TDD: seriously, try it ! 😃

Jan / Feb 2021

dynatrace

First thing first

Why this session?

- Review the basics around TDD
- Understand the benefits of building code designed by tests
  VS
  building the tests after we write our code
  (before VS after)
- See an example
- (I hope) learn some tips
- … so do not run away, please! ☺

Who am I?

Nacho Cougil

- Senior Software Engineer at Dynatrace
- TDD & clean code *fan*
- Started to write in Java before the effect 2000
- Founder of the Barcelona Java Users Group &

  co-founder of the Java and JVM Barcelona Conference (JBCNConf)
- Father, former mountain marathon runner 😆

@icougil

Ask questions at any time, please!

- Raise your hand at any moment!

Warning

Based on my *personal* experience

Agenda

- Where it cames from?
- Advantages & disadvantages
- Process & rules
- Good habits
- Example
- Final Recap

Agenda

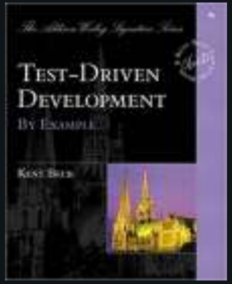
- **Where it cames from?**
- **Advantages & disadvantages**
- Process & rules
- Good habits
- Example
- Final Recap

A bit of history

- Kent Beck "rediscovered" TDD when writing first testing framework SUnit in 1994.

  "Take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output."

- TDD became part of Extreme Programming book in 1999.



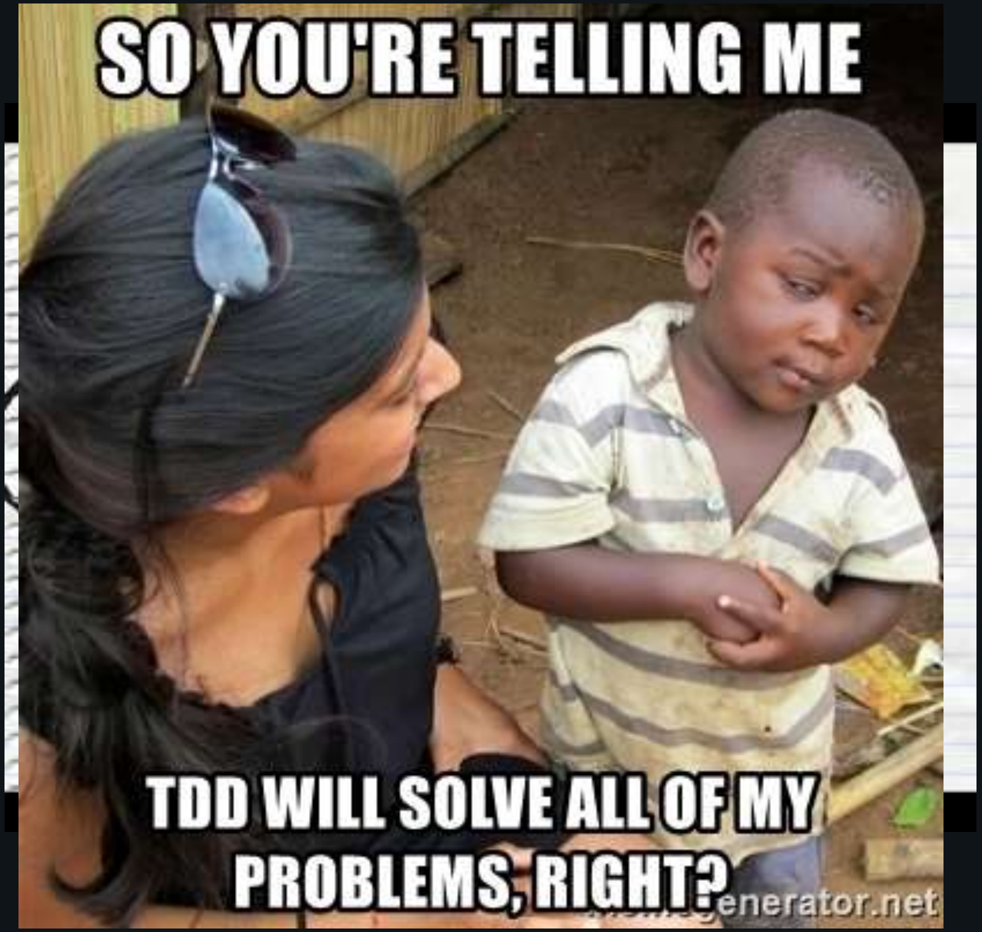
For a moment, imagine...

- ... you can find defects earlier ⏱ (when running or designing your tests)
- ... you can easily detect regression errors ●
- ... you follow a simple process ⚙ that helps you to develop your software in a straightforward way
- ... your software is going to be easier to refactor because of ✓ green (safety net)

For a moment, imagine...

- ... your software guides you how a consumer will use your component
- ... tests are living documentation (really !!)
- ... the software you write it is likely to have less bugs
- ... your development costs 💰 will be lower

References:
- Test-driven development as a defect-reduction practice (IEEE)
- Guest Editors' Introduction: TDD--The Art of Fearless Programming (computer.org)



Yes, TDD has some disadvantages

- It is not so easy to start with 😅
  (not many good places to help, majority of people are not using it daily)

- Has a high learning curve 😟
  (depends directly on the complexity of your code + your design and engineering capabilities)

- Can be a large investment 💸 in a team (depending on the experience of the members + requires time ⌛ & effort 🙇‍♀️🙇‍♂️)

Yes, TDD has some disadvantages

- It is really easy to forget 😓 about it (especially if you are not in an environment / team that does not encourage its use or you cannot experiment with it comfortably)

- It provokes resistance ☹ (in some people)

- Can be corrupted and lead to the syndrome of reaching the highest possible level of *coverage* 🙈

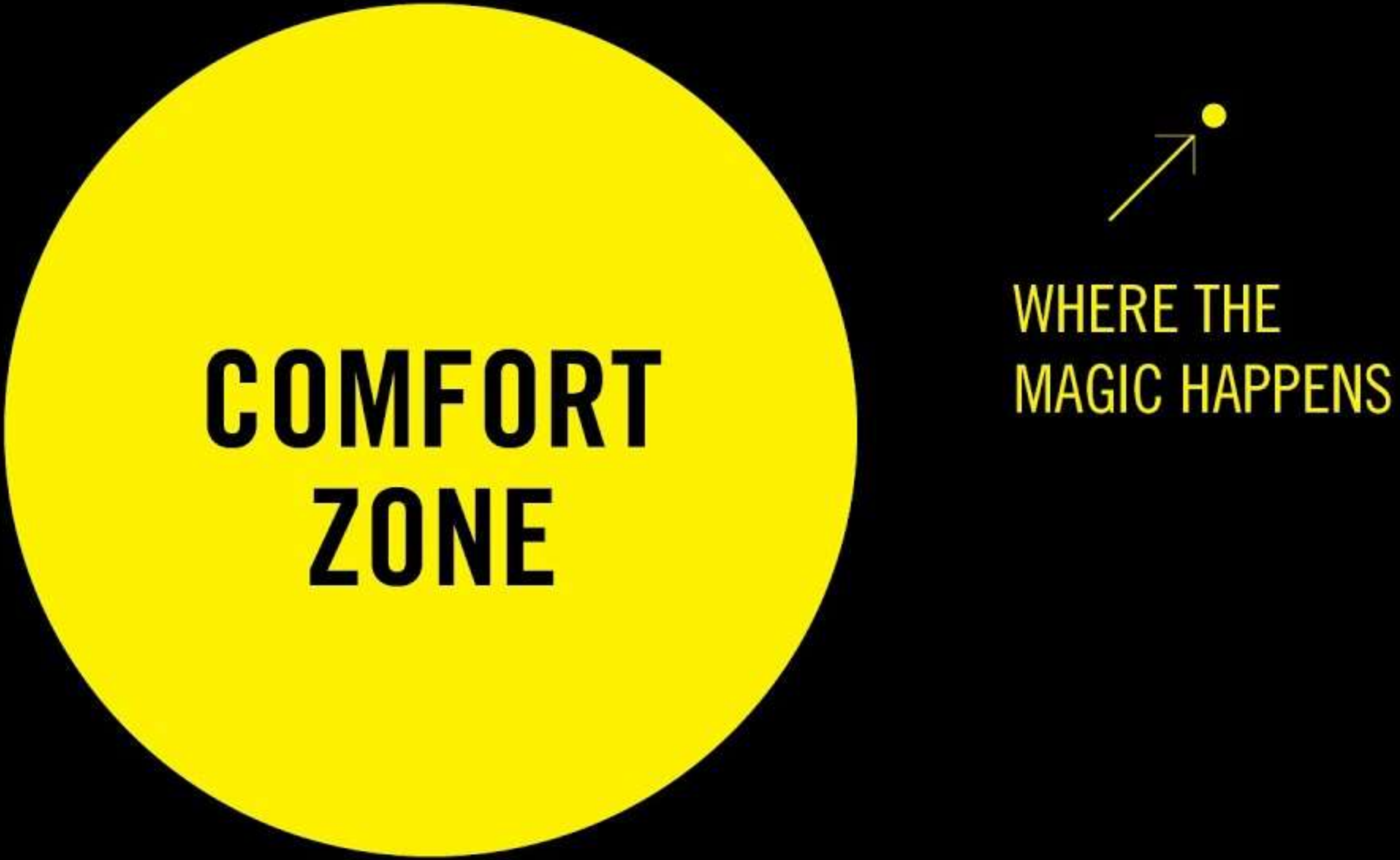
- It is extremely difficult 😃 to master at it

So, … how it is?

- You learned how to write code time ago,…

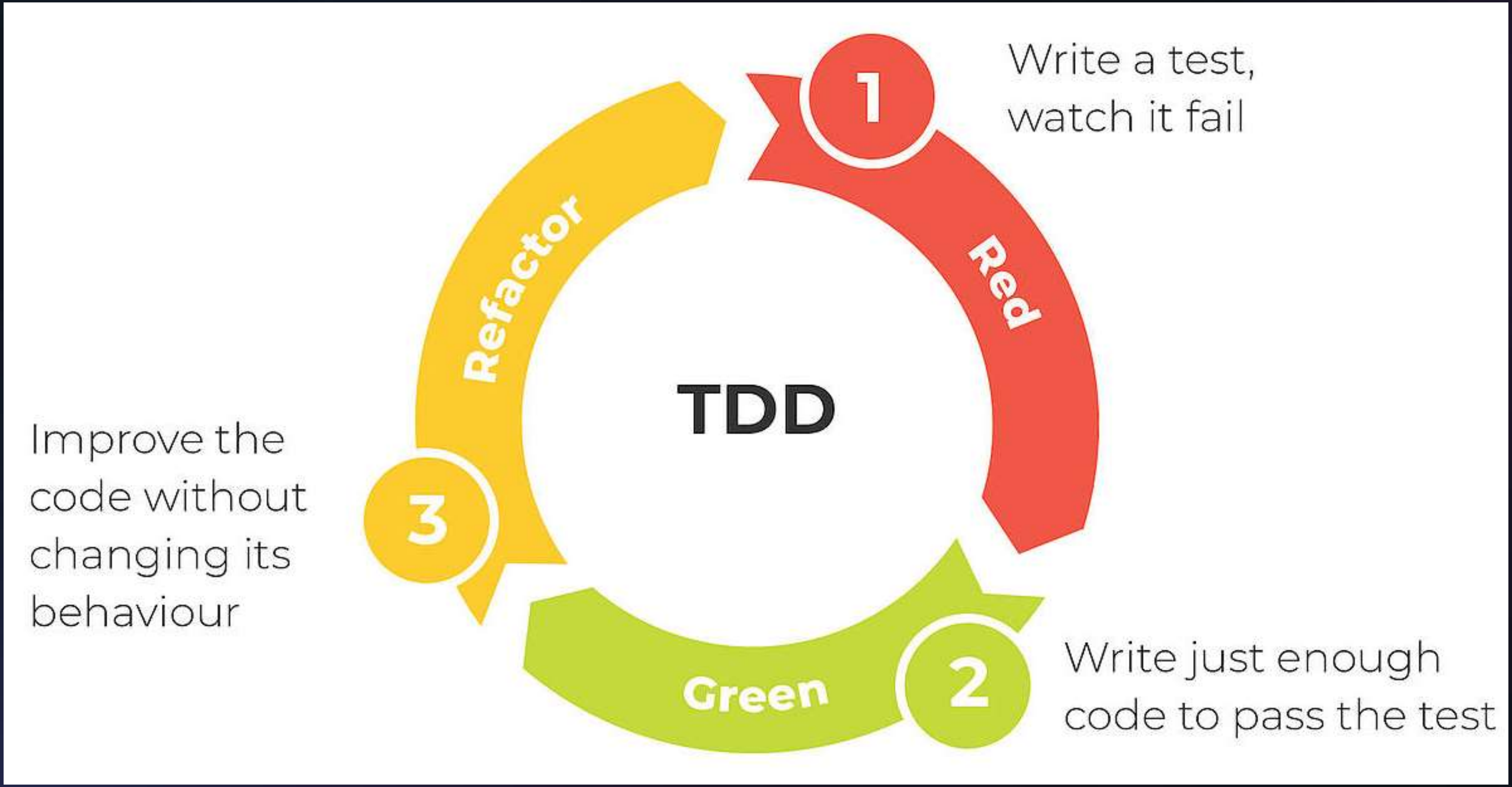
- …and now you'd may learn a different way of writing software.

  Like "*learning to ride a very different bike*",… but being older 😆


COMFORT ZONE
WHERE THE MAGIC HAPPENS

Agenda

The process


1
Write a test, watch it fail
Red
2
Write just enough code to pass the test
Green
3
Improve the code without changing its behaviour
Refactor
TDD

The rules

- You are **not allowed** to write any **production code unless** it is to make a failing **unit test pass**.
- You are **not allowed** to write **any more of a unit test** than is sufficient to **fail**.
- You are **not allowed** to write any more **production code** than is **sufficient to pass** the one failing unit test.

Robert C. Martin

(uncle Bob)

➔ Focus on **building tests 1st** that will help us demonstrate the behaviour we want.

➔ We will start writing tests and we can't write very much apart from a unit test. **Tests are first class citizens**

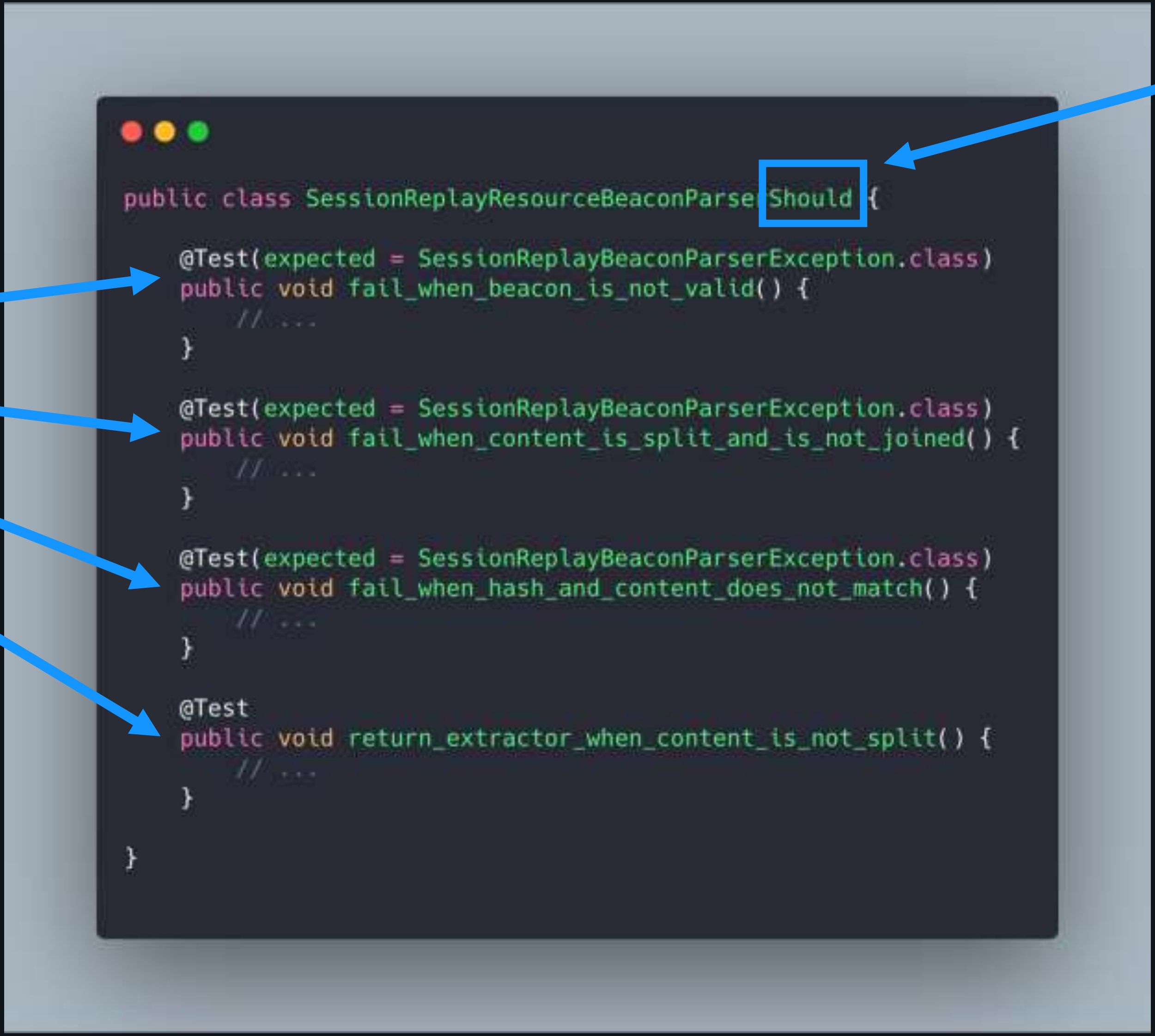
➔ No overengineering, simple code, simple solution: **make the test pass**

Agenda

Good habits

- Before you write production code, check that your test is failing ●!
- Each test has only 1 reason to fail
- Write the assertion first

Tests naming convention



```
public class SessionReplayResourceBeaconParserShould {

    @Test(expected = SessionReplayBeaconParserException.class)
    public void fail_when_beacon_is_not_valid() {
        // ...
    }

    @Test(expected = SessionReplayBeaconParserException.class)
    public void fail_when_content_is_split_and_is_not_joined() {
        // ...
    }

    @Test(expected = SessionReplayBeaconParserException.class)
    public void fail_when_hash_and_content_does_not_match() {
        // ...
    }

    @Test
    public void return_extractor_when_content_is_not_split() {
        // ...
    }

}
```

Ending the class name with *Should* will “force “ you to start describing an action for every test you may create

Describe the expected behaviour, not the internals.

Our tests should describe behaviour in *plain english*

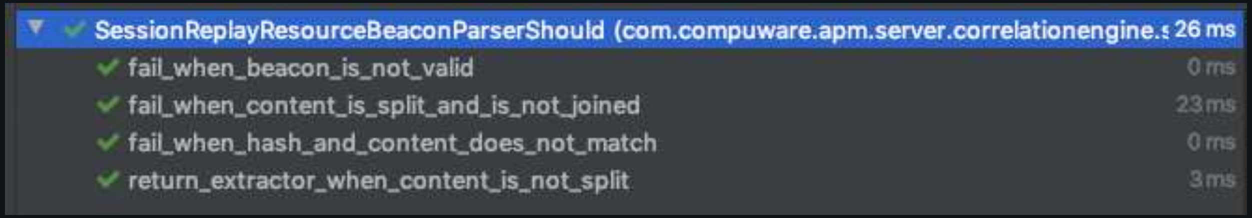
The result will be focused on the business

- Our tests methods will only describe behavior. Therefore we will have a better understanding on what this class does

- Our tests will be *more clear*

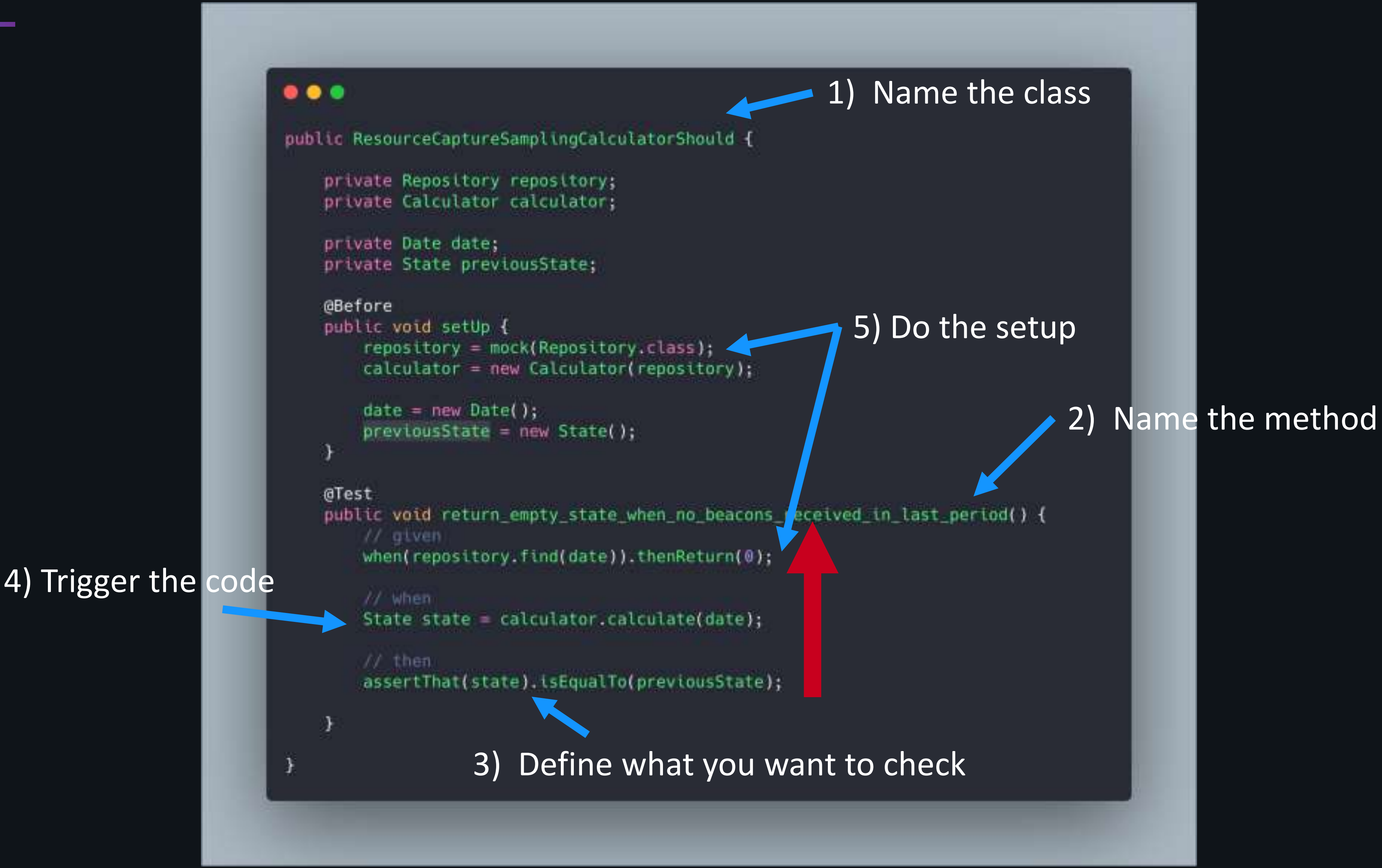
- If some tests fails, we can have a look and see easily which case is failing

- We don't need to get into details on any particular test if we don't need it



```
▼ ✓ SessionReplayResourceBeaconParserShould (com.compuware.apm.server.correlationengine.s 26 ms
      ✓ fail_when_beacon_is_not_valid                                                      0 ms
      ✓ fail_when_content_is_split_and_is_not_joined                                      23 ms
      ✓ fail_when_hash_and_content_does_not_match                                          0 ms
      ✓ return_extractor_when_content_is_not_split                                         3 ms
```

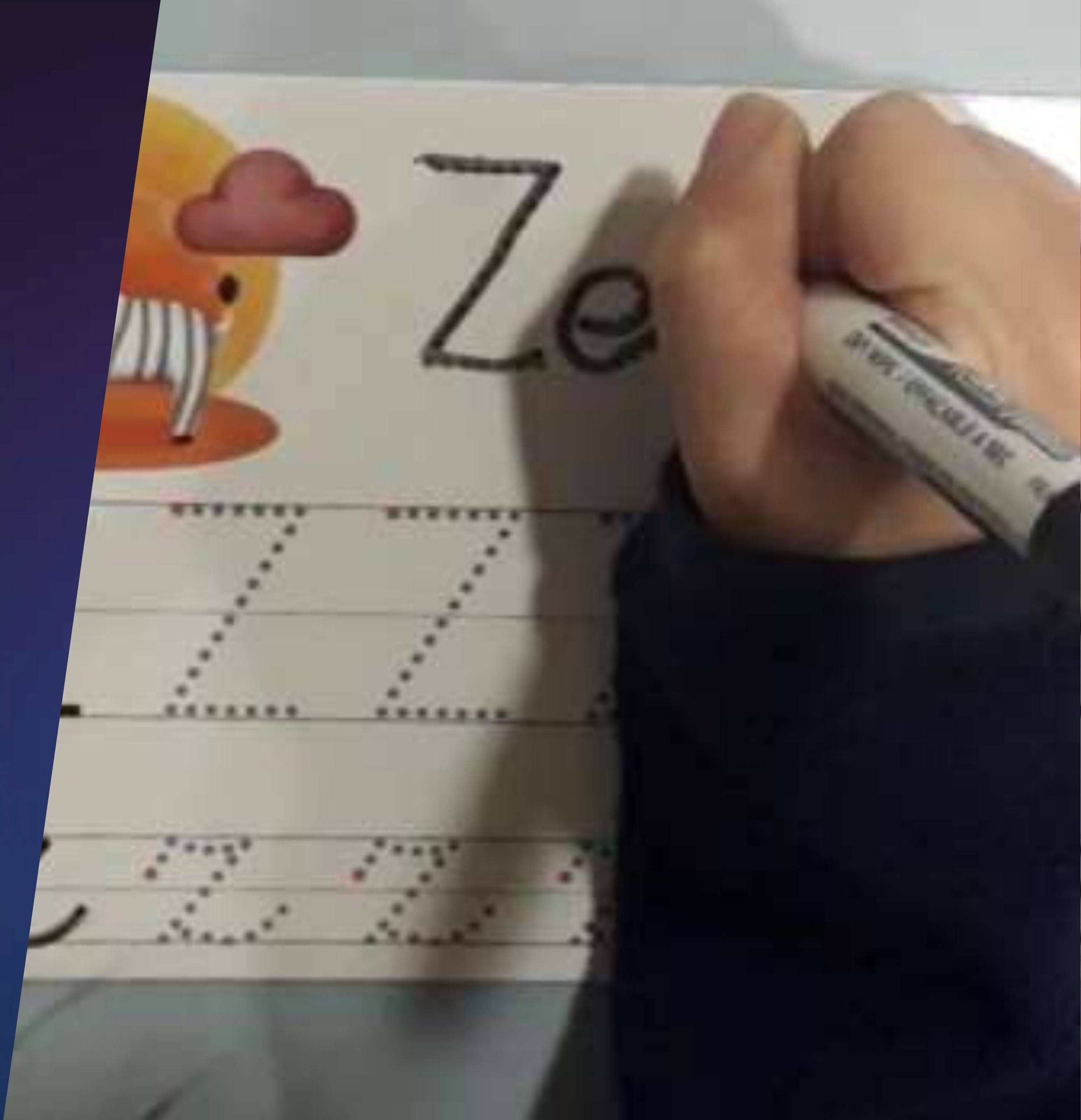
Test creation order



```java
public ResourceCaptureSamplingCalculatorShould {

    private Repository repository;
    private Calculator calculator;

    private Date date;
    private State previousState;

    @Before
    public void setUp {
        repository = mock(Repository.class);
        calculator = new Calculator(repository);

        date = new Date();
        previousState = new State();
    }

    @Test
    public void return_empty_state_when_no_beacons_received_in_last_period() {
        // given
        when(repository.find(date)).thenReturn(0);

        // when
        State state = calculator.calculate(date);

        // then
        assertThat(state).isEqualTo(previousState);

    }
}
```

But, how to start?

- Start little by little...
  - Katas
  - Proofs of concept
  - Personal projects
- Practice...
  - ... and keep practicing
  - ... and continues
  - ... until you finally internalize it □♀😉
- Check online content
- Practice with somebody else (***pair-programming***) - *next slide* ☞-



Pair programming

- An **Extreme Programming** (XP) Practice in which 2 developers participate in one development effort at one workstation.
- One, the *driver*, **writes code** while the other, the *navigator* or *observer*, **reviews the code** as it is typed in. The 2 engineers **switch roles** frequently.
- While reviewing, the *observer* also considers the "**strategic**" direction of the work, coming up with ideas for improvements and likely future problems to address. This is intended to free the driver to focus all of their attention on the "**tactical**" aspects of completing the current task, using the observer as a safety net and guide.

Agenda

- Where it cames from?
- Advantages & disadvantages
- Process & rules
- Good habits
- **Example**
- Final Recap

Warning – Live coding!

Kata: Film Recommendation Service

- We have just started working in a startup that sells and rents out films over the Internet

- Our product managers have told us that they want to add new functionalities by building a service that allows them to *recommend* films to the users of the current platform.

- **Requirements**:
    - Build a service that returns a list of films that are associated with a particular genre
    - By default, the result must be ordered according to the average rating given to the films by the users
    - A film should contain at least a title, a year when it was published, one or more tags and one or more genres

- **Example:**
    - recommendationService.filmsByGenre("science-fiction")

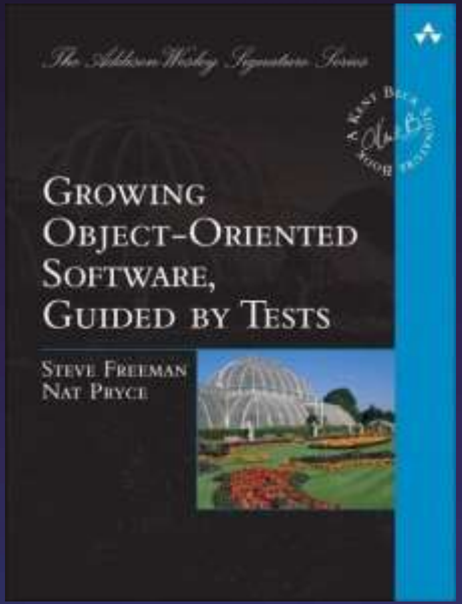
https://github.com/icougil/FilmRecommendationService

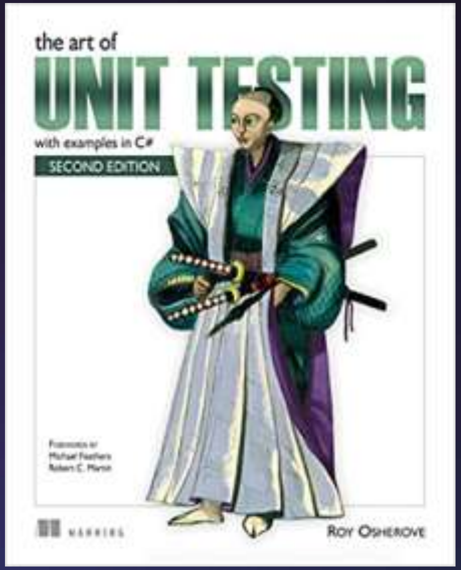
Agenda

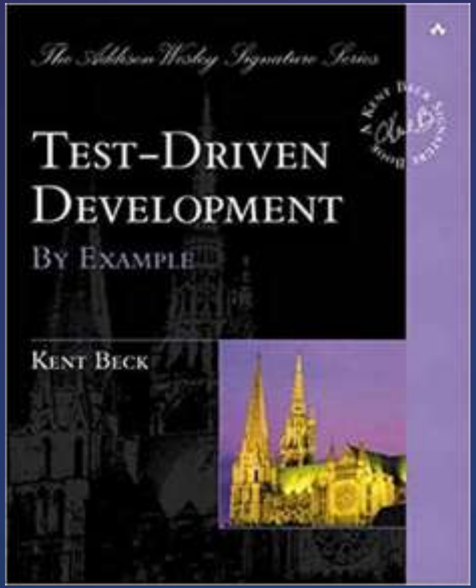
- Where it cames from?
- Advantages & disadvantages
- Process & rules
- Good habits
- Example
- **Final Recap**

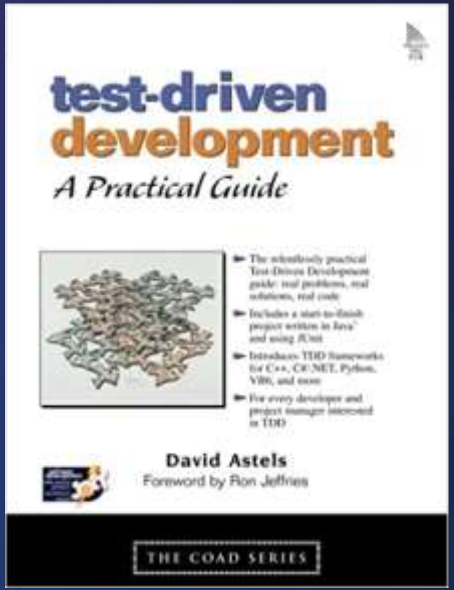
Recommended content

- Martin Fowler (main concepts around testing)
  - https://www.martinfowler.com/bliki/TestDouble.html
  - https://martinfowler.com/articles/mocksArentStubs.html
  - https://martinfowler.com/articles/practical-test-pyramid.html
- James Shore (JS practices)
  - https://www.youtube.com/user/jdlshore/videos
- Jason Gorman (Java practices)
  - https://www.youtube.com/user/parlezuml/videos


The Addison-Wesley Signature Series
Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce







Final Recap

- TDD helps you to develop your code in a simple and effective way (better modular design) 👌
- It is difficult to adapt to its mechanics and sometimes is difficult 😅 to maintain its practice…
- It worth a try 😉 (your software will have less bugs 🐛 & it will be easier to maintain 💰)
- There are some tips 💡 (could make your life easier)
- Try to pair, it will help 😄 you (a lot)
- Practice, practice & practice □ again

Questions?

nacho@cougil.com

https://nacho.cougil.com

@icougil

I need your feedback, please:
https://bit.ly/tdd-seriously-try-it-feedback

ShaytardGIFS
Thank You.