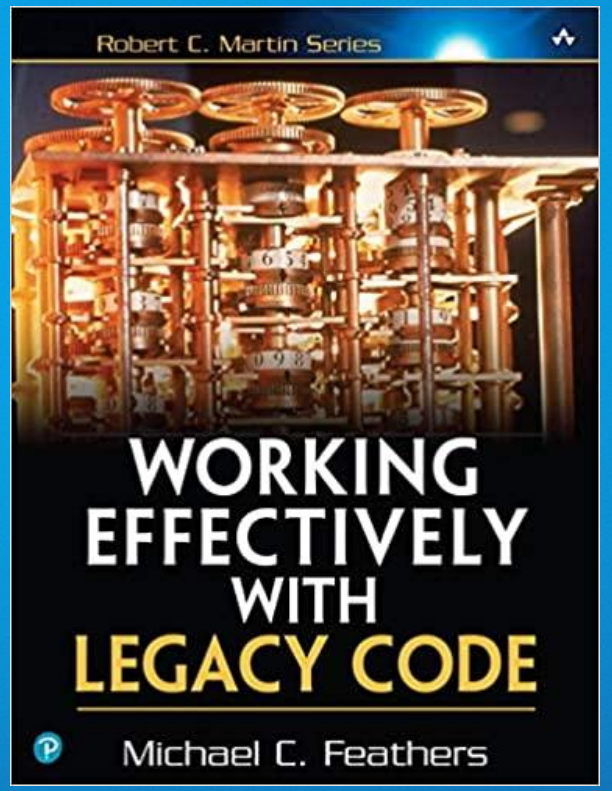
Have a look at «Working effectively with Legacy Code»

by Michael C. Feathers

Luzern, 12. November 2021

ALCOR Academy Training



Motivation

we learned a lot about …

- TDD cycle red-green-refactor
- Transformation Priority Premise
- Object Calisthenics
- Code Smells
- Solid Principles
- Coupling and Cohesion
- Refactoring
- Connascence
- Test Doubles

and so on …

Motivation

but … most of the time a developer has to handle code that
does not follow all these rules!

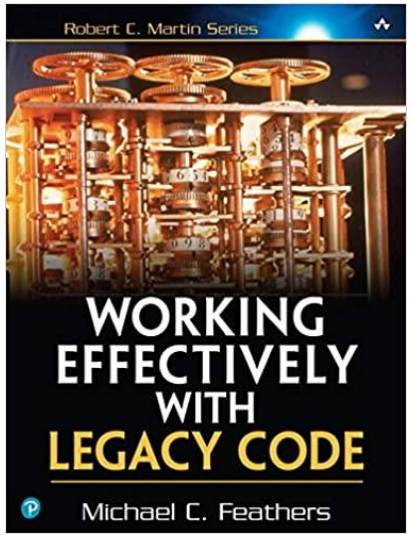
What about legacy code and the need to change it?

Michael J. Feathers:

«Legacy code is often used as a slang term for difficult-to-change code that we don't understand. To me, legacy code is simply code without tests.»

«With tests, we can change the behaviour of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.»

Everything following is based on:



[Feathers] Working Effectively with Legacy Code, Michael C. Feathers, 2005

Changing Software

Four reasons to change: *

1. Adding a feature
2. Fixing a bug
3. Improving the design
4. Optimizing resource usage (time or memory)

.. and there are two ways to do that:

Edit and Pray*

Like «working with care», but safety isn't a function of care.

Working with care doesn't do much for you if you don't use the right tools and techniques.

Cover and Modify*

Like «working with a safty net» when we change it.

Covering software means covering it with tests, giving feedback if we get it right.

* [Feathers] Chapter 1 – Changing Software

The Legacy Code Change Algorithm

The day-to-day goal in legacy code is to make functional changes that deliver value while bringing more of the system under test.

*1. Identify change points
2. Find test points
3. Breaking dependencies
4. Write tests
5. Make changes and refactor (beginning with TDD)

.. after step 5 the code isn’t «legacy» any more!

* [Feathers] Chapter 2 – Working with Feedback

Step 1 and 2: «Identify change point» and «Find test point»

The «**change point**» depends on the architecture of the code and the feature to add or the bug to fix.

For the «**test point**» Michael Feathers introduces a concept called

Seam

* **« A Seam is a place where you can alter behaviour in your program without editing in that place »**

.. and a «Seam» is the starting point for the next step of «breaking dependencies»

* [Feathers] Chapter 4 – The Seam Model

Seam

```
public class CustomSpreadsheet extends Spreadsheet {
   …
   public Spreadsheet buildMartSheet() {
      …
      Cell cell = new FormulaCell(this, "A1", "=A2+A3");
      …
      cell.recalculate();
      …
   }
}
```
*

NO

Is this a seam ?

Can we change the behaviour at this point for test?

Can we change which «recalculate» method is called?

```
public class CustomSpreadsheet extends
Spreadsheet {
   …
   public Spreadsheet buildMartSheet(Cell cell) {
      …
      recalculate(cell);
      …
    protected
   private static void recalculate(Cell cell) {
      …
   }
}
```
*

Is this a seam ?

YES

```
public class TestingCustomSpreadsheet extends
CustomSpreadsheet {
   protected void recalculate(Cell cell) {
      ...
   }
}
```

* [Feathers] Chapter 4 – The Seam Model

Step 3: «Breaking dependencies»*

Dependencies are often the most obvious impediment to testing.

- difficulty instantiating objects in test
- difficulty running methods in test

Often in legacy code, you have to break dependencies to get tests in place!

The challenge is to make the FIRST INCISIONS safe and minimal invasive to bring the code under test.

* [Feathers] Chapter 2 – Working with Feedback

Step 4: «Writing tests»*

Typical for legacy code is:

- no Unittests implemented
- maybe some Integrationtests
- no or less knowledge of the behaviour

For that reason the first tests often will be:

- Golden Master – checking for input output data
- Approval Test – maybe as combination test

… to establish a coverage of lines and behaviour as high as possible

After this step bringing the code under test a **TDD cycle can start** for further changes..

* [Feathers] Chapter 3 – Sensing and Separation

Dependency-Breaking Techniques

Subclass and Override Method*

Use inheritance in the context of a test to nullify behaviour that you don't care about or get access to behaviour that you do care about.

```
public class MessageForwarder {
protected Message createForwardMessage(Session session, Message message) {
    MimeMessage forward = new MimeMessage(session);
    forward.setFrom(getFromAdress(message));
    forward.setReplyTo(new Adress[] {
        new InternetAdress(listAdress)
    });
    // and a couple of further lines like above..
    return forward;
  }
}
```

```
public class TestingMessageForwarder extends MessageForward {
  protected Message createForwardMessage(Session session, Message
message) {
    Message forward = new FakeMessage(session);
    return forward;
  }
}
```

Steps:

1. Identify the dependencies that you want to separate
2. Make the method(s) overrideable
3. Create a subclass that overrides the method(s).

* [Feathers] Part 3 – Dependency-Breaking Techniques, Page 401

Dependency-Breaking Techniques – Part 2

Extract Interface*

Creating an interface for a class to implement this interface within a fake object.

```
public class PaydayTransaction extends Transaction {
  public PaydayTransaction(PayrollDatabas(  ITransactionLog log)  ) {
    super(db, log);
  }

  public void run() {
    for (Iterator it = db.getEmployees(); it.hasNext()) {
      Employee e = (Employee) it.next();
      if (e.isPayday(date)) {
        e.pay();
      }
    }
    log.saveTransaction(this);
  }
  ...
}
```

```
public interface ITransactionLog {
  void saveTransaction(Transaction transaction);
}

public class FakeTransactionLog implements
ITransactionLog {
  @Override
  public void saveTransaction(Transaction transaction) {
    // do nothing
  }
}
```

Steps:

1. Create a new interface. Don't add any methods to it yet.
2. Make the class that you are extracting from implement the interface.
   This can't break anything because the interface doesn't have any methods.
3. Change the place where you use the object so that it uses the interface.
4. Introduce a new method declaration on the interface for each message use.

* [Feathers] Part 3 – Dependency-Breaking Techniques, Page 362

Dependency-Breaking Techniques – Part 3

Extract and Override Factory Method*

Object creation in constructors can be annoying when you want to get a class under test. Having a factory method enables subclassing and overriding.

```
public class WorkflowEngine {
  public WorkflowEngine () {
    Reader reader = new ModelReader(AppConfig.getDryConfiguration);
    Persister persister = new XMLStore(AppConfig.getDryConfiguration);
    this.tm = new TransactionManager(reader, persister);
    ...
  } …
}
```

```
public class WorkflowEngine {
  public WorkflowEngine() {
    this.tm = makeTransactionManager();
    ...
  }
  ...

protected TransactionManager makeTransactionManager() {
    Reader reader = new ModelReader(AppConfig.getDryConfiguration);
    Persister persister = new XMLStore(Appconfig.getDryConfiguration);
    return new TransactionManager(reader, perister);
  }
}
```

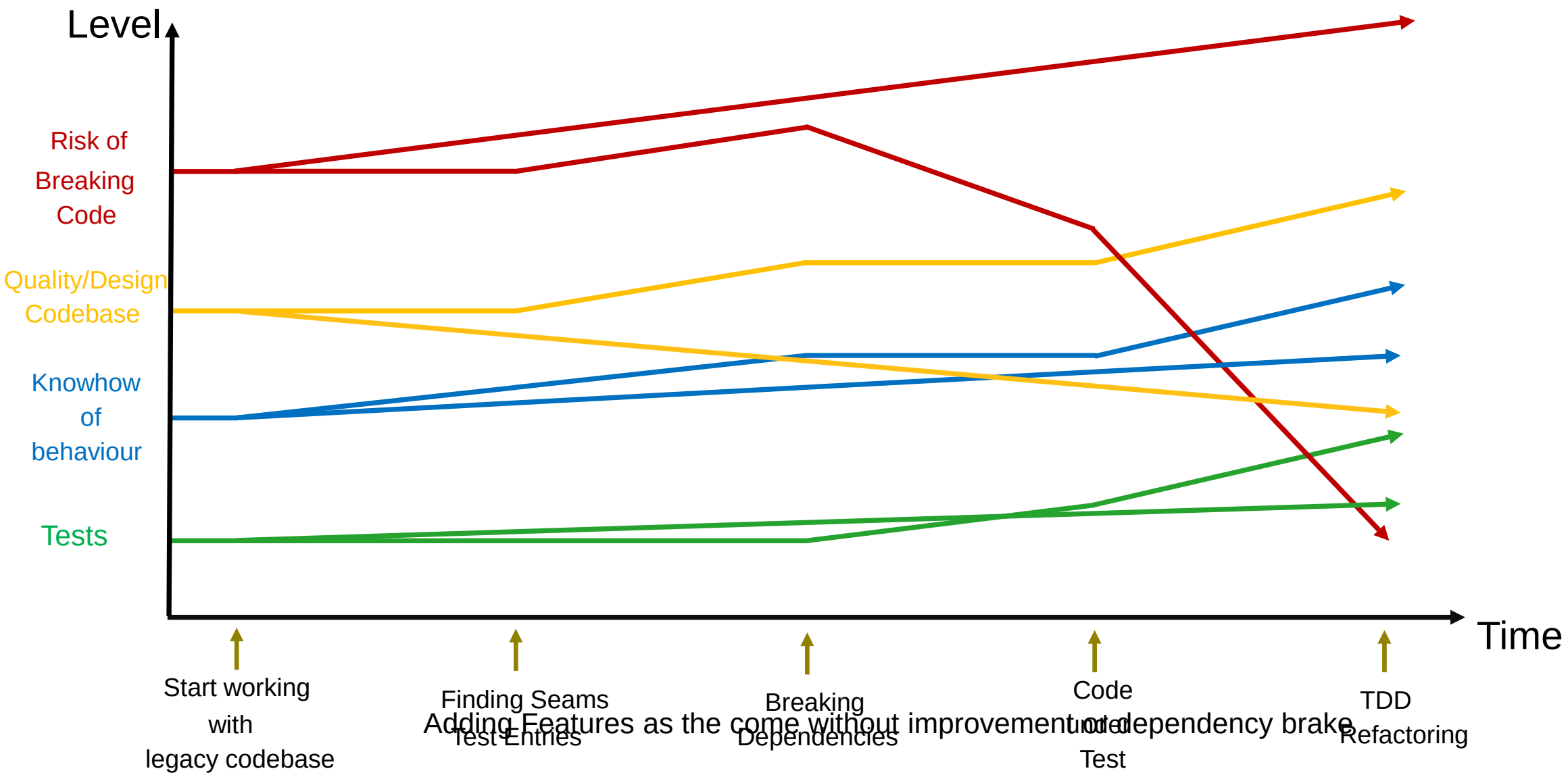
```
public class TestWorkflowEngine extends WorkflowEngine {

  protected TransactionManager makeTransactionManager() {
    return new FakeTransactionManager();
  }
}
```

Steps:

1. Identify an object creation in a constructor.
2. Extract all of the work in the creation into a factory method
3. Create a testing subclass and override the factory method in it.

* [Feathers] Part 3 – Dependency-Breaking Techniques, Page 350

Conclusion



Adding Features as the come without improvement or dependency brake

Any questions ?

Thank you for your attention!

Contact: roderich.gross@css.ch