Introduction

Why this event?

- After my first commit at 1&1 I received 4 "out of office" notification e-mails and was shocked
    - My mentor told me that every commit leads to a diff mail to the whole team in Germany and Romania
    - From this point I reviewed my code twice before i committed

- What I've learned from this was the objective view of source code
    - The main aspect is to separate constructive remarks of code and personal criticism
    - Software development is a team game and the team wins only when they play together
        - ➔ Only possible when all members have the ability to look to source code objectively

- Only few publications, mostly product advertisements
- Not very present in academic world, less present then refactorings
- Practical driven
- But in the real world common and practiced
    - Sometimes ;-)

- I do not claim for completeness, please give me feedback!
- Some IEEE standards, but in my opinion not very prominent used

Theoretical basics

What is a review

What is a review in general?

- One or more people looking (reading) at a reviewed component
- Try to comprehend and follow these thoughts
- Check if they agree or have remarks
- Takes place **after** the factoring process

[Short story: Change something after design?]

- Look at existing things
- Check if reviewed component is correct
- Better chance of correctness the more reviewers
- In a review the chance to gather an objective and mental distanced view of a reviewed component is much higher then to "write time"
    - Developers have a natural identification with "their code"
    - Leads to no objectivity "wish that own code is correct"

Well-known kinds of reviews:

- Film review
- Music review
- Buy review

- Hotel review

Difference to (abstract) software reviews?
- Immutable
- Not necessarily objective
- Looks at an object from a single point of view
- Software reviews have the aim to make the reviewed component better
- Software reviews have not the aim to rate the reviewed component
- Software is a moving target and therefore easy to change and improve

[Short story: Comparison to other engineering disciplines]

Targets

What kind of components can be reviewed in the software context?

- Source Code
    - The most elementary unit, which can be reviewed
    - But the most essential unit as well
    - Quality of software depends heavily on the quality of source code

- Component
    - More abstract and complex units, which can be reviewed
    - Review component structure
    - Review component behavior

- Architecture
    - Most abstract units in software development context
    - Review component interaction
    - Review system structure and behavior

- Design
    - Review the design principles of software components
    - How are design patterns organized within this architecture?

- Refactoring

  [Question: What is a refactoring?]
    - Review if the software has same behavior like before
    - Review if the expected goals was achieved

- Tests
    - Review if you have good test coverage
    - Review if tests are "hard" enough
    - Review if tests are correct

- Project
    - The whole project can be reviewed as well
    - What went wrong, what was good?
    - Targets: design, implementation, documentation, communication
    - Other kind of review
        - ➔ No change for existing project
        - ➔ What can we make better in the **next** project

[Question: What do you think are the most reviewed things?]

Preconditions

- All members must be able to receive or give constructive and objective remarks/criticism
    - Negative example:
        - ➔ Print code, mark red and behave like in an English test

[Short story: True story ;-). Fresh ideas from freelancer]

- Objective remarks are good and should not taken personally

- This ability is supported by the thought of "common code base"
    - The code is our (!) good and we (!) want to take it further
    - "We like our code", "we want that our code is good", …
    - Don't bother how your code get there, take care of improving it

[Illustration: Show obvious and fix it]

- All members want to produce sustainability and high quality code
    - If the quality is good, the management will love you
    - Common code implies:
        - ➔ Common coding styles
        - ➔ Common formatting rules
        - ➔ Common naming convention
        - ➔ Common best practices

[Illustration: Show conventions]

    - Leads to the behavior that it`s not necessary who fixes a bug or who will work on components in the future, because all have one standard.
    - No developer should bother to reformat the whole code before he can start to work or need to understand other naming conventions or or or ...

- It is absolutely necessary to have a diplomatic and good moderator
    - Must be well accepted by all participants of the review
    - Good understanding of problem domain required as well
    - Needs diplomatic sense and objective view
    - "Typical Senior"

[Illustration: Positive event for authors]

- Management must see benefits of reviews
    - Time spend for reviews can be measured in higher quality
    - Time spend for reviews leads to higher productivity afterwards
    - ... and saved time while bug fixing

- You need tool support
    - Source code management
    - Issue tracking (bugzilla, jura, story board, ...)
    - Code review software (jupiter, crucible, gerrit, ...)
    - Further communication tools (chat, wiki, phone, ...)

[Question: What do you use?]

Classification

- Part of the software development process
    - Build in some processes (extreme programming)

[Question: Do you know what extreme programming is?]

- Aims to develop better software
    - Four or more eyes principle
    - Knowledge exchange
    - After factoring process
        - ➔ Not the academic way

- No compensation for tests
    - Unit and integration tests must be done as well!
        - ➔ Tests could be reviewed as well (completeness, quality, ...)
    - Your manager will not accept "we reviewed it, no reason for testing"

[Illustration: Same assumption than “we thought about it and made an invariant, it is correct, no reason for testing”]

    - Natural hope that own code is correct
        - ➔ Some kind of blindness for own suboptimal code

[Question: Who test in your company?]

- Continuous reviews integrated in process
    - Mostly smaller pieces to review
    - Change (diff) of two revisions is reviewed
    - High frequency
    - Regular
    - Dedicated review meeting optionally, mostly skipped
    - Comments visible for all participants while individual phase

- Dedicated at defined points in development process
    - Mostly larger pieces to review
    - A new artifact/component is reviewed
    - Rarely
    - Sporadically, not regular
    - Dedicated review meeting (as good as) mandatory
    - The gag: comments visible first at review meeting
        - ➔ Often leads to discussions about the remarks

- Code reviews are not for free
    - Developers and managers must invest and pay time and energy
    - Author must accept and reviewer must rethink author`s thoughts

[Question: Are you ok with this?]

Why

- If important pieces of software are reviewed, quality raises
    - Code profits from skills of more then one developer

    - Other developers can think about more use cases, exceptions or improvements while the initial developer is mainly focused on getting it done

[Illustration: Connection to extreme programming?]

- Leads to a situation where the team can prevent "online bugs"
    - Each bug found during development time is a good bug
- "only a dead bug is a good bug"
    - Does not prevent the production of bugs, just their releases (hopefully)
    - Higher chance to catch them

[Illustration: Show obvious mistakes]

- The knowledge of the systems must be spread in the team
    - Developer redundancy (illness, fluctuation, promotions, vacation, ...)
    - Parallelize tasks, maintenance
    - Leads to sustainable systems and development
    - See how other sub-teams work
        - ➔ Separation of teams typically in front and backend. Have other coding styles, best practices, ...
        - ➔ Nice to know how the other party works, leads to better understanding, complete overview, ...

[Short story: Typical frontend vs backend separation]

- Usually the first introduction work of a new or junior developer is reviewed
    - Kind of acceptance ritual, in German we would say "Aufnahmeritual"
- "After we all read your code, you are a full-fledged member of our team"

- The work of freelancers should be reviewed as well

[Question: Why?]

    - Sustainability, you have to maintenance it! :)
    - Learn from fresh ideas - discussion and code
    - Show internal best practices (only interesting for further work)

- You can estimate very clear how skilled your co-workers are
    - Specialists, see where workshops are required
- "I will ask him the next time according xyz", “Let´s initiate xyz workshop”
    - Identify accumulated needs

- The coding skills of all participants will rapidly gain
    - Quotation of a co-worker: "In my code review I learned almost as much as I learned in a whole year at university."
    - Requires the presence of al important persons of the team, from professional guides to newbies
    - Experts give their experiences, newbies give their fresh and unconsumed ideas
- "Sometimes a new and simple view is required to break out of fixed habits"
    - All participants take benefit of discussion about best practices of the team and special experiences since the last review

What kind of

There are several kinds of code reviews.

- Classic version is a dedicated activity
    - The most common version
    - Mostly (and unfortunately) very rarely
    - Expensive
        - ➔ large amount, because its so rarely
    - Best for "acceptance ritual"

- Continuous code reviews
    - Through commit mails

[Question: Do you use versioning tools?]

- ➔ Best concept/tool I have ever seen
- ➔ Each time a commit occurs in your repository, a diff mail is send
- ➔ Each subscriber has the chance to read all changes in the system
- ➔ Requires mail filters, of course!
- ➔ Usually decision (if someone wants to read the diff) in according to commit message. Requires good commit messages!

[Illustration: Good vs. bad commit message]

- Good: added “create with” to watermark
- Bad: further implementation

```
Author: pbusch
Date: 2011-11-21 20:29:45 +0100 (Mon, 21 Nov 2011)
New Revision: 375

Added:
   fbfrontend/branches/kozuka-premium/src/main/resources/de/buschstein/fancycover/frontend/image/watermark-white-
transparent.png
Modified:
   fbfrontend/branches/kozuka-premium/src/main/java/de/buschstein/fancycover/frontend/FancyCoverSession.java
   fbfrontend/branches/kozuka-premium/src/main/java/de/buschstein/fancycover/frontend/page/PictureWorkingPage.java
   fbfrontend/branches/kozuka-premium/src/main/resources/de/buschstein/fancycover/frontend/image/watermark-white.png
Log:
watermark with added 'created with'
```

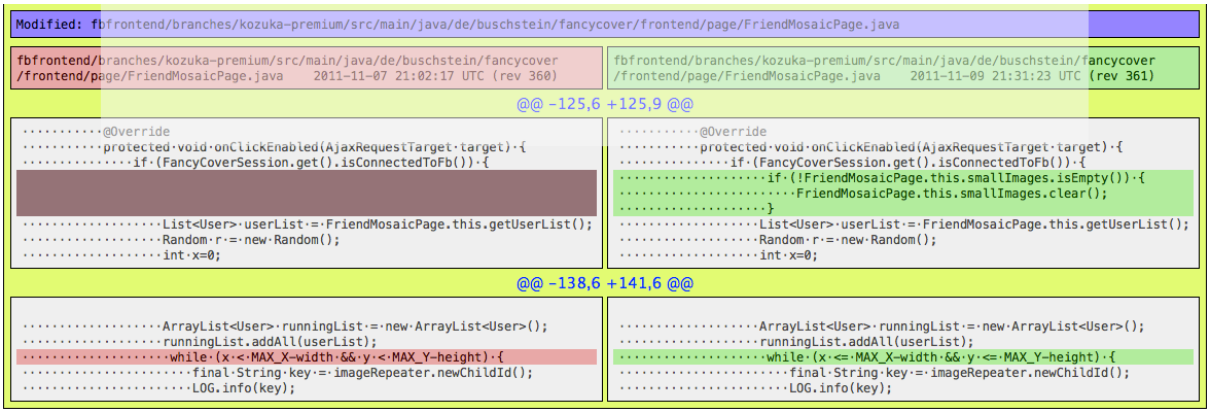
```
Modified: fbfrontend/branches/kozuka-premium/src/main/java/de/buschstein/fancycover/frontend/FancyCoverSession.java

fbfrontend/branches/kozuka-premium/src/main/java/de
/buschstein/fancycover/frontend/FancyCoverSession.java
2011-11-21 08:10:08 UTC (rev 374)

fbfrontend/branches/kozuka-premium/src/main/java/de
/buschstein/fancycover/frontend/FancyCoverSession.java
2011-11-21 19:29:45 UTC (rev 375)

@@ -74,7 +74,8 @@

        byte[] result;
        if (watermarked) {
            ImageBuilder builder = new ImageBuilder(this.re
            builder.overlayWithImage(FancyCoverSession.clas
            result = builder.getFinalImageAsByteArray();
        } else {
            result = this.resultImage;

        byte[] result;
        if (watermarked) {
            ImageBuilder builder = new ImageBuilder(this.re
            builder.overlayWithImage(FancyCoverSession.clas
                    .getResourceAsStream("image/watermark-w
            result = builder.getFinalImageAsByteArray();
        } else {
            result = this.resultImage;
```

- ➔ Requires also clear and well defined commits
- Don't break repository, semantically units, not too huge, ...

[Question: What is your commit behavior?]

- ➔ Requires good know how about the system to estimate quality of changes
- ➔ Good for own reviews as well! "What did I change in this commit?"

[Illustration: Commit mail]



- Through tools like crucible or fisheye
  - ➔ Enables a web based reviews
  - ➔ Smaller amounts, higher frequency
  - ➔ More integrated in development process, not a dedicated activity. Sometimes with coffee on the couch :)

- Extreme programming or agile processes
  - Code reviews in extreme programming build in the process
  - Through pair programming

[Question: Do you remember the concept?]

  - Process:
    - ➔ One develops, the other sits beside and communicates
    - ➔ Has time to think about exceptions, further special cases,...
    - ➔ Has mental distance to think about code objectively
  - Productivity loss? Time spend while developing high quality software is saved in time planned for bug fixing, fluctuation works, …
  - In agile processes (e.g. scrum) not build mandatory in that way
    - ➔ But more accepted as in classic project management processes
    - ➔ Agile processes have the aim to develop software in small steps (sprints). Each small step can be reviewed very well.

- Open source
  - Big point for reviews
  - Expose your software to the whole world
  - Needs fancy project that someone invests effort to review your software
  - Mostly review through usage of your components
  - Maybe you get support from readers of your software

How to use

- Classic code review process and tools
  - Dedicated activity
  - Moderate to larger amount of code
  - Tools mostly IDE integrated
    - ➔ Has benefits of real working environment
    - ➔ ctrl+left click is your friend
    - ➔ Negative: Mostly no chance to view differences

- Web based tools
    - Continuous activity
    - Smaller amount of code
    - Tools web based (who`d think about that?!)
        - ➔ Have benefits of mobility! "Coffee, couch and code review"

[Question: Do you have couches? Some kind of “chilling area”?]

        - ➔ Has mostly the benefit to review differences
        - ➔ Negative: No real working environment

- Commit mails
    - In my opinion the absolute minimum what you should do to improve your quality
    - In the meantime I use commit mails in private projects as well

[Illustration: How to setup commit mails, it is so easy!]
http://blog.netzmeister-st-pauli.com/post/411802119/svn-commit-e-mails-einrichten

    - Useful to keep track of the whole system and the according changes
    - Useful to review own commits, work
        - ➔ Easier to open an email then browsing repository history
        - ➔ “What did I do this morning?”
    - Useful to identify author of dedicated changes
        - ➔ Search functionality of mail clients is very good ;-)
        - ➔ No fingerprinting! Helpful in the way that you can ask the author why he did this change.

- Early recognition
    - Some kind of special use of commit mails
    - You can see when your code moves to a direction where you do not want it (from the point of view of an architect, experienced developer)
    - In combination with offshore development very useful

- Mentor model
    - New colleague gets mentor, which reviews his work/commits
    - Often for junior developers
    - Makes objective remarks and dedicated reviews
    - 1:1 relation
    - Best way to accept objective reviews and learn objective view

[Short story: How my mentor comes with short reviews on normal papers]

- Outsourcing
    - Used in companies with no technical focus (public authorities, super markets [...])
        - ➔ Where developers are just resources
    - After the whole process when something crashes
    - Our developers are not able to build the software we need, lets contact some specialists or consultants
    - Benefit for own developers gains to zero

[Remark: No own experience with this! Just read a lot of this and wondered about…What do you think??]

What to review

- Defined amount of code
    - Someone (usually the author) should think about wisely what components should be reviewed
    - Components should have semantically connection and in the best case wraps a semantically closed piece of software
        - ➔ Introduction work
        - ➔ New feature
        - ➔ Result of a refactoring, …
        - ➔ Subset of the things listed above is also possible
    - Don´t bother about boilerplate code or boring sections :)
        - ➔ Keep track of interesting sections (fancy calculations, ...)
        - ➔ Keep track of critical and important sections

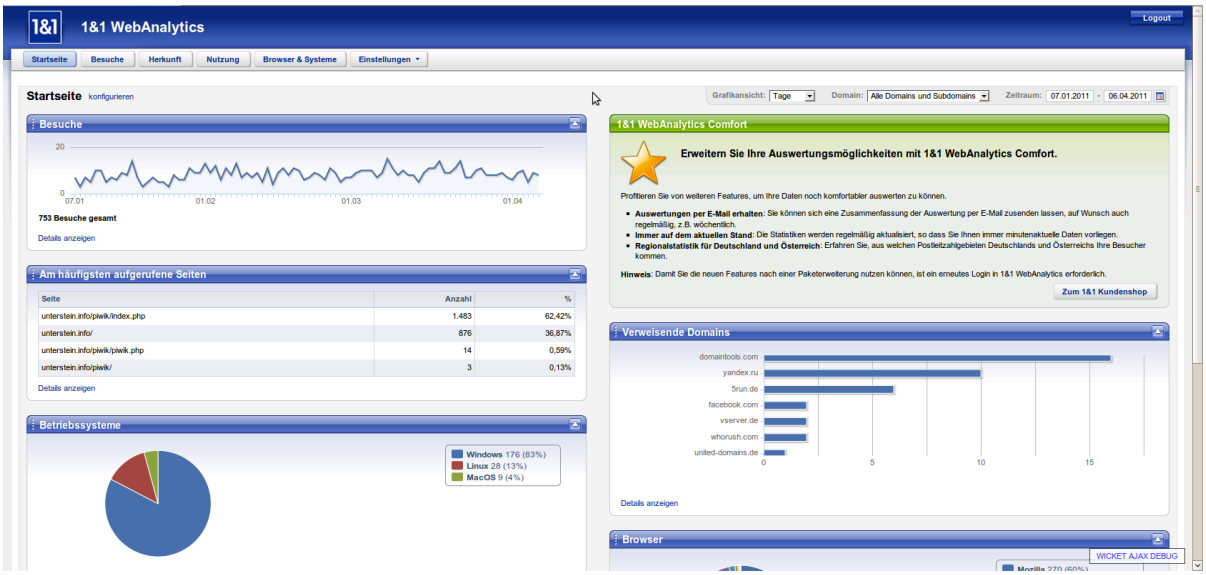
[Illustration: Distribution the burden of critical software pieces!]

- Single commits can be reviewed as well
    - ➔ Via commit mails
    - ➔ In dedicated reviews in the continuous development process
- The difference of two revisions be reviewed either
    - ➔ Useful for comparison before/after of a refactoring, re-theme, …

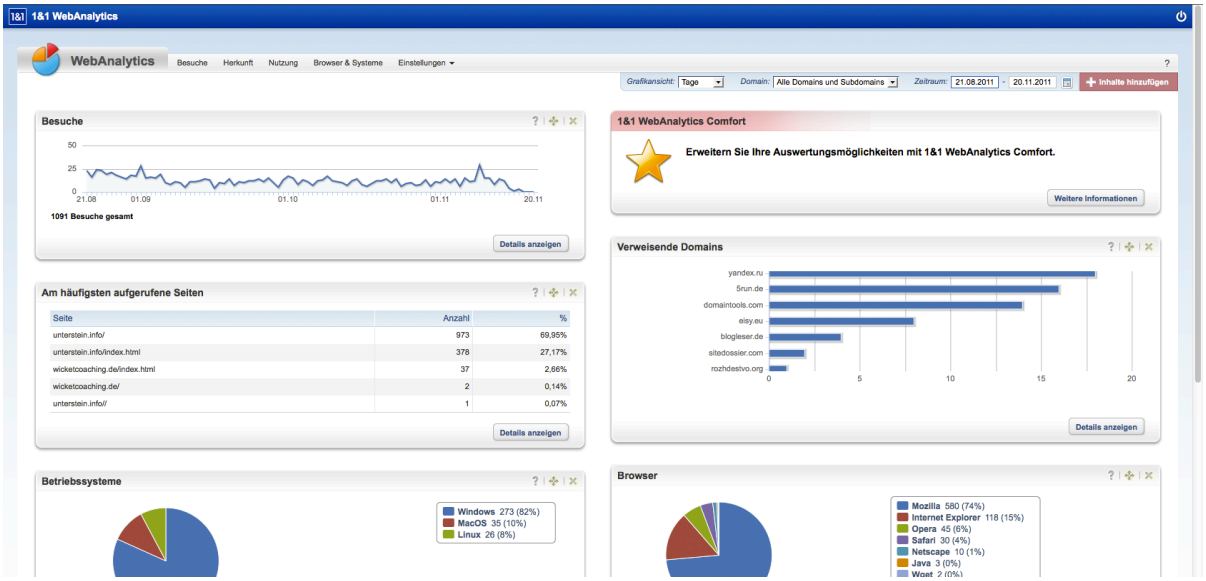
[Question: Difference single commit vs. compare two revisions?]

[Illustration: Webstat before/after]

Before:



After:



➔ Need dedicated tools
➔ And dedicated attention

- It is necessary to define a subset of components for the last two points listed above as well
- It is possible for the reviewers to add new classes while reviewing
  - ➔ Differs through tool support
  - ➔ Very useful!

- Size matters!
  - Developers are expensive, and you need a lot of them for a review :)
  - Extraction of our logs:
    - ➔ I spend about 3 to 4 hours of preparation for each review of a new colleague
    - ➔ Average size: 8 classes with each < 500 loc
    - ➔ Our review meetings took about 4 to 5 hours in average

[Question: What do you think about the return of investment? It is expensive? Do managers spend the time?]

- With about 30€ per hour per developer and 8 developers per review you raise the bill rapidly over 2000€ per review
  - ➔ Price is very moderate in comparison to usual costs when your software has an incident breakdown/critical bug!

[Short story: Maybe a short story about an incident…]

Who

- Roles
- Author or authors
  - Most important persons for the review. Without them there would be nothing to review
  - Role during individual preparation passive (no reviewers)
  - Important while review meeting
  - Difficult to handle
    - ➔ Should not defeat code (fixed on current implementation)
    - ➔ Should discuss objectively

[Question: What do you think? It is easy? You thought about your implementation!]

- Organization (mostly done by the author or mentor)
  - Kickoff: "Yes, we make a review"
  - Find time slot where all needed participants are free
  - Infrastructure: Meeting room with beamer and notebook
    - ➔ Or working station in normal office if review is small enough
- Reviewers
  - Responsible for good preparation
  - Responsible for quality improvement of the review
  - Should make good and constructive remarks

[Illustration: good vs. bad remark]
Situation:

```
/**
 * Gets the name
 * @return the name
 */
public String getName() {
     return this.name;
}
```

Bad: “Not again! Remove this stupid boilerplate comments”
Good, like Adam Bien said: “Don´t waste time writing comments for getters”

➔ Can also make remarks with question marks, when he is not sure
  - Share remarks before review

- Moderator
  - One of the reviewers
  - Controls the mouse ;)
  - Leads discussions
  - Responsible for an objective lead of the discussions while the review meeting
  - Good understanding of the technical problem domain and well accepted by the rest of the team

People

- The according team
  - Try to get as much as possible together
  - Each member profits from participating a code review
  - Each member influences others while participating a code review
  - Paltry excuses are invalid
    - ➔ Individuals should not try to avoid the effort to participate of reviews!
    - ➔ Bad mood in the team if important people are missing
- Good mixture of participants
  - Experts give their experiences, newbies give their fresh and unconsumed ideas
    - ➔ No separation between experts/newbies
    - ➔ All has same voice!
  - Maybe not the best idea to invite the "head of"
    - ➔ Influences the criticism

Common review process

Preparation

- Initialization "Hey, lets do a review"
    - Why? Which reasons?
        - ➔ New co-worker
        - ➔ Fancy new feature
        - ➔ "I made some critical chances and want to get assurance"
        - ➔ The other points from slide "why"

[Short story: My kickoff while team meeting]

- Motivation inside the team
    - Not forced from the outside
    - Must be accepted, pick up unmotivated members
    - Usually kick off during team meeting

[Question: What would you say, when your head of comes and say “Ey, do a review!”]

- Who organizes review?
    - Mostly through the authors of the reviewed code
        - ➔ Have greatest interest at the review
        - ➔ Have technical skills to set up review
    - Could be organized through team assistance either
        - ➔ Notice technical required skills to set up review
        - ➔ Reduces effort in development team

- Decide which code should be reviewed
    - See slide "What to review"
        - ➔ Which components?
        - ➔ What size?

- Decide who will join the review
    - See slide "Who"
        - ➔ As much as possible
        - ➔ For good reasons members could decline in the kick off phase

- Find a reasonable time slot
    - Friday afternoon is a very unthankful time slot
    - Monday morning as well ;-)
    - Not easy to find ~5 hour time slot in 4-10 developer´s calendars
        - ➔ Requires tool to plan meeting and access to calendars
    - Plan meeting one to two weeks in the future
        - ➔ Good for preparation, bad for "real time" behavior

- Do not forget important infrastructural points
    - Meeting room
    - Beamer
    - Notebook

- Plan breaks, fresh air, coffee and other drinks

Distribution

- Distribute the according code to all participants
  - Extremely simple with up to date tools
    - ➔ Take a versioning tool your choice (svn, git, cvs, mercurial, ...)
  - Actually works with .zip distribution...
    - ➔ But ... people without versioning tools typically do not bother about quality improvements through code reviews ;-)

- It is important that all participants review the same revision
  - Usually development goes on after a review is triggered
    - ➔ Unprofitable if participants reviewing trunk
    - ➔ Unprofitable if all participants reviewing different versions
  - Use tags for a clear defined version
  - Mostly tags. Tags are easier to handle then dedicated revisions

- It is also important that all participants gains access to all required tools
  - Versioning tool
  - Review tool
    - ➔ Installation, Web based needs accounts

Individual phase

- The phase where all reviewers review the defined code separately
  - Make your own thoughts about the present code
  - Try to understand and check documentation
  - Check code style, formatting rules, common best practices
    - ➔ Only one return in a method, pattern usage, qualified access, method naming, class naming, method organization, ...
  - Check correctness

- Each reviewer makes personal remarks in the code
  - Meta information, requires tool support
  - Take care of the wording
  - Be polite
    - ➔ Comments will be visible for all

[Remark: See good vs. bad remark]

- Requires a lot of discipline of all reviewers
  - Self-management
  - Objective and constructive criticism
  - Ability to read code where are less points to remark
    - ➔ Benefit for reviewer, reads good code
  - Ability to read code where a lot of points to remark are present
    - ➔ Do not bother about: "Oh no, he did this mistake ten times before", maybe it was not clear for the author that it is a mistake
    - ➔ Endurance to be objective the through the whole code

- First and important step of knowledge distribution
    - Spread knowledge of reviewed component over the reviewers
    - Knowledge about underlying code is present in more then one head
        - ➔ Redundancy, ...
    - Often the bulk of improvements are in the individual phase

- The reviewers must be thorough while reviewing the code
    - Useless to invest hours to receive a superficial result
    - Authors recognize if reviewers did their work good
    - Reviewers should take care of their wording

[Short story: About recognition of bad reviews]

- A good individual phase requires time, take it!
    - Not always easy to manage in the daily business
    - The result excuses the effort! It is worthwhile

The review

- Plan enough time
    - Like said before, discussions requires time
    - Block about 3-5 hours, depends on amount of code
- Do not bother about the usual discussions
    - If you know, that there are points with different and tightened positions ... just limit them to the minimum
    - StringBuilder, getter documentation, @author, null in conditional, ...

[Question: What are your typical discussion points?]

- The team review phase fits not all organizations
    - Open source projects manage quite well by skipping this phase
    - World wide distributed teams are maybe not able to meet

- The actual process
    - Go through all points sequentially
        - ➔ Maybe start with an easy component for warm up
        - ➔ But concentrate then to major components first
        - ➔ Less chance to run out of time for major things
    - The team decides if remarked issue is valid or not
        - ➔ Mostly leads to discussions, except obviously issues
        - ➔ Classifies the issues, e.g.:
            - Valid needs fixing (a real issue that needs to be fixed)
            - Valid fix later (a real issue that you won't fix right away)
            - Valid duplicate (real issue that it's already been mentioned)
            - Valid won't fix (a real issue that you don't want to fix)
            - Invalid won't fix ("it ain't broke, don't fix it!")
            - Unsure validity (needs further investigation)
        - ➔ Rate the issues according their severity, e.g.:
            - major
            - minor
            - trivial
            - enhancement

        - ...
    - If the point is valid, the team develops a strategy how the issue can be resolved
        - ➔ Again: if you have 5 developers, you get 7 strategies to solve the issue

[Question: Do you have examples?]

    - After making clear how to fix the issue, it must be cleared who will fix it
        - ➔ Alternatively all issues or a subset of them is thrown to a gathering pool where different develops picks out them later
        - ➔ No matter who made the defect, it matters who will fix it
        - ➔ Maybe it requires a specialist for solving the issue
        - ➔ Do not waste your time searching the author of a mistake, mostly leads to stupid fingerprinting instead of productivity
            - In the context of a code review, the originator is more or less obvious

[Question: Do you know the situation where you find a defect you did not made? Did you fix it, or did you find out the author to fix it?]

Rework and check

- The developers solve the valid issues
    - Different strategies
        - ➔ Major first
        - ➔ Mass first
        - ➔ Sequentially
        - ➔ Randomly
        - ➔ Ordered by needs

- After solving an issue mark the ticket as resolved

- According reviewer will close the ticket
        - ➔ Should check if the issue was resolved correctly
        - ➔ If the issue was not solved correctly the reviewer reopens the ticket

- Optionally: Review the rework in a dedicated review
    - Mostly done through the check if the issue was solved correctly in a distributed review over the according reviewers

Further aspects

Profit – People

- The authors profit
    - Receives very valuable feedback
        - ➔ Very good chance to gain own coding skills
    - Profits from the essence of experience of the co-workers
        - ➔ Be sure: the others were in your position before

➔ Wants to prevent you from mistakes
➔ Reduces the burden of single developers
- E.g. going online with payment system for thousands of customers is not as easy as it sounds :-)

- The reviewers profit either
  - Reads (hopefully good) code
    ➔ Reading code is always (mostly) an enrichment

[Illustration: Reading code]
Good code: You can learn from
Bad code: You can learn how to improve/refactor code
-> Don`t bother: You will do this a lot in your career
- Leads to know how exchange
  ➔ Reviewers get introduction of what the authors did
  ➔ Profitable for already mentioned points (redundancy, fluctuation, vacation, ..)
- Again: Profits from the essence of experience of the co-workers

[Question: What kind of review intention could you imagine?]
- Newbies or developers where you think "hmm, quality is not the best .."
  - Profit: Quality improvements at reviewed developer
- Experienced developers where you think "quality if quite fine!"
  - Profit: Quality improvement at the reviewing developers

Profit – Code

- Reviews lead to less bugs and prevent you from critical ones
  - Critical bugs often exist because of misunderstanding
  - or bad communication (not always covered through reviews)
  - Critical ones could mostly be identified through a second opinion

- Therefore: Code quality improves through reviews extremely
  - Common coding standards, ...
  - Common acceptance

[Short story: Outsourcing and acceptance afterwards]

- Improves the knowledge about the produce, the code or the project of other team members
  - Sometimes: Project affects only a subset of the whole team
    ➔ One Frontend, two Backend, ... whatever
    ➔ Other members need knowledge about this project as well

- Therefore: Spread the knowledge to be sustainable

[Short story: Critical bugs while vacations]

Psychological aspects

[Short story: Romania – way of work]
- Acceptance from authors

- Review member, who belongs very long to a team, might not lead to acceptance
  - ➔ I`m the hero
  - ➔ "We did it the past years in that way"
  - ➔ "We do it always that way"
- The feeling "oh god, my code is reviewed" is almost new
  - ➔ Needs time to clear the feeling
  - ➔ Against the natural claim of computer scientists:
    - "i produce high quality and bug free code"
    - "i make no mistake"
- You could feel inspected, disgraced, but transparency is always good
  - ➔ If you did a good job, you can show it
  - ➔ If your coding style needs improvements, you´ll get it there
  - ➔ Leads to transparency

- If your code is reviewed, remark that your reviewers try to improve your code and help you
  - Do not take objective view of code as personal criticism
  - Sometimes remarks not easy to handle from co-workers

- Ensure that it is a positive experience for the author
  - He will not accept another review easily otherwise
  - Resistance could grove

Automatic analysis

- Goals
  - Tries to identify some kind of errors while development process automatically
  - Tries therefore to improve the quality
  - While coding (implicitly), triggered manually (explicitly) or during the build process (implicitly)

- Classification of static code analysis
  - Inspects just the static code "plaintext"
  - Formatting, naming convention, memory leaks, code redundancy, ...
  - Simplest occurrence: compiler
    - ➔ Static type conformity
  - Static code analysis tools are meta programs
  - Useful complementation to software tests and code reviews
    - ➔ Very helpful during coding
  - Ensures quality on another level then manually code reviews
    - ➔ Offers basics and ensures preconditions
- Formatting, code styles

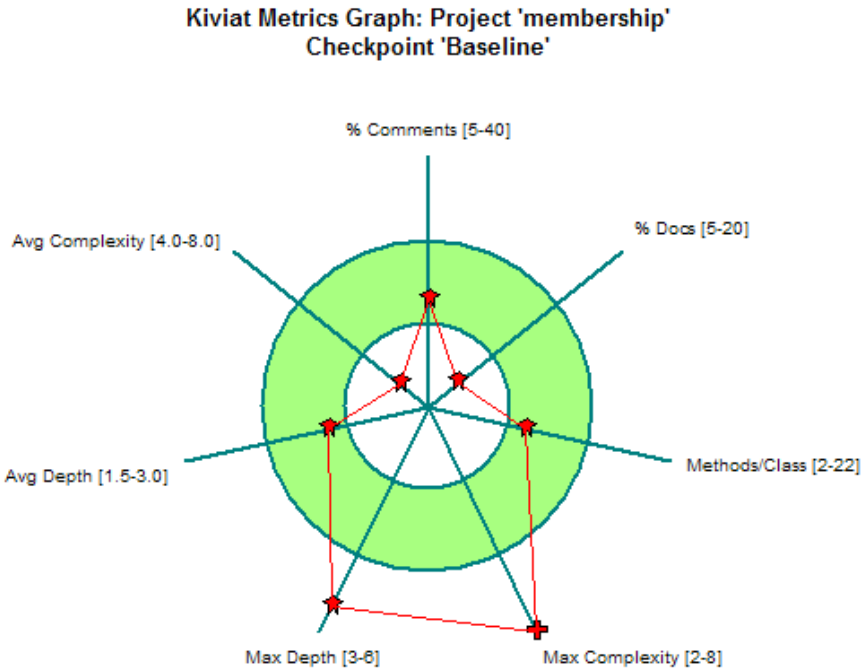
- Tools
  - Checkstyle
    - ➔ Automates checking java code according naming conventions, documentation available, duplicate code, white spaces, ...
  - IDE build in tools, e.g. eclipse java tools

➔ imports, unused vars, qualified access, serial uid, ...

[Illustration: Configure it!]

- Findbug, a little bit more intelligent checks
  - ➔ null pointer checks, instantiation support, makes nice reports
- Other code metric tools like code analysis á la source monitor
  - ➔ % comments, average complexity, methods per class, ...

[Illustration: example code metrics]



- Basic support, should just give static advises
  - Fits nicely in build processing tools (hudson/jenksins, bamboo) and IDEs
  - Should stay as static advisor
  - Not enough for stand alone use

Outsourcing

- Code reviews could also (like everything) be outsourced
- Externals review the piece of software and give result and advise
- Outsourced reviews have only profit for code, no profit for the own developers

- Pros:
  - Externals are mostly more objective then internal developers
    - ➔ Have no emotional binding to the underlying software
  - Externals may be specialized to a certain kind of software, which should be reviewed
  - Externals bring fresh ideas to the software
    - ➔ Maybe invite current a freelancer to a review

- Cons:
  - External code reviews sounds like "yes...our developers are not able to produce the quality we need. Let´s see what externals could do."

- Internal developers have know how either
  - ➔ Nobody knows a system as good as their developer
- Internal developers know best about internal process, standards, best practices, communication partners, ...