LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Agile Software Development

A Planned, Story-Driven, Test-Based, Tool-Supported Process for Well-Designed Software

Philip Mayer

Origins of „Agile“

- Before and around the year 2000
  - Dissatisfaction with heavyweight software processes
  - It was felt that these processes add too much overhead
  - No place left for innovation and adaptation
- 2001
  - „Agile Manifesto“: A definition of agile development by Kent Beck (TDD), Martin Fowler (Refactoring), et al.
- Since then
  - A multitude of agile processes has been introduced
  - XP (eXtreme programming) and SCRUM are among the most well-known ones

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

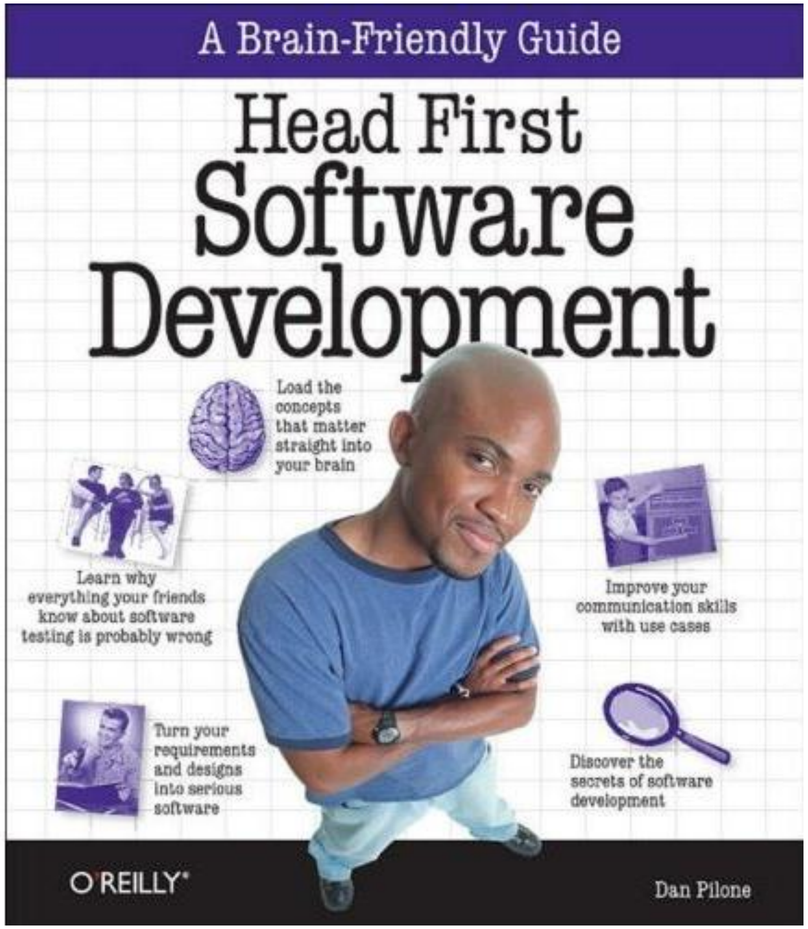
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

The Head First Process

- This talk focusses on one agile process compatible with the manifesto
- The process is
  - intended for groups of up to 10 people
  - consists of simple rules and is thus easy to learn
  - follows the agile manifesto
- The accompanying book, despite being fun to read, describes a rigorous plan for a development process


A Brain-Friendly Guide
Head First Software Development
Load the concepts that matter straight into your brain
Learn why everything your friends know about software testing is probably wrong
Improve your communication skills with use cases
Turn your requirements and designs into serious software
Discover the secrets of software development
O'REILLY
Dan Pilone

The HFSD Process (1)

- **Capturing and Managing Requirements**
  - The Customers Mental Image of the System
  - User Stories and Tasks
  - Estimation and Priority
- **Planning and Controlling the Process**
  - Deciding on user stories and tasks for an iteration
  - Big Board + Burn-Down-Chart: Know where you stand
  - Learning from the process
- **Testing**
  - Testing leads to confidence
  - TDD vs. TBD
  - Test Coverage

The HFSD Process (2)

- **Managing Bugs**
  - Bugs happen.
  - Reproducing bugs: Test for bugs!
  - Tracking bugs
- **Productive (Team-Based) Development**
  - Using IDEs
  - Using Version Control
  - Build Management and Continuous Integration
- **Good Design**
  - Visualising your system design with the UML
  - Design principles like DRY, SRP, etc.
  - Refactoring, Good-Enough Design

Part I/VI. Capturing and Managing Requirements

The Customer

- **Software development is about <u>shipping</u> software that brings the <u>customers ideas</u> to life.**

- Which means
  - **Shipping software**: The software must be completed, executable, and delivered – **on time**, and **on budget**.
  - **Customers Ideas**: The customer has a mental image of his product. The software engineers job is to extract that image and **implement it**.

Requirements

- The customers mental image about his software is captured as a set of **requirements**.
- Capturing requirements is **not simple!**
  - There are usually lots of assumptions both on the customer and IT side about the system to be built
  - Customer might not be an IT expert, he doesn't know what is possible or what is hard/easy to implement
- IT and customer **MUST** work together to extract the correct requirements
- In a sense, IT is also a consultant on the tech parts

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Capturing Requirements (I)

- Requirements are captured in the form of **User Stories**
- A user story captures **one thing** (and one thing only) that the software needs to do for the customer
  - A user story has a **title** and a **short description**
  - The description should fit on a DIN A6 index card (if it is too long, it needs to be split in two)
- User stories are customer-oriented.
  - User stories are written with, and for, the customer
  - They must be written in a language the customer can understand

Capturing Requirements (II)

- User stories are created in a **brainstorming meeting with the customer**
- **Think Big**.
  - While capturing requirements, the sky is the limit. Developers need to think **with** the customer – what else could he want/need in his software?
  - User Stories can be removed later on, but during this phase, IT needs to make sure everything which possibly matters is written down
- This technique is called **blueskying** (the sky is the limit).

Examples

- Good Story (customer-level):

View Games

Users should be able to see games which
are open for participation.

- Bad Story (too technical):

Use MySQL as database

The database will be based on mySQL as
it is a stable and open-source solution.

Other Techniques

- Two other techniques for **eliciting** requirements:

- **Role Playing**
  - Developer acts as the software
  - Customer acts as the user

- **Observation**
  - Developers watch the customer do the tasks he would like the software to support and write everything down

Estimating (I)

- At some point, everybody agrees that the idea of the system to be built is reflected in the user stories.
- The question is: **How long will it take**?
  - Developers need to **estimate** each user story.
  - An estimate is the **number of days or hours** a story takes to implement
- After estimates are available, the customer can prioritise the stories, such that the team knows what to implement in which order.

Estimating (II)

- **Estimation** is done **without the customer**
- Estimation means attaching a number of hours or days to each user story
- Each number should include time for
  - Design
  - Code
  - Test
  - Delivering (building + demoing)
- The whole project is the sum of estimates of all user stories.

How to estimate

- In agile development, the team as a whole is responsible for the implementation
- Everybody should, in principle, be able to implement each functionality. Thus, estimation takes everybody into account

- To arrive at a number **everybody is comfortable** with, developers should
  – **split the story into tasks**
  – **estimate the tasks**

Tasks

- For implementing or estimating a **user story**, it is split into **tasks**.
- A task specifies a piece of development to be carried out by **one** developer
  - Like a user story, it has a **title** and a **description**
  - **Usually written on a post-it attached to a user story.**
- A user story is a combination of tasks
  - Thus, the combined estimates for the tasks yield the user story estimate.

Task Examples

View Games

Users should be able to see games which are open for participation.

Create table for games in the DB

Create a UI for browsing games

Allow retrieval of games from server

Dealing with Assumptions

- The problem with estimates are **assumptions**
  - ...about what is part of a story and what is not
  - ...about the skills required, or the need to acquire them first
  - ...about the complexity of the task
- **No assumption is a good assumption**
  - The aim is to get rid of as many assumptions as possible
  - But they first need to be found
- Planning Poker can help here

Planning Poker

- Planning Poker
  - A certain task is selected
  - Every developer thinks about the task and how long it will take **himself** to implemented it, all things considered.
  - Estimations are done in private
  - The estimate is (covertly) written down on a card; the cards are collected
  - All cards are uncovered, the estimates yield a spread along a timeline
- If the estimates differ a lot, this indicates less condfidence and (probably) hidden assumptions

Planning Poker II

- The goal is convergence
  - The team **must come up with a single estimate**
  - This might require asking the customer for clarification
  - Still, everybody needs to be confident in the estimate
- In the end, estimates are written on a task post-it

- **Note:** A user story should, in general, not take longer than a couple of days to implement

...back to the customer

- The customer wants all of his user stories to be implemented
- Due to the estimates, the developers now know how long this will take
  - Mostly, this will be too long.
  - This means the time available must be extended or functionality removed
  - This is where priorities come into place
- The customer **prioritises** the user stories according to his/her current needs.

Prioritising

- The customer **prioritises the stories**. Important ones get a higher priority and must be implemented first
  - Customer must be assisted as stories might depend on each other
  - Priorities should be taken out of the set of **{10,20,30,40,50}**, with 10 being most important
  - Prorities are written on user story cards
- Based on estimates and priorities, user stories can later be assigned a place in the development cycle.
  - This is discussed later.

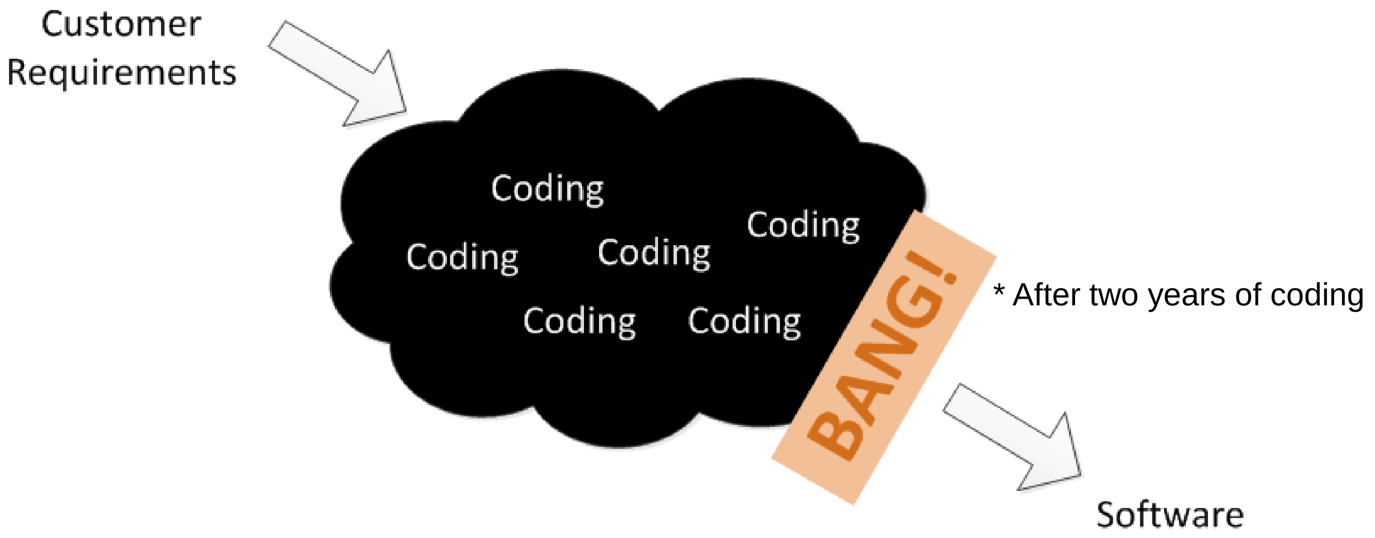
Summary

- Requirements are captured as **user stories**
  - With the customer, for the customer
- User stories are split into **tasks**
  - By the developers
- Tasks, and thereby user stories, are **estimated**
  - the aim is confidence by all developers
  - ...and getting rid of assumptions
- The customer assigns **priorities** to the stories, indicating which functionality should be implemented first

Part II/IV. Planning and Controlling the Process

Big Bang

- The **Big Bang** approach to software does **NOT** work:



- Also known as „going dark“
- No interaction with customer in the black cloud
- **The problem**: Requirements might have been misunderstood, or might change. The resulting system is not what the customer wanted.

We need something else...

Change is the only constant in SW development

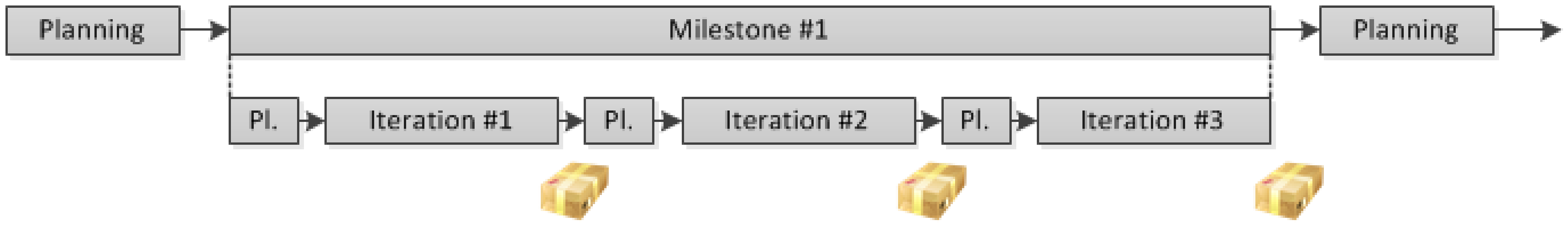
- Agile Development builds heavily on **communication** and the ability to react to **change**
- Requirements, Estimates, and Priorities might change – but this is considered in the process and dealt with **in a controlled way**.

Iterative Development

- **Milestones.** The HFSD process is based on milestones, which take about three months.
  - A milestone, or **version**, is a **major release** of the software which is a self-contained set of functionality.
- **Iterations**. Each milestone is split into iterations, which take about four weeks (= 3 iterations per milestone)
  - An iteration is a short, balanced timeframe for working on a set of user stories, producing a **working piece of software**

User Stories, Milestones, and Iterations

- The HFSD process looks like this:



- In the **Planning** phase(s), **user stories are added to milestones and iterations** based on the time they take, and the priority they are assiged.
- The planning phase is, in general, **not part of a milestone or iteration**; it stands in-between milestones and iterations

Assigning User Stories

- Assigning user stories to an iteration is a **commitment by the team**
  - It says that the team **is able to and will** implement this story in this iteration.
- But: **How many user stories fit into an iteration?**

Reality

- In principle, an iteration is four weeks, which means 20 working days. Multiplied with the number of developers, this yields the available time (say, 3 times 20 = 60 days)
- **However**: Estimates are based on **ideal** days or hours, i.e. actual time spent on solutions. Unfortunately, the real world keeps intruding with
  - Installing Software
  - Team Communication
  - Paperwork
  - Hardware breakdowns
- These things affect the actual time which is available for work

Velocity

- **Solution**: The amount of working days available is reduced to factor in these problems.
- The factor used is called **team velocity**.
  - Velocity is unique for each team. It needs to be monitored and changed over time
- As an initial factor, a value of 0.7 can be assumed.
- Thus, with 3 developers, an iteration has

$$3 \times 20 \times 0.7 = 42 \text{ days}$$

available for development work

Selecting User Stories

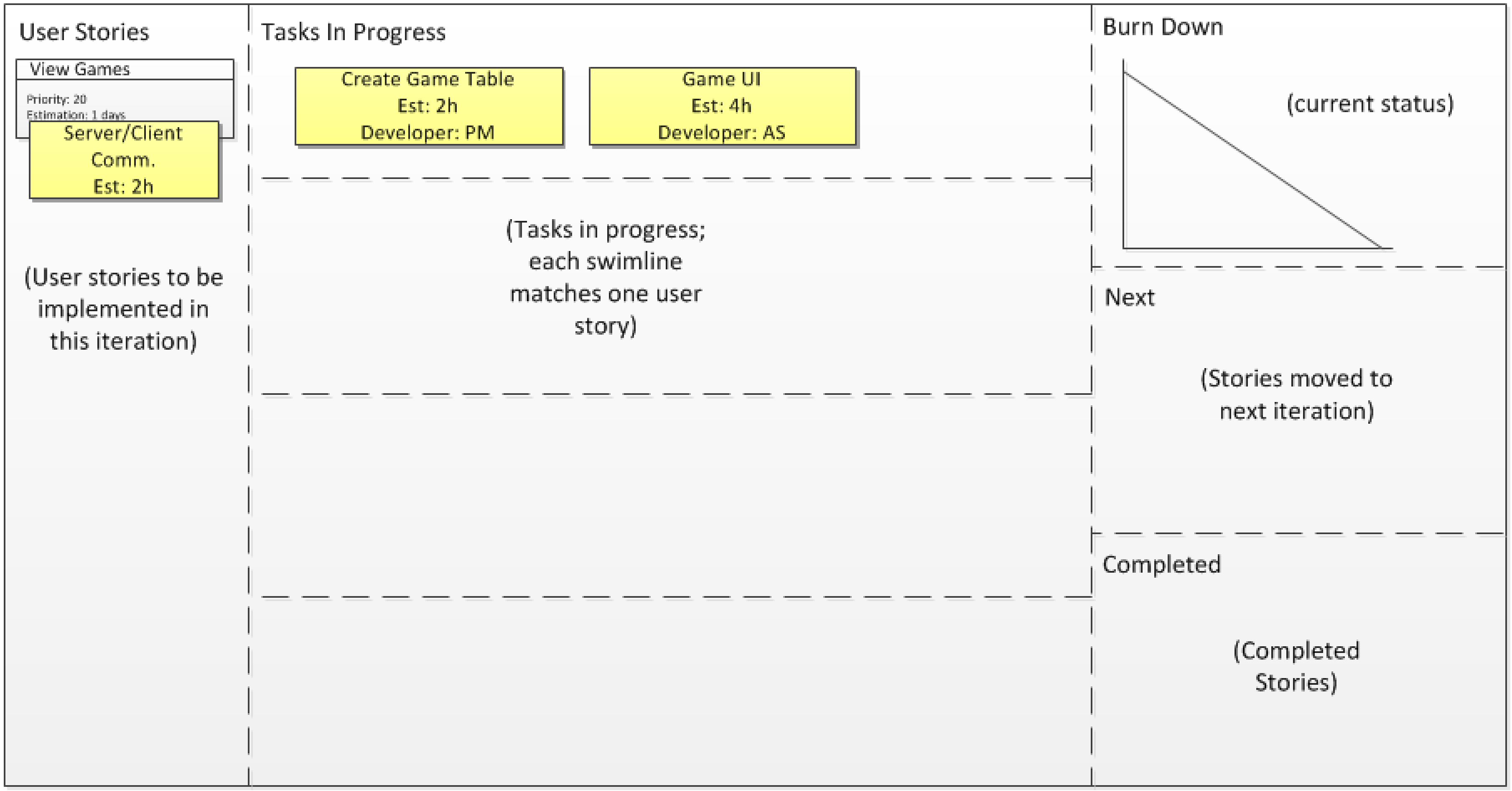
- User Stories can now be selected according to the number of days/hours available
- The highest-priority user stories are selected.
- The idea is to get as close to the **maximum number** of days/hours available as possible
  - However, do not go over the limit
  - It is better to use lower-priority stories than to push the limit

Controlling the Process

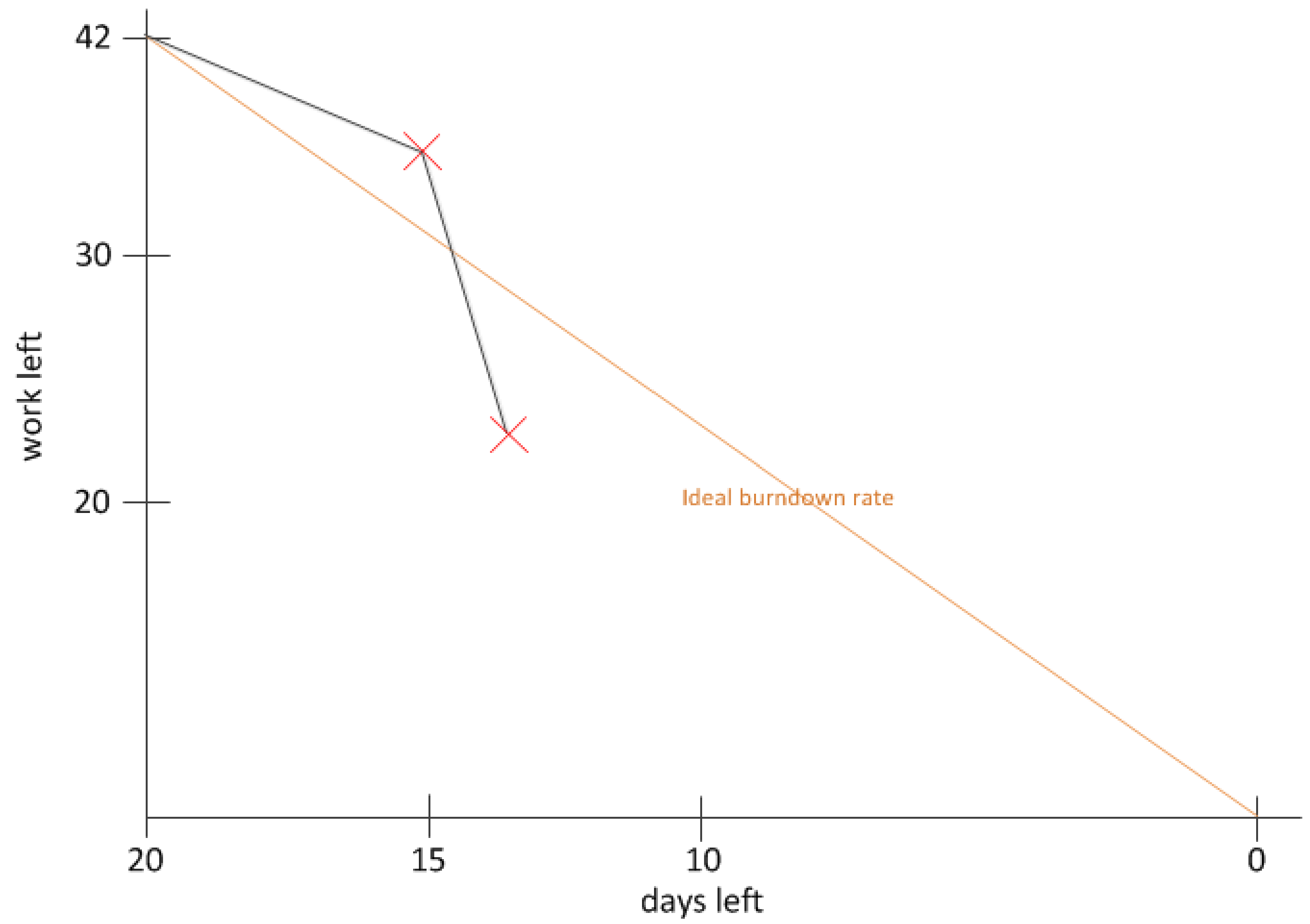
- During an iteration, the user stories and their tasks are realised by developers
  - This includes design, test, implementation, and integration

- It is important to stay on track: If a user story or task takes longer or shorter than expected, or if additional problems come up, the team must **know** about this.
- A **Project Big Board** is used to cover this information

The **Project Big Board (PBB)**


User Stories
View Games
Priority: 20
Estimation: 1 days
Server/Client Comm. Est: 2h
(User stories to be implemented in this iteration)
Tasks In Progress
Create Game Table Est: 2h Developer: PM
Game UI Est: 4h Developer: AS
(Tasks in progress; each swimline matches one user story)
Burn Down
(current status)
Next
(Stories moved to next iteration)
Completed
(Completed Stories)

The Burn-Down Chart



The Burn-Down-Chart (II)

- The chart shows
  - X-Axis: Working days left until the end of the iteration
  - Y-Axis: Sum of task estimates yet to be done
  - The straight line is the ideal burn-down rate: This is how we planned our tasks against the time available.
- During an iteration, the current status is added:
  - The sum of the remaining task estimates are plotted on the intersection with remaining days
  - If the point lies above the ideal burndown rate, the team is behind schedule. Else, it is ahead of schedule
- The chart needs to be updated when tasks change their status, are added or removed, or when estimates change

Problems

- If it becomes apparent that a user story takes longer than expected, this **shows up on the chart**
  - The customer must be notified/asked for clarification. Some user stories/tasks must be scheduled for the next iteration
  - The reason must be noted and taken into consideration for the next estimation phase or, if it was a non-coding related problem, velocity.
- **Unplanned Tasks**, like new but important ideas of the customer, bugs, or other maintenance work need to be considered as tasks as well
  - They get their own story, task, estimation, etc.
  - Depending on their priority, they are handled immediately – in this case, other stories are pushed back – or added to the „next“ section.

Keeping Track of Things

- The Big Board must be continually updated
- This happens in **Stand-Up Meetings**, which should take place every day, or at least once a week
  - Short meetings (20 minutes)
  - Heads-Up on the progress of every developer
  - Changing the board
  - Discussing problems like bugs, new tasks, re-scheduling, etc.
- The intention is to **keep the finger on the pulse** of the project

Ending an Iteration

- An iteration comes to an end when time runs out.
- At this point, a **running version of the software must be available** – if not all tasks/user stories were handled, these have been pushed back before.

- A demo is given to the customer
  - This may lead to new (change) tasks for the next iteration

Moving to the next iteration

- **Learning from the past**
  - Revisit estimates – why did they differ from the actual time? What can be done better next time?
  - Calculate the new velocity (for example, assuming 20 work days, 3 developers, and 36 „estimated task days“ done):

$$36 / (20 \times 3) = 0.6$$

  - Velocity should only account for overhead, not as a buffer for wrong estimates
- **The next iteration** begins just like the last.

Summary

- **Iterative development** is used to stay close on track with the customers ideas
  - Customer is always kept „in the loop“
  - Allows for constant change
    - After every iteration, re-evaluation of user stories and priorities is possible
    - Detection of misunderstandings, wrong assumptions, etc.
    - Keeping up with changes in the customers ideas
- A **Controlled Process** is used to stay on top of problems
  - The team knows the progress. No hidden „bombs“ waiting to go off
  - The customer knows the progress, too
  - Transparent schedule

Part III/VI: Testing

Testing

- Testing is one of the **most important tasks** in software development.
- A test is an **executable** piece of code which **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

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Writing Tests

- Tests are written for, and as part of, the implementation of **tasks.**
  - There should be a test for each important functionality realised by the task
  - A task is **not fully implemented** if there are no associated tests.
- A good test leads to **developer confidence** in code. **Passing tests of a task** should means that
  - The functionality really works as expected
  - If the functionality is refactored, or new code added and the tests still work, nothing was broken („safety net")

How to write tests

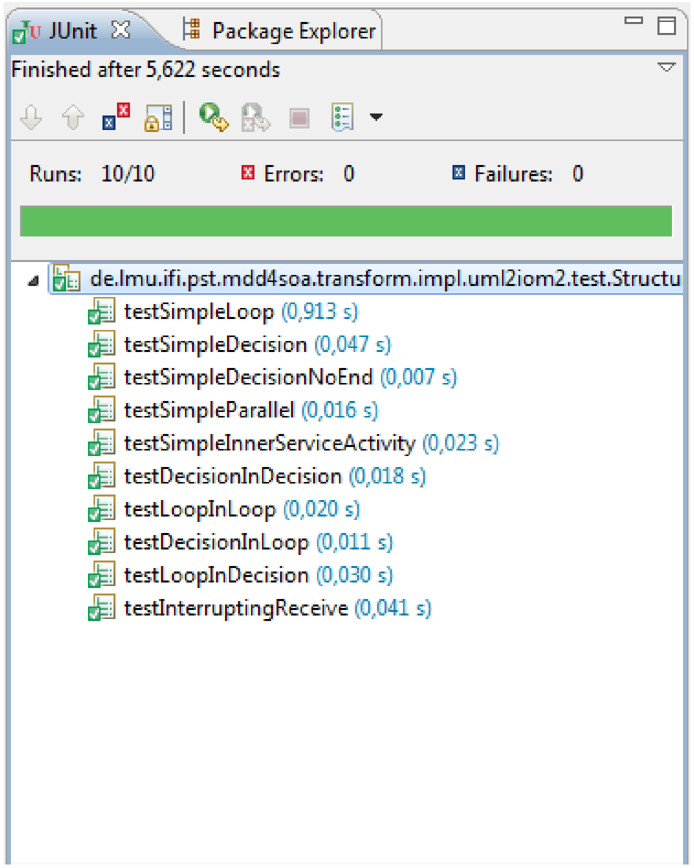
- Ideas for writing tests:
  - **Main functionality** (e.g.: Test that the main path works)
  - **Branch-Based Testing** (e.g.: Check that there is a test for every branch of every condition)
  - **Proper Error Handling** (e.g.: Check methods correctly deal with null inputs, closed resources, failed connections)
  - **Working as Documented** (e.g.: If the APIDOC defines rules for a method, tests these rules)
  - **Resource Constraint Handling** (e.g.: The system should gracefully handle denied requests for resources such as database connections)

Testing in agile processes

- In agile development, tests are an **integral part** of each iteration – 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 proven, solid infrastructure and an existing test-runner with reporting functionality.
- All tests should be **automatable**. This ensures that they can be run again and again (this is called **regression testing**)

Code-And-Test

- The standard method for writing unit 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
- 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**.



JUnit   Package Explorer

Finished after 5,622 seconds

Runs: 10/10   Errors: 0   Failures: 0

```
de.lmu.ifi.pst.mdd4soa.transform.impl.uml2iom2.test.Structu
    testSimpleLoop (0,913 s)
    testSimpleDecision (0,047 s)
    testSimpleDecisionNoEnd (0,007 s)
    testSimpleParallel (0,016 s)
    testSimpleInnerServiceActivity (0,023 s)
    testDecisionInDecision (0,018 s)
    testLoopInLoop (0,020 s)
    testDecisionInLoop (0,011 s)
    testLoopInDecision (0,030 s)
    testInterruptingReceive (0,041 s)
```

TDD

- Another method is **Test-Driven-Development** (TDD)
- In TDD, tests are written **before** the code
  - This means that **all tests fail initally** (due to compile errors or functionality just not being there)
  - The next step is to get the test to pass – implement the simplest thing that could possible to get a green bar
  - Afterwards, refactor the code for the next test.
- **TDD**, in general, leads to more testable code as testing drives the implementation
- The goal here is **red – green – refactor**. First, the code should work, and then it is cleaned up.

Using Mocks

- Ideally, the code under test has no external dependencies
- This is mostly not the case
  - for example, a **currency converter class** might need a database for for retrieving exchange rates
  - To test such the currency converter class, the database access object is replaced by a **mock object**.
- A mock object is a stand-in for real object which implements the same interface, and just returns constant values for a particular test

Tests, Tests, Tests

- Testing is, in principle, a neverending activity
- The main criteria for moving on is **confidence**
  - This is either 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

- Agile 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** (and a safety net) for boldly moving forward to the next task.
- A task is done if the tests pass.

Afterthought

Fear leads to anger, anger leads to hate, hate leads to suffering

No tests lead to fear

Part IV/IV: Managing Bugs

Bugs

- It is a simple, but inevitable fact of life that **bugs happen**.
  - **Bugs** occur all the time during initial development of a task. Such problems can be fixed on-the-go.
  - **More Bugs** are found through testing, which is part of implementing a task. They can be fixed before finishing a task.
  - **Unfortunately, some bugs** are only found after a task or story has been committed or even delivered. The later they are found, the harder they are to fix (usually).

Tracking Bugs

- In agile development, **bugs are accepted as a fact of life** – nothing to be (too) ashamed of.
- A bug is therefore **treated like a normal task**
  - A bug report is made => a task description
  - 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 user story 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 differen 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 treated like tasks
  - They are written down on a post-it and attached to a user story
  - They are estimated, prioritised, and scheduled
  - Tests are written.

Part V/VI: Productive (Team-Based) Development

Productive Development

- **Productive development** 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)

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** and gives developers **control** and **confidence** over their code

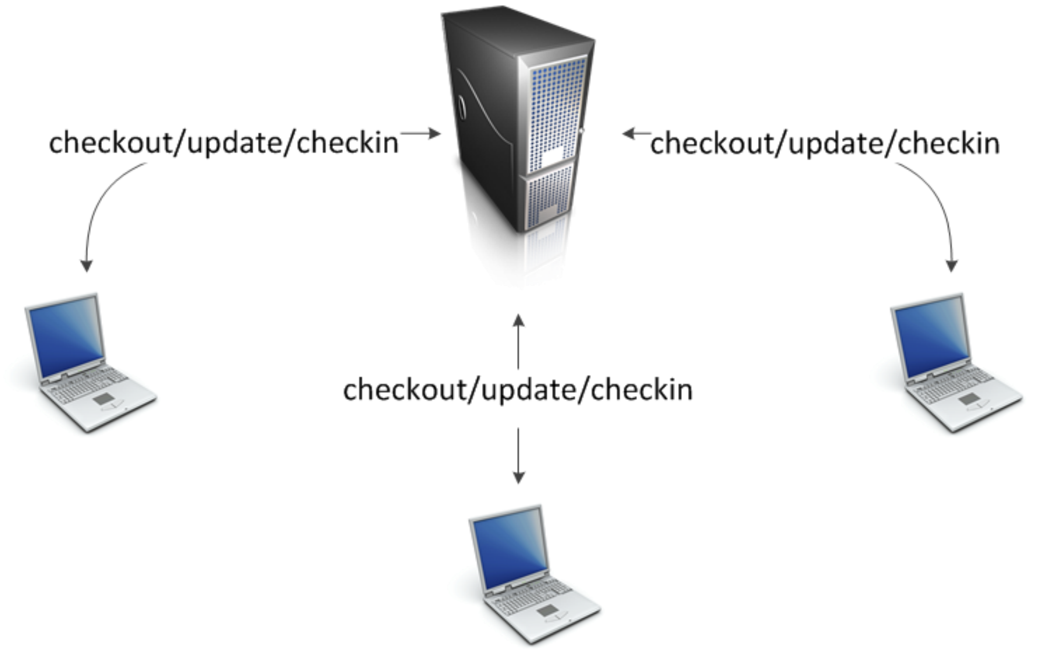
Version Control

- Problems arise when multiple developers work on the same source code:
  - Changes might happen to the same file which must be merged
  - Changes might need to be rolled back because of conflicting features
  - Traceability is needed to be able to determine the origin of an artefact
- **Version Control Systems** exist to address these problems.

How version control works (I)

- We use the centralised version control system **Subversion** (the successor to CVS), for which a client is included in Eclipse. It consists of
  - The **server**, which contains the definitive current copy of all files plus the history of all files since the beginning of the project
  - A number of **clients**, which developers use to work on the source code
    - A developer can download (**checkout**) the current version of the files, edit them, and then upload the files again (**commit**)
    - At any time, the newest version can be re-downloaded while keeping (if possible) local changes (**update**)

How version control works (II)



- The server hosts a **repository** of (source code) files
- Each file (and folder) has a **revision number**
  - The current revision is called the **head** revisision

Basic version control operations

- **Checkout**
  - means copying the repository content to an empty local directory to work on it.
- **Update**
  - means copying the repository content to an existing local directory. This operation attempts to merge changes on the server with local changes. If automatic merging fails (**merge conflicts**), the user is prompted and must **resolve** problems manually.
- **Checkin**
  - means adding the local changes to the repository on the server. This operation attempts to merge local changes into the (maybe also changed) revision on the server. If automatic merging fails (**merge conflicts**), the user is prompted to **resolve** manually.

Other version control features

- Additional features
  - **Tagging**: A certain revision is **tagged** with a name (like `Iteration3Final`, such that it can be found and compared later
    - Tagging should be used for every iteration end or release, and maybe in-between
  - **Branching**: A separate repository based on the head repository is created, which allows for testing out new ideas
    - The main branch is called **trunk**.
    - Branching should be used very sparingly, for example for handling fixes for a version 1.1 while 2.0 is in preparation

Best practises

- 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 task)

- After a commit, perform an update 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 by means of **ant** scripts
- Ensuring passing tests 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

Continuous Integration System

- A continous integration system reacts to checkins or on a timer
  - Checks out all code
  - Builds the project
  - Runs all tests
- The result of the CI run is placed on a website, or mailed to all developers
- A well-known CI tool is **CruiseControl**.

Summary

- **Productive development** can be reached by using effective tools
  - **IDEs** for code control, refactoring, and library management
  - **Version control systems** for handling code merges and conflicts
  - **Continuous integration**, manually or automatically, for ensuring passing tests throughout the development

Part VI/VI: Good Design

Good Design

- Good software design is a science of its own
  - Must match the software type (business, embedded, ...)
  - Must following company style
- **But:** There are rules which apply everywhere
  - **Visualising complicated parts**
  - **Keeping it simple**
  - **Readable Code**
  - **Re-Use** (Design Pattern, Libraries)
  - **SRP / DRY**
  - **Refactoring**

Visualising Software

- The Unified Modelling 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 details
- Diagrams also serve as documentation of the software for new developers

Keeping it simple

- The job of developers is implementing the task at hand – 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

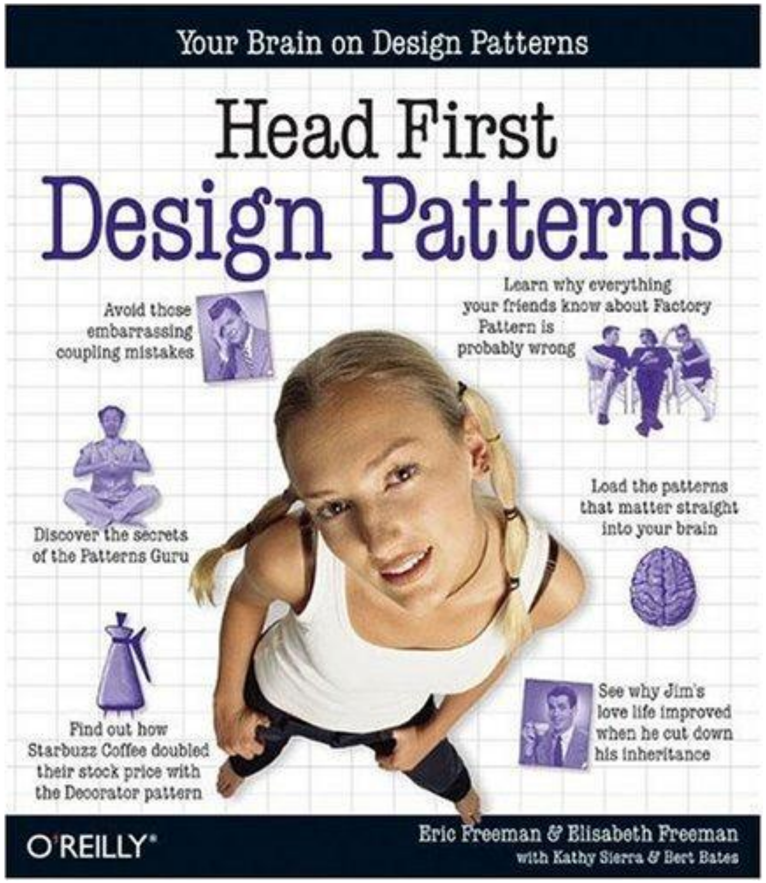
- Code should be designed to be **easy to read**
  - „Speaking“ Code
    - Use long, **self-explanatory** variable and method names
    - Use the formatter to ensure everything looks the same
    - Prefer code to documentation
  - **But:** Use JavaDoc if the code contains *gotchas*
    - i.e. It is not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

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

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!**


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory Pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

SRP

- **Single Responsibility Principle (SRP)**
  - If a certain task is split across several classes, all of them need to change if the task changes (or if new tasks are added). This is called a „ripple effect“
  - **Result**: maintenance nightmare
- **Solution**: Each class should have only **one responsibility**
  - The aim is having high **cohesion**
  - Maintaining the code is easier as one only has to look in one place

DRY

- **Don‘t Repeat Yourself** (DRY)
  - **Never** copy&paste code within your project
  - If a bug is found in copied code, it needs to be changed **everywhere**
  - **Result:** maintenance nightmare (again)
- **Solution**: Use inheritance or delegation to factor out common functionality
  - The aim is to understand which functionality is generic
  - (Again): Maintaining the code is easier as one only has to look in one place

Refactoring

- One of the best thing about modern IDEs is **refactoring support**
- During design, things change
  - Elements change their meaning
  - Elements have to be moved
- **Never** refrain from restructuring and renaming your code to fit the current view of the system
  - Refactorings take care of all references automatically
  - The aim is having **no legacy** („this field is called such-and-such because, at the beginning, we thought...“)
- **Fear Not: The tests ensures the code still works**

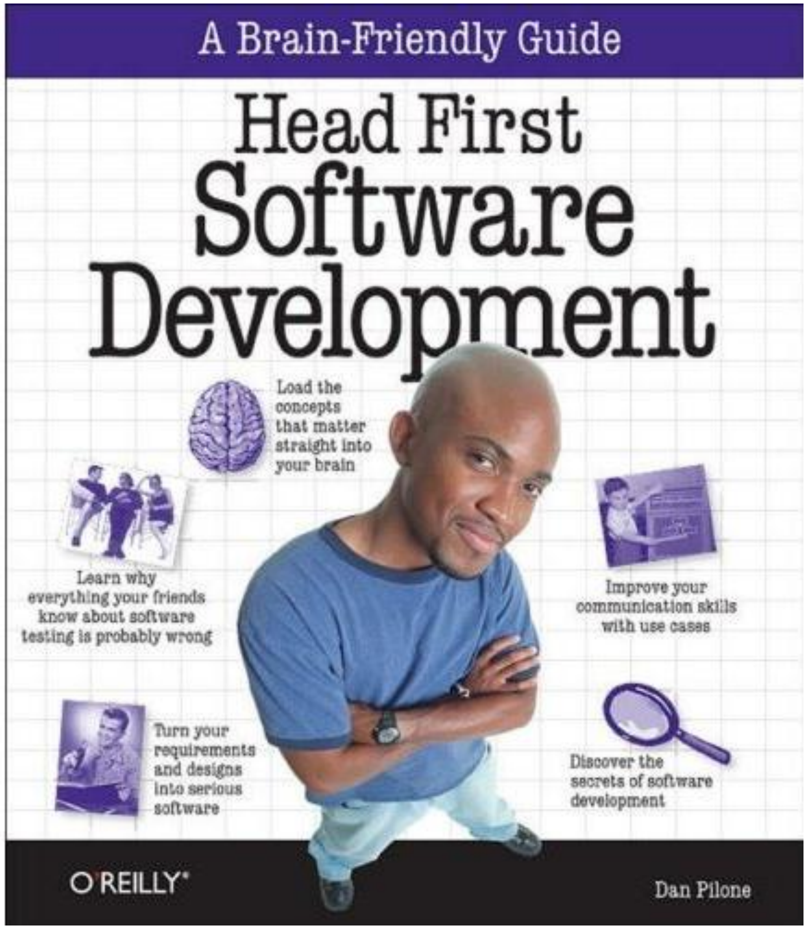
Summary

- Readable Code, Re-Use, DRY, SRP, 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**“
- This is about **drawing a line in the sand**
  – Unfortunately, only experience helps finding the right balance

Conclusion

The Head First Process

- This talk has shown the HFSD agile software development process
- Please make yourself familiar with the process in the remainder of the week.


A Brain-Friendly Guide
Head First
Software
Development
Load the concepts that matter straight into your brain
Learn why everything your friends know about software testing is probably wrong
Improve your communication skills with use cases
Turn your requirements and designs into serious software
Discover the secrets of software development
O'REILLY
Dan Pilone