Software Engineering Project Workshop (SENG202)

Matthias Galster

Reviews and refactoring

September 14, 2020

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Shared space for this session

- Google Docs
  - https://docs.google.com/presentation/d/1PX2rvmsh184ugiB48qiMP6yfKWO9R8fpkODjsaH5B7k/edit?usp=sharing
- Link also on Learn
  - COVID-19 section under “Schedule changes” for 14 September
- Everybody can edit
  - No need to log in

Reminders

- Time left for Phase 2

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
September 14	September 15	September 16	September 17	September 18	September 19	September 20
September 21						

- Submit your weekly individual reflection (Mondays, 5:00pm)

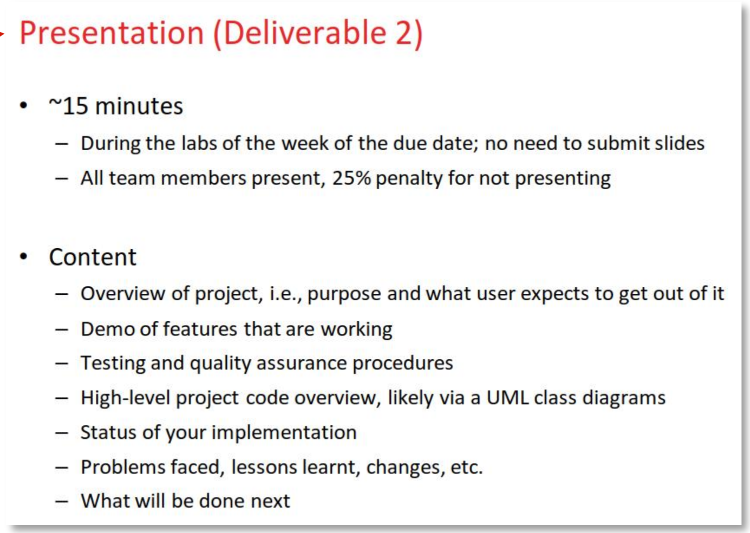
- Keep logging as you go (follow instructions)

- Labs
  - Same allocation as last week
  - Tutorial session and quiz: AT; stand-ups and feedback sessions

Presentations for Deliverable 2

- Record presentation (e.g., as recorded Zoom meeting)
  - Edit if needed (e.g., with OBS-Studio or OpenShot Video Editor)
  - Attend presentation session (either on-site or online)
  - Play recording (either on-site or in Zoom)
  - Be available for questions after presentation
  - Comment and ask questions after presentations of other teams

Remember from introduction to Phase 2



Presentation (Deliverable 2)

- ~15 minutes
  - During the labs of the week of the due date; no need to submit slides
  - All team members present, 25% penalty for not presenting
- Content
  - Overview of project, i.e., purpose and what user expects to get out of it
  - Demo of features that are working
  - Testing and quality assurance procedures
  - High-level project code overview, likely via a UML class diagrams
  - Status of your implementation
  - Problems faced, lessons learnt, changes, etc.
  - What will be done next

Reviews in software engineering

Standard practice in industry

Most efficient way of identifying problems and errors

Types of reviews

Requirements reviews

Code reviews

Architecture reviews

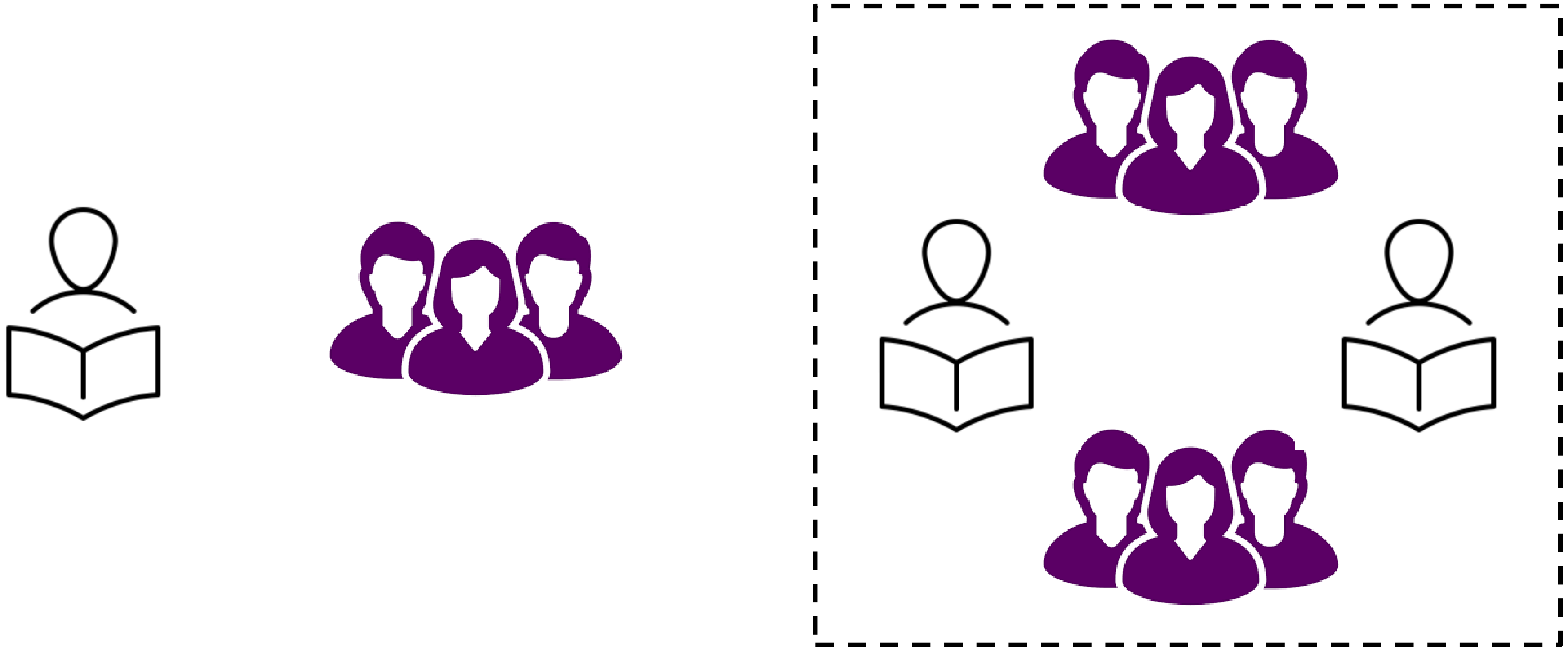
Product reviews

Design reviews

Process reviews

Etc.

Who reviews



Why review

Provide fair and constructive feedback to “reviewee”

- Allow “reviewee” team to improve their work
- Identify problems that “reviewee” is not aware of
- Improve quality of artefacts

Gain insights for own project, reflect on own behavior

- Reading artefacts makes sensitives for own issues
- May get inspired by good ideas of other teams
- Share knowledge

Good reviews

CommitStrip.com

Go's

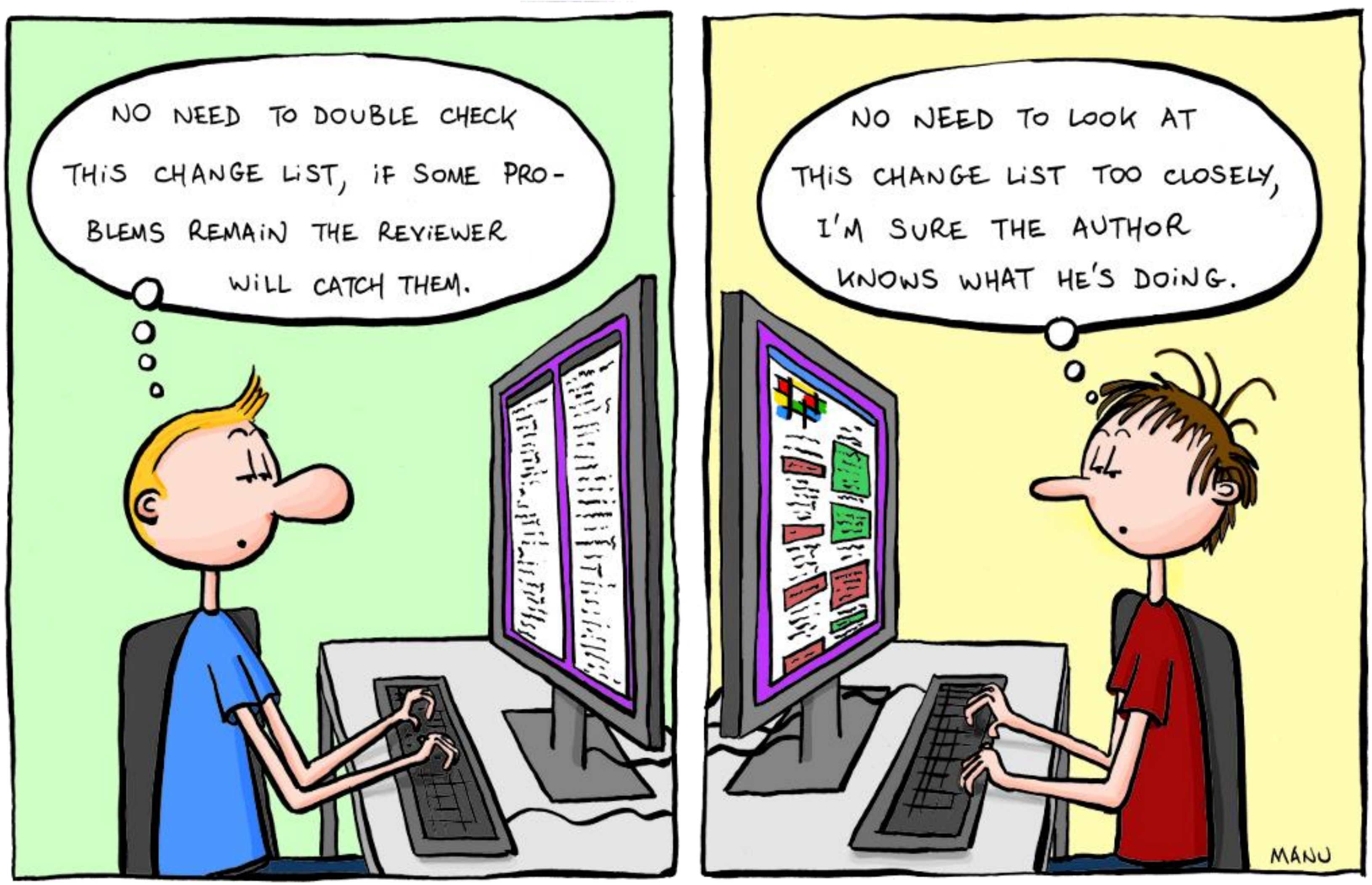
- Content, style, structure
- Constructive
- Objective, not subjective
- Thorough
- Respectful

No-go's

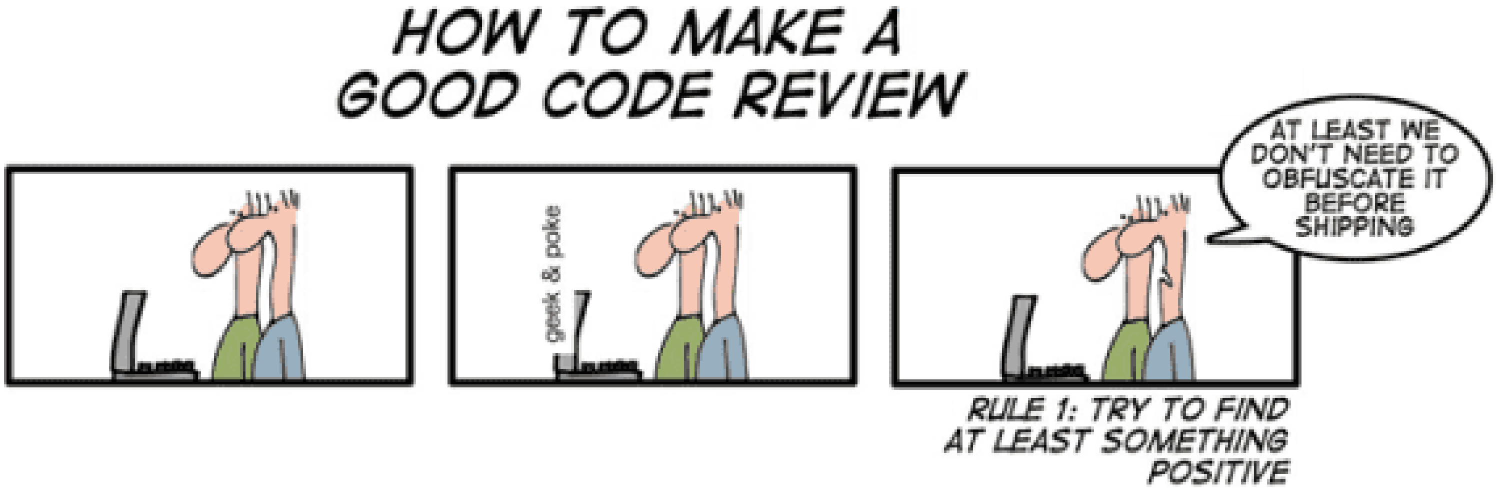
- Ridicule team members
- "So wrong", "This is awful"
- Focus on part of changes
- Personal issues in review
- Skim and accept

Often performed manually, effort-intensive, rely on experience of reviewer

Be thorough

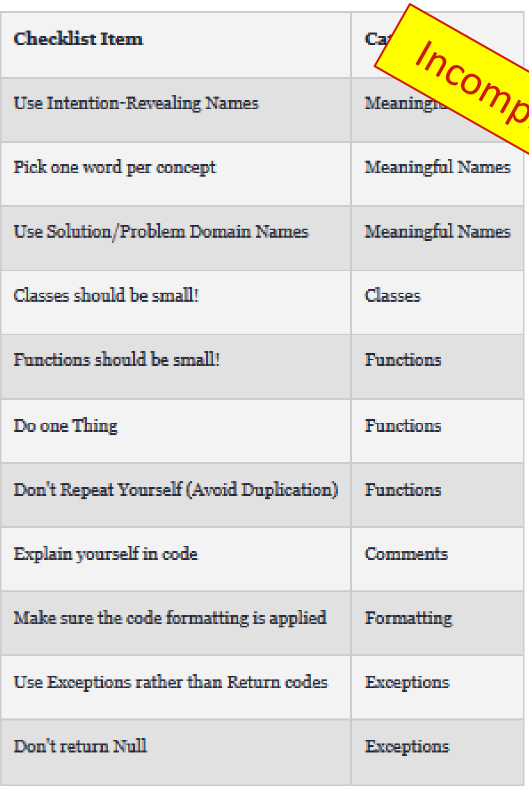

NO NEED TO DOUBLE CHECK THIS CHANGE LIST, IF SOME PROBLEMS REMAIN THE REVIEWER WILL CATCH THEM.
NO NEED TO LOOK AT THIS CHANGE LIST TOO CLOSELY, I'M SURE THE AUTHOR KNOWS WHAT HE'S DOING.
MANU

Be respectful


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE

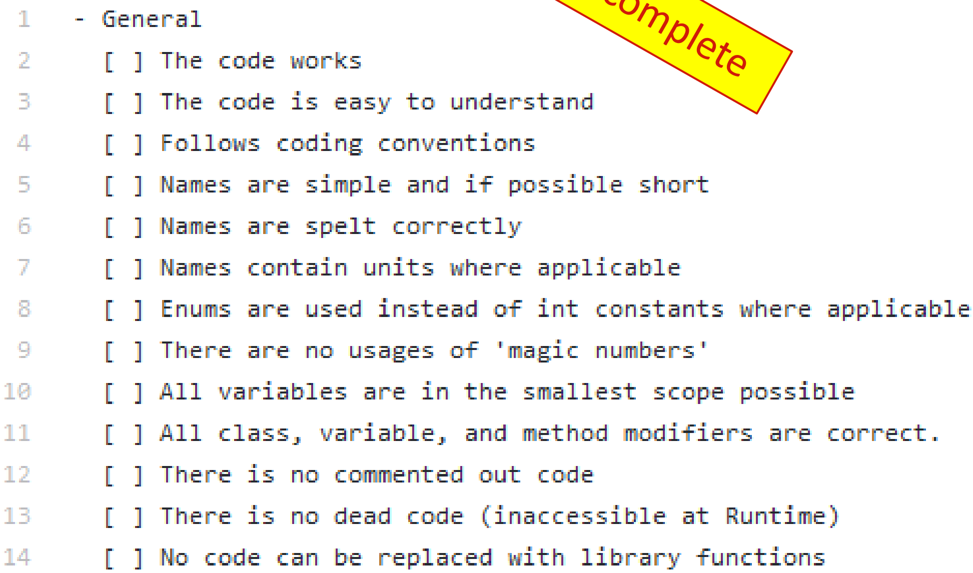
Example code review checklists

- (Java) code review checklist
  - https://dzone.com/articles/java-code-review-checklist
  - https://gist.github.com/kashifrazzaqui/44b868a59e99c2da7b14



Checklist Item	Ca
Use Intention-Revealing Names	Meaningf
Pick one word per concept	Meaningful Names
Use Solution/Problem Domain Names	Meaningful Names
Classes should be small!	Classes
Functions should be small!	Functions
Do one Thing	Functions
Don't Repeat Yourself (Avoid Duplication)	Functions
Explain yourself in code	Comments
Make sure the code formatting is applied	Formatting
Use Exceptions rather than Return codes	Exceptions
Don't return Null	Exceptions

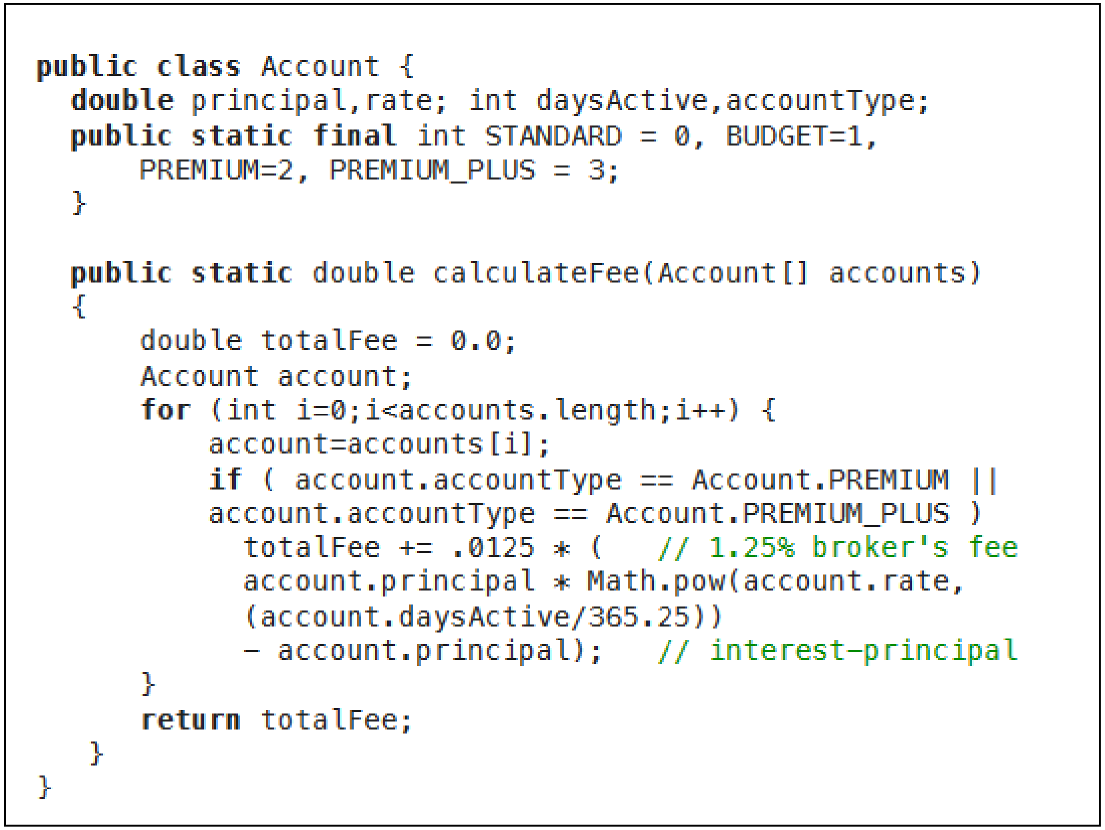
Incomplete



```
- General
  [ ] The code works
  [ ] The code is easy to understand
  [ ] Follows coding conventions
  [ ] Names are simple and if possible short
  [ ] Names are spelt correctly
  [ ] Names contain units where applicable
  [ ] Enums are used instead of int constants where applicable
  [ ] There are no usages of 'magic numbers'
  [ ] All variables are in the smallest scope possible
  [ ] All class, variable, and method modifiers are correct.
  [ ] There is no commented out code
  [ ] There is no dead code (inaccessible at Runtime)
  [ ] No code can be replaced with library functions
```

Incomplete

Example



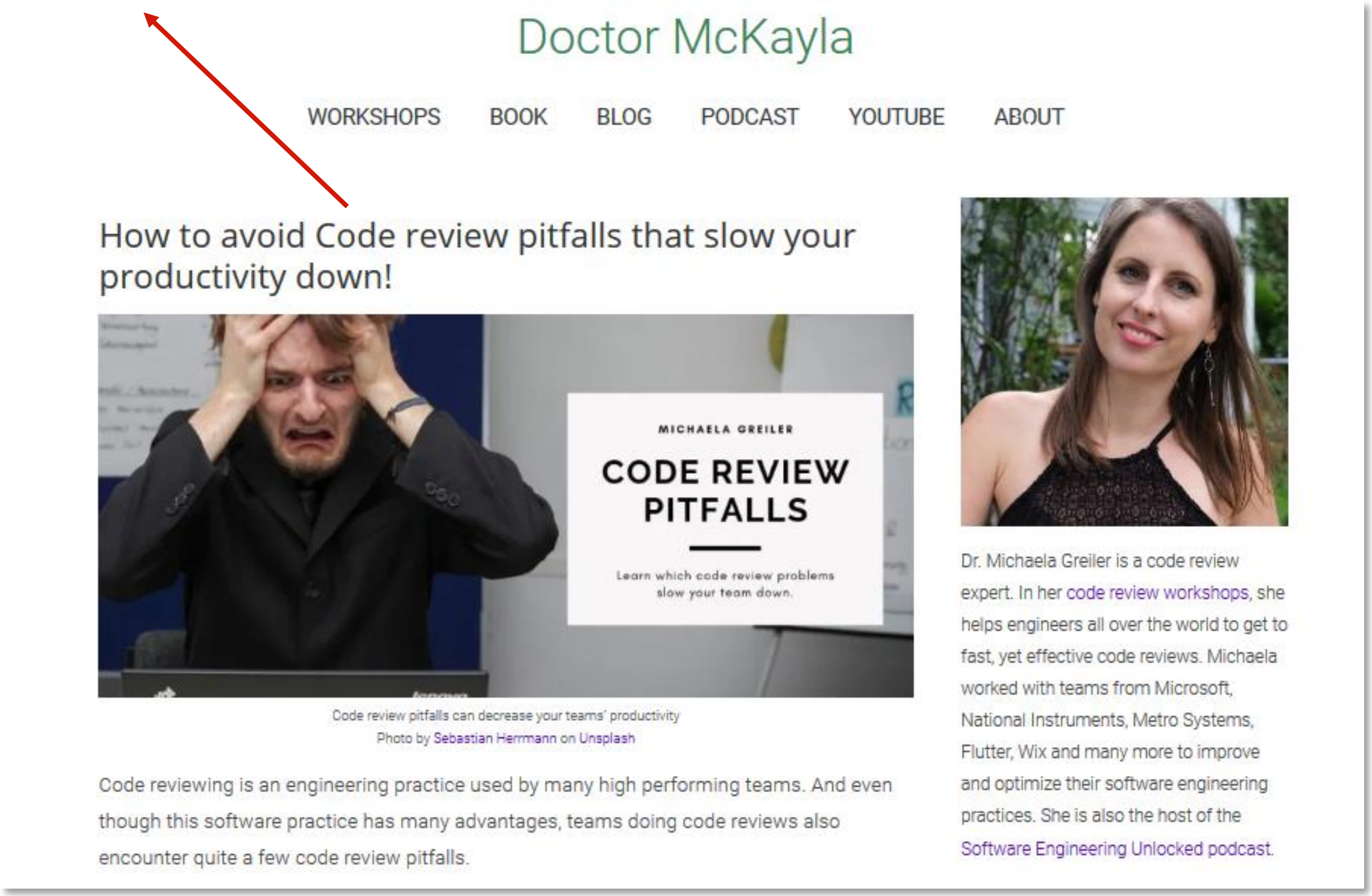
```java
public class Account {
 double principal,rate; int daysActive,accountType;
 public static final int STANDARD = 0, BUDGET=1,
   PREMIUM=2, PREMIUM_PLUS = 3;
 }

 public static double calculateFee(Account[] accounts)
 {
   double totalFee = 0.0;
   Account account;
   for (int i=0;i<accounts.length;i++) {
     account=accounts[i];
     if ( account.accountType == Account.PREMIUM ||
     account.accountType == Account.PREMIUM_PLUS )
      totalFee += .0125 * (   // 1.25% broker's fee
      account.principal * Math.pow(account.rate,
      (account.daysActive/365.25))
      - account.principal);   // interest-principal
   }
   return totalFee;
  }
}
```

Example improvements

- Comment
- Private fields
- Replace “magic” numbers with constants
- Enum for account types
- Consistent white spacing, line breaks, etc.

(Code) review pitfalls

! Most also apply to other types of reviews



Doctor McKayla

WORKSHOPS BOOK BLOG PODCAST YOUTUBE ABOUT

How to avoid Code review pitfalls that slow your productivity down!

MICHAELA GREILER

CODE REVIEW PITFALLS

Learn which code review problems slow your team down.

Code review pitfalls can decrease your teams' productivity
Photo by Sebastian Herrmann on Unsplash

Code reviewing is an engineering practice used by many high performing teams. And even though this software practice has many advantages, teams doing code reviews also encounter quite a few code review pitfalls.

Dr. Michaela Greiler is a code review expert. In her code review workshops, she helps engineers all over the world to get to fast, yet effective code reviews. Michaela worked with teams from Microsoft, National Instruments, Metro Systems, Flutter, Wix and many more to improve and optimize their software engineering practices. She is also the host of the Software Engineering Unlocked podcast.

https://www.michaelagreiler.com/code-review-pitfalls-slow-down/

Reviews and smells

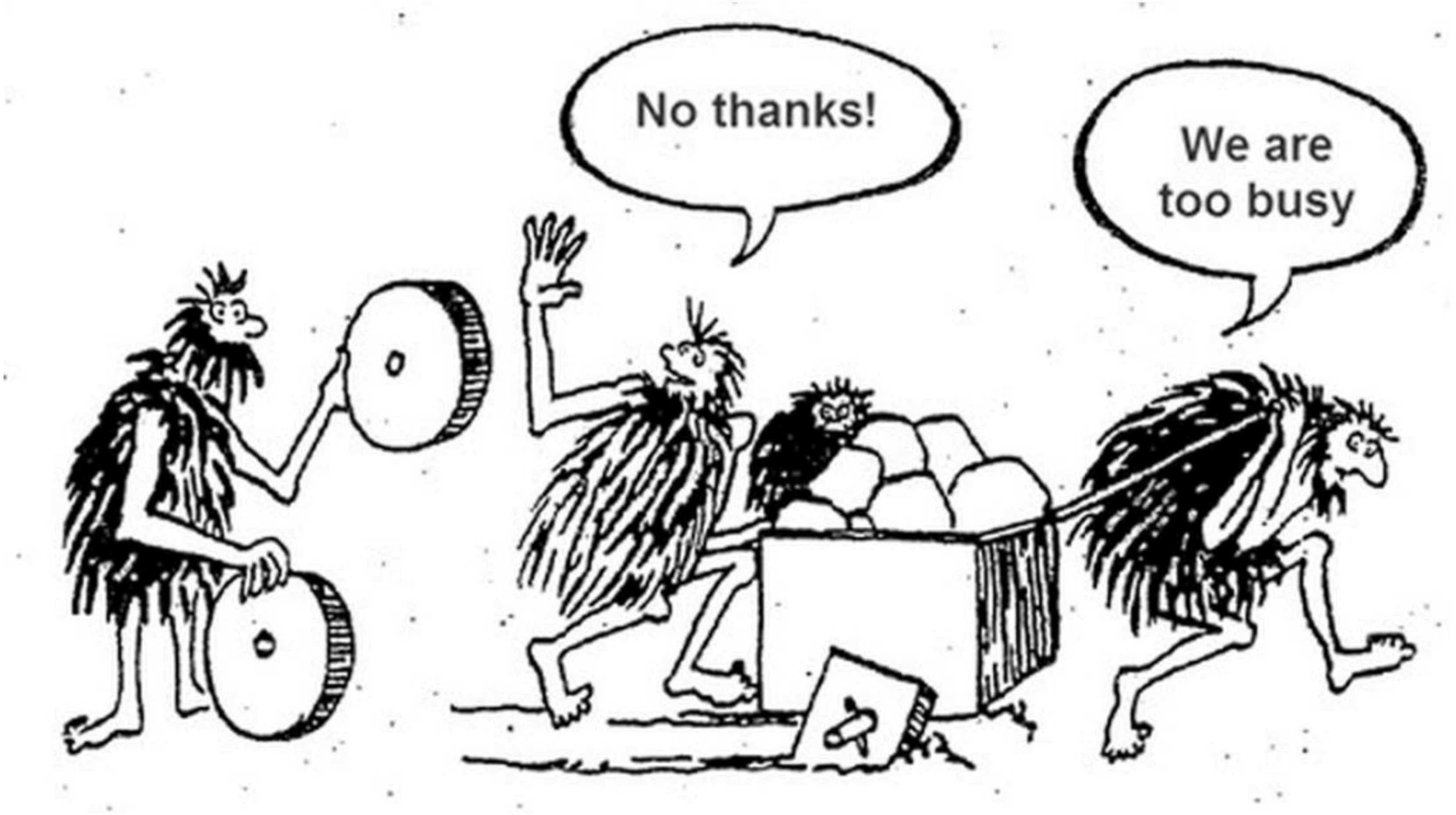
- Smell: not about technically incorrect design or code
  - Do not currently prevent program from functioning
- Indicator that something may be wrong in design, code, etc.
  - May slow down development or increase risk of bugs, future failures
- Violation of principles, e.g.,
  - Tendency of a module to be difficult to understand
  - Clear and expressive code versus convoluted code
  - Elements not currently useful in the design

*Could appear in product code/design or test code/design

What to do about smells: refactor

- “Semantic-preserving” transformation of design and code
  - Improves structure but not behaviour after code has been written
    - Enhance quality
    - Make code easier to read, more flexible, easier to change
  - Applies to design artefacts, such as UML models, classes, code

- Long-term investment in the quality of code and its structure
  - Avoid refactoring may save costs / time in the short term
  - But: penalty in the long run (see also: technical debt)


No thanks!
We are too busy

OO examples – smells within classes

- Comments
  - Comments that illuminate vs comments that obscure
  - Refactor comments

- Long methods
  - Shorter methods are easier to read, understand, troubleshoot
  - Refactor long methods

- Long parameter list
  - The more parameters, the more complex
  - Limit parameters or use objects

OO examples – smells within classes

- Large class
  - Difficult to read, understand, troubleshoot; too many responsibilities
  - Restructure, break into smaller classes
- Duplicated code
  - C+p is useful for test editing, but can be disastrous for code editing
  - Repeating structures that could be unified as single abstraction
- Combinatorial explosion
  - Lots of code that does almost the same thing, but with tiny variations
  - Difficult to refactor (generics?)

OO examples – smells within classes

- Dead code
  - Remove code that is not used anymore

- Temporary field
  - Objects with lots of optional or unnecessary fields
  - Better “calculate” values rather than keeping them as properties?

OO examples – smells between classes

- Primitive obsession
  - Sets of primitive data types instead of classes
- Data clumps
  - Related and unrelated data kept together
- Refused bequest
  - Inherited functionality from a class but never used

OO examples – smells between classes

- Lazy class
  - Class with little functionality
- Message chains
  - Long sequences of method calls
- Feature envy
  - Classes that make extensive use of other class may belong in other class

Catalogues of smells and refactorings

- Martin Fowler’s online catalogue
  - – www.refactoring.com/catalog/index.html

- Some smells
  - – http://mikamantyla.eu/BadCodeSmellsTaxonomy.html
  - – www.codinghorror.com/blog/2006/05/code-smells.html

- Smells / refactorings
  - – http://sourcemaking.com/refactoring
  - – http://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/

M. Fowler, K. Beck. J. Brant, W. Opdyke, D. Roberts. Refactoring: Improving the Design of Existing Code. Pearson Education, 1999.
B. Meyer., Object-oriented Software Construction. Prentice Hall, 1997.
A. J. Riel. Object-oriented Design Heuristics. Addison-Wesley.

Example

```
public class Gorilla
{
          ...
          int paws()
          {
                    return 4;
          }
}
```

Introduce explaining variable

```
public class Gorilla
{
          ...
          int paws()
          {
                    int pawCount = 4;
                    return pawCount;
          }
}
```

Example

```
public class Gorilla
{
        ...
        int paws()
        {
                int pawCount = 4;
                return pawCount;
        }
}
```

Extract interface

```
public class Gorilla implements Primate
{
        ...
        int paws()
        {
                int pawCount = 4;
                return pawCount;
        }
}
```

```
interface Primate
{
        abstract int paws();
}
```

Example

```
public class Gorilla implements Primate
{
        ...
        int paws()
        {
                int pawCount = 4;
                return pawCount;
        }
}
```

```
interface Primate
{
        abstract int paws();
}
```

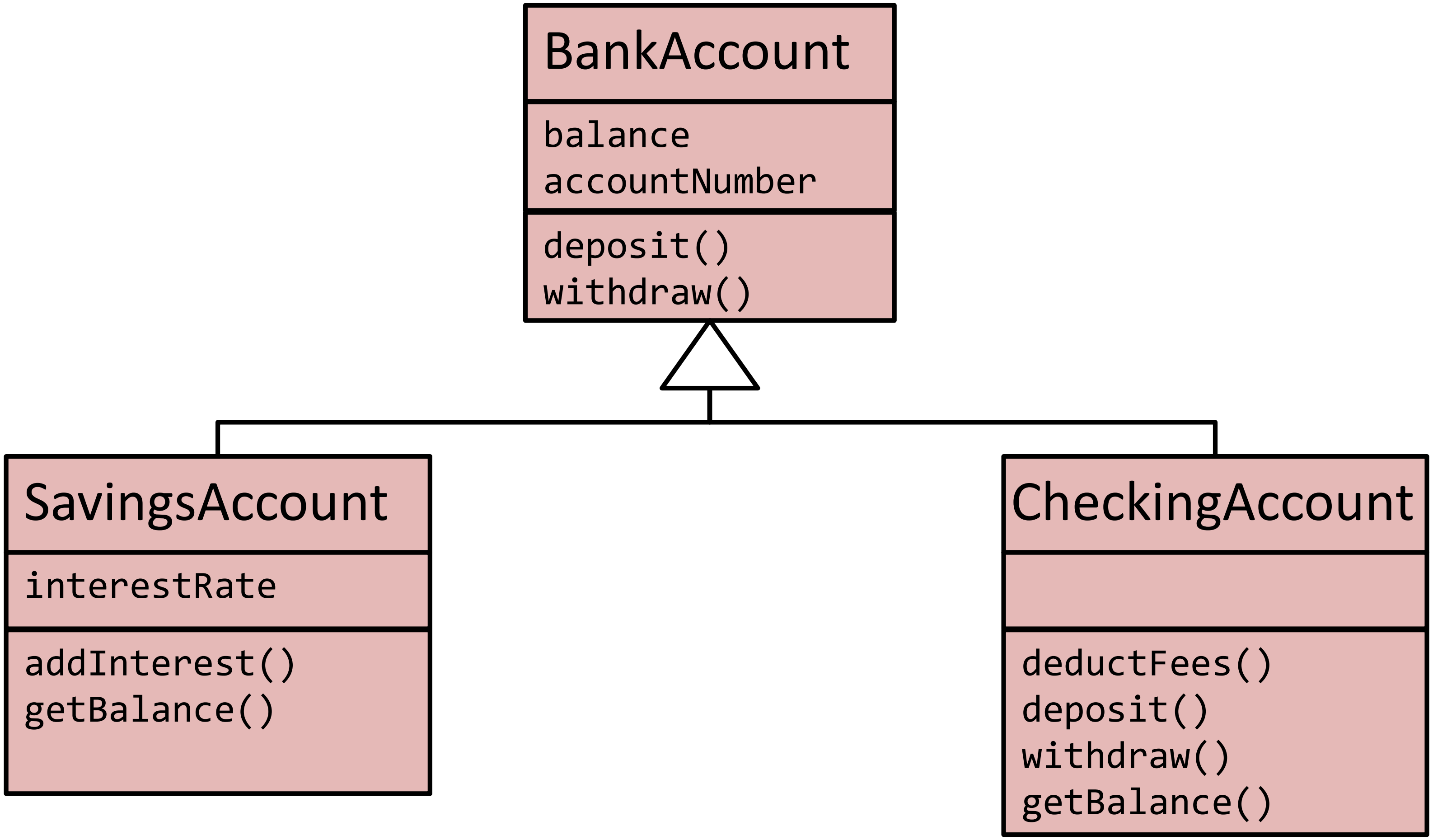
Rename method

```
public class Gorilla implements Primate
{
        ...
        int feet()
        {
                int pawCount = 4;
                return pawCount;
        }
}
```

```
interface Primate
{
        abstract int feet();
}
```

Example

- Pull up Method
  - If identical methods in more than one sub-class, move them to super class



One smell – multiple refactorings

- Smell
  - Duplicate code
  - Code repeated in multiple classes

- Possible refactorings
  - Extract method
  - Extract class
  - Pull Up Method

Schedule until final due date

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
September 14	September 15	September 16	September 17	September 18	September 19	September 20
September 21	September 22	September 23	September 24	September 25	September 26	September 27
September 28	September 29	September 30	October 1	October 2	October 3	October 4
October 5	October 6	October 7	October 8	October 9	October 10	October 11
October 12	October 13	October 14	October 15	October 16	October 17	October 18