Programming Practices

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

AMOS E02

Agenda

1. General principles
2. Technical debt
3. Code refactoring
4. Test-first programming
5. Test-driven development

1. General Principles

General Principles 1 / 4

KISS

(Keep It Simple, Silly)

General Principles 2 / 4

YAGNI

(You Ain’t Gonna Need It)

General Principles 3 / 4

DRY

(Don’t Repeat Yourself)

General Principles 4 / 4

1. **Make it run**
2. **Make it right**
3. **Make it fast**

2. Technical Debt

Definition of Big Ball of Mud [FY97]

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

Benefits of Good (“Well-Factored”) Code

1. **Maintainability**
   (Easier to understand)
2. **Extensibility**
   (Easier to adapt and evolve)
3. **Predictability**
   (Improves planning ability)

Video on Technical Debt [1]

Technical Debt

(Ward Cunningham)

[1] See https://youtu.be/Jp5japiHAs4

Video Lessons

- Technical Debt is a Metaphor
  - Communicates well to manager
  - (Certainly in financial services)
- Taking on debt can speed up development
  - It may be justified to learn faster
  - But you have to pay up later
- If you don’t pay back debt you’ll slow down
  - Paying up means refactoring
  - Still, never write poor code deliberately

Handling Technical Debt

1. **Identify problem**
   (So-called “code smells”)
2. **Identify need to act**
   (Correlate occurrences)
3. **Know how to act**
   (Refactor code)

Code Smell

- According to Fowler [F99]
  - “smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality”
  - also, “a code smell is a surface indication that usually corresponds to a deeper problem in the system”
- Code smells are not bugs

Example Code Smells

- Duplicated Code
- Long Method
- Large Class
- ...

3. Code Refactoring

Refactoring (Practice)

- Definition and purpose
  - Is a behavior-preserving transformation of existing source code
  - It is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
- More on refactoring
  - Change the structure of code without changing behavior
  - Focus on non-functional features within range of specification
  - Are defined techniques that are typically language specific
  - Are ideally supported by IDEs, for example, the Eclipse JDT
- Defined by Opdyke [O92], popularized by Fowler [F99]

Example Refactorings

- Rename class
- Pull-up field
- Extract method
- …

- More at http://refactoring.com

Example Extract-Method Refactoring 1 / 2

```
public class PhotoManager extends ObjectManager {
     protected Map<PhotoId, Photo> allPhotos = new HashMap<PhotoId, Photo>();

     public void addPhoto(Photo photo) {
          PhotoId id = photo.getId();
          assertIsNewPhoto(id);
          allPhotos.put(id, photo);
          ...
     }

     public Photo getPhotoFromId(PhotoId id) {
          Photo result = doGetPhotoFromId(id);
          if (result == null) {
               ...
               if (result != null) { allPhotos.put(id, result); }
          }
          return result;
     }

     public Set<Photo> findPhotosByOwner(String ownerName) {
          ...
          for (Iterator<Photo> i = r.iterator(); i.hasNext();) {
               Photo photo = i.next();
               allPhotos.put(photo.getId(), photo);
          }
          return r;
     }
     ...
```

Example Extract-Method Refactoring 2 / 2

```
public class PhotoManager extends ObjectManager {

     public void addPhoto(Photo photo) {
          ...
          doAddPhoto(photo);
          ...
     }

     public Photo getPhotoFromId(PhotoId id) {
          ...
          doAddPhoto(photo);
          ...
     }

     public Set<Photo> findPhotosByOwner(String ownerName) {
          ...
          for (Iterator<Photo> i=r.iterator(); i.hasNext(); ) {
               doAddPhoto(i.next());
          }
          ...
     }

     protected void doAddPhoto(Photo myPhoto) {
          allPhotos.put(myPhoto.getId(), myPhoto);
     }

     ...
```

When to Refactor?

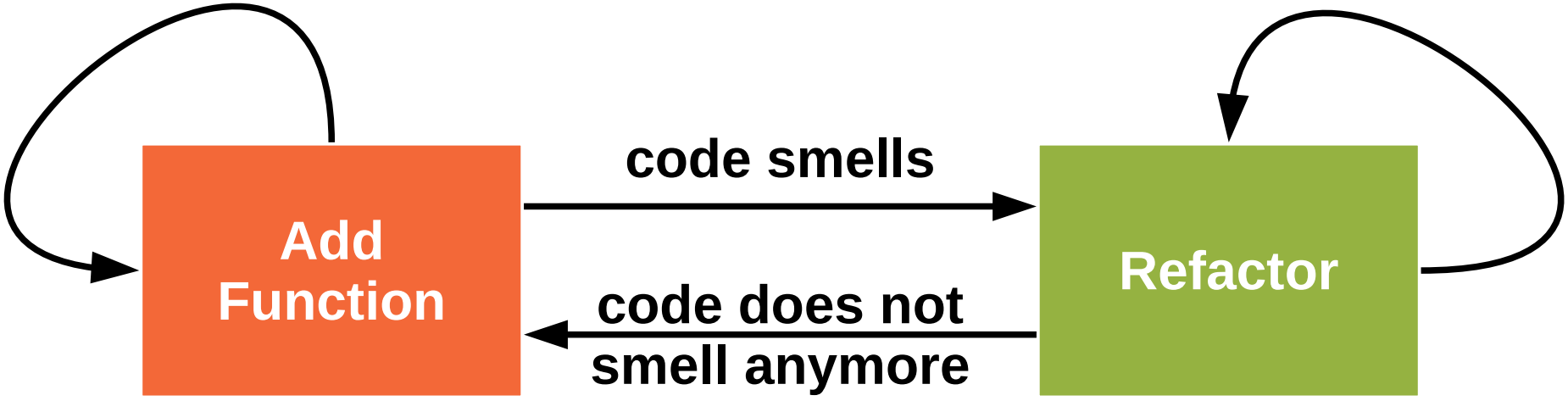
The “three strikes” rule

1st time: Just do it

2nd time: Wince at duplication

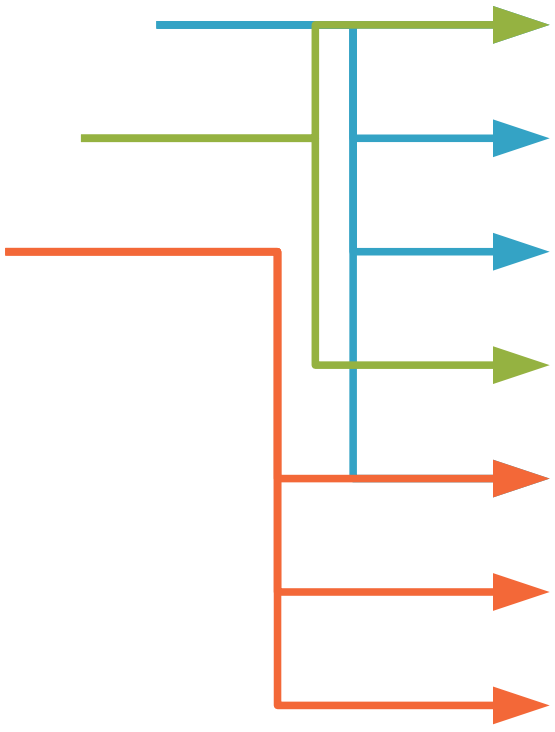
3rd time: Refactor

Refactoring Process (Two Hats)



Code Smells and Refactorings

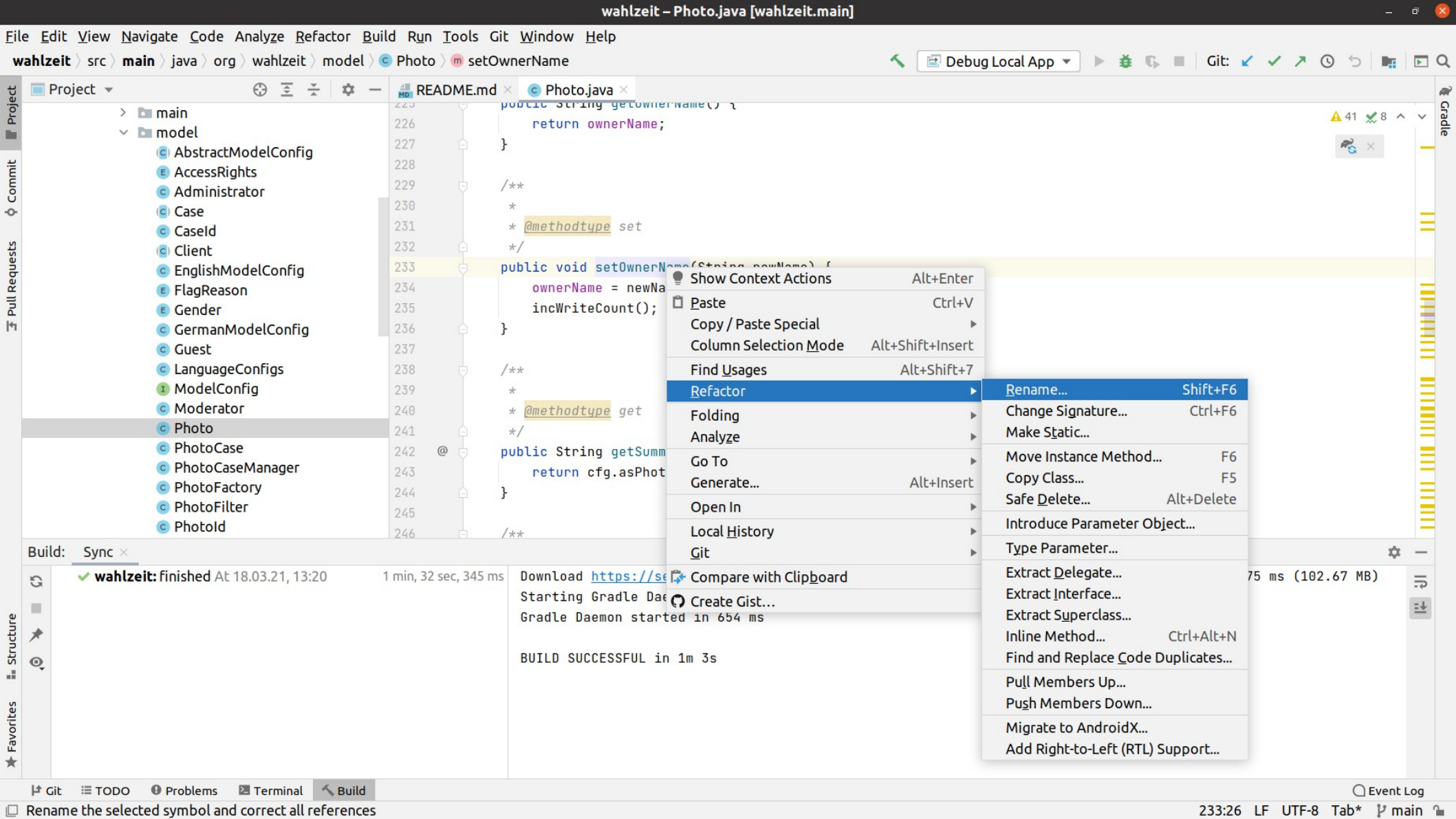
- Duplicated code
- Long method
- Large class
- ...



- Extract method
- Pull up field
- Form template method
- Decompose conditional
- Extract class
- Extract subclass
- Extract interface
- ...

Example Refactoring Process

- The refactoring process can become complex
  - It may turn into a series of refactorings
- Example Removal of Switch Statement
  - Extract Method
  - Move Method
  - Replace Type Code …
    - with Subclass or
    - with State / Strategy
  - Replace Conditional ...
    - with Polymorphism


wahlzeit – Photo.java [wahlzeit.main]
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help
wahlzeit › src › main › java › org › wahlzeit › model › Photo › setOwnerName
Debug Local App
Git:
Project
README.md
Photo.java
main
model
AbstractModelConfig
AccessRights
Administrator
Case
CaseId
Client
EnglishModelConfig
FlagReason
Gender
GermanModelConfig
Guest
LanguageConfigs
ModelConfig
Moderator
Photo
PhotoCase
PhotoCaseManager
PhotoFactory
PhotoFilter
PhotoId
return ownerName;
}
/**
* @methodtype set
*/
public void setOwnerName
ownerName = newNa
incWriteCount();
}
/**
* @methodtype get
*/
public String getSumm
return cfg.asPhot
}
/**
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+Shift+7
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Open In
Local History
Git
Compare with Clipboard
Create Gist...
Rename... Shift+F6
Change Signature... Ctrl+F6
Make Static...
Move Instance Method... F6
Copy Class... F5
Safe Delete... Alt+Delete
Introduce Parameter Object...
Type Parameter...
Extract Delegate...
Extract Interface...
Extract Superclass...
Inline Method... Ctrl+Alt+N
Find and Replace Code Duplicates...
Pull Members Up...
Push Members Down...
Migrate to AndroidX...
Add Right-to-Left (RTL) Support...
Build: Sync
wahlzeit: finished At 18.03.21, 13:20 1 min, 32 sec, 345 ms
Download https://se
Starting Gradle Dae
Gradle Daemon started in 654 ms
BUILD SUCCESSFUL in 1m 3s
75 ms (102.67 MB)
Git TODO Problems Terminal Build
Event Log
Rename the selected symbol and correct all references
233:26 LF UTF-8 Tab* main

Quiz on Refactoring

- Your code smells. All signs for refactoring are given. Under which circumstances should you not start a refactoring?

4. Test-First Programming

Test-driven Development in Context

1. Tests and Testing
2. Test-first Programming
3. Test-driven Development

Tests and Testing

- Testing is a process
  - that tests some concern (the concern “under test”)
  - for correct and expected operation
  - according to a specification
  - usually as part of quality assurance
- Tests can be manual or automated
- Tests verify against a given specification
- Tests increase confidence in correct functioning
- However, tests can never proof a program correct

Types of Tests [1]

- **Components tests** (a.k.a. unit tests)
  - Focus on testing one component out of context
- **Acceptance tests** (a.k.a. functional tests)
  - Focus on testing one cross-cutting functionality
- **Integration tests** (a.k.a. system tests)
  - Focus on testing end-to-end system integrity

[1] This is a simplification for the purposes of this course.

Tests and Testing Terminology

- **Test (Case)**
  - A single test for some particular aspect of the software, succeeds or fails
- **Test Suite**
  - A set of related tests that cover a particular domain of the software
- **Test Set-Up**
  - The data and preparation necessary to run a test as intended
- **Test Result**
  - The result of running a test, either succeed or fail, or a test error
- **Test Harness**
  - A software, like JUnit, that is used to simplify the implementation of tests

Test-First Programming [B02]

- Test-first programming is a practice in which developers
  - Write a test before they implement the actual functionality and
  - Iterate over an “add new or enhance test, make test work” loop

- Functionality is a by-product of making the tests work
  - Test-first programming
    - clarifies code functionality and interfaces
    - improves code quality through second use scenario
    - builds up test suite for continuous integration (later)

Test-First Rules 1 / 2

Only write new code

when a test fails

Then, eliminate waste

Test-First Rules 2 / 2

1. Red
2. Green
3. Refactor

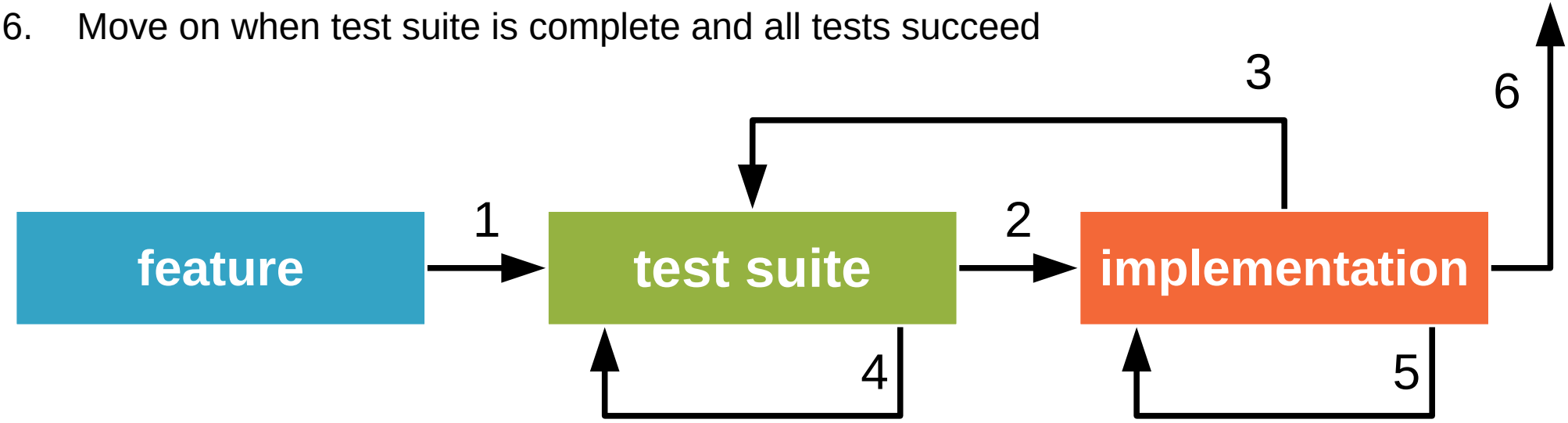
5. Test-Driven Development

Test-driven Development (TDD) 1 / 3

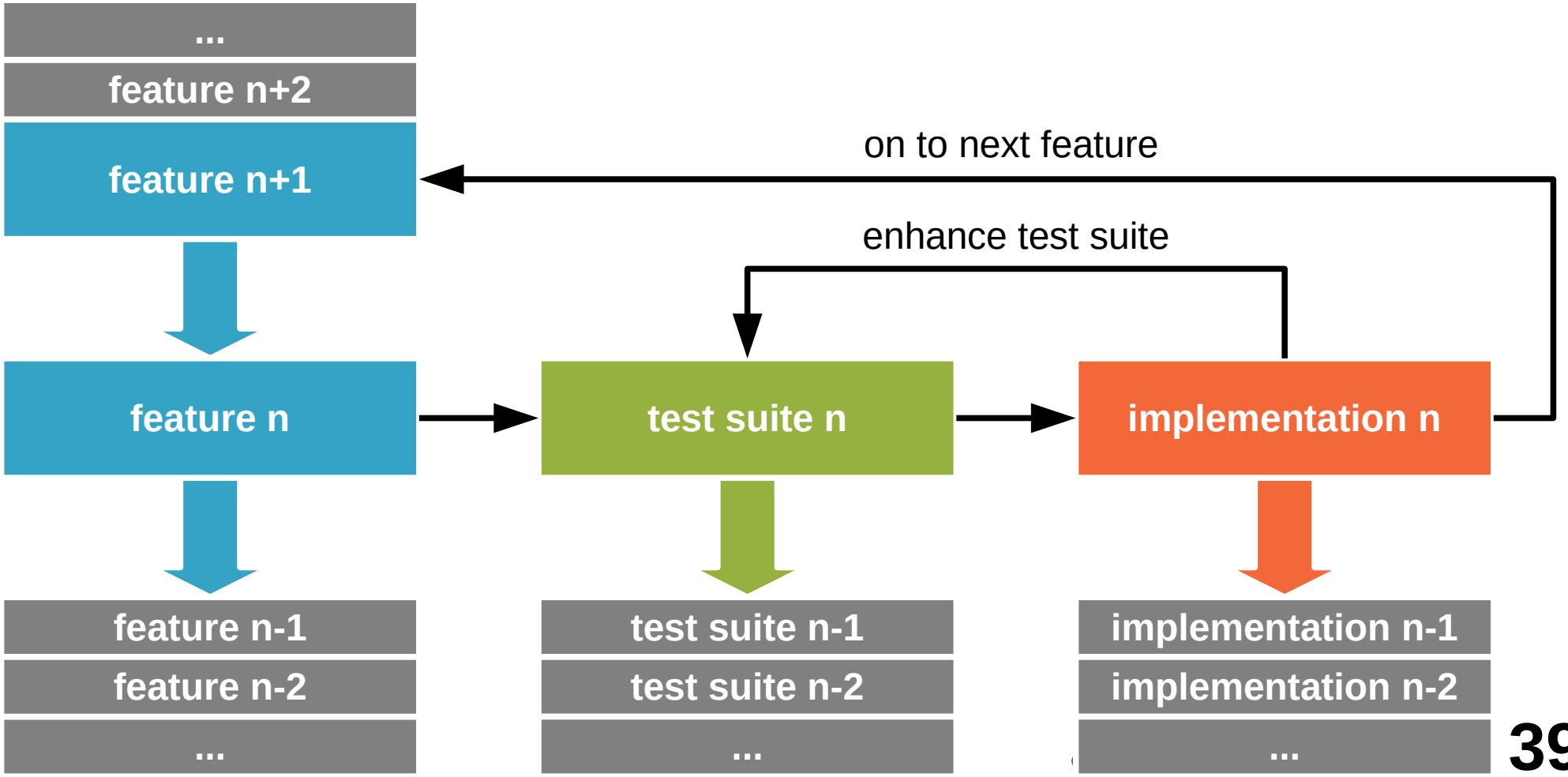
- Test-driven development
  - is a minimal development process based on test-first programming
  - turns feature requests into implementations
- Purpose of test-driven development
  - to grow the product incrementally and steadily
  - to be able to release after every feature implementation

Test-driven Development 2 / 3

1. Translate partial or full feature description into test suite
2. Implement feature to fulfill (“green-bar”) test suite
3. Revise test suite from new insights
4. Refactor test suite to keep design and code clean
5. Refactor implementation to keep design and code clean
6. Move on when test suite is complete and all tests succeed



Test-driven Development 3 / 3



Review / Summary of Session

- Agile principles of programming
  - KISS, YAGNI, DRY
  - Make it run, right, fast
- Technical debt and refactoring
  - Principles, two hats
  - Example refactorings and tools
- Tests and test-driven development
  - Test-first programming
  - The simplest of all processes

Thank you! Questions?

dirk.riehle@fau.de – http://osr.cs.fau.de

dirk@riehle.org – http://dirkriehle.com – @dirkriehle

Credits and License

- Original version
  - © 2021 Dirk Riehle, some rights reserved
  - Licensed under Creative Commons Attribution 4.0 International License
- Contributions
  - None yet