Module 3: Code Review and Documentation in Software Development

CSc3350

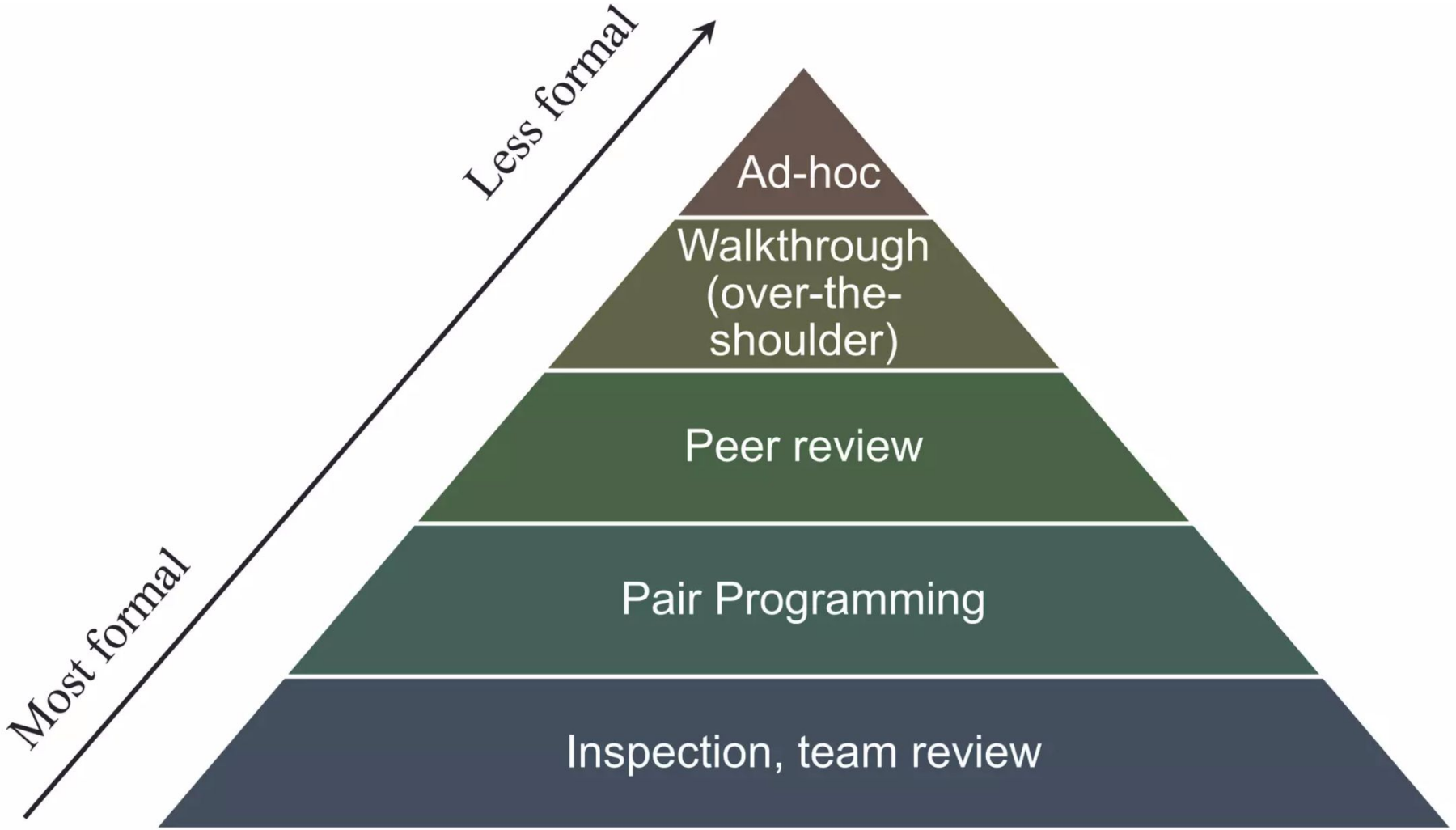
Fall 2023

Dr. William Greg Johnson
Department of Computer Science
Georgia State University

Module 3: Code Review and Documentation

- Goals
  - Improve quality
  - Share knowledge
  - Improve collective code ownership
  - Conformance checking
  - Completeness validation
  - Learn
- Benefits
  - Share knowledge
  - Discover bugs early
  - Maintain compliance
  - Enhance security
  - Increase team collaboration
  - Improve quality of code
- Disadvantages
  - Longer time for completion
  - Distraction from other tasks
  - Large reviews = longer review times

Classification of Code Reviewing



Approaches to Code Review

1. Pair programming
2. Over the shoulder reviews
3. Tool-assisted reviews
4. Email passing around

Code Review: Pair Programming

Benefits

- Transfers knowledge
- Prevents information silos
- Solves complex problems
- Increases morale
- Finds more bugs
- Can be conducted remotely

Drawbacks

- Time-consuming
- Can be overused
- Difficult to measure

Code Review: Over the Shoulder

Benefits

- Easy implementation and completion
- Can be conducted remotely
- Faster than pair programming

Drawbacks

- Reviewer is detached from code
- Review moves at the author's pace
- Lack of objectivity
- No verification that changes were made
- Difficult to measure

Code Review: Tool Assisted

Benefits

- Easier to gather metrics
- Automated tooling frees up developer focus

Drawbacks

- Developers must maintain tools
- Expensive
- Will still require teammate reviews

Code Review: Email Passing Around

Benefits

- Easy implementation and completion
- Facilitates remote, asynchronous reviews
- Automatic reviews via Software Configuration Managers

Drawbacks

- Time consuming to gather files
- Difficult to follow conversations
- No definite review end date
- No verification that changes were made
- Difficult to measure

Code Review: Best Practices

- **Limit code review sessions to keep them productive.** Figure out what works for your team — say, no more than one hour or 200 lines of code — and encourage them to stick to that limit.
- **Include everyone — even new and senior members of the team — in the process.** Gives an excellent way to help newer members of the team get up to speed with the code base — both by reviewing code from others and having their code reviewed by more senior.
- **Distribute code review requests amongst the team.** It is easy for a few developers to get the bulk of code review requests, and this won't be good for them or the rest of the team — or the code base — long term. This puts knowledge in a risk with only a few people having it.
- **Ask questions and provide helpful context.** When you're reviewing someone's code, do your best to help both of you learn during the process. Not sure why they did something a different way than you might have? Ask. Have a suggestion for how to improve their code? Let them know why you're suggesting it in your comment.

Code Review: What to look for?

- Design: Is the code well-designed and appropriate for your system?
- Functionality: Does the code behave as the author likely intended? Is the way the code behaves good for its users?
- Complexity: Could the code be made simpler? Would another developer be able to easily understand and use this code when they come across it in the future?
- Tests: Does the code have correct and well-designed automated tests?
- Naming: Did the developer choose clear names for variables, classes, methods, etc.?
- Comments: Are the comments clear and useful?
- Style: Does the code follow our style guides?
- Documentation: Did the developer also update relevant

Code Review: Why does it matter, in Agile?

- Code reviews share knowledge
- Makes for better estimates in time to deliver and cost
- Creates multiple informed inputs to Sprints
- Enables team members to have ‘time off’
- Eliminates the ‘critical path singularity’ i.e., one team member has all the answers
- Mentorship to newer junior software engineers

More to come…

Documentation in Java: JavaDoc

Importance of JavaDoc

- The JavaDoc tool is a document generator tool in Java programming language for generating standard documentation in HTML format.
- It generates API documentation.
- It parses the declarations ad documentation in a set of source file describing classes, methods, constructors, and fields.

Javadoc is:

- a convenient, standard way to document your Java code.
- a special format of comments. There are two kinds:
  1. class-level comments: Class-level comments provide the description of the classes
  2. and member-level comments: member-level comments describe the purposes of the members.
- all Javadoc comments start with /** and end with */. For example:

```
/** this is a Javadoc comment */
```

Structure:

- A Javadoc comment is set off from code by standard multi-line comment tags /* and */.
- The opening tag (called begin-comment delimiter), has an extra asterisk, as in /**.
- The first paragraph is a description of the method documented.
- Following the description are a varying number of descriptive tags, signifying:
    1. The parameters of the method (@param)
    2. What the method returns (@return)
    3. Any exceptions the method may throw (@throws)
    4. Other less-common tags such as @see (a "see also" tag)

Javadoc tags

Tag & Parameter	Usage	Applies to	Since
@author *John Smith*	Describes an author.	Class, Interface, Enum	
@version *version*	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum	
@since *since-text*	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method	
@see *reference*	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method	
@param *name description*	Describes a method parameter.	Method	
@return *description*	Describes the return value.	Method	

Javadoc tags (continued)

Tag & Parameter	Usage	Applies to	Since
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.	Method	
@deprecated *description*	Describes an outdated method.	Class, Interface, Enum, Field, Method	
{**@inheritDoc**}	Copies the description from the overridden method.	Overriding Method	1.4.0
{**@link** *reference*}	Link to other symbol.	Class, Interface, Enum, Field, Method	
{**@value** *#STATIC_FIELD*}	Return the value of a static field.	Static Field	1.4.0
{**@code** *literal*}	Formats literal text in the code font. It is equivalent to <code>{@literal}</code> .	Class, Interface, Enum, Field, Method	1.5.0

Example using HTML tags

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {
    public static void main(String[] args) {
        /* Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

5

Resources:

- https://web.archive.org/web/20170613233020/http://agile.csc.ncsu.edu/SEMaterials/tutorials/javadoc/
- https://www.protechtraining.com/bookshelf/java_fundamentals_tutorial/javadoc

Attributions

1. Aleksey Solntsev, https://www.slideshare.net/alexsun/code-review-without-animation
2. GitLab, https://about.gitlab.com/topics/version-control/what-is-code-review/
3. GitHub, https://google.github.io/eng-practices/review/
4. Francois Camus, https://www.slideshare.net/FranoisCamus/code-quality-24626716
5. Pallavi Srivastava, https://www.slideshare.net/PallaviSrivastava7/java-docs