AARHUS UNIVERSITET

Software Engineering and Architecture

Refactoring and Integration Testing

The power of automated tests

Two product variants

- Alphatown and Betatown
  - Four models to handle this
    - compositional proposal has nice properties...
- How do we introduce it?

Change by addition

- I state:
  - ***Change by addition, not modification***
- because
  - addition
    - little to test, little to review
    - little chance of introducing ripple-effects
  - modification
    - more to test, more to review
    - high risk of ripples leading to side effects (bugs!)

The Problem Statement

- **But** I have to *modify* the pay station implementation in order to prepare it for the new compositional design that uses a Strategy pattern

- ☹ *Change by modification*

- Problem:
  - How to reliably modify PayStationImpl?
  - How can I stay confident that I do not accidentally introduce any defects?

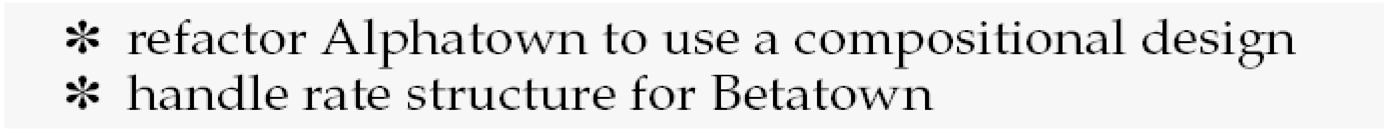
Take Small Steps

- I will *stay focused* and *take small steps!*

- I have **two** tasks
  - 1) Refactor the current implementation to introduce the Strategy and make AlphaTown work with new design
  - 2) Add a new strategy to handle Betatown requirements

- ... *and I will do it in that order – small steps!*

Refactoring



- refactor Alphatown to use a compositional design
- handle rate structure for Betatown

- **Definition:**

- *Refactoring is the process of changing a software system in such a way that is does not alter the external behavior of the code yet improves its internal structure.*
  - *Fowler, 1999*

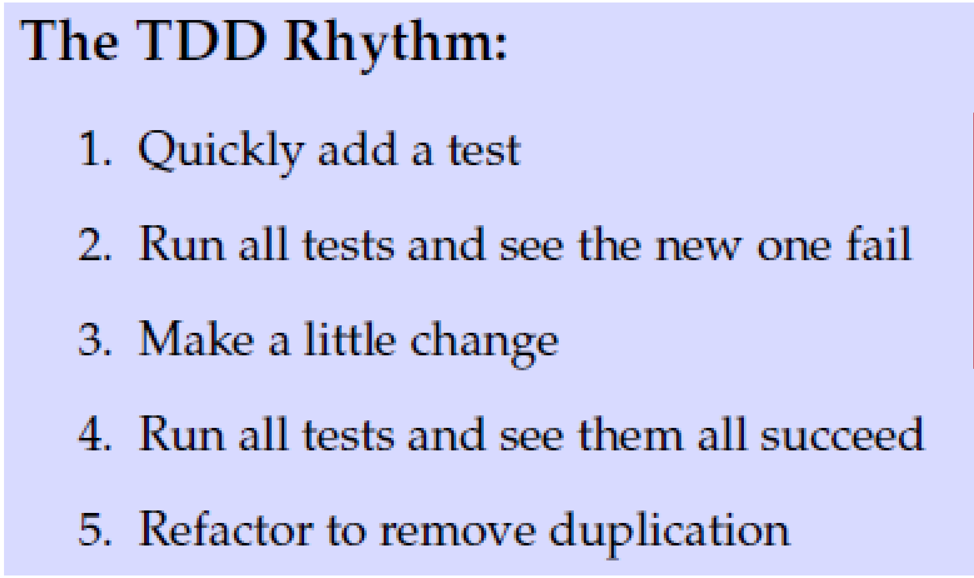
AARHUS UNIVERSITET

Iteration 1

Refactoring step

The Rhythm

- Refactoring and 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

1+2+3: Refactor

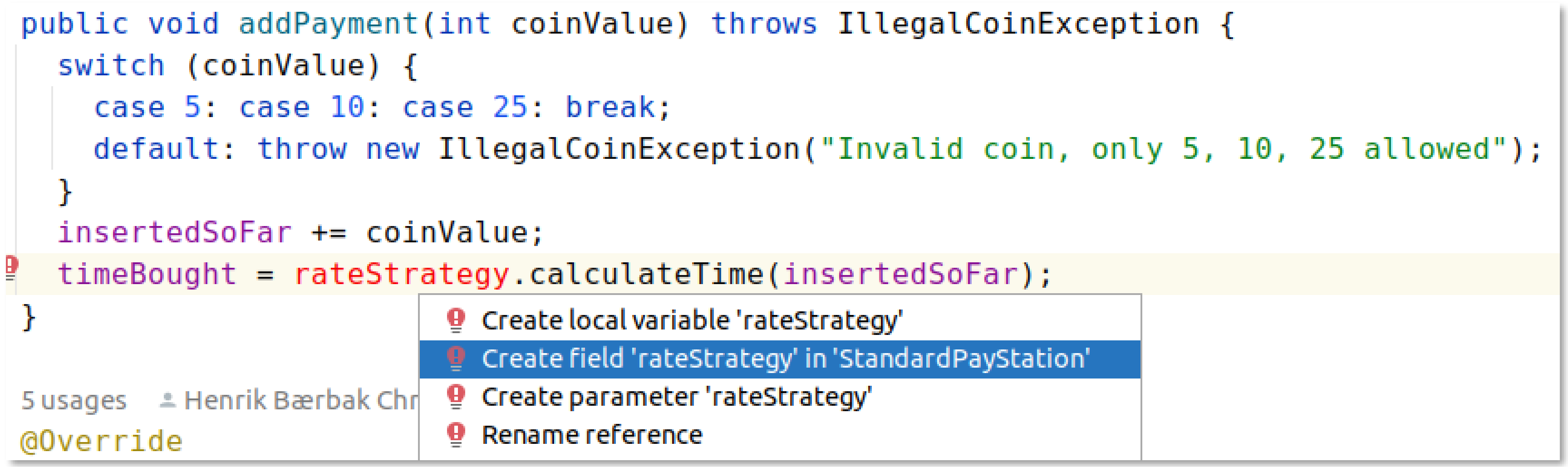
- Same *spirit*, but step 1+2+3 becomes “refactor”

A faster way than in the FRS book

Use the tools in your IDE

AARHUS UNIVERSITET

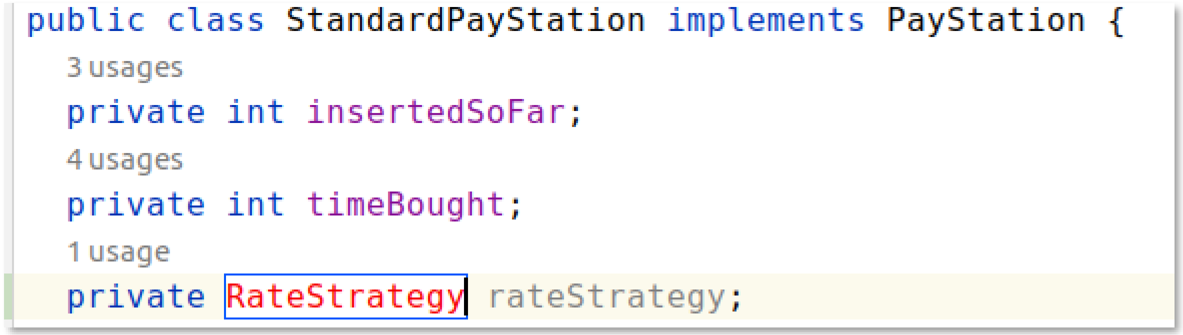
Simply type what you want



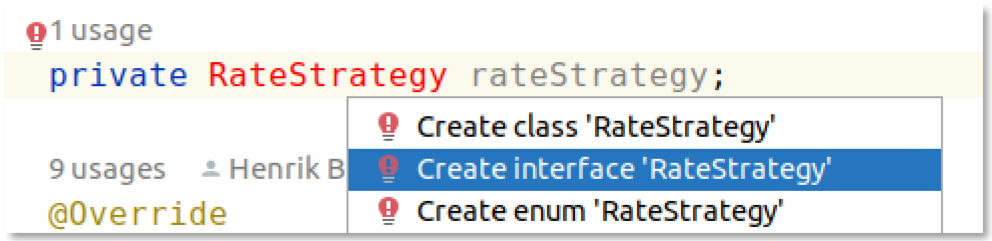
```
public void addPayment(int coinValue) throws IllegalCoinException {
  switch (coinValue) {
    case 5: case 10: case 25: break;
    default: throw new IllegalCoinException("Invalid coin, only 5, 10, 25 allowed");
  }
  insertedSoFar += coinValue;
  timeBought = rateStrategy.calculateTime(insertedSoFar);
}

5 usages   Henrik Bærbak Chr
@Override
```

Create local variable 'rateStrategy'
Create field 'rateStrategy' in 'StandardPayStation'
Create parameter 'rateStrategy'
Rename reference

- And ask the IDE (Alt-Enter) to suggest what to do,
  - And then just *tell it what you want and it will fill in the template*

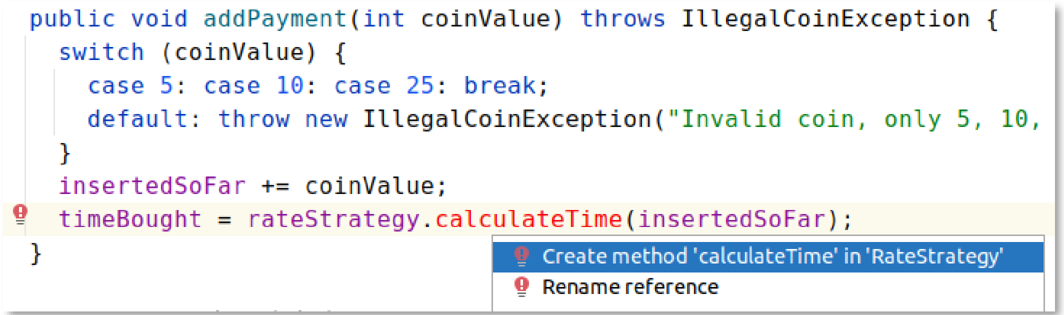


```
public class StandardPayStation implements PayStation {
  3 usages
  private int insertedSoFar;
  4 usages
  private int timeBought;
  1 usage
  private RateStrategy rateStrategy;
```



```
1 usage
private RateStrategy rateStrategy;

9 usages   Henrik B
@Override
```

Create class 'RateStrategy'
Create interface 'RateStrategy'
Create enum 'RateStrategy'



```
public void addPayment(int coinValue) throws IllegalCoinException {
  switch (coinValue) {
    case 5: case 10: case 25: break;
    default: throw new IllegalCoinException("Invalid coin, only 5, 10,
  }
  insertedSoFar += coinValue;
  timeBought = rateStrategy.calculateTime(insertedSoFar);
}
```

The 7 Inch Nail…

- To repeat

Introduce *design changes* in **two** ‘small steps’:

1) Use *existing* test cases to *refactor* code so it has new design
Do not change existing behavior!

2) Only *then* do you start test-driving the *new feature(s)* into your codebase.

Discussion

Why TDD?

- Traditionally, developers see *tests* as
  - boring
  - time consuming

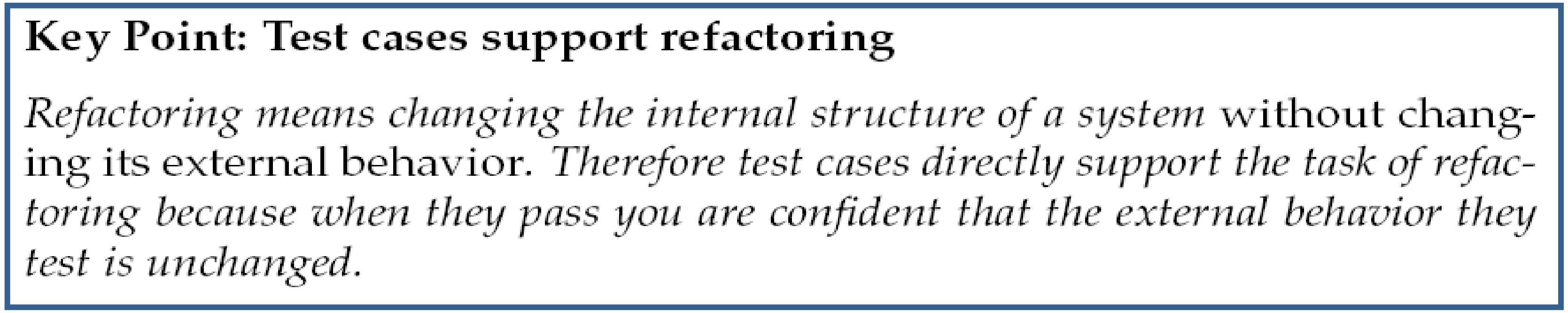
- Why? Because of the stakeholders that benefit from tests are not the developers
  - customers: ensure they get right product ☺
  - management: measure developer productivity ☺
  - test department: job security ☺
  - developers: *seemingly no benefit at all ☹*

AARHUS UNIVERSITET

If it ain’t broke...

- *If it ain’t broke, don’t fix it*
- …is the old saying of fear-driven programming

- Developers and programmers do not dare doing drastic design and architecture changes in fear of odd side-effects.



Key Point: Test cases support refactoring

Refactoring means changing the internal structure of a system without changing its external behavior. *Therefore test cases directly support the task of refactoring because when they pass you are confident that the external behavior they test is unchanged.*

Test Ownership

- Refactoring makes developers want to have ownership of the tests:

• *Automatic tests is the developers' means to be courageous and to dare modify existing production code.*

- Michael Feathers:
  - *Software Vise: Fixating the behavior*

AARHUS UNIVERSITET

…But

- The brittleness of the test cases hinges on *only using the interfaces to the widest possible extend!*

- ☺ assertThat(game.getCardInHand(…), is.…)

- ☹ assertThat(game.getInternalDataStruture()
  .getAsArray()[47], is …)

- Ensure your test cases does not rely on implementation details…

AARHUS UNIVERSITET

When redesigning....

Key Point: Refactor the design before introducing new features

Introduce the design changes and refactor the system to make all existing test suites pass before *you begin implementing new features.*

- TDD often seems like a nuisance to students and developers until the first time they realize that they dare do things they previously never dreamed of!

- The first time a major refactoring is required – the light bulb turns on ☺

AARHUS UNIVERSITET

A Side Note

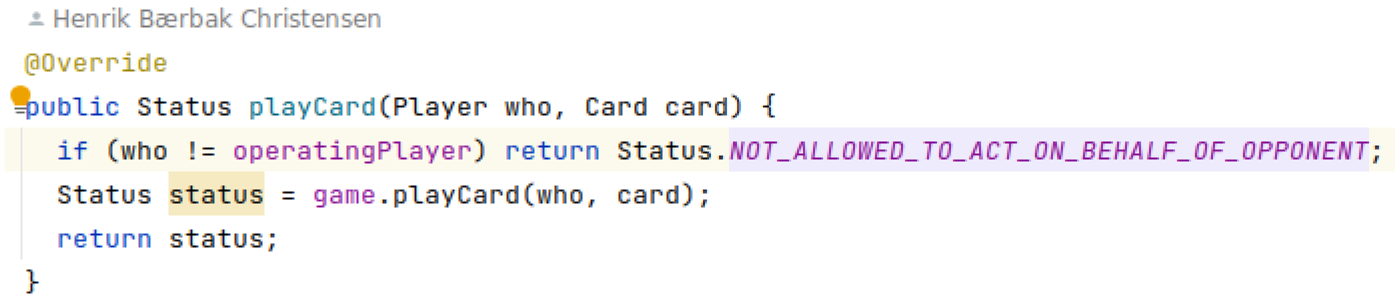
Tests allow ‘hypotheses’ to be verified quickly…

An Example

- 2023 discussion forum question
  - ‘Why that Status.NOT_ALLOWED_TO_ACT_ON_BEHALF…?’
    - That is – what purpose does that particular value serve?
- Good question?
  - The history of that is, well history! Code evolves, ideas are tried out, sometimes they are essential, sometimes not, so is it vital that this status value is kept? Or, can I delete it?
- How to I find the answer to that?
  - **By using my tests! The software Vise…**

An Example

- So – I use IntelliJ to find uses of that value. And find a couple of places, one example being



```
Henrik Bærbak Christensen
@Override
public Status playCard(Player who, Card card) {
  if (who != operatingPlayer) return Status.NOT_ALLOWED_TO_ACT_ON_BEHALF_OF_OPPONENT;
  Status status = game.playCard(who, card);
  return status;
}
```

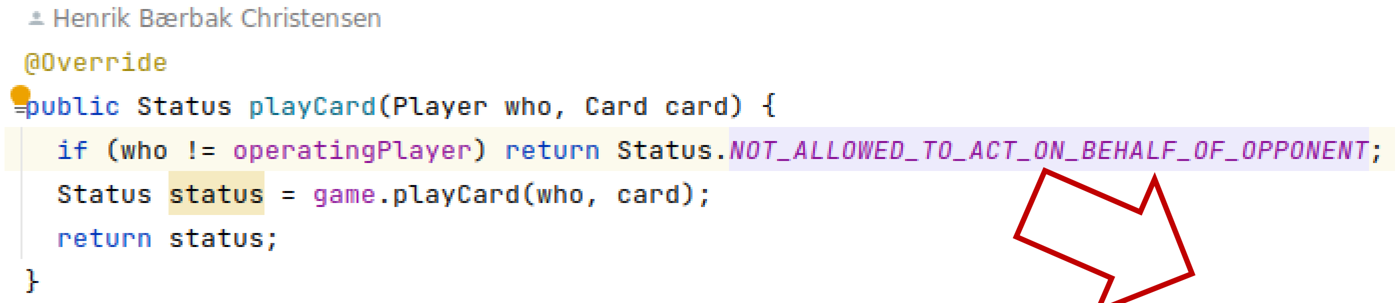
- **The 'what if' scenario**
  - Use your tests to see what happens if…
    - If I replace it by NOT_OWNER???
- **Why can this 'what if' test provide value?**

What If

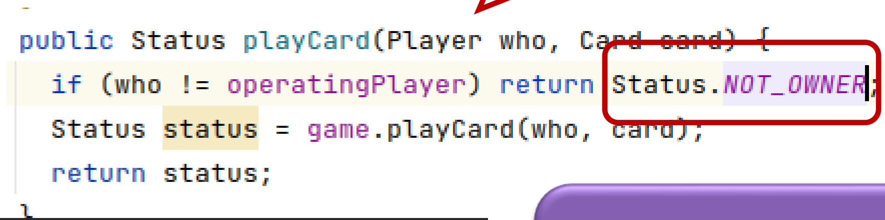
- The point here is:
  - If 1 out of 95 test cases break, then…
  - If 92 out of 95 test cases break, then …
- Yeah – then **what?**
- **The "blast radius" is estimated**
  - **Will this have small or large implications?**

What If game

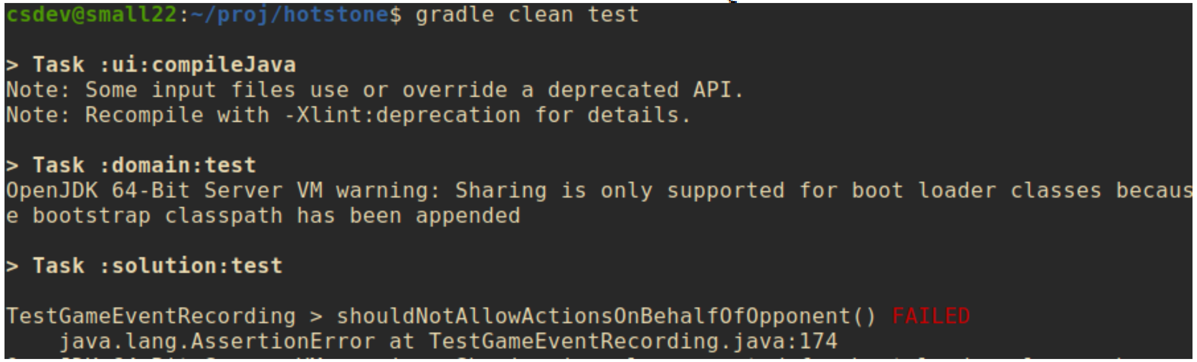
- So I make that change (temporarily) to assess



```
Henrik Bærbak Christensen
@Override
public Status playCard(Player who, Card card) {
  if (who != operatingPlayer) return Status.NOT_ALLOWED_TO_ACT_ON_BEHALF_OF_OPPONENT;
  Status status = game.playCard(who, card);
  return status;
}
```



```
public Status playCard(Player who, Card card) {
  if (who != operatingPlayer) return Status.NOT_OWNER;
  Status status = game.playCard(who, card);
  return status;
}
```



```
csdev@small22:~/proj/hotstone$ gradle clean test

> Task :ui:compileJava
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

> Task :domain:test
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes becaus
e bootstrap classpath has been appended

> Task :solution:test

TestGameEventRecording > shouldNotAllowActionsOnBehalfOfOpponent() FAILED
    java.lang.AssertionError at TestGameEventRecording.java:174
```

Run all tests…
… to see only a single fail!

Conclusion

- For the domain code (game code), it seems that particular Status value does not provide any value beyond what NOT_OWNER does.
  - It can probably be removed, the ‘blast area’ is small
  - Removed in the code base for the E24 instance

- As UI operations are manually tested, I do still need to verify that aspects
  - Issue: Game domain works, but the UI fails…

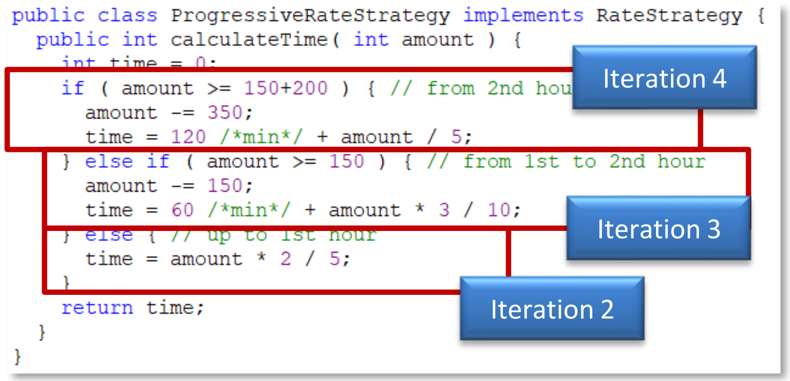
Iteration 2

Betatown Rate Policy

Triangulation at Algorithm Level

- Introducing the real BetaTown rate policy is a nice example of using **Triangulation**
  - Iteration **2:**
    - Add test case for first hour => production code

```
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

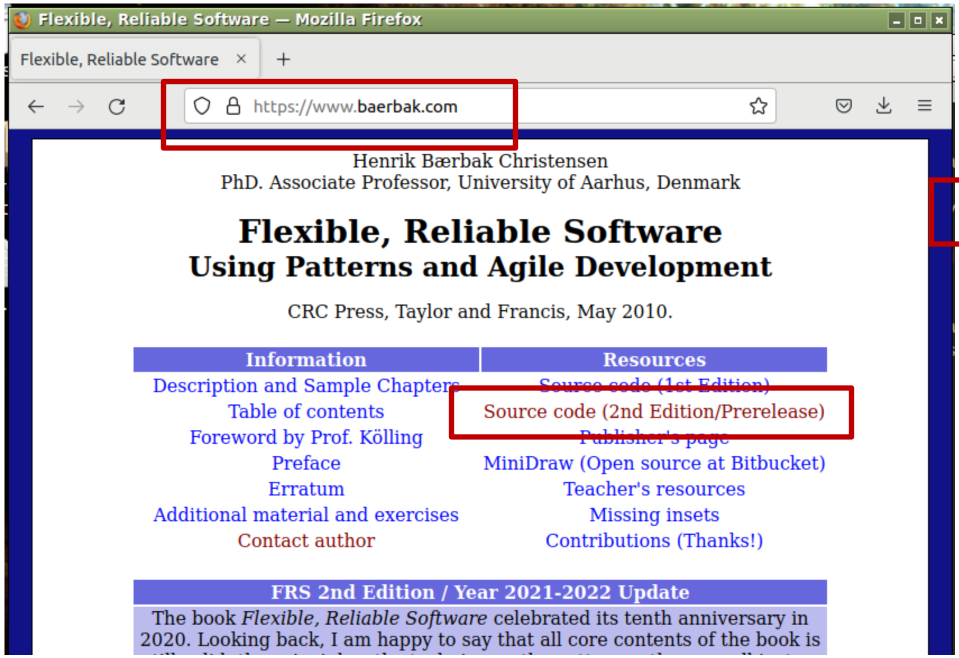


```
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    int time = 0;
    if ( amount >= 150+200 ) { // from 2nd hou
      amount -= 350;
      time = 120 /*min*/ + amount / 5;
    } else if ( amount >= 150 ) { // from 1st to 2nd hour
      amount -= 150;
      time = 60 /*min*/ + amount * 3 / 10;
    } else { // up to 1st hour
      time = amount * 2 / 5;
    }
    return time;
  }
}
```

  - Iteration **3**: Add test case for second hour
    - Add just enough complexity to the rate policy algorithm
  - Iteration **4:** Add test case for third (and following) hour
    - Add just enough more complexity

AARHUS UNIVERSITET

Uhum – the Details?



Flexible, Reliable Software — Mozilla Firefox

Flexible, Reliable Software

https://www.baerbak.com

Henrik Bærbak Christensen
PhD. Associate Professor, University of Aarhus, Denmark

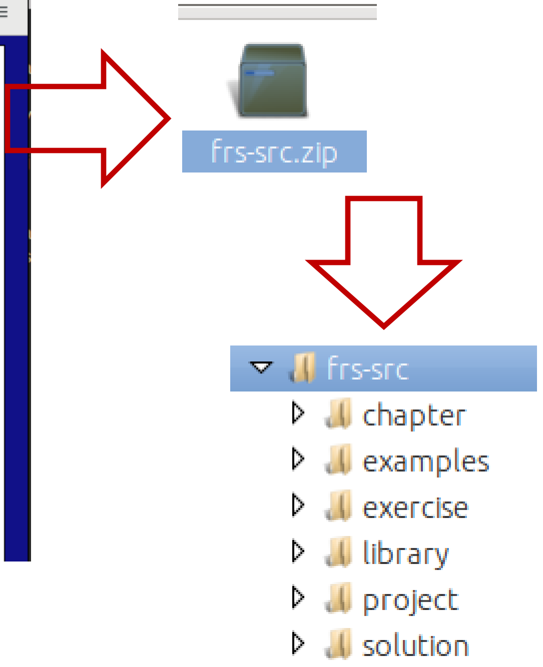
Flexible, Reliable Software
Using Patterns and Agile Development

CRC Press, Taylor and Francis, May 2010.

Information	Resources
Description and Sample Chapters	Source code (1st Edition)
Table of contents	Source code (2nd Edition/Prerelease)
Foreword by Prof. Kölling	Publisher's page
Preface	MiniDraw (Open source at Bitbucket)
Erratum	Teacher's resources
Additional material and exercises	Missing insets
Contact author	Contributions (Thanks!)

FRS 2nd Edition / Year 2021-2022 Update

The book *Flexible, Reliable Software* celebrated its tenth anniversary in 2020. Looking back, I am happy to say that all core contents of the book is



AARHUS UNIVERSITET

Iteration 5

Unit and Integration Testing

Unit Testing

- I can actually test the new rate policy *without using the pay station at all !*

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

```
public class TestProgressiveRate {
  RateStrategy rs;

  @BeforeEach public void setUp() {
    rs = new ProgressiveRateStrategy();
  }
```

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

```
  @Test public void shouldGive120MinFor350cent() {
    // Two hours: $1.5+2.0
    assertThat(rs.calculateTime(350), is(2*60) /* minutes */);
  }
```

Advantages

- The unit testing of the progressive rate strategy is much simpler than the corresponding test case, using the strategy integrated into the pay station.

Fragment: chapter/refactor/iteration-3/src/test/java/paystation/domain/TestProgressiveRate.java

```
  /** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

  assertThat(ps.readDisplay(), is(2 * 60) /*minutes*/);
}
```

Compare to

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

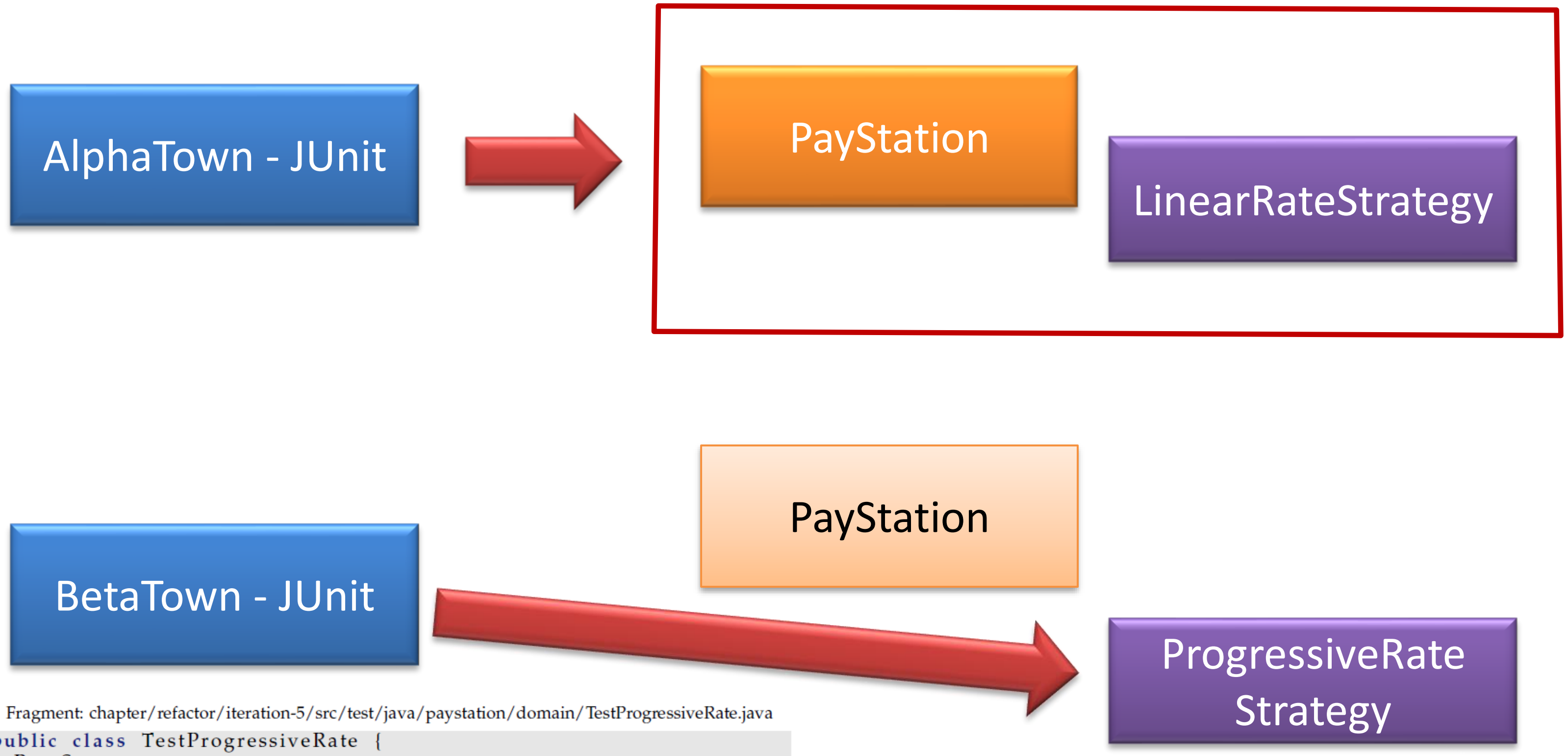
```
@Test public void shouldGive120MinFor350cent() {
  // Two hours: $1.5+2.0
  assertThat(rs.calculateTime(350), is(2*60) /* minutes */);
}
```

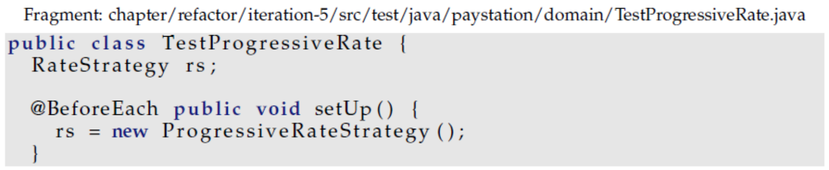
AARHUS UNIVERSITET

Testing Types

- Now
  - I test the ProgressiveRateStrategy *in isolation* of the pay station (Unit testing)
  - The pay station is tested *integrated* with the LinearRateStrategy (Integration testing)
- Thus the two rate strategies are tested by *two* approaches
  - In isolation (unit)
  - As part of another unit (integration)
- And
  - The actual Betatown pay station is never tested!

Visually




Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

```
public class TestProgressiveRate {
  RateStrategy rs;

  @BeforeEach public void setUp() {
    rs = new ProgressiveRateStrategy();
  }
```

AARHUS UNIVERSITET

Definitions

- Experience tells us that *testing the parts does not mean that the whole is tested!*
  - Often defects are caused by *interactions between units* or *wrong configuration* of units!

Definition: **Unit test**

Unit testing is the process of executing a software unit in isolation in order to find defects in the unit itself.

Algorithms – Business Logic

Definition: **Integration test**

Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.

Collaboration between units/modules/services

Definition: **System test**

System testing is the process of executing the whole software system in order to find deviations from the specified requirements.

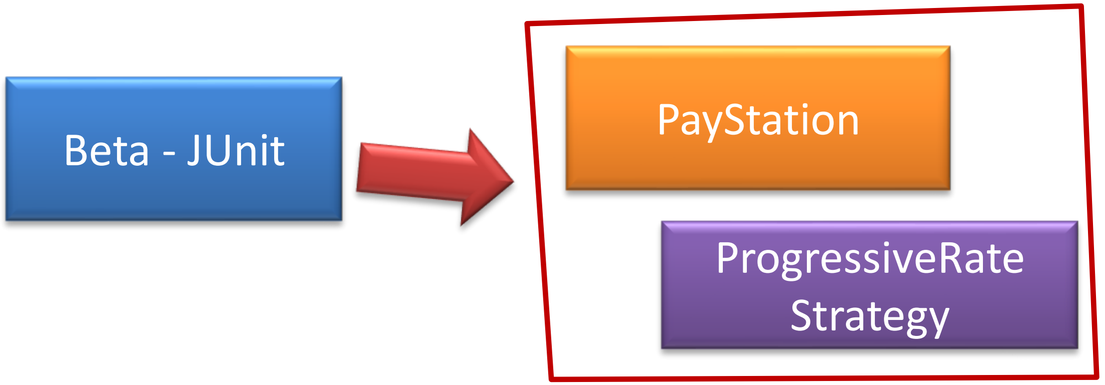
User Expectations

Exercise

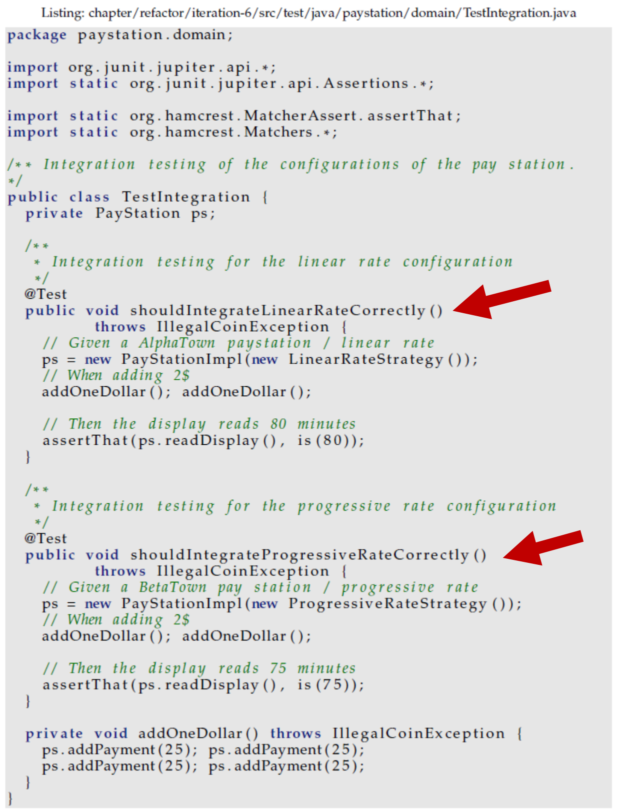
- Tricky – but
  - Give me a concrete example where having tested *all the units in isolation* does not guaranty that the system works correctly!
  - Example: The Mars Climate Orbiter...

Integration Testing the Pay Station

- I must add a testcase that validate that the AlphaTown and *as well as BetaTown* products are correctly configured!



- Just a single test that they integrate!
  - **Not repeating all the tests!**



Listing: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestIntegration.java

```
package paystation.domain;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

/** Integration testing of the configurations of the pay station.
*/
public class TestIntegration {
  private PayStation ps;

  /**
   * Integration testing for the linear rate configuration
   */
  @Test
  public void shouldIntegrateLinearRateCorrectly()
          throws IllegalCoinException {
    // Given a AlphaTown paystation / linear rate
    ps = new PayStationImpl(new LinearRateStrategy());
    // When adding 2$
    addOneDollar(); addOneDollar();

    // Then the display reads 80 minutes
    assertThat(ps.readDisplay(), is(80));
  }

  /**
   * Integration testing for the progressive rate configuration
   */
  @Test
  public void shouldIntegrateProgressiveRateCorrectly()
          throws IllegalCoinException {
    // Given a BetaTown pay station / progressive rate
    ps = new PayStationImpl(new ProgressiveRateStrategy());
    // When adding 2$
    addOneDollar(); addOneDollar();

    // Then the display reads 75 minutes
    assertThat(ps.readDisplay(), is(75));
  }

  private void addOneDollar() throws IllegalCoinException {
    ps.addPayment(25); ps.addPayment(25);
    ps.addPayment(25); ps.addPayment(25);
  }
}
```

Important Note!

- Integration testing is not system testing!

- You typically integration test that A works with B, while using doubles for C, D, and E units!
  - We will return to what ‘doubles’ are next week ☺

- System testing is testing the *full* system: A working with real B, real C, real D, and real E units.
  - Focus: Does system do what it promised to do?

AARHUS UNIVERSITET

More advanced integration testing

- The pay station case's integration is pretty simple as it is all a single process application.

- SkyCave case
  - Automated integration tests use special libraries to start a MongoDB database and a external REST server, in order to test the main server's proper interaction with these.
  - Afterwards the database + REST server is stopped and wiped for contents
  - *Integration tests are often slow to execute*
    - *Which is why they are often performed by a special build server…*

And system testing

- Karibu case
  - (Manual) system test requires
    - Two servers running clustered RabbitMQ
    - Two servers running Karibu Daemons
    - Three servers running replica set Mongo databases
  - Test cases include
    - Shutting down servers and validate data keeps flowing and reviewing log messages for proper handling of shut down events...

AARHUS UNIVERSITET

Iteration 6: Unit Testing
Pay Station

Separate Testing

- I can actually also apply *Evident Test* to the testing of the pay station by introducing a very simple rate policy

Fragment: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestPayStation.java

```
@BeforeEach
public void setUp() {
  // Given a PayStation whose rate strategy is that
  // one cent buys one minute parking time
  ps = new PayStationImpl(coinValue -> coinValue);
}
```

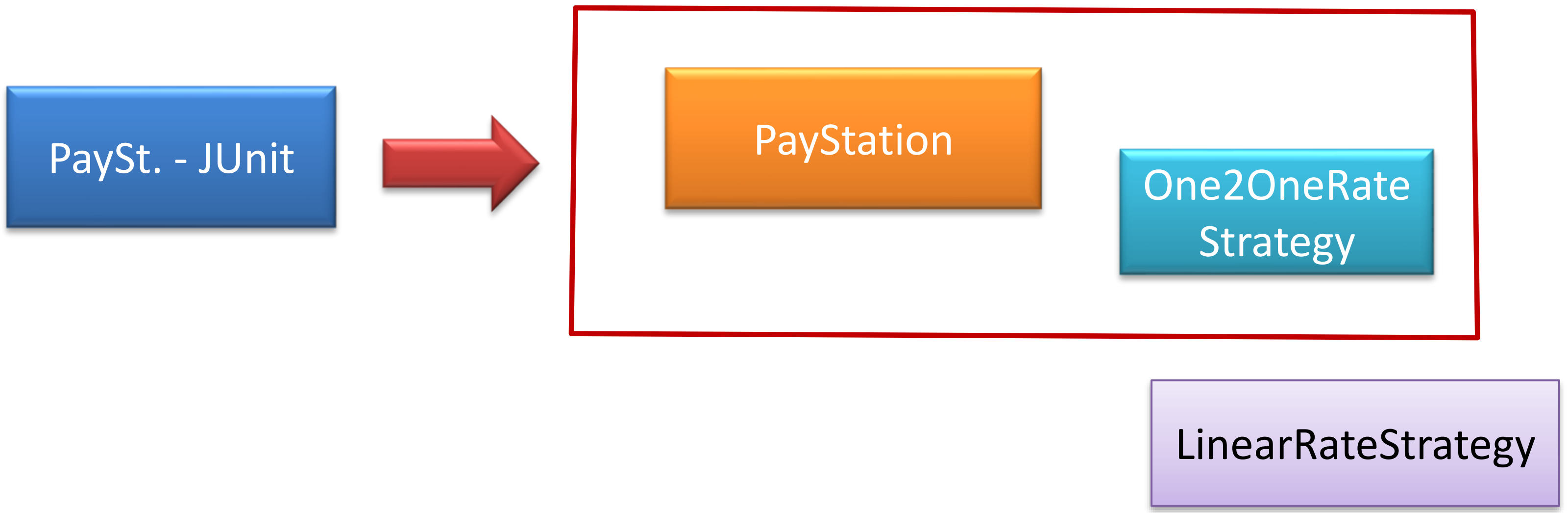
Lambda expression for: one cent = one minute

Fragment: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestPayStation.java

```
/** Test acceptance of all legal coins */
@Test
public void shouldAcceptLegalCoins() throws IllegalCoinException {
  // Given a paystation
  // When I enter 5, 10, and 25 cents
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  // Then the display should read 40
  assertThat(ps.readDisplay(), is(5+10+25));
}
```

AARHUS UNIVERSITET

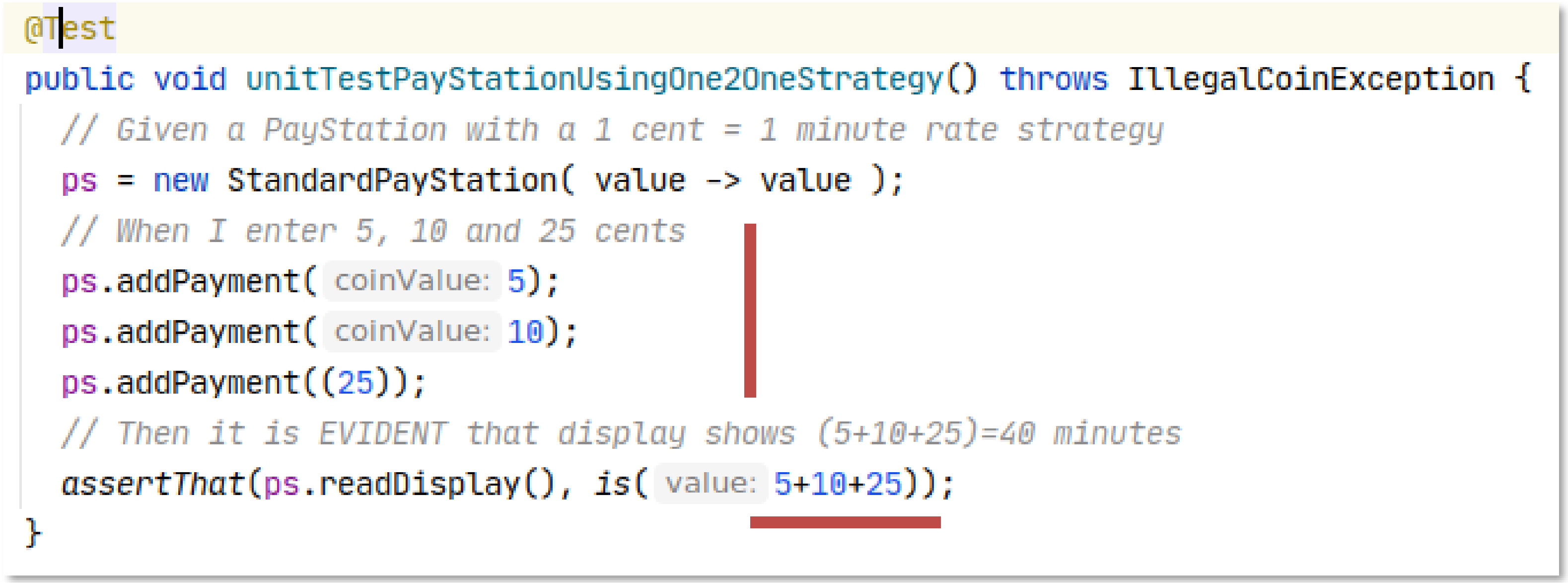
Visually



- Now **unit testing PayStation**
  - As the RateStrategy is ‘doubled’ by a simpler implementation
    - Simpler => No defects there, so any defect *must* stem from coding errors in the PayStation…

Resulting test cases

- Using this rate policy makes reading pay station test cases much easier!



```
@Test
public void unitTestPayStationUsingOne2OneStrategy() throws IllegalCoinException {
  // Given a PayStation with a 1 cent = 1 minute rate strategy
  ps = new StandardPayStation( value -> value );
  // When I enter 5, 10 and 25 cents
  ps.addPayment( coinValue: 5);
  ps.addPayment( coinValue: 10);
  ps.addPayment((25));
  // Then it is EVIDENT that display shows (5+10+25)=40 minutes
  assertThat(ps.readDisplay(), is( value: 5+10+25));
}
```

AARHUS UNIVERSITET

Outlook

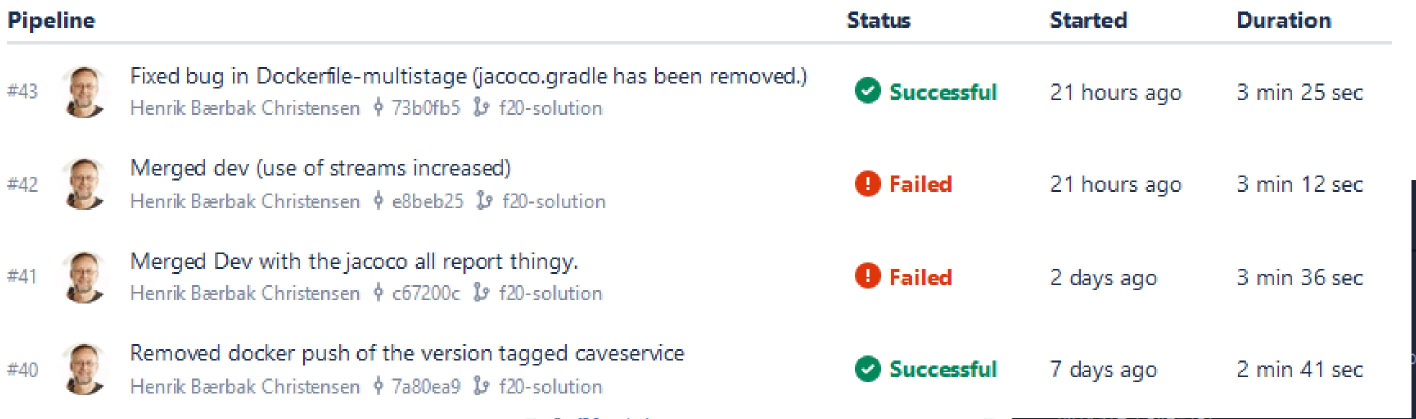
Continuous Delivery and Deployment

AARHUS UNIVERSITET

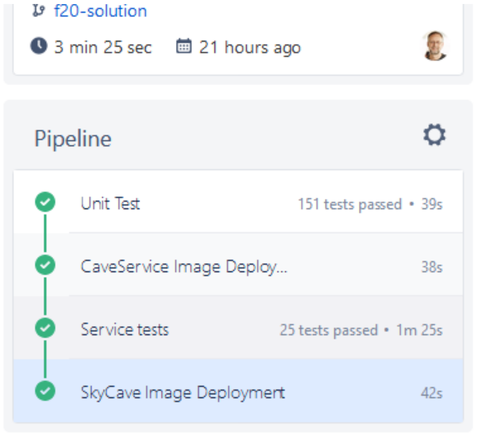
Agile on the Minute Scale

- Many software houses release and deploy software on the minute and hour scale
  - Google, netflix, uber, amazon, microsoft, …
- How
  - Comprehensive unit test suites
  - Comprehensive integration tests
  - Automated ‘build pipelines’ running on dedicated build servers
    - The pipeline will
      - Run all tests, package the system into a virtual machine and release it
      - Potentially deploy the release and put it into production

Example: Bitbucket Pipelines



Pipeline		Status	Started	Duration
#43	Fixed bug in Dockerfile-multistage (jacoco.gradle has been removed.) Henrik Bærbak Christensen 73b0fb5 f20-solution	Successful	21 hours ago	3 min 25 sec
#42	Merged dev (use of streams increased) Henrik Bærbak Christensen e8beb25 f20-solution	Failed	21 hours ago	3 min 12 sec
#41	Merged Dev with the jacoco all report thingy. Henrik Bærbak Christensen c67200c f20-solution	Failed	2 days ago	3 min 36 sec
#40	Removed docker push of the version tagged caveservice Henrik Bærbak Christensen 7a80ea9 f20-solution	Successful	7 days ago	2 min 41 sec

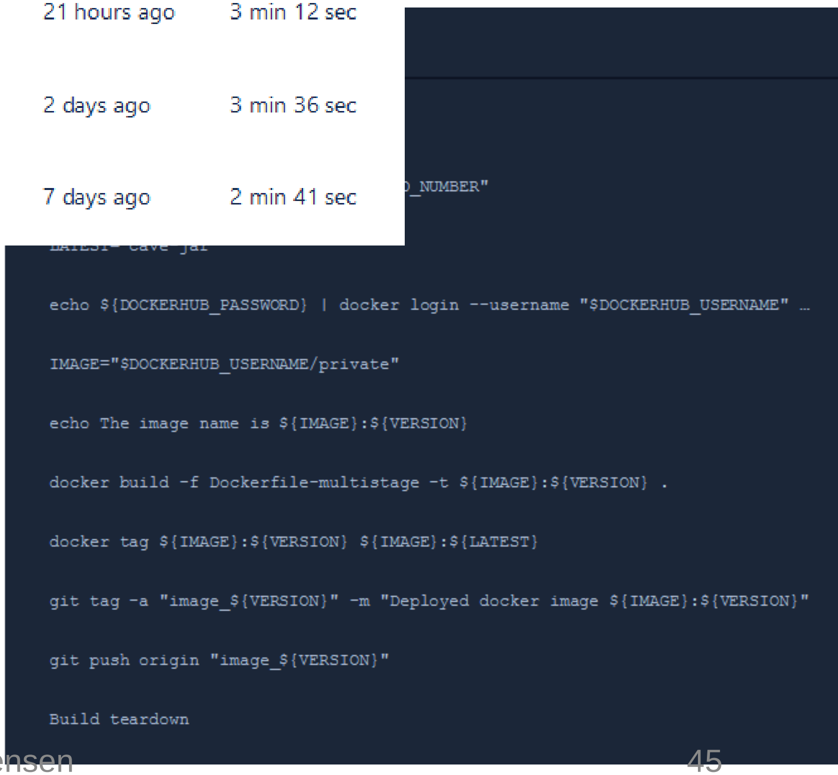


f20-solution

3 min 25 sec 21 hours ago

Pipeline

Unit Test	151 tests passed • 39s
CaveService Image Deploy...	38s
Service tests	25 tests passed • 1m 25s
SkyCave Image Deployment	42s

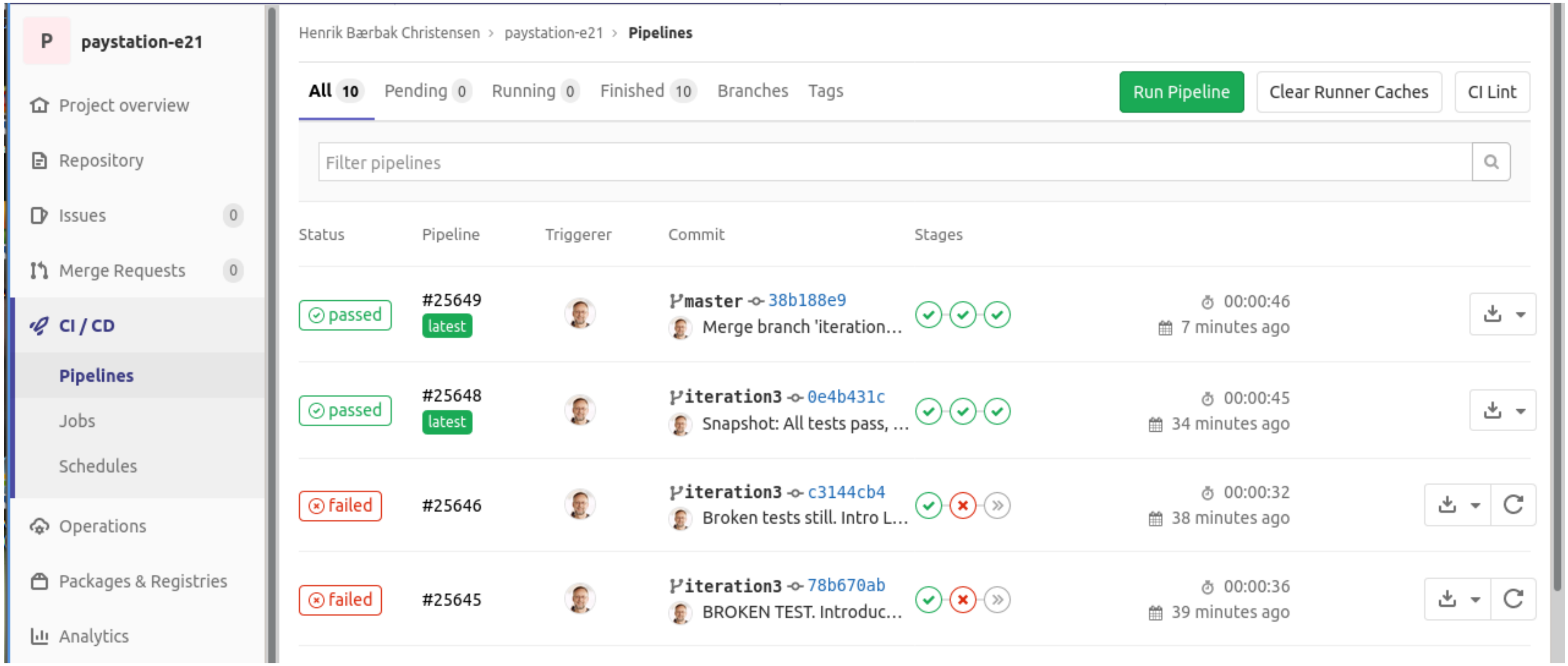


```
_NUMBER"

LATEST="cave-jar"

echo ${DOCKERHUB_PASSWORD} | docker login --username "$DOCKERHUB_USERNAME" …

IMAGE="$DOCKERHUB_USERNAME/private"

echo The image name is ${IMAGE}:${VERSION}

docker build -f Dockerfile-multistage -t ${IMAGE}:${VERSION} .

docker tag ${IMAGE}:${VERSION} ${IMAGE}:${LATEST}

git tag -a "image_${VERSION}" -m "Deployed docker image ${IMAGE}:${VERSION}"

git push origin "image_${VERSION}"

Build teardown
```

AARHUS UNIVERSITET

AU GitLab supports it

- You *can* enable it by adding a special ‘yml’ file…



Henrik Bærbak Christensen › paystation-e21 › Pipelines

All 10 | Pending 0 | Running 0 | Finished 10 | Branches | Tags

Status	Pipeline	Triggerer	Commit	Stages	
passed	#25649 latest		master 38b188e9 Merge branch 'iteration…	✓ ✓ ✓	00:00:46 7 minutes ago
passed	#25648 latest		iteration3 0e4b431c Snapshot: All tests pass, …	✓ ✓ ✓	00:00:45 34 minutes ago
failed	#25646		iteration3 c3144cb4 Broken tests still. Intro L…	✓ ✗ »	00:00:32 38 minutes ago
failed	#25645		iteration3 78b670ab BROKEN TEST. Introduc…	✓ ✗ »	00:00:36 39 minutes ago

Conclusion

AARHUS UNIVERSITET

Advice

- *Do not code in anticipation of need, code when need arise...*

- Automatic tests allow you to react when need arise
  - because you dare refactor your current architecture...

Refactoring

- When 'architecture refactoring' need arise then

- A) Use the old functional tests to refactor the architecture **without** adding new or changing existing behavior
- B) When everything is green again then proceed to introduce new/modified behavior
- C) Review again to see if there is any dead code lying around or other refactorings to do.

AARHUS UNIVERSITET

Discussion

- These refactorings shown here are very local, so the ‘architecture decisions’ are also local.
- However sometimes you need to make larger architectural changes that invalidate the test cases ☹
  - Changing API or the way units are used
  - Ex: Changing persistence from file to RDB based
- What to do in this case?
  - *Define a path (even a long one) of small tasks that keep tests running! Even if it means making code that later must be removed*