



adaptTo()

APACHE SLING & FRIENDS TECH MEETUP
BERLIN, 25-27 SEPTEMBER 2017

How to write clean & testable code without
losing your mind - Andreas Czakaj

How did you learn what you know today?

“There are three kinds of men:

1. The one that learns by **reading**.

2. The few who learn by **observation**.

3. The rest of them have to **p****
on the electric fence for themselves.”

this is us
learning AEM...

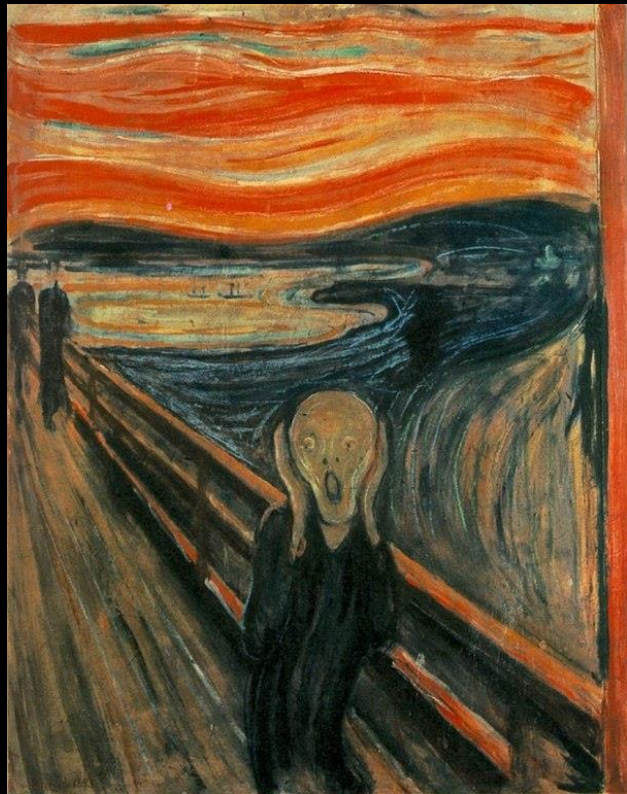
-- *Will Rogers*

Our early experience with AEM



- Painful upgrades
- Slow progress
- Hard to change code
- Hard to fix bugs
- Regression bugs

Our early AEM code



- Low coverage
- Expensive tests
- Slow & fragile tests
- Extensive logging
- High coupling
- Low cohesion

What we actually wanted

- If you need to maintain, extend and adapt over several years you'll want...
 - reliable software
 - confidence in code
 - understanding
 - control
 - adaptability, extensibility

How we actually wanted to work

- If you need to maintain, extend and adapt over several years you'll need...
 - regression tests
 - high coverage
 - knowledge management
 - control over dependencies
 - high cohesion + low coupling

Refuse, Resist

Test Driven/First Development

- In 2005 I was a freelancer working on a project using Servlets & JSPs, Hibernate & JPA, SOAP etc.
- They *forced* me to do TDD
- After some weeks of *futile* protest...
- ... I realized how TDD works & why it's **great**

What IS great about TDD?

- Fewer bugs (duh!)
- Permanently up-to-date documentation
- You work faster (fast response, no maven)
- You have a tool to check your **design decisions**
- ... in a straightforward way

Test **Driven** Development

- If it's **hard to test** it's likely **poorly designed**
- Focus on creating testable code
- For your design decisions you should ask:
 - what makes the code more testable?
 - which of my options yields more testable code?

Test **First** Development

Test	Prod. Code
<pre>assertEquals(6, fac(3));</pre>	<pre>int fac(int n) { return n > 1 ? n * fac(n-1) : 1; }</pre>

Account	Debit	Credit
Furniture	€ 1,500	
Cash		€ 1,500

- Write the test first
- Write prod. code
- Run test
- Refactoring
- ~ Double-Entry Accounting

Refactor production code, keep existing tests

Test

```
assertEquals(  
    6, fac(3)  
);
```

Prod. Code

```
int fac(final int n) {  
    int out = 1;  
    for (int i = n; i > 1; i--) {  
        out *= i;  
    }  
    return out;  
}
```

As a result...

- fewer bugs -> reliable software
- high coverage -> control, confidence in code
- tests as documentation -> understanding
- refactoring -> adaptability, extensibility
- -> That's what we were looking for, right?
- -> I'll tell my AEM developers about it!

However, the team was not impressed



Your boss telling you to
“**get 100% coverage**”
does not work...
... especially, when you
have to deal with code
like **this**:...

How do you test THIS?

```
public void onEvent(final EventIterator events) {  
    while (events.hasNext()) {  
        final Event event = events.nextEvent();  
        Session session = null; Node node = null;  
        try {  
            String path = event.getPath();  
            if (path.endsWith(JcrConstants.JCR_CONTENT)) {  
                session = repository.login(new Credentials() { /*...*/ });  
                node = session.getNode(path);  
                if (node.hasProperty("cq:template") &&  
                    "...".equals(node.getProperty("cq:template").getString())) {  
                    processExport(node);  
                }  
            }  
        } catch (RepositoryException e) { /*...*/ } finally { /*logout*/ }  
    }  
}
```

Team: “Isn’t it obvious?”

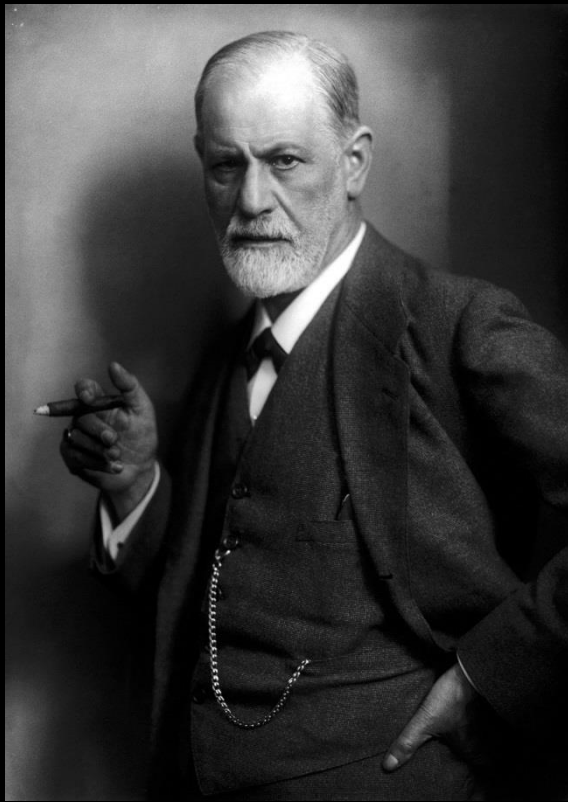
- “We already tried everything™” ...
 - mocks
 - end-to-end tests
 - in-container tests
- “We can reach *some* coverage...”
- “...but, obviously, it will be
a lot of tedious work and pretty expensive”

Team decision



- “You can’t test everything in AEM without losing your mind”

No need to lose your mind



- TDD is **NOT** about **testing** cleverly
- TDD is about writing **code** in a different way:
clever == testable

Digging into the problem - together



- We spent some time figuring out new design approaches
- Here's what we found out
really works...

Some AEM Examples

Busy, busy method

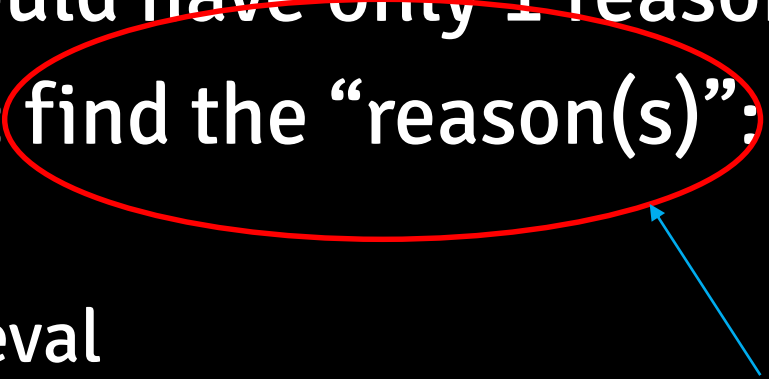
```
public void onEvent(final EventIterator events) {  
    while (events.hasNext()) {  
        final Event event = events.nextEvent();  
        Session session = null; Node node = null;  
        try {  
            String path = event.getPath();  
            if (path.endsWith(JcrConstants.JCR_CONTENT)) {  
                session = repository.login(new Credentials() { /*...*/ });  
                node = session.getNode(path);  
                if (node.hasProperty("cq:template") &&  
                    "...".equals(node.getProperty("cq:template").getString())) {  
                    processExport(node);  
                }  
            }  
        } catch (RepositoryException e) { /*...*/ } finally { /*logout*/ } }  
    }
```

... in a busy, busy class

```
private void processExport(final Node node) { // original method: ~100 loc
    try {
        String group;
        if (node.hasProperty(PROPERTY_GROUP)) {
            group = node.getProperty(PROPERTY_GROUP).getString();
        } else {
            LOG.warn("There is no group. Stop export.");
            return;
        }
        /*...*/
        File csvFile = File.createTempFile("...", "csv");
        exportToFile(group, /*...*/, csvFile);
        /*...*/
    } catch(/*...*/) { /*...*/ } finally { cleanupTempFiles(); /*...*/ }
```

Single Responsibility Principle & Clean Code

- “A class should have only 1 reason to change”
- Clean Code: find the “reason(s)”:
 - Event loop
 - Data retrieval
 - Data processing
 - Export to FileSystem
 - ... in specialized format



this part might
lead to
philosophical
debates...

Single Responsibility Principle & TDD



- ... or you can look at it from a TDD point of view
- TDD: imagine writing tests for it...
 - -> *meh*

The TDD way

- Don't be clever at testing...
- ... instead, aim at fixing the code
- Write the tests first...
- ... then find the code that works best for the tests
- Start simple – but be thorough & complete

Specify your rules with plain unit tests

```
@Test
public void test_toList_allEmpty() throws Exception {
    List<String> row = exporter.toList(new MyExportData());
    assertEquals("It should export nulls as empty strings",
        Arrays.asList("", ""), row);
}

/** production code*/
/* ... */
items.stream()
    .filter(item -> item != null)
    .map(this::toList)
    .forEach(rowConsumer::accept); // List::add in tests,
/* ... */                          // OutputStream.write in prod code
```

Don't be clever at testing, fix the code instead

```
class ExportEventListener {
    public void onEvent(final EventIterator events) {
        final Dao dao = new DaoJcrImpl(repository);
        final MyService service = new MyService(dao);
        while (events.hasNext()) {
            final Event event = events.nextEvent();
            final ProcessingContext ctx = toProcessingContext(event);
            service.process(ctx);}}}

public class MyService {
    private final Dao dao;
    private final Exporter exporter;
    public void process(final ProcessingContext ctx) {
        final Data data = dao.getData(ctx);
        exporter.export(data);}}
```

- TDD:
 - Event loop: simple setup
 - DAO: AemMock / MockJcrSlingRepository
 - Data processing: POJOs -> unit testable
 - Export to FileSystem: using POJOs
 - Specialized format: plain Java -> unit testable
 - Service: each “piece” can be replaced by stub

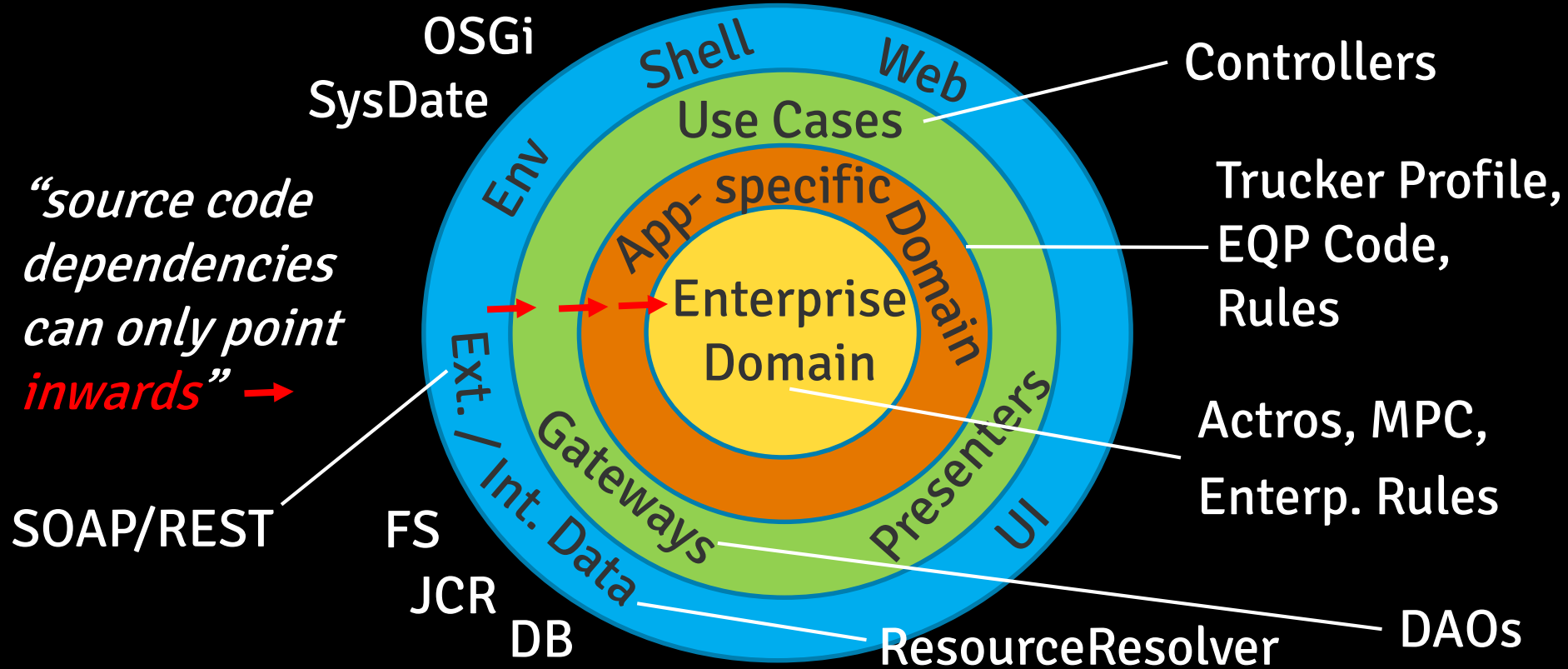
- Clean Code: 1 class per “reason” / “concern”
 - Event loop (Entry point)
 - Data retrieval (DAO + Adapter)
 - Data processing (Domain logic)
 - Export to FileSystem (Gateway)
 - Specialized export format (Strategy)
 - Service: integrate pieces (Dependency Injection)

TDD & Clean Code

- Bottom line:
 - To make TDD work you need to decouple code
 - ... using SRP, DI, “Ports” etc.
 - ... i.e. Clean Code principles
 - -> TDD will lead you to Clean Code
 - (... if you stick to the rules...)
- But how does it work on application-scale?

Application-scale Dependency Management

The Dependency Rule – for AEM



Simple Example

- **Approach 1**

```
package com.mercedes-benz.trucks.domain;

public class Actros {
    static Actros fromRequest(HttpServletRequest req) { /* ... */ };
}
```

- **Approach 2**

```
package com.mercedes-benz.trucks.integration.web;

public class ActrosAdapter {
    Actros toActros(HttpServletRequest req) { /* ... */ };
}
```

- **=> which one works according to the Dependency Rule?**

Simple Example

■ Approach 1

```
package com.mercedes-benz.trucks.domain;

public class Actros {
    static Actros fromRequest(HttpServletRequest req) {/* ... */};
}
```

■ Approach 2

```
package com.mercedes-benz.trucks.integration.web;

public class ActrosAdapter {
    Actros toActros(HttpServletRequest req) {/* ... */};
}
```

- => Keep outer layer dependencies out of inner layers

Ports & Adapters / Dependency Inversion

- “But the Use Cases and Domain objects need data from the outer layers!”
- “How can I read from and write to the outer layer without depending on it?”
- -> Dependency **INVERSION**
- -> “Ports”
(simplified: outer layers behind **interfaces**)

TDD + Dependency Rule = key to success

Identify the Domain

- ... yes, it exists
- Start at the core: start at the Domain
 - ... it's the code that's the easiest to test
 - -> you'll start at 100% coverage
 - ... then stay at 100%
- -> E.g. keep the ResourceResolver / JCR out of your Domain, Rules and APIs

This approach scales & works also for...

- Services
- Models
- Components
- Workflows
- Listeners
- ...

How does it pay off for us in the real world?

- We reach 100% coverage (in new code)
- Sometimes, we still have bugs...
 - ... but mostly in the front end / JavaScript
 - ... or because of production data mismatch
- We're faster
- Developers no longer lose their minds ;-)

- [Ports & Adapters](#)
- [DAO / Repository Pattern](#)
- [Clean Architecture / Dependency Rule](#)
- [3 Rules of TDD / Bowling Game Kata](#)
- [Single Responsibility Principle](#)
- [“The more testing you do...”](#)

<https://github.com/mensemmedia/adaptTo2017>

- **Exporter:**
Refactored version of a real life project
(w/ 100% coverage)



#thx

Questions?

Andreas Czakaj (CTO)

mensemedia Gesellschaft für Neue Medien mbH

Neumannstr. 10

40235 Düsseldorf

Germany

Email: andreas.czakaj@mensemedia.net

Twitter: @AndreasCzakaj

Blog: <https://blog.acnebs.com/>