Agile Software Development Techniques

Testing, Managing Bugs, Teamwork, and Software Design in Agile Software Development

(Slides by Prof. Dr. Matthias Hölzl, based on material by Dr. Philip Mayer with input from Dr. Andreas Schroeder and Dr. Annabelle Klarl)

Development Techniques

- We now introduce some useful techniques for (not only agile) development

- We revisit:
  - Testing
  - Managing Bugs
  - Productive Development in a Team
  - Software Design

Agenda

I. Testing
II. Managing Bugs
III. Development in a Team
IV. Software Design

Testing

- Testing is one of the **most important tasks** in software development.

- A test is an **executable** piece of code which **automatically executes part of the system** and **verifies** the output
  - For example, test code might start a new game and verify afterwards that is has indeed been started.
- A test may have two results:
  - **Pass (Green)**: Everything went as expected.
  - **Fail (Red)**: The system failed to meet requirements.

Why Testing?

- A good test leads to **confidence** in code.
- **Passing tests of a task** should mean that
  - newly implemented functionality really works as expected
  - refactored functionality or added functionality does not break any previously working code (regression test)
- Thus, tests have to be written for, and as part of, tasks**.**
  - There should be a test for each important functionality realized by the task.
  - A task is **not fully implemented** if there are no associated tests.

How to write tests I

Ideas for writing tests:

- **Main functionality** (e.g. test that the main path works)
- **Branch-Based Testing** (e.g. check that there is a test for every branch of every condition)
- **Proper Error Handling** (e.g. check that methods correctly deal with null inputs, closed resources, failed connections)
- **Working as Documented** (e.g. if the documentation defines rules for a method, test these rules)
- **Resource Constraint Handling** (e.g. check that the system handles denied requests for resources such as database connections)

How to write tests II

Granularity of tests:

- **Unit tests** test the smallest testable part of the software (e.g. a single method in Java).
- **Integration tests** test the interaction between components
  (e.g. public interfaces).
- **System tests** test the software as a whole.
- **System integration tests** test whether the software is correctly integrated into its environment.

Using Mocks

- Ideally, the code under test has no external dependencies.
- Unfortunately, this is most often not the case.
  - For example, **a currency converter class** might need a database for retrieving exchange rates.
  - To test such the currency converter class, the database access object is **replaced by a mock object**.

- A mock object **mimics a real object** by implementing the same interface and just returning constant values.
- We will use Mockito for mocking purposes. (more details in the talk on technologies)

Testing in Scrum

- In Scrum, tests are an **integral part** of each Sprint – they are **NOT** deferred to the end of the project!
- Tests can be written by hand or using **Test Frameworks**.
  - The most well-known one for Java is **JUnit**.
  - The advantage is a good infrastructure and an existing test-runner with reporting functionality (more details in the talk on technologies)
- All tests should be **automatable**. This ensures that they can be run again and again if new functionality is added.

Code-and-Test

- The standard method for writing tests is **Code-and-Test**.
  - The code for the task is written.
  - Immediately afterwards, the tests for the task are written.
- This ensures that each task has tests.
- But, it also holds the danger of designing tests according to the code and not to the requirements.
- JUnit uses a **bar** for showing passed and failed tests:
  - Red Bar: At least one test failed.
  - Green Bar: All tests passed.
- The aim is to **keep the bar green**.

Test-Driven Development

- In agile methods, **Test-Driven-Development** (TDD) is used.
  - The test code for the task is written.
  - Afterwards, the simplest code is written to get the test pass.
  - At last, the code is refactored.
- TDD is claimed to lead to
  - … more testable code as testing drives the implementation
  - … more reasonable tests as tests are designed according to the requirements
- The aim is **red – green – refactor**. All tests fail initially, Then, the code should work and at last it is cleaned up.

When did we test enough?

- Testing is, in principle, a never-ending activity.
- The main criteria for moving on is **confidence**. That is
  - … the feeling that the tests adequately cover the functionality implemented in a task
  - ...or reaching a certain **code coverage** with the tests.

- **Code Test Coverage** is the percentage of code tested.
- Tools like **EclEmma** for Eclipse calculate this percentage based on the test cases.

Summary

- Software development does **not work** without tests!
  - Tests are **executable requirements.**
  - Tests ensure that existing functionality still works after changes (**regression testing**).

- Testing gives developers **confidence** for boldly moving forward to the next task.
- A task is implemented if the tests pass (but not yet done!)
  - See Definition of Done

Agenda

I. Testing
II. Managing Bugs
III. Development in a Team
IV. Software Design

Managing Bugs

- It is a simple, but inevitable fact of life that **bugs happen**.
- In agile methods, **bugs are accepted like that** – nothing to be (too) ashamed of.
- A bug is therefore **treated like a normal Backlog Item**
  - A bug report is made => a new Item for the Sprint Backlog
  - The task is given an estimate and a priority (as usual).
  - It is scheduled (as usual).
- A bug task is attached to an existing User Story or a new Issue is created for it

A bug report

A bug report should consist of:

- **Summary** – one sentence
- **Steps to Reproduce** – from a **well-defined** state of the system, what needs to be done to reproduce the bug?
- **What was expected, and what did happen** – to ensure everybody knows what was perceived as a problem
- **Version, Platform, Location Information** – bugs may be different in different versions, on different platforms or on different URLs
- **Severity and Priority** – how disastrous is the bug? How soon should it be fixed?

Testing for Bugs

- Bugs have a nasty habit of reappearing.
- Therefore,
  - Like a usual task, a bug-fixing task **MUST** include a test which reproduces the exact circumstances the bug was found in.
  - The test is added to regression testing (as usual) to ensure the bug does not occur again.

- **Finally:** When fixing a bug, look out for similar issues in the code.

Summary

Bugs are nothing to be ashamed of.

Bugs are treated like normal Backlog Items.

- They are written down and either attached to a User Story or a new Issue is created.
- They are estimated, prioritized, and scheduled.
- Tests are written.

Agenda

I. Testing
II. Managing Bugs
III. Development in a Team
IV. Software Design

Team Development

Productive development in a team means

- … using an **IDE** for managing and controlling code, dependencies and libraries
- … using **version control** to merge the work of multiple developers in a controlled fashion
- … using **continuous integration** for ensuring up-to-date, tested builds (manually or automated)
- … performing **code reviews** for ensuring high code quality and bug-freedom

Using IDEs

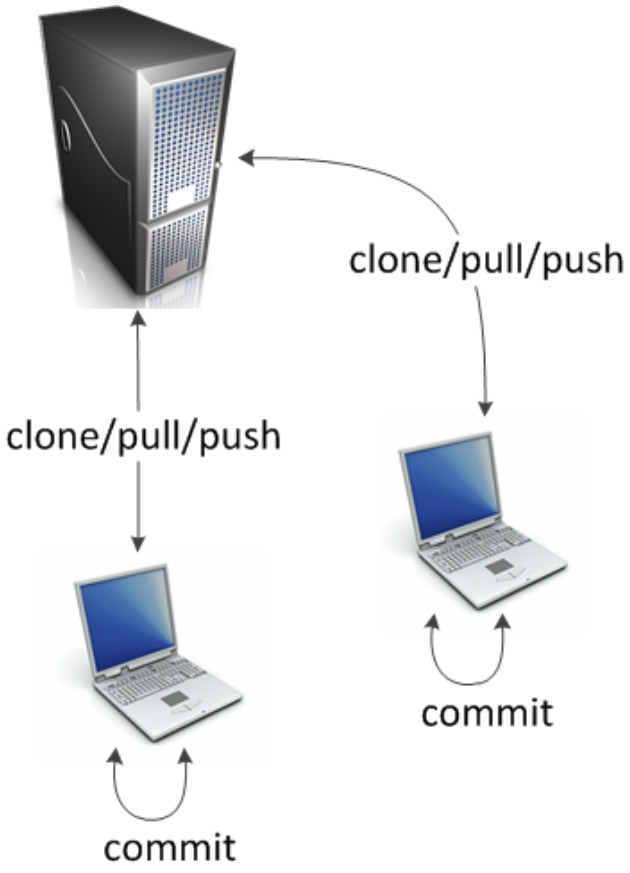
- An **integrated development environment** offers much more than just a code editor...
  - Integrated build system (background building)
  - Refactoring support (includes changing references)
  - Integrated documentation
    (source code of the entire Java API and libraries)
  - Code Navigation
    (jump to definition, references, call hierarchy, etc.)
  - Integrated test runners (JUnit and others)
  - Version Control support (CVS, Subversion...)
- An IDE makes programming **productive!**

Version Control

- Problems arise when multiple developers work on the same source code:
  - Changes happen to the same file which must be merged.
  - Changes might need to be rolled back because of a faulty implementation (e.g. overridden or conflicting features)
  - Traceability is needed to be able to determine the origin of an artifact (e.g. the developer can be asked for clarification)
- **Version Control Systems** exist to address these problems
- … and even more.

How version control works

- We use the distributed version control system **Git** for which a client is included in Eclipse. It consists of
  - … **multiple local copies** of a repository which developers might use to work on the source code and which provide the full functionality of a revision control system
  - … often, one copy is **marked** as the official repository



Version control best practices

- Committing a new revisions should only be done
  - ... if the code **compiles.**
  - ... **after** running all test cases
  - ... with a **commit message** which precisely says what has been changed or newly implemented (with a reference to the issue tracker task)

- **Before** pushing to the official repository, perform an update (pull) and **run all test cases again** to ensure nothing was broken.

Building Code

- Eclipse already contains mechanisms for **building software**
  - This includes compiling java source code, ...
  - ... an export mechanism as an executable JAR file
  - ... and building arbitrary other elements with ant scripts.

- Ensuring that all tests pass is still the responsibility of the developer.
  - In small and simple projects, this can be done **manually**.
  - For larger, more complex projects, a dedicated system for compiling and testing might be necessary that performs **automatically regular** builds and test runs.

Continuous Integration

- A continuous integration system reacts to commits or on a timer and performs
  - … checking out all code
  - … building the project
  - … running all tests
- The result of the CI run (e.g. compilation or tests failed) is placed on a website or mailed to all developers.
- Well known CI tools:
  - CruiseControl (little bit old-fashioned)
  - Hudson/Jenkins
  - **GitLab CI tool**

Reviewing Code I

Option 1: Peer Code Review (on one machine)

- Peer code reviewing means getting your code checked by your peers before assuming an issue is fixed.
- Code reviews are the single biggest thing that improve code quality. The average defect detection rate is 55 – 60%
  (vs. 25% for Unit Testing)
- Peer code reviews entail **increased** productivity
  - Less time spent with reproducing and fixing bugs
  - Increases knowledge transfer about the code base

Reviewing Code II

Option 2: Pass-around Review (using version control)

- The developer commits code to version control and informs the chosen reviewer via Mail or IM.
- The reviewer checks the changes, asks questions, discusses with the author, notes problems and bugs found.
- The developer responds and addresses the issues, and commits changes to version control.
- The review is completed.

Reviewing Code III

Option 3: Pair Programming (i.e. all coding done in collaboration)

- Two developers collaboratively writing code
- One has the keyboard and codes – the “pilot”
- One checks code on the fly and reflects about alternative approaches – the “co-pilot”
- Roles switch constantly back and forth
- Pilot and co-pilot constantly discuss the code, and the review is performed on the fly.

Summary

Productive development in a team means

- … using an **IDE** for managing and controlling code, dependencies and libraries
- … using **version control** to merge the work of multiple developers in a controlled fashion
- … using **continuous integration** for ensuring up-to-date, tested builds (manually or automated)
- … performing **code reviews** for ensuring high code quality and bug-freedom

Agenda

I. Testing
II. Managing Bugs
III. Development in a Team
IV. Software Design

Good Design

- Good software design is a science of its own e.g.
  - … it must match the software type (business, embedded, ...)
  - … it must follow the company style
- **But:** There are rules which apply everywhere
  - **Visualize complicated parts**
  - **Keep it simple**
  - **Readable Code**
  - **Re-Use** (Design Patterns, Libraries)
  - **SRP** / **DRY** / …and SOLID
  - **Refactor**

Visualize complicated parts

- The Unified Modeling Language (UML) is a visual design tool for software.
- The static parts, in particular **class diagrams**, are a great tool for **planning** (parts of) the software.
  - **Idea:** Focus on the overall structure, not on every detail
- Diagrams also serve as **documentation** of the software for new developers.
- On a higher level of abstraction, even the customer can get some **insights into the architecture** of the software.

Keep it simple

- The job of developers is implementing the task at hand
  … and nothing more.
- This means:

Implement the simplest thing that could possibly work!

- The aim is not to get caught up in „what might be needed in the future“.
- Instead, implement the task at hand, and implement it well.

Readable Code

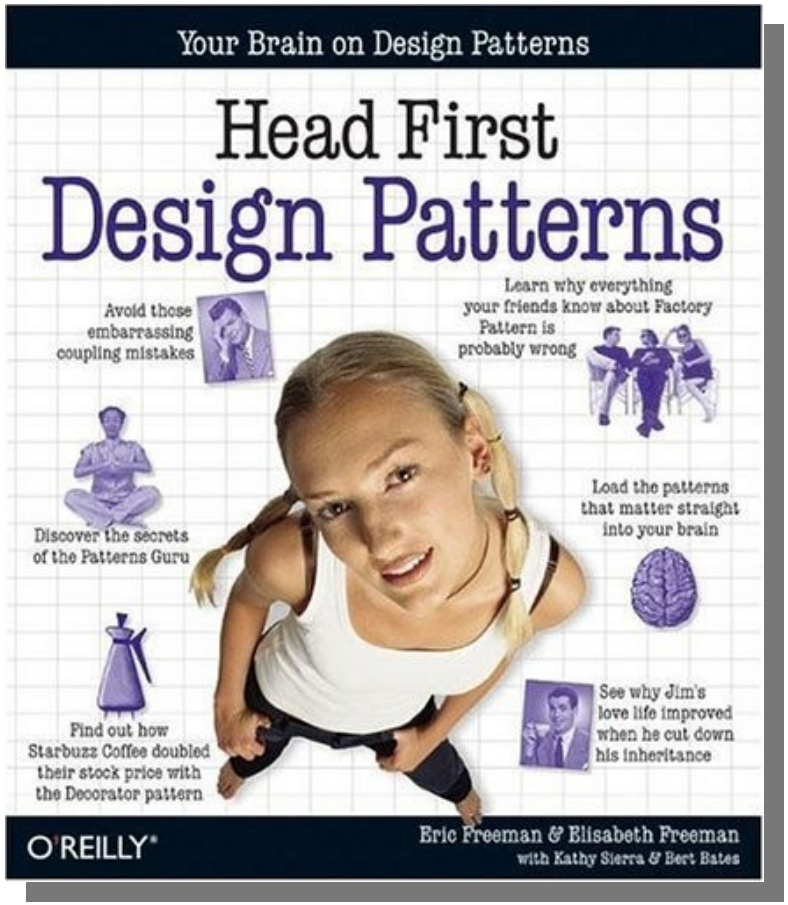
- This is (obviously) **WRONG:**

~~**It was hard to write, it should be hard to read!**~~

- Code should be designed to be **easy to read.**
  - "Speaking" and "Readable" Code:
    - Use long, **self-explanatory** variable and method names.
    - Use the formatter to ensure everything looks the same.
    - Prefer code to documentation.
  - **But:** Use JavaDoc if the code contains pitfalls
    - i.e. it is not obvious why it was written this way

Re-Use

- Do not reinvent the wheel!
- Mostly, there are already solutions for your problems:
  - Check for applicable **design patterns**
  - Check the (Java) **API**
  - Check for **external libraries**
- **Talk to your team!**



SRP/DRY

Maintaining code is easier if you only have to look in one place for each feature!

- **Single Responsibility Principle**
  - If a task is split across several classes, all of them need to change if the task changes.
  - **Result**: maintenance nightmare
- **Solution**: only **one responsibility** per class
  - Aim: high cohesion and low coupling
- **Don‘t Repeat Yourself**
  - If a bug is found in copied code, it needs to be changed **everywhere.**
  - **Result:** maintenance nightmare (again)
- **Solution**: Use inheritance/delegation to pull out common code
  - Aim: Find generic functionality (Hint: copy&pasted code)

Five basic principles in OO Programming & Design

- **SRP** = Single responsibility princle
  - a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
- **OCP** = Open/closed principle
  - software entities … should be open for extension, but closed for modification.
- **LSP** = Liskov substitution principle
  - objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- **ISP** = Interface segregation principle
  - many client-specific interfaces are better than one general-purpose interface
- **DIP** = Dependency inversion principle
  - Dependency injection is one method of following this principle.

Refactor

- One of the best things about IDEs is **refactoring support.**
- Due to design purposes, code may change:
  - Elements change their meaning.
  - Elements have to be moved.
  - Elements have to be split or merged.

- **Never** refrain from restructuring and renaming your code to fit the current view of the system.
  - Refactoring take care of all references automatically.
  - The aim is having **no burdens of the past.**
    (“this field is called xy because, at the beginning, we thought...“)
  - And **the tests ensure that the code still works**.

Summary

- Visualizing, Creating simple and readable Code, Re-Using, SRP/DRY/SOLID, and Refactoring are tools waiting to be applied.

- But: Do not go too far!
  - Even a “Perfect Design“ is obsolete tomorrow.
  - Aim for “**good-enough design**“.
  - Unfortunately, only experience helps to find the right balance.

Final Words

- We’ve seen how to use and apply
  - Testing
  - Managing Bugs
  - Productive Development in a Team
  - (Agile) Software Design

- Try to apply these principles, and learn through that experience.

Thank You.