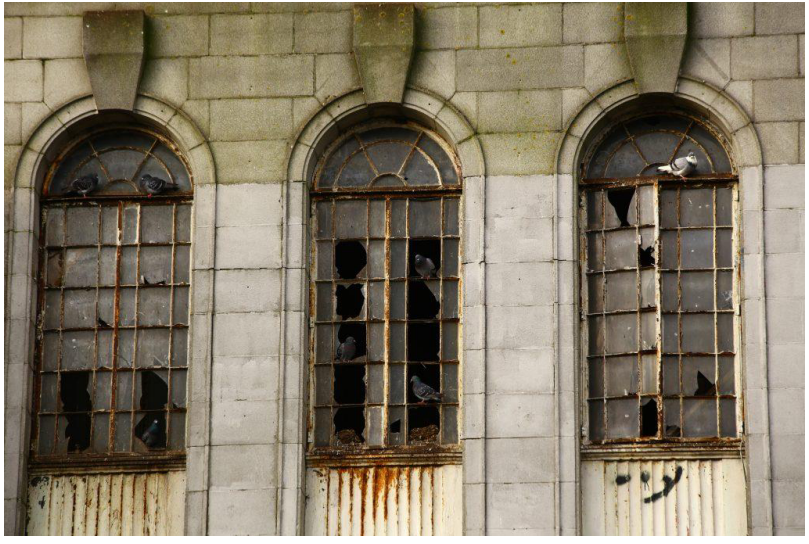


AARHUS UNIVERSITET

Central Principles Overview

Refactoring

- *Fix your broken windows*
  - *Clean up **now*** or your code *will* deteriorate quickly!



Definition: Refactoring

Refactoring is the process of modifying and restructuring the source code to improve its maintainability and flexibility without affecting the system's external behavior when executing.

TDD Principle: **One Step Test**

Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

TDD Principle: **Fake It ('Til You Make It)**

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

TDD Principle: **Obvious Implementation**

How do you implement simple operations? Just implement them.

TDD Principle: **Isolated Test**

How should the running of tests affect one another? Not at all.

TDD Principle: **Evident Tests**

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.

TDD Principle: **Assert First**

When should you write the asserts? Try writing them first.

TDD Principle: **Break**

What do you do when you feel tired or stuck? Take a break.

TDD Principle: **Do Over**

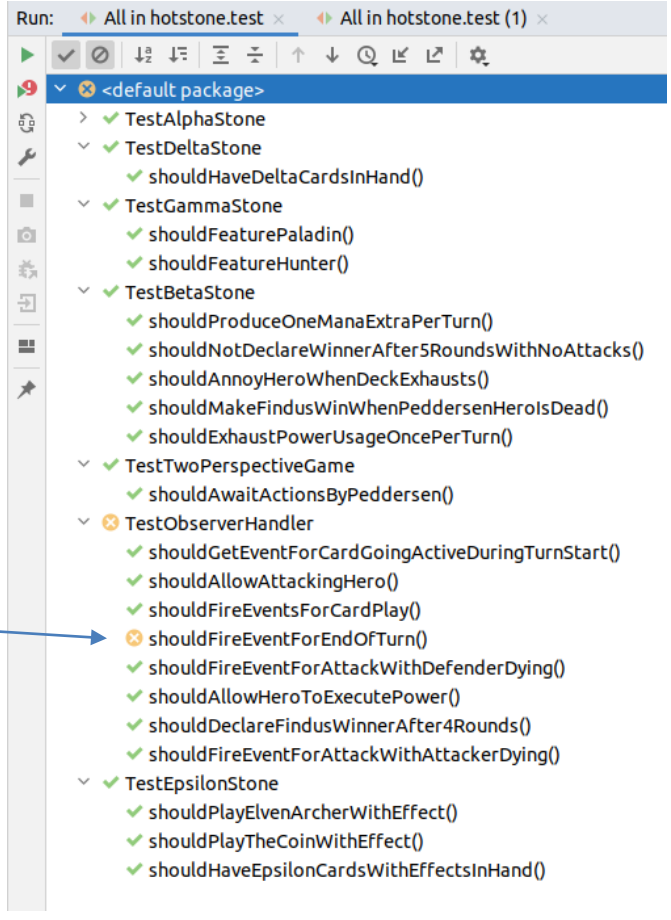
What do you do when you are feeling lost? Throw away the code and start over.

AARHUS UNIVERSITET

Run *All* tests and see new fail

- IntelliJ can lure you into working two hours in just one test class, without exposing a bug introduced that make all *other test classes fail.*

- Run *all tests all the time…*
  - *Only one should fail at a time!*



AARHUS UNIVERSITET

IDE Assistance

Avoid Typing

AARHUS UNIVERSITET

IDE Support

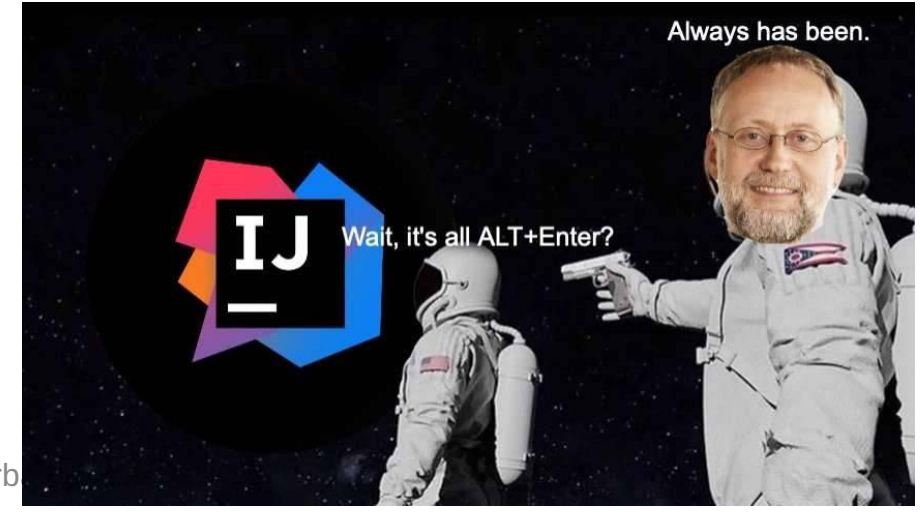
- Use it! Ctrl-Space and Alt-Enter your code into existence!

```
@Test
public void shouldShow2MinFor5Cents() throws IllegalCoinException {
    // Given a paystation
    PayStation ps = new StandardPayStation();
    // When entering 5 cents
    ps.addPayment( coinValue: 5);
    // Then 2 minutes (each 5 cent is
    assertThat(ps.readDisplay(), is(
```

Create class 'StandardPayStation'
Create record 'StandardPayStation'
Create inner class 'StandardPayStation'
Create inner record 'StandardPayStation'

```
public class StandardPayStation implements
    PayStation {
}
```

- *See my ‘IDE’ slides…*
- My wonderful students have meme’ified it ☺



AARHUS UNIVERSITET

AI Assistance

Things are moving fast…

CoPilot & ChatGPT

CoPilot

Write “should…”
Wait. Hit Tab.
Fix defects.

TestAlphaStone.java

```
68
69      // Test case: players alternate turns
        no usages
70      @Test
71      public void shouldAltern
72
73      // Example of a later, more complex, test case:
74      // Card handling
75
76      // The HotStone specs are quite insisting on how
77      // the cards, drawn from the deck, are organized
78      // in the hand. So when drawing the top three cards
79      // from the deck (uno, dos, tres) they have to
80      // be organized in the hand as
81      // index 0 = tres; index 1 = dos; index 2 = uno
82      // That is, a newly drawn card is 'at the top'
83      // of the hand - always entered at position 0
84      // and pushing the rest of the cards 1 position
85      // 'down'
        ⚓ Henrik Bærbak Christensen (m1.coffeelake) +1
@Test
public void shouldHaveUnoDosTresCardsInitially() {
  // Given a game, Findus has 3 cards in hand
  int count = game.getHandSize(Player.FINDUS);
  assertThat(count, is( value: 3));
  // And these are ordered Tres, Dos, Uno in slot 0
```

```
// Test case: players alternate turns
new *
@Test
public void shouldAlternateBetweenPlayers() {
  // Given a game
  // When I ask for the current player
  Player player = game.getPlayerInTurn();
  // Then it should be Findus
  assertThat(player, is(Player.FINDUS));
  // When I end the turn
  game.endTurn();
  // Then it should be Peddersen
  player = game.getPlayerInTurn();
  assertThat(player, is(Player.PEDDERSEN));
  // When I end the turn
  game.endTurn();
  // Then it should be Findus
  player = game.getPlayerInTurn();
  assertThat(player, is(Player.FINDUS));
}
```

82:15 LF UTF-8 2 spaces master

AARHUS UNIVERSITET

Also Funny at Times

- Where the h… did Pingo come from???

```
1 usage  new *
private void tooglePlayerInTurn() {
    if (playerInTurn == Player.FINDUS) {
        playerInTurn = Player.PINGO;
    } else {
        playerInTurn = Player.FINDUS;
    }
}
```

And downright wrong!

- Sometimes I struggle more when CoPilot invents non-sense that does not correlate in any way to my code…

```
Player me = mutGame.getPlayerInTurn();
// Iterate all own cards and increment health by 2
mutGame.getBoard().getCards(me).forEach(card     -> {
      },
```

getBoard() ?

```
// Iterate all own cards and increment health by 2
for(int i=0; i < mutGame.getNumberOfCardsInHand(me);)      i++) {
      },
```

getNumberOfCardsInHand() ?

```
// Iterate all own cards and increment health by 2
for(int i=0; i < mutGame.getFieldSize(me);i++) {
  mutGame.incrementHealth(me, i, 2);
}
```

incrementHealth()?

AARHUS UNIVERSITET

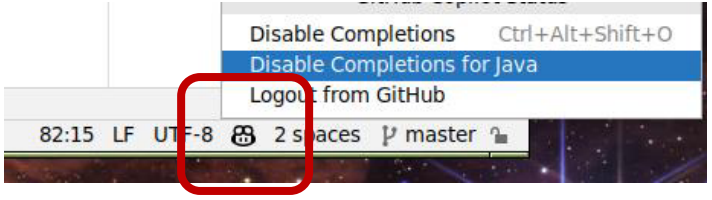
So – What to make of it?

- Use it – or Not? In SWEA? In future work?
- Positive side
  - ACM talk: 30% more productive programmers
    - CoPilot fills in the gaps quickly
- Negative side
  - Spaghetti code generator spewing wrong code out at high speed
    - If you is not careful, and you have the overview
  - **In a learning and training situation you loose the initiative**
    - CoPilot puts you in the *reviewer role* rather than *the coder role*
      - And when learning new stuff it does not sharpen your skill properly
- ***Do you drive CoPilot – or is CoPilot driving you?***

AARHUS UNIVERSITET

So – My Requirement

- Start out by **disabling CoPilot** (or not install it)



- At the Autumn vacation and when you master Java and the TDD process and JUnit and Hamcrest, I find it OK to turn it on and *take it out for a spin…*

- And by all means – it is a great tool in the long run and you should use it to your advantage

AARHUS UNIVERSITET

Outlook

Tests as Specifications

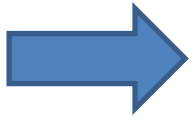
AARHUS UNIVERSITET

Insight

- An early insight was that *the tests* are actually a *specification of the requested behaviour*
  - *Aka* a **requirements specification!**
- What if costumers could write them themselves !?!?

- Gave rise to much experimentation
  - Behaviour-Driven Development

As a [X]
I want [Y]
so that [Z]



Given some initial context (the givens),
When an event occurs,
then ensure some outcomes.

GWT/Gherkin notation

AARHUS UNIVERSITET

I Use GWT !

- FRS §5.12,
  - I write my GWT's as comments, marking the 3 sections

```
/**
 * Entering 5 cents should make the display report 2 minutes
 * parking time.
*/
@Test
public void shouldDisplay2MinFor5Cents()
        throws IllegalCoinException {
  // Given a pay station
  PayStation ps = new PayStationImpl();
  // When I enter 5 cents
  ps.addPayment(5);
  // Then the display reads 2 minutes parking
  assertThat(ps.readDisplay(), is(2));
}
```

```
@Test
public void shouldClearAfterBuy()
        throws IllegalCoinException {
  // GIVEN the pay station from the setup() method

  // WHEN customer 1 enters 25 cents and buys parking
  ps.addPayment(25);
  ps.buy();
  // THEN the display should read 0
  assertThat(ps.readDisplay(), is(0));

  // WHEN customer 2 adds 35 cents
  ps.addPayment(10); ps.addPayment(25);
  // THEN the display reads correctly
  assertThat(ps.readDisplay(), is((10+25) / 5 * 2));
  // WHEN customer 2 then buys parking
  Receipt r = ps.buy();
  // THEN he/she receives a parking receipt of proper value
  assertThat(r.value(), is((10+25) / 5 * 2));
  assertThat(ps.readDisplay(), is(0) );
}
```

```
@Test
public void shouldMoveBlueFrom00To01() {
  // Given a game in which blue is ready to do stuff
  game.rollDie();
  game.turnCard();
  assertThat(game.getPlayerInTurn().getColor(), is(PlayerColor.Blue));
  // When blue moves 3 units to (0,1)
  CommandResult result = game.move(blueSettlement, new Position( r: 0,  c: 1),  count: 3);
  // Then that move is valid
  assertThat(result, is(CommandResult.OK));
  // And then the (0,1) tile is blue with 3 units
  Tile t = game.getTile(new Position( r: 0,  c: 1));
  assertThat(t.getOwnerColor(), is(PlayerColor.Blue));
  assertThat(t.getUnitCount(), is( value: 3));
  // Then the settlement has only 7 units left
  t = game.getTile(blueSettlement);
  assertThat(t.getUnitCount(), is( value: 7));

  // Then the game's state is buy
  State state = game.getState();
  assertThat(state, is(State.buy));
}
```